A Paradigm for Decentralized Process Modeling and its Realization in the Oz Environment

Israel Z. Ben-Shaul

CUCS-014-95

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences.

Columbia University
1995

© Israel Z. Ben-Shaul 1995
ALL RIGHTS RESERVED

ABSTRACT

A Paradigm for Decentralized Process Modeling and its Realization in the Oz Environment

Israel Z. Ben-Shaul

This dissertation investigates decentralization of software processes and Process Centered Environments (PCEs), and addresses a wide range of issues concerned with supporting interoperability and collaboration among autonomous and heterogeneous processes, both in their definition and in their execution in possibly physically dispersed PCEs.

Decentralization is addressed at three distinct levels of abstraction. The first proposes a generic conceptual model that is both language- and PCE-independent. The second level explores the realization of the model in a specific PCE, Oz, and its rule-based process modeling language. The third level addresses architectural issues in interconnecting autonomous PCEs as a basis for process interoperability.

Two key concerns guide this research. The first is maximizing local autonomy, so as not to force a priori any global constraints on the definition and execution of local processes, unless explicitly and voluntarily specified by a particular process instance. The second concern is tailorability, dynamicity and fine-grained control over the degree of interoperability.

The essence of the interoperability model lies in two abstraction mechanisms — *Treaty* and *Summit* — for inter-process definition and execution, respectively. Treaties enable to specify shared sub-processes while retaining the privacy of local sub-processes. To promote autonomy, Treaties are established by explicit and active participation of the involved processes. To promote fine granularity, Treaties are defined pairwise between two collaborating processes and formed over a possibly small sub-process unit, although multi-site Treaties over large shared sub-processes can be constructed, if desired. Finally, Treaties are superimposed on top of pre-existing instantiated processes, enabling their dynamic and incremental establishment and supporting a decentralized bottom-up approach.

Summits are the execution counterparts of Treaties. They support “global” execution of shared sub-processes involving artifacts and/or users from multiple sites, as well as local execution of private sub-processes. Summits successively alternate between shared and

private execution modes, where the former is used for synchronous execution of shared activities, and the latter for autonomous execution of any private subtasks emanating from the shared activities as defined in the local processes.

Contents

List of Figures

List of Tables

To my dear parents, Rivka and Simcha Ben-Shaul, for their support, devotion, and love.

Acknowledgements

First and foremost, I would like to thank my advisor Gail Kaiser, for her continuous guidance and support, for conveying to me the essence of research, for teaching me how to write research papers through her always comprehensive reviews, and for convincing me to enroll in the PhD program. I would also like to thank my other committee members John Kender, K. Narayanaswamy (Swamy), Lee Osterweil and Ken Ross. In particular, I thank Lee for his insightful critique of my thesis, and Swamy for his continuous contributions through formal and informal discussions over the past three years.

Special thanks to George Heineman, my colleague, office mate and friend throughout my PhD studies, for the endless discussions, long programming nights, and for reviewing the entire dissertation. Many thanks to Peter Skopp, for his numerous contributions to the communication infrastructure and to the overall system-level aspects of Oz. Thanks to the following past and present project students whom I had the pleasure to work with over the years: Steve Linde, who implemented Treaties; Yong Su, who extended the client with multi-site support; Hsin Liu, for her implementation of the cache manager; John Hinsdale, for his numerous contributions to the Marvel project that were used in Oz, including schema evolution, ad-hoc query processor, and feasibility study of PCTE; Will Marrero, for his work on process evolution; Moshe Shapiro for his work on access control; and Tony Bunter for his work on query evaluation. Other students who contributed to the Oz project include Shelly Tsellepis who worked on file transfer, Andrew Lih who worked on connectivity through secure sites; Andrew Tong, who implemented the CMarvel process that was used for the development of Oz; and Jack Yang, for his work on the ISPW demo.

Last but not least, I would like to thank my dear wife, Tamar, for her continuous support, encouragement and endurance through difficult times, and my daughter Mika, for giving me wonderful moments of joy.

1

Introduction

Software Engineering (SE) is aimed at constructing cost-effective and high-quality large-scale software. As such, it is concerned with methodologies, tools, and frameworks that can assist groups of developers throughout the lifecycle of a software product.

Software Development Environments (SDEs) is a sub-field within SE that is more specifically concerned with providing frameworks, or infrastructures, for supporting the development of software products. Thus, the SDE community is less concerned with the development of specific tools but rather with the integration and interoperability of these tools within an environment; it is not only concerned with support for an individual user but also with support for coordination and interaction among multiple users participating in the development of a product; and, finally, it is concerned with the management and integrity of the various *artifacts* involved in a software product (e.g., modules, libraries, design documents), which are manipulated by diverse *users* using diverse *tools*.

Early research in SDEs, dating back to the mid seventies, focused on support for coding and debugging [9]. Called programming environments, these systems often included a set of language-specific tools (e.g., language-based editors) that assisted an individual programmer to code programs.

There was a transition in the eighties, when language-based editors were used as front-ends to integrate a set of language-based tools. Examples included Pecan [91], which emphasized visualization; and generator environments like Gandalf [41], and the Synthesizer Generator [93].

More advanced SDEs focused on general purpose tool-integration techniques. An example system was Field [92], in which tools interact with each other by sending and receiving messages to a centralized Broadcast Message Server, effectively implementing a "software bus" from which tools can be easily added or removed, thereby providing extensibility. Another direction in SDE research was to provide comprehensive support to the entire lifecycle of a software project, not only programming. Termed Integrated Project Support Environments (IPSEs) [101], these systems had to also support multiple users (as opposed to earlier single-user systems), and often included management of the software artifacts. ISTAR [27] is a representative IPSE that emphasized integration of managerial as well as engineering tasks and tools[1]. Other systems from that era include DSEE [69] and NSE [53].

1.1 Process Modeling

In order to provide comprehensive project support, SE researchers and developers had to observe and understand project development processes. This marked the beginning of the "**process**" era [70], pioneered by Humphrey [55] and Lehman [71]. The realization that the process of constructing and maintaining software was crucial in determining the success of the project — while at the same time widely different from project to project, depending on the nature of the product being developed, management policies, tools used and so forth — has led the SDE community to shift its focus towards support for the process as a key factor in increasing productivity and improving the quality of software.

The term "**software process**" can be defined as an orderly approach to applying methods and tools to software development. It includes:

- The set of activities carried out during the development process. These activities can be low-level activities, such as invoking a compiler on source code, or they can be high-level and decomposable activities (also called tasks), such as the integration test phase of a large system. Process activities can be further categorized by the degree of computer support they require, ranging from purely computer-oriented activities, such as compilation, to activities with partial computer support, such as code inspection, to purely human-oriented activities, such as design meetings.

---

[1] A more detailed account of ISTAR will be given in Chapter 2.

- Local constraints on activities, typically in the form of prerequisites to, or implications of, activities. This includes constraints imposed on the execution of activities and obligations that must be carried out as a result of the execution of activities. For example, a process might restrict depositing a modified source code to a repository that holds a stable version of the system, only if the source code had been statically inspected. An example of an activity implication might be that after the release of a new version of a product, all current licensees must be notified and get the upgraded version.
- Global constraints on tasks. For example, a release deadline constraint might affect all activities related to preparing the release.
- Partial ordering among tasks. For example, the well-known waterfall model [101] for software development implies sequential ordering among the various phases with feedback to previous phases.
- Synchronization among concurrent tasks. It is common for software processes to allow for concurrent execution of tasks. This implies that the process can also specify points where dependent tasks synchronize.

The interest in the software process has consequently led SDE researchers to explore ways to represent the process with a formal notation in order to support it. Pioneered by Balzer [2] and Osterweil [81], this research direction has become to be known as **process modeling**[2].

The advantages of using a formal notation for the definition of software processes are: (1) Understanding — clearly, by explicitly and rigorously defining processes one can clarify and gain a better understanding of the processes themselves. A written process also helps in explaining it to the personnel involved in the process, thereby achieving a better understanding among the project participants. Moreover, by using programming language concepts such as data and control *abstraction*, *modularity*, and *encapsulation*, processes can be better defined and understood much in the same way that programming and design in general benefit from these techniques. (2) Analysis — Static process analysis can be performed on processes modeled in a formal notation using well-understood techniques

---

[2] Although Osterweil referred to it as "process programming", this term later became associated with a more specific approach to process modeling.

from the programming languages domain, which can result in process improvement. For example, program optimization techniques can be used to eliminate redundant activities, to point out potential for increased concurrency among independent activities, to identify "dead ends" from which the process cannot proceed, and so forth. Furthermore, when processes are represented as state machines (e.g., as in StateMate [54]) formal automata techniques can be employed, for example for reachability analysis. (3) Execution — Perhaps the dominant aspect, at least within the SDE community, and the one that is emphasized in this thesis. Once there is a formalism that encodes a (software) process, there is a potential for the process to be interpreted by a "process machine" (or process engine) that is sensitive to the defined process, and can assist in its execution[3].

However, while there is an obvious resemblance between software processes and ordinary software, there is also a fundamental distinction between them. Whereas the operational semantics of the latter are fully and completely defined by the software program, the compiler, and the underlying machine (and operating system) on which the program runs, a software process model only defines a (possibly small) subset of the overall process. More importantly, the software process is not being executed completely on a physical machine, and it involves (unpredictable) humans carrying out significant portions of the process. As a consequence, the process engine cannot (and should not) control all aspects of the process, and all we can hope for is to find ways in which it can *assist* users in carrying out the process. This view has implications on the choice of the modeling paradigm and on the types of support that can and should be provided. Thus, in the remainder of the thesis we will use the term process modeling to encompass definition of the model regardless of whether it is executable or not.

The process community invented the term **process enactment** to describe assistance in process execution and distinguish it from the notion of program execution. Enactment is also sometimes confused with the concept of simulation (although process simulation might be a viable option) in that while a simulation might also involve nondeterministic agents and behaviors that describe natural phenomena, process enactment involves support of *real* execution of the process, involving real devices, tools, artifacts, and real users. There are several forms of enactment, or assistance:

1. *Enforcement* — This refers to the capability of the process engine to ensure

---

[3] In some cases, such as in APPL/A [104], process models are compiled and executed directly.

that constraints, obligations and general process invariants are maintained consistently. For example, a specific process might impose a constraint that no source file can be edited unless it has been properly checked out to a private workspace using the process' configuration management subsystem. In this case, the process engine will enforce this constraint and disallow violations, thereby maintaining process consistency as promised. More complicated consistency constraints might span a group of artifacts and a set of interrelated activities. For example, if a function signature has been modified, all callers (from different modules) should be outdated, to force recompilation. This constraint is likely to reduce errors from interface mismatch.

It is, of course, not desirable to enforce all aspects of the process on all individuals. Indeed, much of the criticism about process enactment stems from the impression that individuals are "controlled" by the process and must operate within strict rules that severely restricts their work and creativity[4]. The goal is to enforce those constraints and invariants that constitute process consistency (the "law of the system" [78]), but relax or even leave undefined other activities in the process. While different environments provide different enforcement capabilities, the degree of enforcement also depends largely on the specific process model, as defined on a project-specific basis. A system known for its enforcement support is Darwin [78].

2. *Automation* — This refers to the environment's capability to carry out some activities of the process automatically on behalf of users. Automation might be explicitly specified by the process model, or it can be inferred by the system. Note that automation can be used for enforcement. On the other hand, enforcement can be supported independently from automation. SDEs known for their automation support include CLF [89] and Marvel [59].

3. *Guidance* — The environment builder might choose to guide users in performing tasks in the process, without actually forcing them to do any of them. For example, the process might maintain a "to do" list of pending tasks. SDEs known for guidance support include ProcessWEAVER [32] and

---

[4] In that respect, the choice of the term "enactment" is misleading, but since it is so widely used in the process community we will stick to this term throughout the thesis.

Merlin [109].

4. *Monitoring* — This refers to the environment's capability to monitor the progress of the process, and accurately assess the state of the process at any particular point in time (other mechanisms can be used to actually extract the process state). An extension to process monitoring is process diagnostics, i.e., when monitoring detects problems that can be diagnosed and later repaired. Note that process diagnostics in this context refers to inspecting active enactable processes analogous to a debugger, and is different from static analysis mentioned earlier in the context of modeling. SDEs known for their monitoring support include SMART [34] and Provence [67].

Process Centered Environments (PCEs) are SDEs that provide a **Process Modeling Language** (PML) in which project-specific software processes are defined by a process administrator (as opposed to environment end-user), and a corresponding process enactment engine that is sensitive to the defined process and supports its execution in some or all ways described above.

Most, but not all, PCEs support some form of *data modeling*, both for the product artifacts which are being manipulated by the process (*product data*), and for the data used by the PCE itself to keep track of process state (*process data*). Moreover, some of those PCEs support data modeling on a per-project basis; in these cases, data modeling is considered to be part of the process, although it may be specified by a separate Data Definition Language (DDL).

Process modeling has increasingly attracted attention in the software engineering community, as evidenced by the Ninth International Software Process Workshop [37] and the Third International Conference on the Software Process [85]. Various PCEs have been constructed as research prototypes and non-commercial systems, and some have been recently released as commercial products. Examples of relatively well-known academic and other research PCEs include Arcadia [57], Common Lisp Framework (CLF) [89], Melmac [24], Merlin [109], Spade [3] and TEMPO [11]. Examples of commercial products include ProcessWEAVER [32], HP SynerVision [51], and Lion [73].

The state-of-the-art in PCE technology (including all the systems mentioned above), however, has been supporting centralized and homogeneous processes, for moderate-sized and often co-located groups.

1.2 Why Decentralization ?

Large-scale product development typically requires the participation of multiple people, often divided into multiple heterogenous groups, each concerned with a different facet of the product. For example, one software product may require different teams for requirements elicitation, functional specification, design, coding, testing, documentation, and maintenance; another product may also involve multiple teams, in this case with each responsible for full development of a distinct component of the system. Each team uses its own selection of tools, its own private artifact database, and its own management policies and development workflow — all parts of the process. At the same time, the teams need to cooperate in order to develop the product, and as studies in software engineering have shown [33], the interaction among team members accounts for a significant fraction of the total cost of the product being developed.

The degrees of team *autonomy* and *collaboration* between teams both depend on the nature of the product being developed and on organizational policies (e.g., centralized vs. decentralized management). Sometimes multiple independent organizations with preexisting processes need to collaborate on a product, in which case autonomy (privacy or security) is a “hard” constraint that cannot be compromised.

In recent years, there has been an explosive growth in telecommunication technologies and infrastructures that enable global communication (most notably, the Internet). This “globalization” provides immense opportunities for growth and collaboration among teams that are geographically dispersed and time shifted. Indeed, as a result of these enabling technologies, the field of Computer Support for Collaborative Work (CSCW) has gained popularity in recent years, providing tools and platforms for collaborations among multiple users (see [82] and Section 2.3.3). However, these technologies also introduce the hard problems of *heterogeneity* and *decentralization*, which will have to be taken in the near future as a given requirement, as opposed to a design by choice.

1.3 A Motivating Example

The following is a sample process that illustrates the problems in modeling and enacting decentralized processes. Assume there are three development teams working in separate sub-environments (henceforth SubEnvs) **SE1**, **SE2**, and **SE3**, who are responsible for three disjoint components of a system **S**, labeled **S1**, **S2**, and **S3**, respectively (see Figure 1.1). The teams operate in different sites[5] and reside in different geographical areas. They each maintain and develop their own private artifacts (represented as rectangles in the figure), using their private tool-set (triangles in the figure) and their own methods and policies, i.e., process (clouds in the figure).

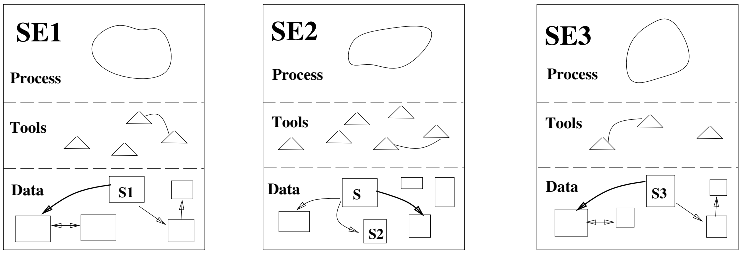
Each component can be coded and unit-tested independently, and the components are interconnected through published, well-defined, interfaces. Suppose **S2**'s interface has to be modified in order to enhance some of its functionality, thereby requiring the other components to change. The following steps are then taken (corresponding to the numbers in figure 1.1): (1) the proposed change has to be reviewed and approved by all SubEnvs; (2) the interface of **S2** is actually modified; (3) The affected components are modified to correspond to the new interface; (4) a local test of each component is performed; and (5) an integration-test with all revised components is performed. For simplicity, only the "successful" path, i.e., assuming that all the steps were carried out successfully, is described. (One example of an unsuccessful path would be a failure of the local-testing at one of the SubEnvs, which might require reiteration to step 1.)

While the global modification and integration test must be performed synchronously (with respect to all sites) and at one site, the review, local modification, and local test activities can be performed asynchronously in the local sites, and they can differ at different sites. For example, one site might employ "white box" local testing, while another site might use "black box" testing. Moreover, even identical operations might trigger different related operations when issued at different sites.

At the modeling level, there should be a conceptual framework that allows for the definition of interoperability of the autonomous processes on a per SubEnv basis, in terms of interactions and information exchange among them. At the enactment level, a DEcentralized PCE (henceforth DEPCE) should enable and support the execution of a decentralized task that possibly involves data from multiple sites. At the architectural level, there must exist an infrastructure that is capable of providing mechanisms for consistent and reliable access to shared data, communication protocols and capabilities for accessing remote data in a proper manner, and a decentralized enactment engine that *performs* well, both in terms

---

[5] Although the term *site* usually refers to an administratively cohesive Internet domain sharing a single network file system, we will use it throughout the thesis more liberally to denote *logical* cohesiveness of a computing unit, and therefore will use at times site and SubEnv interchangeably. In order to distinguish between logical and physical sites, we will refer to the latter as an Internet domain, or simply domain.



Change

1. **Review and approve (decentralized)**
2. **Modify interface (centralized)**
3. **Modify affected components (decentralized)**
4. **Unit–Test change (decentralized)**
5. **Integration test (centralized)**

Figure 1.1: A Motivating Example

of functionality and efficiency.

For this particular example, a *wrong* and in some cases impossible solution would be to collect all the necessary data from the remote sites to the coordinating site (i.e., **SE2**), and then carry out all the process steps (and all implied and triggered steps) on all data as defined in **SE2**'s process. Besides the obvious performance limitations, this approach would be a clear violation of autonomy, since each site has its *own* sub-process for its local activities (e.g., local testing), which may not even be known to the coordinating site. Another possible problem might be that some of the tools do not exist in all SubEnvs (e.g., because licensing binds a tool to a specific site or host), and other tools can be executed only in specific SubEnvs (e.g., special-purpose hardware), which necessarily binds the execution of a process step to a specific SubEnv. On the other hand, some of the operations involve data from multiple sites and must execute at a common location, and all sites have to agree on it. Therefore, the solution should enable handling of such work in a manner that retains maximum process autonomy and operational independence while still providing for collaboration and interaction among the processes as needed.

1.4 Research Focus

This thesis focuses on the *modeling and enactment of interoperability and collaboration among independent, autonomous, and possibly pre-existing processes.* Within this focus, it is important to clearly identify what issues are addressed in this research, and equally important, what issues are beyond the scope of this research.

1.4.1 Decentralization vs. Distribution

Broadly speaking, a distributed system is defined as one that provides a single, homogenous, logical perspective to its applications, but is physically distributed into multiple computing units, usually across machines of a single site. That is, a distributed system transparently shields the distribution from its applications [21]. In contrast, a decentralized system is comprised of relatively *independent* and heterogenous subsystems with some degree of correlation between them, perhaps (although not necessarily) spread among multiple sites. In particular, transparency is intentionally not supported, for several reasons. First, it inherently violates site autonomy since it implies unrestricted access to at least some remote resources and repositories. Second, transparency is simply irrelevant when the involved entities are heterogenous at the application level (i.e., they run different programs, see Section 1.4.2). Finally, transparency is undesirable when the entities are geographically dispersed, since it is often necessary to distinguish between different access costs, given that they can vary widely depending on the available bandwidth, available computing resources, etc. For example, if a component has a timeout mechanism on fetching objects for fault tolerance purposes, then it should be aware of the "degree of remoteness" of the object being accessed in order to determine when to time out.

Observing the evolution and scaling up of large systems, the natural order tends to be: (1) centralized control, (2) distributed control, (3) decentralized, or "federated" control (the best representative of this kind of evolution is the database field). The reader might wonder why skip over step 2 and jump directly into step 3, when the problems of step 2 are not yet resolved ? The answer is, that if transparency shields users and applications from knowing where the data is, and retains a uniform view of the data and the process, then the main problem becomes to provide this transparency. From the PCE research aspect, it is much more interesting to look at loosely coupled and autonomous systems that allow for different processes to coexist. Furthermore, environment distribution is a form of "vertical" scale-up, in that it allows for more users to work, but under the same process and within some bounded physical distance (typically a local-area network). This thesis explores mainly "horizontal" scale-up, where the number of users per group sharing the same process may not grow much (and in fact may consist of a single user), but the number of *groups* may be arbitrarily large, each group with its own private process and data but collaborating in a concerted effort with the other groups.

1.4.2 Process, Language and System Heterogeneity

Heterogeneity can be categorized into three levels: system, language, and application. For example, in a heterogenous database system, the multiple local databases can differ from each other in their inner structure (system heterogeneity), they can differ in their front-end Data Definition Language (DDL) while still having similar inner structure (language heterogeneity), and they can support different schemas at the different databases, written using the same DDL (application heterogeneity). Similarly, DEPCEs can have system heterogeneity by allowing different product databases to be used in different sites. They can support language heterogeneity by allowing different sub-processes to be written in different PMLs, and they can allow different processes written with the same PML to interoperate (application heterogeneity).

Each level can be further categorized based on the degree of heterogeneity. For example, system heterogeneity can vary from component difference (e.g., different databases) to more substantial architectural difference (e.g., message bus vs. client server). Also, there can be different combinations of the above. For example, there could be support for multiple languages to define different aspects of a single, centralized process, or there could be support for interoperability between different subsystems with different corresponding PMLs that are used for support of different aspects of the same process. Thus, while decentralization usually implies heterogeneity, the reverse is not necessarily true.

Support for heterogeneity is, in general, an extremely difficult problem. This thesis explores a limited aspect of heterogeneity within the context of decentralization. The system and language levels are fixed, that is, it assumes, for the most part, homogeneity at the system level and at the PML level, and focuses on heterogeneity at the process level.

That is not to say that the other aspects of heterogeneity are totally ignored. For example, it is hoped that the language level can be addressed by translating various PMLs to an underlying "assembly" PML, similar to the approach taken in some Heterogenous Distributed Data Bases (HDDBs) (e.g., [66]). In fact, some evidence that this approach is feasible was given by the work described in [60]. Furthermore, in investigating architectural support for decentralization, some level of system heterogeneity, namely componentization, is considered and is one of its guidelines (see Chapter 5). Nevertheless, multi-PML support and componentization are, by and large, guidelines (and constraints) on this research as opposed to subjects of the research, and are partially addressed in the theses of Popovich [88] and Heineman [46], respectively.

Finally, while this research focuses on process heterogeneity and assumes homogeneity at the language level, it does not restrict itself to a specific PML, but instead attempts to provide a high-level abstract model that can be implemented by a family of PMLs.

1.4.3 Bottom-up vs. Top-down

As mentioned above, this thesis looks into interoperability between multiple, possibly pre-existing, processes. This implies a bottom-up view on the construction of a multi-site environment, without necessarily having any *a priori* knowledge of the "neighboring" processes at the time of construction. This is in contrast to decomposing a single process in a top-down fashion into sub-processes with predefined and coordinated interfaces. There are architectural implications which further distinguish between the two approaches, as will be seen in Chapter 3.

1.4.4 Inter-Process Collaboration vs. Intra-Process Coordination

Intra-process coordination is concerned with coordinating concurrent activities that might violate the consistency of the project database, assuming that all participants use the same process, the same schema, and most importantly, share the same centralized, project database. In contrast, this research is about collaboration between users or teams with different processes, different schemas, and most importantly, different project databases. (Work on intra-process coordination has been investigated in the Marvel project, primarily by Barghouti [6], and is discussed briefly in Section 2.2.)

1.4.5 Logical and Physical Decentralization

Logical decentralization refers to multiple autonomous and heterogenous processes which are enacted separately but are physically co-located (i.e., operate within the same local area network), and physical decentralization adds the dimension of physical separation between the SubEnvs, with arbitrary communication bandwidth between them. Physical decentralization obviously has implications on the architecture (e.g., variable bandwidth, no shared file system, security, see Section 5.6), but is likely to also affect the general model. Nevertheless, the problem of providing interoperability among heterogenous and autonomous processes can be examined independently of the additional constraints and problems associated with having those processes enacted in arbitrary physical separation. Thus, while this thesis discusses at length the architectural considerations and the actual design of a DEPCE that supports physical dispersion among the SubEnvs, the generic model is at the logical level and applies to both.

1.5 Requirements

We now formulate the general problems and motivations discussed above into a set of well defined "research requirements" which guide the design of the model and its realization. This is a high-level overview of the requirements. More detailed refinements of these requirements are given in the relevant chapters, as the context to understand them builds up.

1. *Process Locality* — A basic requirement is that as far as local work is concerned, a DEPCE should provide the same capabilities and same support as a PCE does. The underlying assumption is that most of the work done by a site is local to that site, and therefore each site should still be optimized towards local work. Thus, a DEPCE subsumes the capabilities of a PCE. We address here mainly the additional requirements for a DEPCE that are not PCE-specific, and extensions to existing PCE requirements. ([17] discusses general requirements for a PCE).

2. *Process Autonomy* — Each local SubEnv should have complete control over its process and data, while allowing access by remote SubEnvs under restrictions that are solely determined by the local SubEnv. Access to a process

has two perspectives: (1) access to the local artifacts owned by the process through a process *interface*; and (2) access to, and use of, the process itself. Autonomy constraints imply that, by default, a site allows no sharing of process or data. Moreover, once defined, sharing should be minimized to the degree necessary for interoperation. Autonomy is a strong requirement with major influence on all aspects of this thesis.

3. *Process Collaboration and Interoperability* — Autonomy trivially exists when there is no possible interaction between the SubEnvs. However, we are interested in allowing interoperability between multiple independently-defined heterogenous SubEnvs, each serving a group of users. Collaboration in this context refers to not only enabling "read-only" access to remote process and data, but also enabling operations that might affect the process state and the product data of remote processes. Such interoperability is particularly difficult in the context of processes, because of the richness of the semantics associated with process and its modeling power.

4. *Independent Operation and Self-Containment* — Related to autonomy, this means that a SubEnv should be able to behave as a complete environment by itself when not collaborating with any other SubEnvs, and SubEnvs must be able to operate concurrently and independently, except when their processes explicitly collaborate. The most fundamental implication of this requirement is that it requires a "share-nothing" architecture. That is, no service, mechanism, or data, in the environment can be centralized or shared across SubEnvs, and all interaction should be based solely on message passing. An additional implication of this requirement is that SubEnvs must be prepared to be dynamically disconnected/reconnected from/to each other when interoperating, without disrupting the operation of local tasks in individual SubEnvs, and moreover, they cannot rely on having all sites always active. Besides the architectural implications, this requirement also effects the conceptual model in various ways, as will be seen in Chapter 3.

5. *SubEnv Interconnectivity* — To support process interoperability, there must be an infrastructure that supports connectivity between the participating

SubEnvs. This includes a name service to identify and address SubEnvs and a communication service to exchange messages between them. Autonomy and independence constraints exclude a centralized name service and require a mechanism to (re)connect to, and automatically invoke, remote SubEnvs.

6. *Dynamic Reconfiguration* — A related issue to interconnectivity is that of site configuration. A DEPCE should have the capability to dynamically add new SubEnvs and remove inactive SubEnvs without disrupting the operation of the currently active SubEnvs. The concept of an inactive SubEnv is a direct implication of requirement 4. An inactive SubEnv is one that is temporarily dormant, although it may have recently been active, and could potentially be active in the future. In contrast, an active SubEnv is one that is currently under execution.

7. *Support for Pre-existing SubEnvs* — A DEPCE should enable a SubEnv with a pre-existing process to "join" a global environment with other pre-existing process(es), with minimal overhead. Similarly, a "split" of a SubEnv from its current global environment should be supported. This requirement is important when two or more organizations with established processes need to collaborate for a limited time.

8. *Data Sharing, Querying, and Presentation* — While remote data access should not be transparent in a DEPCE and governed by the process operating on it as discussed earlier, there should still be a mechanism that enables sites to access, query, and browse through data residing at remote sites, provided that the access is granted by the owner process. Moreover, since PCEs often support complex and highly structured data models, it is especially desirable to be able to display graphically the types of data and the relationships among them. This represents a challenge both in user interface design, and in the communication protocols that are responsible for updating the user's view(s). Chapter 5 elaborates on this issue.

9. *Transaction Support*— Multi-user SDEs in general, and PCEs in particular, require sophisticated and flexible concurrency control (CC) and failure recovery mechanisms [7, 18]. DEPCEs add the dimension of remote vs. local

access and potential heterogeneity of transaction management policies. This complicates CC and recovery because extended transaction models devised for centralized, and even distributed systems, might not be adequate. For example, if semantics-based CC is employed, then different processes impact their local CC policy differently, requiring some sort of negotiation between local CC engines. Furthermore, operational independence excludes any sort of centralized transaction service.

10. *Flexibility* — This is perhaps the most important (meta) requirement. It is concerned with the general approach to be taken to fulfill all the above requirements. That is, analogous to one of the most important characteristics of PCE technology — modeling a process on a per-project basis and its enactment by a process-sensitive engine — a DEPCE should similarly possess such flexibility and be able to specify the degree of autonomy as well as the collaboration on a per-SubEnv basis, and not by a hard-wired policy. Flexibility, autonomy and independent-operation requirements have been most influential on this research, and are henceforth referred to as the "core research requirements".

To summarize, this research attempts to provide an enabling technology for process-interoperability. The underlying theme in the requirements is to provide both autonomy and interoperability, which are often conflicting goals. Obviously, some compromise between these two is necessary. The idea is to apply suitable modeling and enactment facilities that will minimize the impact that SubEnvs can have on one another besides what they have explicitly agreed upon.

1.6 Thesis Organization

The main body of the thesis addresses modeling and enactment of multiple interoperating processes at three levels of abstraction:

1. A conceptual framework, given by a formal and generic (i.e., system- and language- independent) model (Chapter 3).

2. A comprehensive realization of the model in Oz[6], a specific PCE with a rule-based PML (Chapter 4).

3. An architectural framework and infrastructure that supports the decentralized model (Chapter 5).

1.6.1 Chapter 2: Previous and Related Work

Section 2.1 surveys state-of-the-art PCEs, particularly those that address distribution, interoperability, and heterogeneity. Section 2.2 gives a detailed description of Marvel, the predecessor to Oz, with emphasis on the system characteristics and principles that were carried over to Oz. Section 2.3 presents work in related fields, namely databases, heterogenous processing, and CSCW.

1.6.2 Chapter 3: The Formal Decentralized Model

This chapter presents the formal decentralized model, independent of any specific PML or PCE. It is the cornerstone of this research. Section 3.1 defines basic terms and concepts which are used throughout the thesis; Sections 3.2 and 3.3 present the **Treaty** and the **Summit** models for defining (modeling) and executing (enacting) inter-process collaboration, respectively. Section 3.4 revisits the motivating example in the context of the decentralized model. Section 3.5 applies the model to three families of PMLs which represent the paradigms of choice in many existing PCEs, namely, rules, Petri-nets, and grammars, and briefly discusses its applicability to APPL/A. Section 3.6 discusses extensions of the model to address integration of groupware technology. And Section 3.7 concludes the chapter with other potential extensions and alternatives to the model.

1.6.3 Chapter 4: Realization in Oz

The generic model, as a high-level abstraction, leaves many aspects undefined and unresolved, both technical and conceptual. Chapter 4 addresses these issues by describing the realization of the model in the Oz DEPCE, with its rule-based PML. Section 4.1 starts with a conceptual and operational overview of the system. Section 4.2 is devoted to introducing the structure of Oz objectbases. Section 4.3 covers all aspects of *modeling*

---

[6] Why Oz ? as a continuation to the Marvel project, named after Professor Marvel from "The Wizard of Oz".

process interoperability in Oz, including the algorithms that implement the Treaty protocol for defining common sub-processes, the associated problems of preserving the consistency of both local- and inter-process definitions while allowing local evolutions, and other issues related to the definition of shared schema and the accessibility of data instances. Section 4.4 covers all aspects of multi-process *enactment*, focusing on the operational and transactional semantics of the Summit protocol in Oz. Section 4.5 discusses an effective (though preliminary) implementation of language and system mechanisms for integration of groupware technologies into the Oz framework. Section 4.6 summarizes the implementation status with respect to what has been described in this chapter.

1.6.4 Chapter 5: Oz Architecture

This chapter discusses the architectural support for the decentralized model. Section 5.1 begins with an architectural overview, and the underlying principles that guided the construction of the system. The focus is on multi-server infrastructure support rather then on a single server architecture (which is covered in-depth in [17]). Section 5.2 describes the communication infrastructure including the decentralized repository for maintaining (dynamically changing) communication information, the actual communication protocols, and decentralized naming schemes. Section 5.3 presents the mechanism for dynamic (re)configuration of sites participating in a global environment. The unique approach taken here is that configuration is modeled and enacted as a process, and as such, it may be tailored (to some degree) on a per environment basis. As a by product, this chapter also shows that process modeling and enactment can be used not only for software processes, but also, for example, for configuration processes. Section 5.4 explains the context switching mechanism that is required in order to avoid communication deadlocks and starvation during the concurrent execution of Summits. Section 5.5 presents the remote object cache in Oz that significantly enhances the performance of Summits. Section 5.6 discusses the architectural extensions which were made in order to support arbitrary geographical dispersion over the Internet. Finally, Section 5.7 summarizes the implementation status of the architectural aspects of Oz.

1.6.5 Chapter 6: The ISPW Example: Validation and Methodology Issues

This chapter validates the decentralized model and its implementation in Oz by discussing an Oz multi-site environment that was built to support an example "benchmark" process written at the International Software Process Workshop (ISPW). The discussion of the solution is focused on design issues (the actual process code is given in Appendix B). In addition, a methodology for modeling decentralized processes is given, based on examples from the ISPW solution environment.

1.6.6 Chapter 7: Summary, Evaluation, and Future Work

This chapter evaluates the thesis, summarizes its contributions and the conclusions of this research, and points to future directions.

2

Previous and related work

While decentralization, heterogeneity and interoperability have been active research topics in several communities — databases, engineering, and distributed systems in general — these issues have been mostly unexplored in the PCE community until recently, mainly because the field is relatively young and the state-of-the-art in PCE technology was too immature. In the last two years, however, there has been a surge of activity in this area, and several PCEs and ideas have been developed to address some of these issues. These issues have also been acknowledged recently as one of the main future research directions [84], and seem to be a natural evolution of PCE technology. Nevertheless, most of the work to date is still on interoperability and heterogeneity under a single process (modeling) and/or under centralized control with centralized shared database (enactment). Further, much of the current work is still "on-paper", i.e., at the theoretical stages of development.

This chapter is divided into three major parts. The first part (Section 2.1) surveys SDEs and PCEs that address some levels of heterogeneity and/or decentralization.

The second part (Section 2.2) gives a detailed account of Marvel, the predecessor to Oz, from which many concepts (and code) were inherited. This section is important for the understanding of Chapters 4 and 5, where the Oz system is presented. Finally, the third part (Section 2.3) surveys work that has been done in related fields, namely databases, heterogeneous processing, and CSCW.

2.1 PCEs

ISTAR [27], one of the earliest SDEs (or "Integrated Project Support Environments"), provided comprehensive support to the software development lifecycle, including both management and software engineering. The main idea in ISTAR is the *contractual* approach, in which a "contractor" (e.g., a group of programmers) provides services to a client (e.g., manager). The contract must have well defined deliverables and acceptance criteria, and might include additional constraints imposed by the client. A contractor can further delegate some of the tasks to a sub-contractor, creating a "contract hierarchy" in a top-down fashion. In addition, the ISTAR architecture permits for sub-contracts (and all of their sub-contracts, recursively) to operate autonomously in different sites, since the contract databases are distinct and can be operated independently. Although ISTAR was not a PCE (it had a somewhat hard-coded process), its architecture is an important (and somewhat neglected) step towards decentralization.

Shy, Taylor, and Osterweil were among the first to explicitly identify decentralization as a key environment technology [97]. Their theoretical work draws an analogy between software development and the business corporation, and they advocate a "federated decentralization" model for PCEs with global support for environment infrastructure capabilities and local management with means to mediate relations between local processes. Among the arguments made for this model (as opposed to "corporate autocracy" or "radical decentralization") are: (1) The level of global support is not rigid. (2) While the communication is established under guidelines determined by the global process, the actual communication is provided and maintained under the control of the local entities. (3) Extensibility, because integration of processes and services can be implemented gradually. This preliminary model, while advocating decentralization, still considers every sub-environment to be strongly affiliated with the corporation and necessarily abiding by some global rules. Thus, autonomy is necessarily restricted a priori.

Heimbigner argues in [43] that just like databases, "environments will move to looser, federated, architectures ... address inter-operability between partial-environments of varying degrees of openness". He also notes that part of the reason for not adopting this approach until recently was due to the inadequacy of existing software process technology. However, his focus is on support for multiple formalisms. His proposed ProcessWall [44] is an attempt to address heterogeneity at the language level. The main idea in the Process-

Wall is the separation of process *state* from the *programs* that construct the state; in theory, multiple process formalisms (e.g., procedural and rule-based) can co-exist and be used for writing fragments of a process. However, decentralization as a concept is not addressed, and in particular, the process state server is inherently centralized.

Peuschel and Wolf explain why current client-server architectures are inadequate to support distributed software processes [87]. They identify four alternatives for distribution: (1) hierarchical process organization; (2) distributed process data; (3) distributed process engines over local area network; (4) distributed process engines over wide area network. They further propose four architectural approaches to meet these requirements, concluding that distributed process engines with only partially distributed process data — including a common process database that serves as a communication platform among the process engines — is best. Once again, this proposal excludes the possibility of a "shared-nothing" architecture.

Kernel/2r [52], from the Eureka Software Factory project, supports a special case of process formalism interoperability. The system identifies and divides the process into three distinguished kinds of process fragments, each with a separate process engine (and PML). The *interworking* process engine, MELMAC [24], supports cooperation between teams or within a team. An instance of the *interaction* process engine, WHOW, supports a single user working with a variety of tools to create, manipulate and delete development materials. The *interoperation* support, through the MUSE software bus, behaves like a process engine in that it controls partially ordered sequences of tool invocations where human intervention is not required. Although Kernel/2r does not directly support collaboration among multiple independent processes, MELMAC can, in principle, interface to teams who use another PCE (or who are not concerned with process at all).

ProcessWEAVER, another spin-off from the Eureka Software Factory, is a commercial product of Cap Gemini Innovation, with a Petri-net based PML. Fernström describes "...in a process, which consists of a set of cooperating sub-processes, every sub-process can be characterized by the set of 'services' it provides and requires from the other sub-processes" [32]. This sounds remarkably similar to our approach. However, in the ProcessWEAVER system, "...processes are recursively structured into sub-processes of finer and finer granularity and detail." In other words, processes are defined top-down, and provide essentially for fine-grained decomposition of one global process, whereas in our approach, what is in effect the decentralized process of a global environment should be defined bottom-up from the (collaborating) processes of the constituent SubEnvs. Finally, autonomy concerns for local process and their artifacts, which is a fundamental requirement in our approach, is not considered.

SMART [34] is an attempt to provide a methodology and a supporting technology for the *process* (as opposed to product) lifecycle through multi-formalism support, whereby different phases in the lifecycle are supported by different formalisms and corresponding (sub)systems. Specifically, SMART views the lifecycle of a process as consisting of a development phase; followed by analysis and possibly a simulation phase; followed by an embedding phase, in which a process model is instantiated with actual tools and product data bound to it; followed by an execution and monitoring phase, which feeds back to the development phase. Modeling, analysis, and simulation are performed with the Articulator system [76], process execution is performed by HP's SynerVision, and Matisse [35] (also from HP) is used to maintain a knowledge-base containing the artifacts that represent the process models developed in the Articulator, and serves as an integration medium between Articulator and SynerVision. Thus, the emphasis is on multi-paradigm support for the process, and on bi-directional translation: from process models to process (executable) programs, and from the process execution state back to the process model level. From a heterogeneity standpoint, SMART can be categorized as having some degree of system heterogeneity, since it integrates three different systems, and formalism heterogeneity, although not for defining different aspects of the process (as in ProcessWall), but rather for supporting different phases of a predefined lifecycle. However, there is no support for multiple processes with distinct instantiated products.

TEMPO [11] is a PCE that is designed to support "programming-in-the-many", i.e., projects that involve a large number of people, and therefore its emphasis is on modeling and mechanisms for supporting collaboration, coordination, and synchronization between project participants. TEMPO provides three main abstractions that facilitate modeling multi-user aspects of the process: (1) hierarchical decomposition of processes to subprocesses in a top-down fashion, similar to ProcessWEAVER; (2) support for multiple private views of the process, through the *role* concept which allows to define private constraints and properties; and (3) active and programmable *connections* between role instances, which are defined and controlled by *rules* with temporal constraints in addition to pre- and post-conditions. TEMPO is data-centered, and is built on top of Adele [10], an active object management system with data-driven triggering, which enables to realize rule processing in

TEMPO. While TEMPO provides for definition of "personal" processes and supports coordination among them, it is still inherently centralized, in that it requires a single database as the coordination platform, and supports multiple views of essentially a single group process, defined in a top-down fashion.

2.2 Marvel: The Predecessor PCE

This section gives a relatively detailed overview of Marvel, for two reasons. First, to introduce concepts and terms which will be used throughout the thesis, since large portions of Marvel were (re)used in Oz. Second, to clearly distinguish the work that was done in this thesis from the work that was done earlier in the context of the Marvel project.

In a nutshell, Marvel [59, 17, 50] is a highly tailorable rule-based PCE that supports project-specific definitions for the *data model*, *process model*, *tool envelopes*, and *coordination model*. The runtime environment (i.e., the process engine) has a client-server architecture that supports multiple users and enacts a centralized process on a centralized project database. We now discuss each major aspect of the system separately.

2.2.1 Data Model

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

Marvel supports four types of attributes: state, file, composite, and reference links. The first two attributes contain the contents of objects whereas the last two attributes are used to denote relationships to other objects. *State* attributes are used mainly for process data (although they can be used to hold product data as well), and can be formed from a set of primitive types such as integer, string, enumerated, etc. *File* attributes can be either text or binary, and are used usually to maintain product data which is held in files. File attributes are implemented as a file-system path pointing to the file in a "hidden" file

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

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

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

Figure 2.1: Several Classes from C/Marvel

system. Thus, end-users access objects in the objectbase which abstract the file system. *Composite* attributes are used to denote an "is-part-of" relationship between objects to form the *composition hierarchy*. Finally, *reference link* attributes (or simply links) allow any arbitrary semantic relationship between two objects. Both composite and link attributes are typed, and both allow one to specify whether arbitrary number of objects can be linked (by the `set_of` construct) or whether only a single object is allowed to be linked. The general structure of an instantiated Marvel objectbase can be viewed as a forest of trees, each of which represents a composite object, with additional links between objects across and within the trees.

Figure 2.1 shows some representative classes from C/Marvel, the process used for developing Marvel itself. The `MINI_PROJECT` class inherits attributes from its superclasses `BUILT` (not shown here) and `PROTECTED_ENTITY` (part of access control support, see Section 2.2.6); it contains two state attributes, `config` and `options`, both of type `string`; a file attribute named `log` with a postfix `.log`; two composite attributes, one single and one set, and three link attributes. For example, the `afiles` attribute specifies a link to a set of objects of type `AFILE`.

2.2.2 Process Modeling

The process model (in addition to the data modeling, which is also usually considered part of the process but for the purposes of this discussion is treated separately) is specified in a rule-based *process modeling language*, called the Marvel Strategy Language (MSL). Each process step is encapsulated in a *rule*. A Marvel rule has a (not necessarily unique) name, typed formal parameters, and three optional constructs: *condition*, *activity*, and *effects*.

The condition consists of two parts: *bindings* which are used to select objects by querying the objectbase, and a *property-list* which is applied to the binding set and must evaluate to true (in which case the condition as a whole is said to be satisfied) prior to invocation of the activity. A rule binding specifies a quantified variable[1] (or a *derived parameter* in Marvel terminology, to distinguish it from regular rule parameters) to which objects are bound, a class restriction on the allowed bindings, and a query that determines the binding set. The query consists of a possibly complex clause with nested conjunctions and disjunctions of *predicates* of two kinds — structural and associative. *Structural* predicates navigate the objectbase to obtain ancestors or descendants of specified types, containers or members of aggregate attributes, and objects linked to or from other objects. *Associative* predicates query the objectbase to obtain those objects satisfying a relation (equality, inequality, less than, etc.) specified between attributes of two objects, or between an attribute and a literal. At the end of the binding phase, each variable is bound to a set (zero, one, or more) of objects. The property list is similar in its syntax to the query part of the binding, consisting of a complex logical clause of associative predicates, but it is applied over the actual and derived parameters and returns a boolean value.

An *activity* involves invocation of an external tool to operate on the product data encapsulated within the bound objects. Tools are encapsulated via an *envelope* mechanism written in a Shell Extended Language (SEL) [38]. If there is no activity, then by definition there can be only one effect. If there is an activity, then in general the invoked tool may have several possible results mapped one-to-one with the given effects. A non-empty activity specifies an envelope and its input and output arguments, which may be literals, status attributes and/or (sets of) file attributes. In addition to output arguments, each envelope

---

[1] The specification of variable quantification at the binding phase is merely due to a flaw in the design of MSL, since it is only used later in the property-list. If the quantifier is universal, then all objects in the binding set of that variable must satisfy the condition, and if it is existential only one object in the set must satisfy the condition for the whole condition to yield a true value.

returns a code that uniquely selects one of the specified effects.

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

A sample rule, taken from C/MARVEL, is shown in Figure 2.2. This `archive` rule accepts one parameter of class `MODULE`. It has six composite binding expressions (lines 1-17), a property-list expression (lines 19-25), an activity that takes three arguments, each of which can be possibly bound to a set of objects (line 27), and two effects (lines 28-33). More explanations about this rule will be given shortly.

Rules are interrelated by means of matchings between assertions in the effect of one rule and predicates in a condition of another rule, which are are compiled into a static *rule-network*. Thus, operations between steps in a process can be implicitly formed by matching predicates in the condition of one rule and an effect of another rule, and the enactment engine enforces and/or automates the sequencing. However, the process is not in any sense limited to a deterministic sequence of steps. (See [60] for discussion of the specification of alternatives, iteration and synchronization through the conditions and effects of rules.)

2.2.3 Process Enactment

Enactment is provided in Marvel by *chaining*. Forward and backward chaining over the rules enforces consistency in the objectbase and automates tool invocations. Enforcement and automation are the two main forms of enactment supported in Marvel.

Marvel's process enactment is user-driven, with reactive control. When a user enters a command with the arguments, the environment applies its overloading mechanism to select the rule with the same name and "closest" signature to the provided actual parameters considering multiple-inheritance [50]. Then it dynamically binds objects to the derived parameters, and evaluates the condition. (dynamic, or late, binding, is an essential feature of Marvel that allows it to separate rules from the underlying objects.) If the condition of the selected rule is not satisfied, backward chaining is attempted, recursively. If the condition is already satisfied or becomes satisfied during backward chaining, the activity is initiated. After the activity has completed, the appropriate effect is asserted. This

```
archive [?m:MODULE]:

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

  ### activity
   { ARCHIVER mass_update ?m.log ?a.file ?a.history }

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

Figure 2.2: Example Rule from C/MARVEL

triggers forward chaining to any rules whose conditions become satisfied by this assertion. The asserted effects of these rules may in turn satisfy the conditions of other rules, and so on. Eventually, no further conditions become satisfied and forward chaining terminates. Marvel then waits for the next user command. Because of the event-driven nature of the enactment model, the actual parameter-selection for rules invoked through chaining is done by an algorithm that "inverts" the logic of the bindings of the chained-to rules [50]. This is in contrast to the data-driven approach, in which the parameters are supplied directly by the database as a result of data updates. We will refer from now on to this algorithm as the inversion algorithm.

Predicates in effects of rules are each annotated as either *atomicity* or *automation*. By definition, all forward chaining from an atomicity predicate in an asserted effect to rules with satisfied conditions and empty activities is mandatory. In contrast, forward chaining from an automation predicate or into any rule with a non-empty activity is optional, and can be explicitly restricted through `no_forward`, `no_backward` or `no_chain` directives on individual automation predicates. It is important to understand that only automation *chaining* is optional; users are still obliged to follow some legal process step sequence implied by the conditions and effects of rules, whether through manual selection of commands or automation chaining.

An automation predicate is enclosed in parentheses "(...)", and may optionally be preceded by a chaining directive. An atomicity predicate is enclosed in square brackets "[...]". For example, the property list of the `archive` rule shown in Figure 2.2 (lines 19-25) consists solely of automation predicates. It first checks that the `MODULE` parameter (represented by the `?m` symbol) has not already been archived, and then permits backward chaining to attempt to compile the `MODULE` parameter and/or to archive any of the `CFILE` (`?c` symbol), `YFILE` (`?y`) or `LFILE` (`?x`) components, or nested `MODULE`s (`?child`). However, the "no_forward" directive in lines (21-25) prevents from automatically chaining *into* this rule from other rules whose assertions might otherwise satisfy these predicates. Thus, there is full control over the degree of automation in rule invocations.

The first effect of this `archive` rule (lines 28-31) has two automation predicates and one atomicity predicate. The atomicity predicate guarantees that whenever this `archive` rule is successfully executed and its first effect selected, then any other rules whose conditions are satisfied by setting the status of a related `AFILE` to `Archived` will also be executed. If for some reason one of these rules — or one of their own such implications — could not be completed, then the whole recursive *atomicity chain* would be rolled back as if none of its rules had ever been fired. In contrast, no such atomicity requirements are imposed by the assertion of automation predicates. We will return to discuss automation and atomicity in more detail later in Section 4.4.5.

2.2.4 Synchronization and Coordination Modeling

There has been extensive work on support for advanced Concurrency Control (CC) in Marvel. In fact, Marvel's architecture is heavily influenced by, and geared towards, supporting flexibility in the selection and application of CC policies (see [17]). For example, the separation between data and transaction management, as well as the separation between conflict detection (lock management) and conflict resolution within transaction management, enhances the flexibility in tailoring concurrency control policies. An additional innovation in Marvel's support for concurrency is that it allows both the dimensions and the contents of the lock compatibility matrix to be modified. Consequently, Marvel can determine, on a per-project basis, which locks (from the matrix) should be applied on certain operations. Thus, concurrency control can be configured to support a wide range of policies and lock modes. Finally, Marvel provides a programmable interface to model coordination among team members, by means of a Coordination Rule Language (CRL) (due to Barghouti [6]) that defines how to resolve lock conflicts in accessing data. This allows specific semantics-based CC policies to be implemented. However, as mentioned in the introduction, the coordination is among users that operate within the same process. Moreover, this can be viewed as an *a posteriori* coordination, i.e., coordination rules are called only *after* a conflict has arisen, which limits the modeling capabilities. These aspects are currently addressed by Heineman [46].

2.2.5 Process and Schema Evolution

Process evolution in PCEs is analogous to schema evolution in a database management system. An initial process model is developed based on a requirements analysis for the project, but changes are often needed later on. By this point, however, the process may be already instantiated with process state reflecting the progress through the installed process. To replace the process model, it is necessary to modify this state so that it is possible to continue work using the new process from the point at which work using the old process left off, while ensuring that the process state is semantically as well as syntactically appropriate with respect to the new process. In particular, it is usually undesirable to “start over” with a pristine state and, in general, incorrect to continue work using the previous process state “as is”. Further, it is tedious and error-prone to modify the process state manually.

In PCEs that support data modeling, *schema evolution* is a necessary adjunct to process evolution, because changes in the process model often mandate changes in the schema specifying the types and composition of the process state and possibly also product data representation. The need for a schema evolution mechanism is clear: when a structure of a data element (e.g., a relation in a relational database, or a class in an object-oriented database) is modified, the pre-existing data that was defined according to the previous definition of the structure must be upgraded to conform with the new definition in order to be accessed properly. Some structure modifications, like adding primitive fields to a class, can be handled relatively easily by adding those fields to all instances with some predefined default values. Other structure changes, like renaming fields (but wanting to keep the old values), changing the types or the allowable ranges of fields, and deleting fields, are harder to implement. And finally, changes that update the class hierarchy (adding or removing superclasses from a class definition) or the composition hierarchy (e.g., removing a child attribute which might imply disconnecting the hierarchy) are hardest to implement.

In most cases, however, the differences between the old and new schemas can be syntactically analyzed to enable subsequent automatic update of the database. This is the gist of the schema evolution mechanism in Marvel, implemented as part of the Evolver utility. It consists of two components: a front-end, based on an algorithm adapted from the Orion object-oriented database management system [5], that compares the old and new schemas and either produces a “delta” (which is also displayed to the administrator to allow him/her retraction if evolution is unacceptable) or rejects the evolution if it contains changes that are not supported by the Evolver; and a back-end that actually updates the objectbase according to the new schema. For more details on Marvel’s schema evolution, see [68].

In contrast to schema evolution, process evolution is much more complicated. First, it is not clear how to technically analyze the syntactic “delta” between the two process models and represent it in a form that can be used for evolution. Second, it is far from clear how to analyze and identify the semantic differences between the old and the new models in order to properly update the process state. Third, even if a semantic “delta” is feasible, it is not clear whether to apply the changes on none, some, or all of the relevant process states.

The general approach in Marvel is based on the notion of process consistency, and on ways to identify inconsistencies that might be introduced as a result of changes that were made to the process model. We summarize here our general approach and solution to this problem. A detailed discussion of this topic can be found in [58, 68].

2.2.5.1 Process Consistency and Enforcement

Process consistency refers to *constraints* that are defined in the process (either implicitly or explicitly), and are assumed to always hold for any relevant process state in any instantiation of the process. The process state is deemed *consistent* if all constraints have indeed been enforced on all past process steps, and *inconsistent* otherwise. Thus, an underlying premise here is that under normal circumstances, process constraints are *enforced* by the process engine. Adding process steps (or tasks) to an existing process might introduce new constraints, some of which could potentially make the process inconsistent with respect to the existing process state. For example, suppose that we want to add to a development process a new static code-inspection step. The associated new constraint in the process is that source code can be checked-in to a stable "master" repository only after it passed successfully code-inspection. Then, the source code which is already in the master repository violates the new constraint and thus introduces some inconsistency. Thus, a systematic method to identify inconsistencies and generate a "process delta" can be used as a basis for process evolution. However, such analysis does not imply necessarily how to repair it. In the above example, for instance, it may not be reasonable to require manual code-inspection of all source code in the master area that was there before the new constraint was introduced.

The gist of our approach is to generate a list consisting of every process step affected by the new constraints, and give the user the opportunity to enact each such process step on none, some or all data items of the relevant type. The goal is to apply the new constraints retroactively to the existing process state, in order to make the state consistent with respect to the new constraints, but not necessarily by following all the steps that would be required under normal process execution.

2.2.6 Access Control

Access control is a mechanism that allows to specify by whom, and in what manner, artifacts can be accessed, independent of a particular application that accesses them or a particular time at which they are accessed. In other words, it is a persistent property of the artifacts. An example of such a mechanism is the Unix permissions on the file system.

Marvel employs a similar access control mechanism at the object level. The original idea was to build a flexible mechanism that could be tailored on a per-project basis to meet the demands of a particular environment, much like other aspects of process modeling. However, since such a mechanism necessarily involves low-level operations on the objectbase — including interaction with the hidden file system and dynamic checks on the objectbase each time an object is accessed — this approach would require to expose those operations to the modeling language, involving extensive modifications to both MSL and the process engine. Alternatively, separate notations and interpreters could be built for access control specifications.

As neither of these approaches was feasible and since access control was treated in Marvel less as a research topic and more as a bare necessity, the approach that was taken in Marvel was a compromise. The *definition* of access-control is done using the notations used to define normal data attributes and classes, but the *manipulation* of the permissions data is done through a set of built-in operations, and the actual checking of permissions is also hardwired. Representing the permissions data by normal class/attribute definitions enables one to potentially access and manipulate the permissions through MSL rules, analogous to the way structural operations in Marvel have both built-in and rule-based interfaces. However, this approach also introduces the problem of potential security violations through the process, which would have to be addressed before allowing such rule-based access.

In order to realize protections, two important concepts were added to Marvel: the notion of a *user* object, to which protection information can be attached, and the notion of a *permission group*, similar to group permissions in Unix. The natural way to represent users and groups was to use Marvel's data definition language and objectbase to define and store user objects, respectively. This also avoided the need to hard-code any notion of users or groups within the kernel. The protection model thus defines built-in classes to represent users and user-groups (called `USER` and `USER_GROUP`, respectively) that contain the necessary permission information (e.g., a `mask_string` attribute that defines the default permissions on an object created by a user) and are structured in a "user tree". The representation of users and groups in the objectbase goes beyond access control, as it can contain any information that pertains to that user (e.g., personal data). Indeed, several instantiations of Marvel environments defined specialized sub-classes of the `USER` class with additional information, for example to represent roles.

In addition to defining users and groups, the protection model must associate permissions with each individual object in the objectbase. Once again, to avoid hard-coding of permissions, it was implemented by adding a generic class called `PROTECTED_ENTITY`, such that only classes that are defined as subclasses of `PROTECTED_ENTITY` are protected. Therefore, the protection mechanism is entirely optional. In particular, if there is no need for such a mechanism (for example, in a single-user instantiated process) then the overhead associated with protections is totally eliminated. Another benefit of this approach is that by using the Evolver, protections can be easily added to, or removed from, an environment instance by simply adding `PROTECTED_ENTITY` as a superclass to all classes and evolving the objectbase (the addition of the user tree should not require evolution if no such tree pre-existed). The MSL data definitions for protection are given in Figure 2.3.

The runtime behavior of the protection model is as follows: when a user logs in to a Marvel environment, the server associates the user with his/her appropriate `USER` object, if there is one. The matching between a user and his user object is done via the Unix user-id which is stored in each user object (there is no such association with Unix groups, though). If there is no `USER` object for that user, he/she is associated with the `anonymous` user object with default guest permissions. If a user logs in as an administrator, he/she is associated with the special `administrator` object rather then the user object. When a user accesses any object either directly from the client, or indirectly through chaining, the server enforces the protections by matching the mask of the user's `USER` object with the accessed object's permissions attribute. The actual checking takes place in the transaction manager to ensure that any access to an object is checked. The check occurs before any other operation, and if the objects' permission denies the requested access to the object, the associated transaction aborts. For a detailed discussion of the access control mechanism in Marvel, see [68].

2.2.7 Marvel 3.1 Architecture

```
strategy protection

imports none;
exports all;

objectbase

PROTECTED_ENTITY:: superclass ENTITY;
owner: user;
perm_string: string = "rwad rwa*";
end

Login_Register:: superclass PROTECTED_ENTITY;
group_list: GROUP_LIST;
user_list: USER_LIST;
end

GROUP_LIST:: superclass PROTECTED_ENTITY;
glist: set_of USER_GROUP;
end

USER_LIST:: superclass PROTECTED_ENTITY;
ulist: set_of USER;
end

USER_GROUP:: superclass PROTECTED_ENTITY;
grp_name: string;
end


USER:: superclass PROTECTED_ENTITY;
groups: set_of link USER_GROUP;
mask_string: string = "rwad r*a*";
end

end_objectbase
```

Figure 2.3: Class definitions for protections

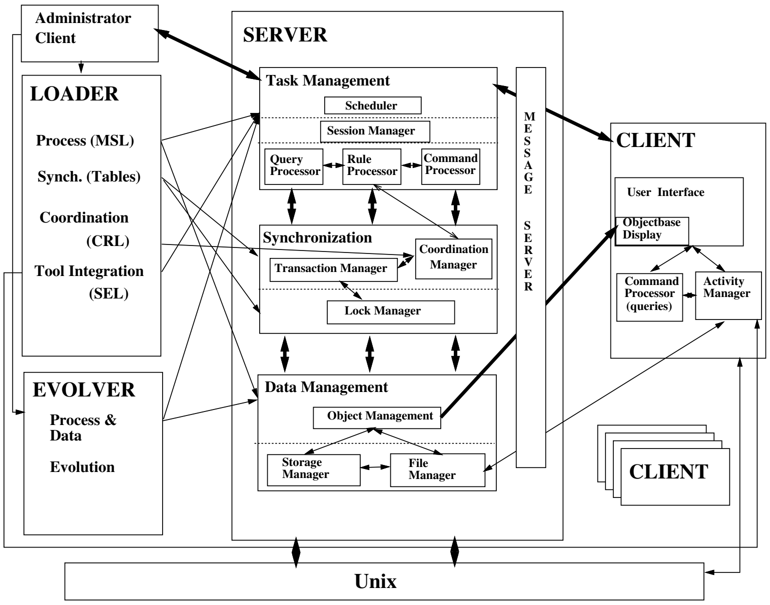


Figure 2.4: Marvel 3.1 Architecture

Marvel's architecture is illustrated in Figure 2.4. The architecture follows the client-server model, where the server is centralized and manages the data, process, and synchronization, and clients manage the user interface (including objectbase browsing) and activity invocation by forking operating system processes to execute external tools using the envelope wrapping mechanism. Each client can support multiple threads of control and clients can be distributed across machines, but the server and all of its clients must reside in the same Internet domain and share the file system. Every user command (besides a small number of commands that are handled solely at the client) is transferred to the server, which validates the request and possibly backward-chains to other rules, manages the access to objects (including concurrency control), sends the activity to be carried out to the client, and switches context to service new requests. (The actual scheduling algorithm is a simple FIFO queue.) Upon completion of an activity, the server attempts to forward chain to other rules, which in turn may lead to more interactions with the client to execute more activities, and so forth.

Various built-in (process-independent) operations are available in every instantiated environment, with proper user interface. Some of the important services include: a set of commands for structural objectbase manipulation (implemented as built-in rules that embed the corresponding built-in effects, e.g., `add object`), which could be tailored on a per-process basis but the default basic operations are always available; an ad-hoc query processor; a browser; and several process-inspection and animation utilities, such as the rule-network navigator and the display of class and composition hierarchies of a given schema.

Finally, additional components of the system include: the Loader, available to administrators, which translates the process specifications and loads them into the server; the Evolver, discussed earlier; and `marveld`, a daemon responsible for activating a server on a given environment upon client request when the environment is inactive (the server normally shuts down when no clients are connected to it). The rationale behind the Marvel architecture can be found in [17].

This architecture is adequate for a small to medium number of people interacting with the server through clients in the same local-area network (we have experienced using Marvel successfully with up to 20 concurrent clients and 8 users). But as the number of (simultaneous) clients grow, the server becomes a bottleneck. More importantly, the architecture is inherently single-process, and dictates that all users must work essentially within the same process, or at best allow minor deviations but still from the same process[2]. And finally, Marvel requires all entities to reside in the same domain (although clients can reside in different hosts within the same domain). Thus, Marvel lacks the necessary architectural support for scale and heterogeneity.

2.3 Other Domains

Autonomous decentralization, heterogeneity, and interoperability, are very active research areas in several related fields, including the database community (more specifically Heterogeneous Distributed Database Systems (HDDB)), Computer Support for Collaborative Work (CSCW), and Heterogeneous Processing (HP). We briefly summarize each field, give some examples, and differentiate them from the research on PCE decentralization.

2.3.1 Heterogeneous Distributed Data Bases (HDDBs)

The relevance of work in HDDB to DEPCEs is analogous to the relevance of (centralized and distributed) database research to PCEs. Just as PCEs generally impose specific requirements on the representation, storage, and (concurrency) control of the persistent

---

[2] The actual implementation does not support deviations at all. See [8] for a design of such a mechanism for Marvel.

artifacts being developed [18] (also known as software engineering databases) — DEPCEs might similarly require special-purpose modeling and mechanisms to support various degrees of heterogeneity and site autonomy with respect to the persistent artifacts which are manipulated by the environment. The database community has defined this field as federated, or heterogeneous databases, which permit a high degree of site autonomy [90]. The heterogeneity is usually with respect to one or both of two criteria, system and schema (the third "standard" criteria, namely language heterogeneity, is embedded in the system criteria since the data definition and manipulation languages are usually strongly associated with the underlying system): the sites may employ the identical system but devise their own schema independently (also known as a homogeneous federation); and/or they may select different database systems from among those supported by the federation "glue" (heterogeneous federation).

UniSQL\M is an HDDB that assumes a common relational data model to which all component database systems convert their schemas. In [66], Kim describes a complete framework for classifying schematic and data heterogeneity as a basis for a later "homogenization" of the databases. The general approach to addressing heterogeneity is by providing an underlying common formalism into which the various formalisms translate. The main difficulty with this approach is that the common data model and the formalism (in this case, relational and SQL, respectively) must be expressive enough to support a wide variety of data models and languages — which in many cases might not be feasible.

Pegasus [29], from HP laboratories, is an HDDB that uses object-oriented technology to extend the schematic integration approach in an attempt to alleviate the difficulties with arbitrary mappings of data models and to increase local autonomy. First, local schemas need not be mapped completely, only imported (sub)schemas are integrated. This allows one to hide parts of the local schema (and the instantiated data) that are either hard to map or for privacy concerns. Second, the integrated schema is not necessarily global. Instead, Pegasus builds a hierarchy of integrated schemas that apply to a subset of the local databases and enables a more refined integration. The approach for integration is based on the notion of "upward-inheritance", where types can be superimposed to generalize on local types in different schemas (and different databases). Finally, in addition to object-oriented data abstractions, Pegasus also exploits object-oriented function abstractions to enable non-trivial integration of schemas by attaching optional functions to support the mapping.

2.3.2 Heterogeneous Processing

Heterogeneous Processing is an emerging field concerned with architectures for distributed systems that support heterogeneity and interoperability of autonomous entities. The main focus in this field is on (1) investigating *system* level heterogeneity, and (2) providing solutions to the general heterogeneity problems by exploring integration at the system level. The main relevance of this field to PCE research and technology is in the area of architectures for decentralized and heterogeneous PCEs. The analysis and identification of requirements for DEPCE-specific architectures is another research issue that involves both communities.

One of the prominent specifications for heterogeneous processing systems is OMG's Common Object Request Broker Architecture (CORBA) [80]. The key idea in CORBA is to insert a programmable intermediary level between *clients* that request to invoke operations on some (active) objects (which may be viewed as servers) and the object implementations, thereby providing an infrastructure that enables to "glue" heterogeneous components and mix and match between them. The heart of the intermediary mechanism is the Object Request Broker (ORB), which interconnects objects and clients (location, message transfer, etc.). Object implementations specify their interface in an Interface Definition Language (IDL) that is independent of the programming language in which the object is implemented, and is understood by the rest of the system.

InterBase [20] is a system that addresses the (mostly data) heterogeneity problem by using integration at the system level. This is in contrast to most HDDBs that provide integration at the schema level. Moreover, it can be viewed as a *control*-oriented approach as opposed to data-oriented. The main idea in InterBase is that each subsystem supplies a programmable Remote System Interface (RSI) that serves as an intermediary between the local sites and the "federation", and global transactions are supported through a distributed transaction manager that interacts with these RSIs and coordinates the concurrent execution of global transactions, thereby serving also as a coordination platform.

To summarize, it can be seen that the field of heterogeneous processing, is also seeking to use data and control abstractions to cope with system heterogeneity and interoperability, by trying to hide everything that is not pertinent to interoperability, minimize the "exposure" to global control, and determine the desired exposure at each site autonomously.

2.3.3 Computer Support for Collaborative Work (CSCW)

CSCW focuses on support for human-human interactions, including social aspects of collaboration and tools to enhance collaboration and coordination, possibly among physically remote group members. As such it also borders with the user interfaces (or more generally Human Computer Interaction (HCI)), Multimedia and the Virtual Reality communities. CSCW is related to PCE technology because software development is inherently a collaborative task. The interesting challenge from the PCE perspective is in embracing CSCW technologies into the process framework. Some work in that direction is given in Sections 3.6, 4.5, Chapter 6, and [14].

Suite [25] is a system that provides an infrastructure for building multi-user and multimedia tools. Suite provides mechanisms to support flexible and fine-grained concurrency control (needed to enable simultaneous sharing of data in multi-user applications), synchronous and asynchronous collaboration facilities, caching of user-interface state at local workstations (to reduce communication overhead), coupling of user-interface states of different collaborators, and audio and teleconferencing annotations for multi-media collaboration. FLexible Environment for Collaborative Software Engineering (Flecse) [26] is a set of collaborative software engineering tools built on top of Suite, intended to support product development of a group of geographically-dispersed engineers, with focus on multi-user tools such as editing, debugging, and versioning. The technological idea is to provide the local user an interface front-end and a cached state of the tool (provided by Suite), while manipulating the actual tool's data and state in a central location. Flecse can be considered essentially as a toolset, without integration mechanisms or policies regarding their invocation. This, however may be beneficial when considering the integration of the tools within a PCE.

An example of a full blown CSCW system is Conversation Builder (CB) [30]. The main concept in CB is that of a *conversation*, which is a context in which a user performs its actions ("utterances"), and can potentially affect other users participating in the same conversation through a shared conversation space yet still protect their private conversation space. An interesting capability of CB that most CSCW systems lack, is the ability to specify activities and their interrelations using *protocols*, which are state-machine descriptions of the flow of the conversations. Thus, a limited form of tailorability and collaboration modeling is provided. The architecture of CB is centralized, with a shared conversation engine (analogous to process engine in PCEs) with which all clients communicate. It is based on a multi-user FIELD-like message bus that serves as the control-integration mechanism. CB has been used in several application domains, including configuration management and code inspection, and has some overlap with PCE concepts. However, the emphasis in CB is on user-user collaborations, not the software process in general, and it does not have any form of enactment.

Finally, Media Spaces [19] is a multi-media project at Xerox PARC that supports a virtual environment in which people that are physically dispersed can feel and operate as if they were co-located. The specific approach to reaching virtual reality is based on providing non-activity-specific oriented environment (such as "chance-encounters"), in addition to the standard activity-oriented support (e.g., video conferencing). The (remote) relevance of this work to our research is mainly in motivating the PCE community to explore ways in which such technologies could be useful for software development, and in integrating them into PCEs, with proper modeling and enactment support.

2.3.4 Summary

What distinguishes research in DECPEs from the above domains, is the fact that heterogeneity, autonomy and interoperability have to be addressed not only in the context of architectural support (Heterogeneous Processing), data integration (HDDBs), and tools for human collaboration (CSCW), but in the context of the modeled *process* that oversees, integrates and assists in the invocation of multi-user tools, on heterogeneous data, and on behalf of, and with the participation of, multiple collaborating human users. Furthermore, the challenges in this research are to find suitable notations, mechanisms, and infrastructures that support all the above in a flexible and project-specific manner.

3

The Formal Decentralized Model

Chapter 1 provided the motivation for *why* decentralized environments should be investigated, and imposed some requirements on how to build them. In this chapter we will show *how* this can be done, by introducing a general formal model for process interoperability. The model is both language and system independent and is in principle applicable to a wide range of Process Modeling Languages (PMLs) and Process centered Environments (PCEs) (as will be seen in Section 3.5), although in a particular language/system only a subset of the model's capabilities might be realized. Chapters 4 and 5 will show a particular realization of the model in Oz.

The high-level approach taken in this thesis to meet the challenges described in the introduction, is to supply an abstraction mechanism whereby multiple possibly pre-existing processes can be encapsulated and retain security of their internal software artifacts, tools, and steps, while agreeing with other processes on formal interfaces through which all their interactions are conducted on shared data. Thus, another perspective on process modeling is that process models, being encoded in formal notation, can be used as a sound basis for formally modeling *interoperability* among processes. Furthermore, multi-process enactment engines can support the execution of collaborative inter-process activities in the same way that single-server enactment engines support the execution of single processes. This thesis, then, starts out from the premise that process modeling is in general a viable technology, and asserts that inter-process modeling and enactment, as invented in this thesis, is a viable technology as well.

Some intuition to the decentralized model may be gained by the "international alliance" metaphor which will be used occasionally in the thesis. In such an alliance, the default is for independent countries (processes) to operate autonomously and collaborate (interoperate) only in accordance with predefined *treaties*. The actual collaboration is modeled as a *summit* meeting, where preparations to the summit and consequences of the summit are performed independently (according to private sub-processes) and the summit itself is performed cooperatively (according to a shared sub-process).

We begin with definitions of terms and a formalization of concepts that are used in the rest of the model, followed by discussion of the formal model for definition and execution of decentralized processes, followed by application of the model to various types of PMLs, discussion of groupware support, and conclude with possible extensions to the model.

3.1 Definitions

The purpose of this section is to establish a common terminology to be used consistently throughout the thesis. It is particularly important to clarify the meaning of the heavily overloaded term "environment", and to distinguish it from other concepts.

3.1.1 PCEs, Process Models, and Environments

As mentioned in the introduction, a Process Centered Environment (PCE) is a system in which processes are modeled and enacted, and a process model is an actual definition of a project-specific process, as specified in the Process Modeling Language (PML) supplied by the PCE. The PCE may supply "base" process models which are then tailored to project-specific needs, or processes can be written from scratch to support a specific project, but this distinction is irrelevant to the ensuing discussion. While a process model tailors a generic process (if any), it is a static entity which does not represent execution, only definition.

A process model can be *instantiated*, by loading it and binding to it real artifacts, tools, users, and any other system bindings which are required by the PCE in order to initialize it for execution[1]. An instantiated environment (or environment instance) is an enactable process model. It can be viewed as the loaded "core image" of a process model. However, it usually maintains persistent data and state that lasts across (operating system

---

[1] A process can also be initialized for simulation, in which case the bindings are to virtual or simulated artifacts, users, tools, and so forth.

process) executions of the process model. For brevity, we shall call an instantiated environment simply an **environment**. This term should not be confused with the term PCE, which refers to the system on which (instantiated) environments run.

Thus, a process' lifetime begins when it is initially defined as a process model. Afterwards it is loaded and instantiated for execution (perhaps with intermediate testing and analysis steps), at which point it turns into an instantiated environment. Note that the same process model can be instantiated in multiple environment instances. At some point during its execution, the process model might need to be refined, e.g., because of feedback from the environment or new requirements, in which case it is modified and then reloaded, although this time the persistent process state and product database have to be evolved to conform to the new process model.

3.1.2 A Generic Process Context Hierarchy

The following is a generic definition of a three-level hierarchy of nested contexts within a single process. A particular PML might have more or fewer levels, but we assume that there is some mapping into these core levels:

1. *Activity* — This level is where the PCE interfaces to actual tools, including input/output data transfer with respect to the tools. This is sometimes done through wrappers, or envelopes.

2. *Process-step* — This level encapsulates an activity with local prerequisites and immediate consequences (if any) of the tool invocation, as imposed by the process. For example, in the FUNSOFT Petri-net based PML [39], a process step corresponds to a transition along with its (optionally) attached predicates; in the Articulator task graphs [77], this level corresponds to a node with its predecessor and successor edges; and in rule-based PMLs, a process step is represented by a rule with pre- and post-conditions. The process-step level may also supply the mechanism to interface among multiple activities in a process. For instance, in rule-based PMLs, a post-condition of one rule is matched against a pre-condition of another rule to determine possible chaining; similarly, the firing of a Petri-net transition can enable another transition.

3. *Task* — This level is defined as a set of logically related process steps that represent a coherent process fragment. Depending on the specific PML and PCE: (1) there are typically some ordering constraints, or *workflow*, among the activities or process steps of a task; (2) parts of a task might possibly be inferred dynamically, emanating from an entry activity or process step selected by the user; and (3) a task might be partially carried out automatically by the PCE on behalf of the user, usually by triggering the inferred activities or steps. The task level may be explicitly defined in the PML through a special notation, or may be implicitly defined through the local prerequisites/consequences in the process-step level, or both. For example, the Activity Structures Language [60] specifies "local constraints" using rules (the form of process steps), and "global control flow" using constrained expressions (explicit tasks). In a Petri-net PML, the task level typically corresponds to a subnet, if such a construct exists. Tasks may be further decomposed into subtasks.

3.1.3 A Multi-User, Single-Process Environment

The following is a formal definition of a single-process environment. It is a *minimal* definition, in that it specifies only the ingredients which are necessary for our model. Thus, our goal here is to include as many PCEs as possible, but at the same time identify (families of) PCEs which cannot possibly fit in the model.

An (instantiated) environment E is defined as a quintuple:

$$E = < S, D, T, U, P >$$

Where:

- S — A *schema* representing data types for modeling the product and process data manipulated by the environment. Note that this requirement excludes environments with no data modeling support (e.g., Synervision [51]). Furthermore, the schema must support the notion of an "object" as explained below. However, there are no further requirements on the expressiveness of the data definition language, and in particular, it can either be a sublanguage of the PML, or a separate language.

- D — A *database* storing a set of objects, each belonging to a certain type (or class) from S. This component requires persistent storage for at least the process data, and possibly for the corresponding product data (the latter could alternatively be maintained in a separate database or in the native file system, but should be identifiable from the process data). As noted above, the database should be object-based, in the sense that data elements are typed (or classified), they have unique identity, and they can be referenced and manipulated by the process. Requiring an object-based project database does not seem to be a severe restriction, though, as most existing PCEs tend to use object-based databases (Adele-2 [10], Merlin [109], SPADE [3], Arcadia [57], Matisse [35], to name a few).

- T — A set of *tools* being used in the environment. The tools can be off-the-shelf, or customized to work in the PCE, but in either case it is assumed that the PCE has means to invoke those tools from within the environment through process activities.

- U — A set of users using the environment. No built-in roles or hierarchies are assumed to be attached to users, except for the concept of environment *administrator*, who defines and can modify each of the elements in E (analogous to the role of a database administrator).

  Note that this component implies an important requirement on the underlying architecture: it must support multiple users sharing the instantiated process, possibly simultaneously.

- P — A set of activities/steps/tasks and their inter-relationships, which together comprise the *process* model. They can be invoked either manually by human end-users, or automatically by the process engine. Each activity encapsulates a tool from T, with formal parameters from S, and actual parameters from D. An activity is not required to be bound to specific users or roles from U, although such a requirement can be imposed by a specific implementation or a specific process definition.

As can be seen, the definition above imposes some architectural requirements on the PCE. These will be fully discussed in Chapter 5. For the time being, it is sufficient to note that a PCE supporting this definition of an environment must include:

- Data modeling, repository, and management (including concurrency)
- Communication services
- Tool integration mechanisms
- Task/Process management
- User interface
- Translators and/or interpreters for the process and data models

Hence, with the exception of the last item, these are the same components as in the "toaster" reference model for general SDEs [28]. The key difference is that by using the translator and loader, some or all of the above components can be tailored, and are not hard wired into the system. Indeed, one of the architectural challenges in building a DEPCE is to preserve the modifiability property, which, as evidenced in [63], is the most important property in determining the quality of the architecture.

Based on the above definitions and requirements, a high-level view of an architecture of a single-process PCE with an instantiated environment is depicted in Figure 3.1. It consists of a database server managing the process schema and data, a tool server integrating the project's tools, a process server enacting the defined process, a client-user interface, and a communication layer connecting all components. A typical interaction with the PCE is as follows: an end-user from U initiates a task from P by invoking an activity that encapsulates tool(s) from T, on a set of data arguments from D that belong to classes from S. The process server receives the request, and depending on the specific installed process and other ongoing activities, determines what to do before, during and after the requested activity, involving the data and tool servers which can also interact directly with the client.

3.1.3.1 A Sample Single-Process Environment

The following is a specification of a sample environment, E_1, that supports a *code change* sub-process (taken from an actual Marvel process model used to develop Oz). For simplicity, only that sub-process is presented here. E_1 is defined as:

$$E_1 = < S_1, D_1, T_1, U_1, P_1 >$$

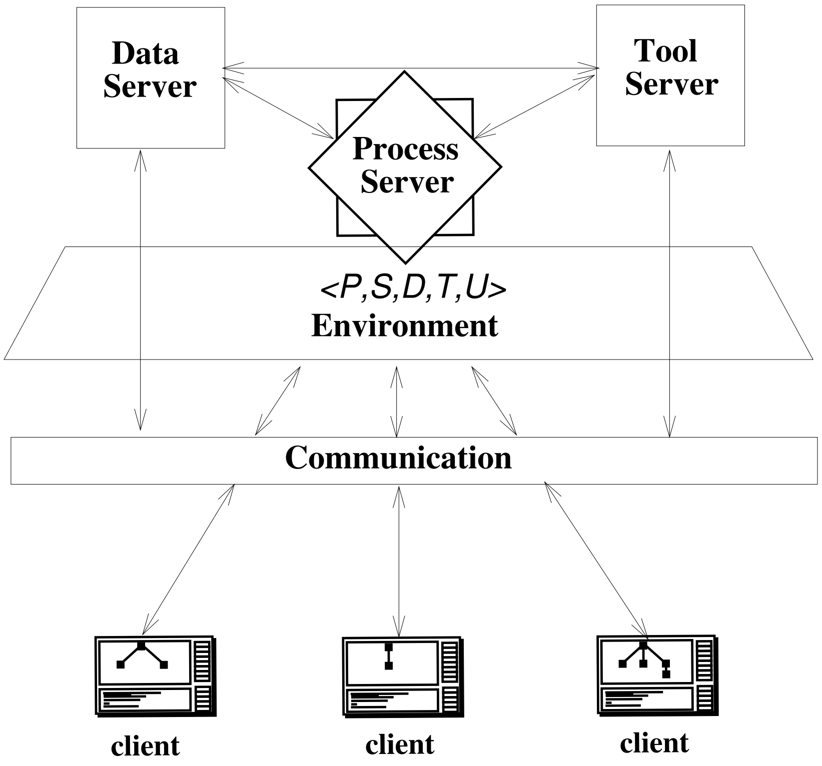


Figure 3.1: A Generic Single-Process Environment

where:

S_1 ={ FILE, LIB, WORKSPACE, EXEFILE}

D_1 ={ IssyWorkSpace, IssyExe, ServerLib, ClientLib, server.c, client.c, server.h, client.h, server.h.local }

T_1 = { rcs, emacs, cc, lint, gdb, inspect_tool, tags, ar, find_dep, list_dep, latex }

U_1 ={ israel, heineman, pds, popovich, tong }

P_1 is represented by the graph in Figure 3.2.

A change sub-process consists of issuing a `Reserve` activity to check-out a source file from some master-area (maintained by another sub-process not mentioned here) to a local workspace; followed by the `Outdate` activity that out-dates the local workspace, thereby invalidating any local binaries that were constructed in prior changes; followed by an `Edit-LocalRef-Analyze-Compile-Build-Debug-Edit` cycle[2], followed by the `unit-test` task (marked in black-box in the figure to denote an entire sub-task which is not expanded in

---

[2] `LocalRef` is a utility that checks that relevant master-area artifacts have not been updated since the reservation took place.

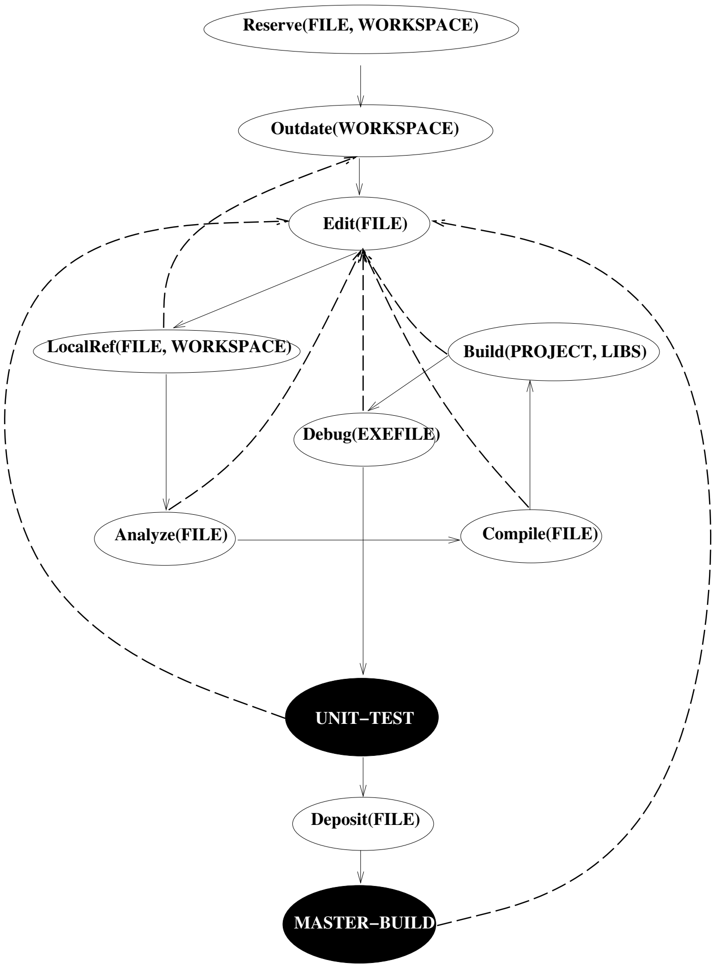


Figure 3.2: Change Sub-Process

this example), a `deposit` activity, and a `master-build` task, also not given here. Forward edges represent the expected "successful" flow of the process, and backward (dashed) edges represent the flow of control when activities "fail" (e.g., unsuccessful compilation).

3.1.4 A Multi-Process Environment

A multi-process decentralized environment is formally defined as

$$\{E_i\}\ i = 1 \ldots n$$

where each E_i is a single-process environment as defined in the previous section. In addition, a multi-process environment has some (modeling and enactment) facilities for

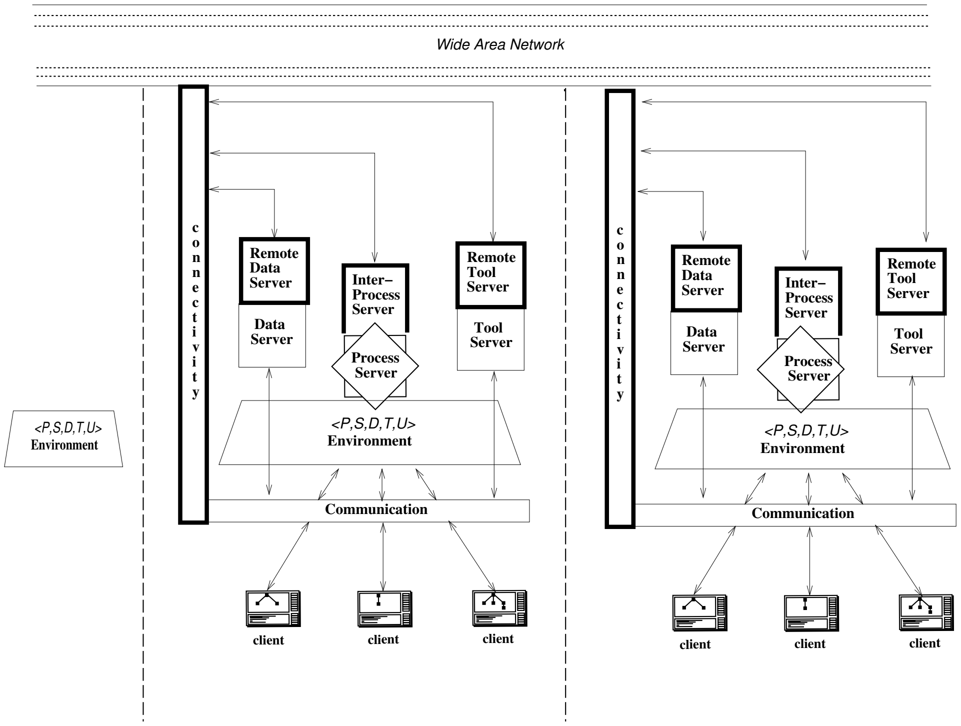


Figure 3.3: A Decentralized Environment

environment and process inter-connection and inter-operability. This is the main subject of this chapter.

Site autonomy and operational independence impose a strong architectural requirement: it must be a "share-nothing" architecture. This means not only that processes are private, the data is also disjoint, and all inter-process communication is performed through message passing. This is in sharp contrast to the "blackboard" or shared-memory approach adopted in the single-process environment with respect to the multiple clients, in which multiple entities operate on shared data using a centralized process. While the data is disjoint, it must nonetheless be accessible by remote SubEnvs in order to enable process-interoperability. Thus, we assume that the underlying PCE has the necessary mechanisms to reference and bind remote data objects to local activities, consistent with the "global-browsing" requirement (see Chapter 5 for an actual implementation of these). Driven by autonomy requirements, however, the data in each SubEnv is private by default, and is said to be "owned" by its local process. Thus, access to both process and product data cannot be made from a remote process without prior "permission" from the owner process.

The high-level architectural view of a generic decentralized PCE with a three-site decentralized (instantiated) environment is depicted in Figure 3.3.

Each local environment consists, in addition to the single-process components (as outlined in Figure 3.1), an inter-process server, a remote-data server, a remote tool server, and a connectivity server (along with a possible connectivity database) that enables SubEnvs to connect to, and communicate with, other SubEnvs participating in the same (global) environment. These elements together form the necessary infrastructure support needed for process-interoperability. Notice the "no sharing" property, which enables full operation of some sites when some of the other sites are inactive or disconnected (e.g., the leftmost SubEnv in the figure is inactive).

A *multi-site activity* is an activity that involves, when executed, data objects from remote SubEnvs. Note, however, that this is a dynamic property of an activity, in that a given activity may or may not be considered a multi-site activity at different invocations, depending on whether the data bound to it includes remote objects. Multi-site activities are the building blocks of any process-interoperability in this model.

Referring to the context-hierarchy described in the previous section, it is important to note that there is intentionally no fourth level that represents a local process as part of a global process. This reflects our concept of *independent* collaborating (local) processes. While this model of a DEPCE provides global infrastructure support to enable interoperability among local processes, it explicitly avoids the need for a global "super" process — although such a process can be implicit.

3.2 Defining Process Interoperability: the Treaty

3.2.1 Motivation and Requirements

The following is a set of requirements specific to modeling interoperability, driven by the high-level requirements presented in Section 1.5.

1. In order to enable invocation of multi-site activities, there must be a way to define and agree on a *common sub-process* that would become an integral part of each local process intended to collaborate during that sub-process (but not necessarily by all SubEnvs in a global environment). A common sub-process determines what actions can be taken in the multiple participating SubEnvs.

At the very least, the multi-site activities must be commonly specified so that they can be identified during execution. But the "unit of commonality" might also be the process step, or even the task. In any case, this unit has to represent those process fragments that potentially involve multiple local processes. The decision as to what level (in the context hierarchy) to choose as the unit of commonality depends on the modeling primitives of the specific PML. For example, in a Petri-net formalism the transition (along with its input and output places) seems a natural choice, whereas in rule-based PMLs the rule (process step) is likely to be chosen. In PMLs that support task hierarchies and modularization (e.g., Articulator [77]), a subtask might be the right choice.

It is important to recognize that the activity portion of a decentralized sub-process need not be executable in every participating SubEnv, e.g., since the encapsulated tool may not be physically available everywhere. Instead, the activity only needs to be executable in one of the SubEnvs intended to collaborate, which would hence always serve as the invoking, or *coordinating* process. This means that common sub-processes are not necessarily reciprocal, in the sense that not all participant SubEnvs have identical process "privileges" on multi-site activities. This issue has direct implications on the model, as will be seen shortly.

2. In order to enable the definition (at least in strongly typed PMLs) and the execution (in all PMLs) of multi-site activities (as part of a multi-site common sub-process), the SubEnvs which are involved in that sub-process must have a *common sub-schema*, so that the types of the parameters specified in the activity are known at the SubEnvs. For example, if an activity A_1 is invoked from SubEnv E_1 on remote data from E_2, then E_2 must have the proper types in its schema and consequently the properly instantiated objects that are required by A_1.

   Note, however, that a common sub-schema does not necessarily imply that the corresponding data instances are shared — only their types (i.e., their schema) are shared. Defining common data schema and allowing access to data instances are separate concerns which should not be confused or coalesced.

3. Following the above argument, there must be a way to define (and subsequently, control) which data instances are allowed to be accessed, in what way, and by which SubEnv. That is, local databases are by default private, consistent with the autonomy requirement, and parts of them can be made accessible to enable remote access by multi-site activities.

4. It must be possible for a common sub-process (and the corresponding common sub-schema) to be shared among only some of the local processes (SubEnvs) of a given global environment, not necessarily all of them. Further, the same local processes must be able to participate in multiple common sub-processes, together with the same or different collections of remote processes. There is usually some portion of each local process that is not shared with any other process (a *private* sub-process). Similarly, it must be possible to specify access to subsets of the data instances to only some but not all participating SubEnvs, as opposed to allowing data to only be either totally private or universally public.

5. Finally, the PML must allow for both *dynamic inclusion and exclusion* of common sub-processes, as well as *independent evolution* of private sub-processes. The former is particularly important when independent pre-existing processes decide to collaborate, perhaps only temporarily, while the latter is important for preserving the autonomy of local processes. One of the architectural implications of this requirement is incremental process compilation capabilities[3].

3.2.2 Alternatives, Design Choices, and Justifications

In considering the possible alternatives to expressing common sub-processes within otherwise private and encapsulated processes, we can draw an analogy between our problem and similar problems in the "neighboring" domain of programming languages and distributed systems and investigate alternatives there:

---

[3] The meaning of "compilation" depends on the specific PCE, but most translate their processes into some internal format rather than repeatedly reparsing and reinterpreting the text.

1. Process interface specified within the PML — This approach includes programming language abstraction mechanisms in which all control and data of a unit are by default private (or hidden) unless specified explicitly as public in the unit's interface. For example, the *body/specification* distinction in **Ada** could be used to expose only the common sub-processes (or sub-*tasks* in Ada terminology) in the specification and hide the private sub-process in the body. Another example is the *export-import* mechanism in **Modula-2**, in which a subset of the activities (functions) could be exported by one process (*module* in Modula-2 terminology) and imported by another, while the rest of the local process (module) is by default hidden. A third example is the object-oriented approach to encapsulation, whereby a class denotes the public methods in its *interface*, and hides all other methods (which are part of its *implementation*).

   The main disadvantage with this language-based approach is that it is static in nature, conflicting with requirement 5 from the previous section. That is, the interface specifications cannot be changed while the program is executing, and all the bindings among the different modules are made at "compile" time. Another problem with this approach is that the underlying motivation for it is to provide abstraction for distinguishing between a unit's external (public) interface and its internal (private) implementation. While this might be the case in process inter-operability, more often the distinction is along the lines of shared versus private sub-processes, regardless of whether the private process is an "implementation" of the shared process.

2. Process interconnection language, separate from a specific PML — This is analogous to Module Interconnection Languages, in which a separate notation is used to denote how modules are inter-connected. For example, the Darwin [74] configuration language, the successor to Conic [75][4], enables (operating system) processes to interconnect independently of the specific language in which they are written, by means of typed *ports* through which data is exchanged between the processes. Ports are protected and made accessible through an import-export mechanism (the actual notation in Darwin

---

[4] Not to be confused with the Darwin environment mentioned earlier.

is `require` and `provide`).

The advantage of this approach over the previous one with respect to (software) process interoperability is that it can be made dynamic, as is the case with Darwin. That is, the nature and kinds of bindings between the processes can be changed dynamically. However, since this is still essentially a language-based approach, dynamic changes impose a problem in terms of comprehensibility: either the changes do not correspond to the original source definitions, which is an obvious problem, or the interconnection is not explicitly declared, defeating in some sense the purpose of using a language-based approach to begin with. The latter approach is taken in Darwin, where the references to the services (or control constructs) are passed in messages, allowing to change their behavior, but as the authors point out, this feature is not recommended for long-term or semi-permanent bindings.

3. Other distributed programming languages — This community produced numerous languages that support some form of dynamic program configuration among relatively independent (operating system) processes. One representative is Hermes [103], another port-based language in which new ports can be added to an executing (operating system) process and existing *port connections* can also be changed, by statements executed from within the existing Hermes code. New processes can also be added using the *create of* statement, but only from within an existing process. Thus, it is not possible to add new facilities that were not anticipated in the original program.

One aspect which is not addressed in either of these language-based approaches is the independent-operation requirement: as processes are defined (and later enacted) in separate SubEnvs, there must be facilities for enabling such dynamic cross-SubEnv bindings, which imply some degree of system support.

Our solution then is *system-based*, not language-based. The idea is to take advantage of the underlying virtual machine specialized for supporting process modeling, and extend the available PCE's enactment engine with mechanisms to support definition of the model. As such, this approach does not require the invention of a whole new PML intended for decentralization, nor does it make any assumptions about a particular PML, making it generically applicable.

The formal model presented below attempts to address all of the requirements presented in Section 3.2.1. The central concept here is the *Treaty*. Some intuition to the model can be gained from the "international alliance" metaphor mentioned in the beginning of this chapter. Multiple countries collaborate by signing "treaties" determining *what* kinds of artifacts are allowed to be exchanged and *how* to perform the exchange/collaboration. Once signed, treaties have to be ratified by the local parties, so that the full impact of the treaty is reflected in each country when enacted.

The Treaty model addresses directly requirements 1, 4 and 5 from Section 3.2.1. Requirements 2 and 3 are addressed in Sections 3.2.4 and 3.2.5, respectively.

3.2.3 The Treaty

In the following discussion, the following notation is used:

- E_i denotes an instantiated environment as defined earlier.
- A_i is used to denote a set of process steps that form a common sub-process as explained above. Note that in terms of the definition of an environment, A_i is a subset of P, i.e., it does not necessarily contain a subset of T, D, U, but it does imply a subset of S (schema) through the types of the formal parameters to the activities in A_i. Furthermore, A_i may consist of a set of unrelated steps, all of which are part of the common process, or they can be interrelated, for example representing a single common task.
- $A_i(E_j)$ denotes sub-process A_i of environment E_j, i.e., a fragment of E_j's process model.

We define the following operations:

1. $export(A_1(E_1), E_2)$ — Export A_1 from E_1 to E_2, enabling E_2 to *import* A_1. This operation executes locally at E_1.
2. $import(A_1(E_1), E_2)$ — Get A_1 from E_1, and integrate it with E_2's process. This operation executes at E_2 and involves also E_1. The successful outcome of this operation generates $A_1(E_2)$, a local version of A_1, fully integrated with the rest of E_2's process. The exact meaning of "full integration" is intentionally left out here, since it is PML-specific. Intuitively, the idea

is that the newly imported sub-process gets interconnected with the local process and becomes an integral part of that process. For some concrete examples, see Section 3.5.

These operations form the mechanism to implement common activities. However, as mentioned earlier, a separate concern is to determine execution privileges on the common activities, such as which SubEnv is entitled to execute a multi-site activity on remote data. In some cases, invocation of specific activities cannot be made from some of the SubEnvs, for example, due to tool invocation restrictions (e.g., licenses, platforms, location of tool experts, etc.).

It appears at first that such "execution privileges" semantics could be attached to the *export* and *import* operations in some fashion. However, early experiments with our implementation revealed that these are indeed separate and orthogonal concerns. That it, we separate the issue of how to provide common multi-site activities from the concern of how to restrict or control their application.

Therefore, we define the following two directives, each of which could be used in conjunction with either of the above operations:

1. *request*(A_1, E_1, E_2) — specify an intent of E_1 to use A_1 on data from E_2. Note that A_1 can be either exported by E_1 or imported from some other SubEnv.

2. *accept*(A_1, E_1, E_2) — allow A_1 to be used by E_1 on data from E_2. Once again, A_1 could be originally defined at E_1, in which case it was imported by E_2, or it could be exported by E_2 and imported by E_1. The latter case resembles the concept of a process interface, where the process publishes the tasks that can be used by other processes to access its own data.

To summarize, the four combinations and their intuitive meanings are:

1. *export_request*$(A_1(E_1), E_2)$ — I (E_1) want to use my A_1 on your (E_2) data.

2. *import_accept*$(A_1(E_1), E_2)$ — I (E_2) allow you (E_1) to use your A_1 on my data.

3. *export_accept*$(A_1(E_1), E_2)$ — I (E_1) allow you (E_2) to use my A_1 on my data.

4. $import\_request(A_1(E_1), E_2)$ — I (E_2) want to use your A_1 on your (E_1) data.

A (simple) **Treaty** (denoted as T) is a binary relationship between two sites, defined as either one of these two possibilities:

$$T_{A_1}(E_1, E_2) = export\_request(A_1(E_1), E_2); import\_accept(A_1(E_1), E_2)$$

or

$$T_{A_1}(E_1, E_2) = export\_accept(A_1(E_2), E_1); import\_request(A_1(E_2), E_1)$$

In words, this Treaty allows users operating at E_1 to execute activities defined in A_1 on data from E_2. We shall refer to this Treaty as "a Treaty *from* E_1 *to* E_2, *on* A_1". Both definitions lead to the same outcome, the difference being the origin of A_1: in the first expression A_1 is initially defined in E_1 and is exported to E_2, which imports it; whereas in the second expression A_1 is initially defined in E_2 and exported to E_1, which imports it.

Thus, a Treaty between two SubEnvs consists of one requester and one acceptor, as well as one exporter and one importer. The *export-import* pair of operations establishes a common step (containing multi-site activities), and the *request-accept* pair defines which site is eligible to invoke activities from the common step (the requester) and which one allows access to its data (the acceptor). The gist of the Treaty is that it requires both sides to actively participate (and perhaps negotiate) in the agreement that determines their inter-process interactions. In particular, a *request* on an activity without a corresponding *accept* on the same activity has no effect on either SubEnv (regardless of whether the activity is properly imported-exported). As for the order of the operations in a Treaty, the main reason for them not being commutative is to protect the privacy of the exporting process. This means that any implementation of *import* should restrict its visibility only to activities which have been already exported by other SubEnvs.

It is important to understand that the Treaty relationship is not symmetric. For example, the Treaty above does *not* imply that E_2 can run activities from A_1 on E_1, i.e., it is only uni-directional. This property of Treaties addresses the concerns raised in requirement 1 in Section 3.2.1. Furthermore, the Treaty is not transitive, and each Treaty between two sites must be formed explicitly. (Treaties can be considered reflexive, though, if self-export and self-import are defined as "no-ops".)

The extension of a Treaty to multiple sites is defined as:

$$T_{A_1}(E_1, (E_2 \ldots E_n)) = \bigcup_{i=2}^{n} T_{A_1}(E_1, E_i)$$

This multi-site Treaty allows users operating in E_1 to run activities defined in A_1 on remote data from some or all of E_i, $i > 1$.

To enable symmetric Treaties, we define a (binary) *Full Treaty* (denoted FT) as:

$$FT_{A_1}(E_1, E_2) = T_{A_1}(E_1, E_2); T_{A_1}(E_2, E_1)$$

and similarly, a multi-site full Treaty is defined as:

$$FT_{A_1}(E_1, E_2 \ldots E_n) = \bigcup_{i<j} FT_{A_1}(E_i, E_j)$$

This consists of the union of all unordered pairs of binary full Treaties (or all ordered pairs of regular Treaties). While symmetric, full Treaties are still not transitive, to protect the privacy of sites.

A Full Treaty allows any participating SubEnv to invoke a multi-site activity on data from any other SubEnv in the Treaty. Note that when multiple sites are involved, there are many combinations of possible Treaties between the sites on the same set of activities, not only simple or full. For example, the Treaties:

$$T_A(E_1, (E_2, E_3))$$

and

$$T_A(E_2, (E_1, E_3))$$

allow either E_1 or E_2, but not E_3, to invoke multi-site activities from A on data from some or all of the three sites.

As can be seen, this model provides maximum flexibility in expressing interprocess collaboration, and each participant in a Treaty must explicitly "sign" it by invoking the proper operation that reflects its role in the Treaty.

In order to retract from Treaties, the following operations are defined:

1. *unexport*$(A_1(E_1), E_2)$ — This operation executes in E_1. It removes A_1 from further being available to E_2 and invalidates possible previous Treaties. In addition, it revokes any privileges which were associated with the *export* (see below).

2. *unimport*$(A_1(E_1), E_2)$ — This operation executes in E_2, effectively removing A_1 from E_2's process. Like *unexport*, it invalidates any previous Treaties and privileges which were attached to the *import*.
3. *cancel*(A_1, E_1, E_2) — has the opposite effect of *request*, i.e., it disallows to further use A_1 at E_1 on E_2. It is issued at the requester end of a Treaty.
4. *deny*(A_1, E_1, E_2) — The opposite of accept, it disallows E_1 to further access E_2's data through A_1. It is issued at the acceptor end of the Treaty.

Since *export* and *import* are the mechanismfor establishing shared common subprocesses, when *unexport* (*unimport*) is executed on a previously exported (imported) activity, the corresponding execution privileges property (either *request* or *accept*) is also revoked (by *cancel* or *deny*). The opposite is not true, though. A *cancel/deny* does not imply *unexport* or *unimport*. For example, a requester activity could be transformed to an acceptor activity by issuing a *cancel* followed by *accept*, regardless of whether it is an exported or imported activity.

In order to enable unilateral withdrawal from the treaty — which fits well both with the operational independence and the process autonomy principles — all retracting operations are local, not involving remote interaction. However, this results in an overhead in execution time, as every invocation has to be validated at run time, because even if at some point at the past such an invocation was well formed in a Treaty, it might not be the case at the time of the invocation. This issue is discussed in Section 3.2.6.

To summarize, Treaties are the abstraction mechanism used to *define* process interoperability. The only way by which a SubEnv can collaborate with other SubEnvs is through these pre-defined arrangements that determine how to collaborate, and on what artifacts. Consequently, the degree of collaboration (vs. autonomy) between each pair of SubEnvs is determined by the "size" of their common sub-process. This can range from total isolation (no common sub-process is defined) — where the SubEnvs have no means to access each other's data but are entirely autonomous — to total collaboration (the entire process is common) — where the SubEnvs lose any autonomy and logically share the same process and data and are perhaps only physically distributed.

By splitting a Treaty into two independent operations and the Full Treaty into four operations (as opposed to bundling them to one global operation) we ensure that both ends agree on the Treaty and join it on their own terms. Not requiring synchronous execution of *export* and *import* enables Treaties to be formed incrementally and *when* each party wants to join them. In fact, of all of the primitive operations, *import* is the only operation that requires both sides to be simultaneously active. This independent multi-step protocol also enables SubEnvs to retract from, and join to, a Treaty, independently and dynamically.

Finally, it might appear that this approach suffers from being too low-level in that it makes it difficult and somewhat awkward to define Treaties between sites. However, this formalism ensures maximum process autonomy in all involved sites. A particular implementation might use "macros" or "scripts" that perform all the necessary operations automatically to form Treaties between "friendly" sites in cases that privacy can be compromised for simplicity and convenience. Alternatively, an implementation might decide to bundle some of the operations. For example, it could always implicitly associate *export* with *request* and *import* with *accept*, or vice-versa, but not both. Or it could set defaults for the combinations but allow the expert process administrator to modify them. Finally, the PCE can make provisions for enabling a user to be an administrator on multiple SubEnvs, so that in environments that allow multi-site administrators (e.g., when the interoperability is between tightly-coupled SubEnvs), it is possible to bundle the Treaty as one operation, without violating autonomy. Several of these alternatives were in fact implemented in Oz (see Chapter 4 for details).

3.2.4 Defining Common Sub-Schemas

There are several alternatives to defining common sub-schemas, with differing degrees of flexibility and complexity. The simplest and most restrictive approach is to require a global schema, i.e., all SubEnvs must have identical schema. This is obviously too restrictive and counter-autonomous, and in particular it prevents the bottom-up approach of forming Treaties over pre-existing environments. While some common sub-schema has to be formed ultimately, the goal is to minimize its extent. In addition, given the share-nothing architecture, it would be very hard to guarantee that the the complete schemas, which are maintained locally at the SubEnvs, are kept identical.

A second alternative would be to require global product-data sub-schema, but to allow variation in the process-data sub-schema. (Recall that product data are the actual artifacts under development, e.g., source files, design documents, and so forth, and process data is used by the PCE to manage the project, e.g., a source file's version, its compilation status, etc.) The rationale behind this division is that in cases where the product data is heavily shared, it provides for more freedom in defining the process model while still requiring a common denominator for definition of the product. The opposite alternative would be to require global process data and local product data — in cases where a global process needs to be defined but security and privacy of local data is important. While these approaches support a larger degree of autonomy (process and data, respectively), they are still too restrictive. In the former approach, some local processes might still need to retain additional private product data, and in the latter approach some SubEnvs might need private process data for extensions to the local process. In addition, data might not always be clearly classifiable in to one of the two categories. Thus, while the distinction of process and product data is conceptually important, it seems like the wrong criteria for determining what parts of the schema to make common.

The third and most general approach would be to have both common and private sub-schemas, regardless of the type of attributes/classes involved. Here again, a restricted approach would be to require the common sub-schema to be shared by all SubEnvs, and a more general approach would require only pairwise common sub-schemas that match the corresponding pairwise Treaties. This approach is the most flexible, and fits well with the overall research requirements, but it is also the hardest to realize. The problems associated with this approach fall in general under the domain of *schematic heterogeneity*, a topic that is investigated by the heterogeneous database community (e.g., see [66]) and is largely beyond the scope of this thesis as a research topic, although a practical solution is given in Section 4.3.3.

3.2.5 Sharing Data Instances

Requirement 3 in Section 3.2.1 indicated the need to identify data instances which should be made accessible to Treaty subtasks. At the highest level, there are two main alternatives to address this issue: (1) tie the export of data instances with the export of common-sub tasks; or (2) treat export of data instances as orthogonal to defining common-processes. The first alternative implies that processes or tasks are defined on particular data instances. While this might be true in some cases, it is a narrow view that necessarily restricts the notion of a process and its scope. A more generic and realistic view of a process is that it is defined over classes of instances, and may over time be bound to, and execute on, different instance sets of the project database. In addition, it is quite possible that Treaties (as well as local processes) are defined before the creation of instances that are used in that Treaty, in which case the first alternative is impractical.

Taking this view, our approach is to define an access-control mechanism that defines which SubEnv can access which data, through which operation. Formally, we define the operation

$$export\_data(D_1(E_1), mode, E_2, A_2)$$

to denote: make data instances D_1 of the local SubEnv E_1 accessible to remote SubEnv E_2 in the specified access *mode* (e.g., read, write) under operations from A_2. Note that this operation is meaningless if A_2 is not a part of a Treaty between E_1 and E_2, and therefore could be checked at definition time.

It is important to note that this operation, like the rest of the model, ignores issues that have to do with particular databases, PMLs, and PCEs, which might require prohibitively expensive implementation of these operations. In some cases it might be necessary to restrict the model in order to make it feasible. For example, the Oz implementation (given in Section 4.3.4) does not specify the last parameter, only allowing to distinguish between SubEnvs, not specific activities. Another PCE-specific issue is the granularity of access control.

3.2.6 Independent Local Evolutions

We discuss now briefly how the Treaty model can support local evolutions, and some tradeoffs. A fully detailed account of this subject is given in Chapter 4.

We assume that environments are evolved by the process administrator, who modifies in some way the process and/or the schema and reloads them into the environment, possibly upgrading the existing populated product and process database.

The most important integrity constraint associated with the validity of a Treaty is what we refer to as the *common-subprocess invariant*, which simply states that a common sub-process which was defined through a Treaty, must remain identical in all participating SubEnvs to retain its validity. Since there is no shared space in which Treaties are stored, this is the only way to guarantee that the original Treaties have not been altered by the time they are invoked on remote data. Thus, if due to local evolutions a Treaty step is altered only in some but not all SubEnvs, the Treaty should be invalidated.

Given a Treaty $T_{A_1}(E_1, E_2)$, several kinds of evolutions could make it invalid:

1. E_1's process is evolved, possibly modifying activities in A_1. Then the treaty might no longer be valid since A_1 could differ arbitrarily from the version agreed upon when the Treaty was signed. Therefore, whenever a multi-site activity is invoked from E_1 on data from E_2, E_2 has to dynamically check whether its version of the imported activity is identical to the version invoked by E_1. A convenient mechanism is to associate a *timestamp* with each activity (see Section 4.3.2.1).

2. E_2's process is evolved, possibly modifying activities in its own version of A_1. In this case the treaty is again no longer valid, and $A_1(E_2)$ should be outdated and its validity checked at execution time.

3. E_1 *cancel*s A_1 on E_2, disallowing users in E_1 to further invoke activities defined in A_1, from E_1, on data from E_2. To cope with this evolution, each time a multi-site activity is attempted at E_1, E_1 must check locally whether it is still allowed to invoke such an activity on data from E_2. The reader might wonder why to check locally for the validity of one's own operations. The answer is that SubEnvs in general involve multiple users, and even though a *cancel* was carried out by the local administrator, other users, perhaps not aware of the cancellation, might still try to invoke A_1.

4. E_2 *denies* E_1 to further execute A_1 on its own data. This evolution has the most significant implications with respect to Treaties, as it enables an accepting site to leave the Treaty unilaterally. Note that "leaving" the Treaty is not "breaking" it, since the Treaty is not misused. It is simply prohibited. To allow such one-sided *deny* operation, each time a multi-site activity issued at a coordinating site requests to access remote data, the remote SubEnvs must check dynamically whether the issued activity is still *accepted*, and reject it if it is not.

The approach outlined above is consistent with our concerns for maximum locality and autonomy, both logically and physically. An alternative approach at the other end of the spectrum is to attempt to immediately notify all involved SubEnvs when any of the above operations occur. For example, a *deny* operation would notify the denied SubEnv, which in turn would invalidate the Treaty at the "source" SubEnv. This approach has the advantage of reducing the number of invalid invocation requests, and more importantly, possibly eliminating the need for dynamic validation of Treaties. However, since it is possible that at a given time only some but not all of the SubEnvs are active or reachable on the network, it is in general impossible to eliminate altogether the dynamic validation mechanism. Moreover, this approach unnecessarily tightly binds the SubEnvs and incurs significant communication overhead, which is unacceptable when the SubEnvs are arbitrarily distant. For example, each local evolution in a SubEnv would imply immediate broadcast to all other SubEnvs that have any Treaty arrangement with it. Finally, for this "immediate-update" approach to be effective, cross-site operations must be atomic (all or nothing), or otherwise all dynamic checks would still have to take place, defeating the purpose of the update in the first place. But preserving atomicity, particularly for distributed operations, involves significant overhead (for example, supporting context sensitive rollback) and require global control, two good reasons to avoid this approach.

3.2.7 Inter-process Consistency

The final requirement in Section 3.2.1 regarding incremental and dynamic inclusion and exclusion of common sub-processes, introduces the problem of preserving the local process consistency when a process is augmented with a new decentralized sub-process, or such an extension is removed. In particular, violations of process consistency that cannot be tolerated must somehow be rejected.

The key to the solution of this problem lies in the observation that this is a subset of the more general problem of process evolution, where a local process is modified by adding (removing) new activities to (from) it. Since there is no global process, there are no global consistency considerations, reducing this problem to the same problem as in single-site PCEs. Thus, this issue is not discussed here any further. The solution in OZ is to use the Evolver utility, which was covered earlier in Section 2.2.5.

3.3 Enacting Process Interoperability: the Summit

Given the Treaty model for *defining* process interoperability, the second major issue is to support the *enactment* of multiple interoperating processes. Once again, the major requirements that affect this model are *autonomy, independent operation*, and *flexible interoperability*. Although a Treaty requires some mechanisms that enable its operation, inter-process enactment requires much more infrastructure support, since it implies extending significantly the process engine.

3.3.1 Alternatives, Design Choices, and Justifications

At first glance, there are two ways in which a multi-site task can be executed: (1) one SubEnv (call it the coordinating SubEnv) copies remote data into its own space and executes locally, or (2) the task leaves the data where it is, and requests that its activities be executed by the remote SubEnvs. This is similar to the two main approaches to distributed program execution: fetch the data and execute locally, or send a request for remote function execution. There are obvious tradeoffs between the two approaches, and the superiority of one over the other largely depends on the nature of the program and the volume of the data involved.

However, since a multi-site task inherently involves more then one process, neither of these approaches is always feasible or desirable: (1) Process autonomy restricts application of the data fetching approach, since some of the remote data might not be accessible to the executing process, and even if it is, the prerequisites and consequences determined by the coordinating process might not maintain consistency with respect to the remote process(es). (2) The function sending approach does not address activities that manipulate data from multiple (local and remote) processes, but instead assumes that an activity's arguments all reside in the same SubEnv. In addition, as mentioned earlier, tools invoked by an activity may not be available at a remote SubEnv (in fact such a scenario might be the initial motivation for running the activity in the originating site), and even copying the tools might not work if the SubEnvs operate on heterogeneous platforms or if there are licensing restrictions.

We devised a third hybrid approach, which combines the two approaches mentioned above in a manner that ameliorates their limitations. At the activity level, remote data is fetched and modified locally; but at the process-step level, any subtasks emanating from prerequisites and consequences are executed at the remote SubEnvs. This permits activities with arguments from multiple SubEnvs, executes tools at the same site as their process, and maintains consistency in process and product data according to the local processes owning the data.

3.3.2 The Summit

Following the "international alliance" metaphor mentioned in the introduction to this chapter, our decentralized enactment model can be described as a "summit meeting". Before the meeting (multi-site activity), each party (process) handles local constraints (prerequisites) that are necessary for the meeting to take place; then the meeting is held at one location (SubEnv), where the various parties send representatives (data) to collaborate; once the meeting is over and agreements were made (results of the activities), all parties return home (to their SubEnvs) and carry out the implications (consequences) of the meeting locally. Summits can lead to subsequent Summits, each involving a subset of the parties, possibly with different representatives (data arguments).

Similarly, process interoperability takes place when an activity is invoked (either manually by an end-user or automatically by the process engine) on data from one or more SubEnvs. The case of only local data from the same SubEnv does not lead to inter-process collaboration, and is handled however it would normally be done by the underlying single-process PCE. We call the process from which the decentralized activity is invoked the *coordinating process*. The Summit protocol consists of the following phases (not all of which must necessarily exist in all implementations):

1. *Summit Initialization and Verification* — First, the coordinating process in which the Summit request was issued, establishes a task context (necessary to support interleaved execution of multiple activities) and allocates the necessary resources needed for the Summit. It then binds the actual parameter objects (at least one of which is remote, or otherwise this would not be considered a Summit) to the formal parameters of the activity.

   Initialization is followed by verification, checking whether the Summit is allowed to be executed. For example, a multi-site activity A_1 is eligible for execution from site E_1, on data from site E_2 and E_3 only if it conforms with the Treaty protocol, i.e.:

(a) A_1 was defined as a common activity by means of any valid combination of *export* and *import* operations. For example, it could have been exported by E_1 and imported by E_2 and E_3, but it could also have been imported by E_1 and *exported* from E_2 or E_3.

(b) A_1 has been *request*ed in E_1 and *accept*ed in E_2 and in E_3.

However, while a necessary condition, this static property is not a sufficient condition for allowing the Summit, since by the time the Summit is invoked for execution, local evolutions might have violated the Treaty. Thus, an additional runtime Treaty validation phase is required. Validating whether the specific data instances are accessible to the remote processes is determined at the remote SubEnvs by checking their local access-control mechanism to see if the instances are properly exported to the coordinating SubEnv, as outlined in Section 3.2.5.

2. *Pre-Summit* — The involved processes (i.e., those that own some of the data requested by the multi-site activity) are notified, and all of them (including the coordinating process) perform simultaneously and asynchronously pre-Summit process actions, *each according to its local process, with its local data, in the local SubEnv.* Examples of pre-Summit actions include, not necessarily in this order: (1) Verification that prerequisites imposed by the process step enclosing the activity are satisfied (locally); (2) Verification that the activity can be executed with respect to the task workflow; (3) Active invocation of related activities, e.g., to satisfy (1) and (2); and (4) Deriving and binding data arguments that are required by the activity but were not specified as parameters (for example, a `compile` activity on a `C` source file may require to bind all the `#include`d header files for the activity). Pre-Summit requires that all involved SubEnvs identify the same requested activity, in order to know what to verify/satisfy. This is guaranteed through the `import` mechanism of the Treaty.

   It might be possible in some cases (depending on the PML as well as the specific activity) for the coordinating process to determine locally whether or not launching remote pre-Summit is necessary for each participating SubEnv, in which case no “fan-out” to the local sites is required. In general, however, the local SubEnvs need to be able to decide for themselves whether or not they need to undertake any work. The main point is the locality of the enactment, which is determined solely by each SubEnv on its local data, without “global” intervention.

3. *Summit* — If pre-Summit is successful in all involved processes, the requested activity is invoked in the coordinating process, with all the necessary local and remote data arguments. Note that for the time being, we restrict the invocation of a summit to occur not only at one time, but also at one location (i.e., the coordinating process). An extension of this phase that enables to perform summit activities across multiple sites simultaneously (e.g., groupware activities) is discussed separately in Section 3.6.

4. *Post-Summit* — When the Summit completes, all involved SubEnvs are notified, and all of them (including the coordinating SubEnv) perform simultaneously and asynchronously Post-Summit process actions, again *each according to its local process, with its local data, in the local SubEnv.* Examples of Post-Summit actions include, not necessarily in this order: (1) Assertions on the process and product data that reflect the fact that the various activities were executed (depending on the PCE, it may not always be possible to directly modify such data within the activities themselves); (2) Binding and assignment of data affected by the activities that were not supplied as arguments; (3) Verifying that consequences imposed by the steps in the Summit can be fulfilled (this is not always a logical implication of the pre-Summit verification); and (4) Triggering execution of further activities, e.g., as part of (3).

5. *Summit Completion* — When Post-Summit completes in all local sites (including the coordinating SubEnv operating in “local” mode) the coordinating SubEnv checks whether further Summits are pending (see below on how composite Summits are formed). If any Summit is pending, the algorithm returns to step 1. If no Summits are pending, the Summit is completed by releasing all resources associated with the Summit. Note that this is a

"tail-recursive" algorithm, in that its last step calls itself.

Thus, all participating SubEnvs act as if the activities in the Summit took place in their local process with local data only — in the sense that they all carry out the (pre and post) implications of the Summit activity — although only the coordinating SubEnv really executes the activities. The interesting point here is that both pre- and post- Summit phases occur in each SubEnv *only* according to its local process, while execution of the Summit phase involves collaboration among the participating SubEnvs. This design minimizes the interference between the processes (and hence maximizes autonomy) while still allowing them to carry out the desired common activities as agreed upon in the Treaty.

3.3.2.1 Composite Summits

The Summit algorithm presented above indicated the possibility of multiple related Summits executing as one unit. Indeed, the capability to enact multiple Summits (or multi-activity Summits) is crucial, as can be evidenced even from the example given earlier in Chapter 1. While a complete answer necessarily delves into the particulars of a specific PML (and indeed such a solution is given in Chapter 4) — the general idea is that subsequent Summits are invoked through whatever enactment and binding mechanisms the PCE and its associated PML provide. They might be specified explicitly in an imperative-like PML with static binding, and therefore invoked in a straight-forward manner, or inferred implicitly through the PCE's inference mechanism (e.g., rule chaining, or Petri-net transition) with dynamic binding capability to bind remote objects to new Summits. The point is to be able to distinguish between derivation of local activities (which are therefore still part of Post-Summit) versus new Summit activities.

Thus, a composite Summit (e.g., consisting of multiple Summit activities) can be viewed as alternating between "local" mode — in which each participating site (including the coordinating site) performs local operations — and "global" mode in which the coordinating process carries out operations involving remote data, with the approach intended to minimize the "global" mode and maximize the "local" mode.

Finally, a note on concurrency: the Summit protocol is primarily designed for increased autonomy. Nevertheless, this design also contributes to increased concurrency (and hence throughput) since the local (and thus independent) operations can potentially execute in parallel.

3.3.2.2 Transactional Semantics

The details of the operational and transactional semantics of the Summit are left undefined in this chapter since they are inherently PML- and PCE-specific. A detailed solution is given in Section 4.4.5. Nevertheless, some of the issues that come up are generic, and are worth mentioning here.

For one thing, it is mandatory that some transactional semantics be attached to the Summit. In particular, there must be mechanisms that preserve the atomicity of at least a single Summit, which means that distributed abort and rollback as well as distributed commit of Summits must be supported. While autonomy concerns should play a role in the design of distributed transaction management supporting our model, it is in general impossible to provide global consistency as well as local autonomy in transaction management (as shown by Korth in [102]). The approach to be taken, then, should be directed towards minimizing the global atomicity requirements of Summits.

Another problem with supporting transactions in our model is that interleaving among distributed tasks is inherent and unavoidable, making it harder to preserve atomicity in general and serializability in particular. (It is trivial to preserve the serializability of non-interleaving, and thus serial, execution.) The reason is that when a Summit task "fans-out", the coordinating SubEnv must switch contexts, or else the SubEnv would be blocked indefinitely (until all remote SubEnvs finish their local execution). This effectively creates an interleaving among distributed transactions each time a Summit alternates between global and local modes of execution.

The transactional semantics of Summits in Oz are presented in Chapter 4, where they are tied to the notions of consistency and automation forms of enactment. However, the actual implementation of the decentralized transaction manager that supports these semantics is for the most part beyond the scope of this dissertation, and is part of Heineman's dissertation, covered in [49].

3.4 The Motivating Example Revisited

Recall the motivating example introduced in Section 1.3 (Figure 1.1). Figure 3.4 illustrates its enactment using the Summit protocol. Note that each box in the Figure does not necessarily represent a single activity, but rather a subtask that might be broken down

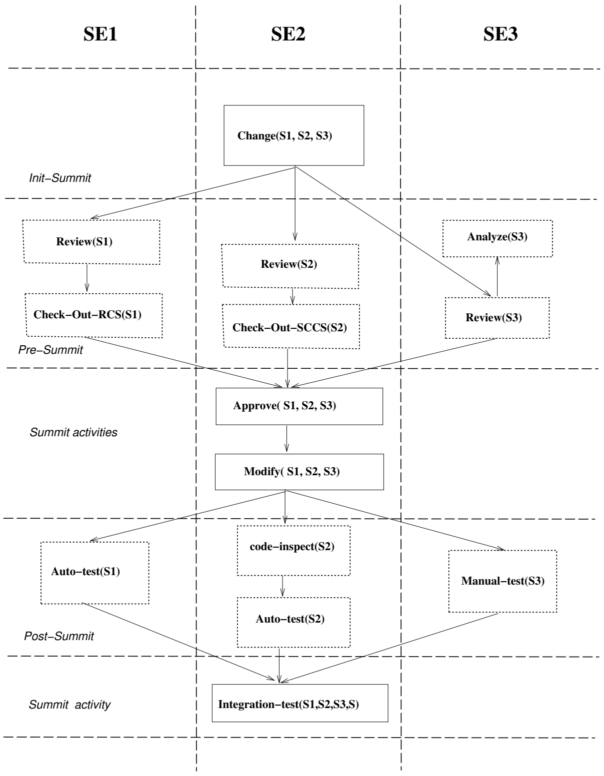


Figure 3.4: Enactment of Motivating Example

into a fine-grained set of process steps.

The `change` activity is initiated by the coordinating SubEnv **SE2**. Pre-Summit takes place in a decentralized manner, where each SubEnv performs the `Review` activity locally according to its own process. For example, **SE3** requires an additional `analysis` step before the review and both **SE1** and **SE2** require a check-out phase using different configuration managers (RCS and SCCS, respectively). Once reviewed by all sites, the Summit activity `approve` is executed, determining whether to approve or disapprove the change based on the local reviews. If the approval step succeeds, the `modify` activity is executed, where the objects are modified. When finished, Post-Summit begins, again in a decentralized manner. All SubEnvs are engaged in a unit-test step, but each one does it according to its own process. For example, **SE3** employs a manual-test procedure (e.g., for testing the user interface) which involves human users that actually perform the tests (devising the input sequences for the test suites can be also done manually or automatically for either manual or automatic testing), whereas the other SubEnvs perform automatic

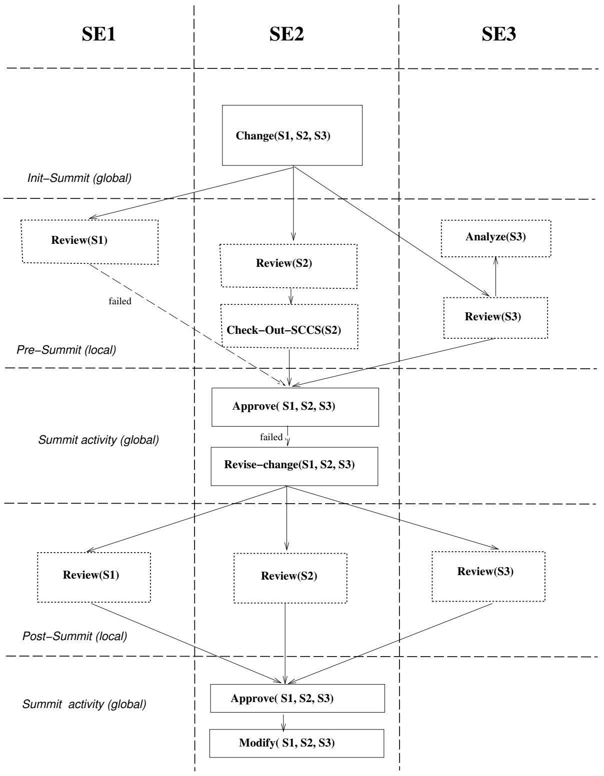


Figure 3.5: Another Enactment of Motivating Example

testing, but **SE2** has an additional code-inspection step. Completion of the local testing leads to `integration-test`, another Summit activity in this composite Summit.

It is important to understand that Figure 3.4 depicts a particular execution trace of the process, not the whole process. For example, figure 3.5 shows a different execution trace of the same process, where this time the `review` phase fails at **SE1**, requiring a revision in the proposed change, after which a second review succeeds and leads to the `modify` activity. This example illustrates how the Summit protocol can support some form of process negotiation, where the Summit activities represent the negotiation table, and the local implications represent private consultations. A full example of such a negotiation-based process is given later in Chapter 6.

3.5 Application of the Model

We describe now how the model can be applied to three families of PCEs categorized by the paradigm underlying their PMLs, namely rules, Petri-nets, and grammars, and

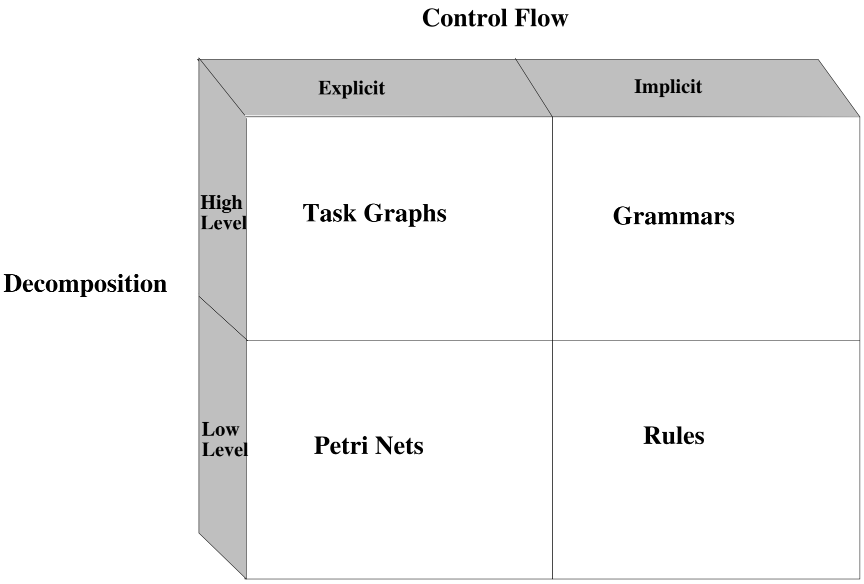


Figure 3.6: Comparison of PMLs

outline how the model might be applied in the APPL/A process programming language. These families were chosen to cover most of the known PCEs [64].

Each PML style has its own strengths and weaknesses with respect to process modeling, and there is no "perfect" process modeling (some of these differences are overviewed in [60]). Further, a particular PML might arbitrarily divert from the properties that identify its family. Nevertheless, these PMLs can be generally characterized in terms of their support (or lack there of) for explicit (implicit) control-flow, local constraints, and modular decomposition (other important aspects of the PML are ignored in this discussion). In general, rules are best suited to defining local constraints and automatic inference of process steps, but multi-rule tasks are implicit and rule are inherently low-level with respect to supporting modularity and process decomposition. Petri-nets are good at explicitly defining the control flow of the process, but even though some versions enable Subnets to be defined as means of decomposition, they are still inherently low-level, like rules. Grammar-based PCEs give implicit control-flow support, and modular decomposition support. Finally, Task Graphs provide both explicit control flow and modular decomposition.

Figure 3.6 summarizes these characterizations along two axes: implicit vs. explicit control flow, and low-level vs. high-level decomposition.

Since enactment in PCEs is heavily influenced by the type of PML used to model processes, we cover in the sequel Summits in full. As for modeling, though, since the Treaty approach is not language-oriented, most of the operations are not considered here. The main issues that are covered with respect to Treaties are: (1) identification of appropriate units of commonality and (2) implementation of the *import* operation.

Finally, since we take the existing PMLs as given, the uninitiated reader should see the cited references for background and justification of each approach to process modeling.

3.5.1 Rule-Based PMLs

In general, a rule represents a process step in our context hierarchy, consisting of an optional *action* (activity) with its *pre-condition* (prerequisites) and *post-condition* (immediate consequences). Some rule formalisms consider the action to be optional, permitting "inference rules" where the pre-condition directly implies the post-condition, while other formalisms have only two parts, usually with the action and post-condition merged together. Parameters and variables are represented by *symbols* used in the rule, which are often but not necessarily typed (the restriction to data of the appropriate type is the simplest form of prerequisite). The process step corresponding to a rule is enacted by first evaluating the pre-condition; the action is initiated only if the pre-condition is true. Completion of the action leads to asserting the post-condition.

Tasks are implicit in the possible rule chaining. *Backward chaining* involves matching the pre-condition of a rule with some rule whose post-condition might cause some subpart of the pre-condition to be satisfied. Then the firing of the second rule is considered recursively. *Forward chaining* arises when the action or post-condition of a rule fulfills the pre-conditions of some rules, which are then fired recursively. Rule-based PMLs can be roughly divided into backward-chaining oriented such as Prolog-based Darwin [78], forward-chaining oriented such as AP5 [23], and those that incorporate both, like Merlin [109].

The rule is the natural unit of commonality for import-export, although in practice the *import* can be carried out on a set of logically related (but nevertheless independent from each other) set of rules. Implementing *import* is relatively straightforward for rules, since the relationships to other rules is determined implicitly through predicate matchings as outlined above. If the PML supports static compilation of the rule set into a rule network, a re-compilation is necessary. Otherwise, the new rule has to simply be added to the rule base and no further recompilation is needed.

The Summit protocol applies to rules as follows: when a multi-site rule is fired, either directly by a user or indirectly through automatic chaining (we assume that the PCE has mechanisms to bind remote objects as rule parameters), the following takes place:

1. *Summit Initialization and Verification* — The participating SubEnvs supply data to be bound to the symbols of the rule, and the validity of the Summit request is performed as outlined in the generic model. The details of how this is accomplished depends on the PCE.

2. *Pre-Summit* — The condition of the rule is evaluated. Although in some cases (depending on the PML and the particular rule) the condition evaluation could be broken down to local sub-conditions which could be then evaluated in a distributed manner, single conditions that involve data from multiple sites must be evaluated centrally in the coordinating site. For example, if a and b are symbols bound to objects $a1$ and $b1$, respectively, where $a1$ and $b1$ belong to different SubEnvs, then a condition of the form:

   $$if(a.status = b.status)$$

   must be evaluated in the coordinating SubEnv. Thus, while part of the condition evaluation can be possibly distributed for optimization purposes, it cannot always be fully distributed.

   When the evaluation completes, if the condition is not satisfied (or at least is not already known to be satisfied), all SubEnvs with data that does not meet the condition are then notified. In backward-chaining PCEs, each SubEnv may then activate other rules in its local process, in an attempt to satisfy (or verify) the failed pre-condition on its own data — possibly in a backtracking manner trying multiple alternatives. In any case, if the pre-condition cannot ultimately be satisfied, then the rule execution is halted in the coordinating process.

3. *Summit* — The action is executed in the coordinating SubEnv, involving both local and remote data.

4. *Post-Summit* — On completion of the Summit, the coordinating process fans-out with the relevant output to the remote SubEnvs. All sites (including the

coordinating site in "local" mode) in turn assert the post-condition of the rule on their own data. In forward-chaining systems, this leads to triggering of other rules in the local SubEnvs whose pre-conditions have become satisfied. However, while inferring rules for forward chaining, any discovered rules which are Summit rules (i.e., they fulfill the necessary Treaty requirements and have remote objects bound to them) are deferred until after the Post-Summit phase completes (see below). Each SubEnv notifies the coordinating SubEnv when it completes its local Post-Summit phase.

5. *Summit Completion* — At this phase the coordinating SubEnv checks whether there are further Summits pending, in which case it starts another Summit, or if no more Summit rules are pending the (multi-rule) Summit is completed.

Since the Oz PML is rule-based, we defer further discussion of rules to Chapter 4.

3.5.2 Petri-Nets

The Petri-net [86] is a powerful formalism for modeling concurrent systems, and it has been widely applied to software process modeling. The application of our decentralized model to Petri-net-based PCEs is influenced primarily by SLANG [3] and FUNSOFT [39], and their corresponding PMLs SPADE and MELMAC, respectively. Each of these PMLs is based on extended Petri-net formalisms (specifically, SLANG is based on ER nets, and FUNSOFT on predicate/transition nets), but we will stick for the most part with the general Petri-net formalism.

Transitions usually represent our notion of activities (note that our activities are different from SLANG's notion of activities, which are more like our notion of a task). The equivalent of a process activity that involves (possibly external) tools is termed in SLANG a black transition, and in FUNSOFT it is called a regular agency.

Places represent the activity's formal parameters. When places are typed, the input places can be viewed as prerequisites on the transitions, and the output places as immediate consequences on transitions.

A *predicate* on the actual parameters (tokens, see below) can be attached to a transition and must be satisfied prior to firing the transition. The predicates define local constraints on an activity, as opposed to the general control flow expressed by the topology of the net. Both languages support the notion of a predicate. In SLANG they are called guards, and in FUNSOFT simply predicates.

Tokens (or the marking of the net) represent the current state of the process under execution and the product data used in the activities. A transition is said to be *enabled* when its input places contain the sufficient quota of tokens (with the right types) and the predicate on the transition is satisfied.

Finally, a single net can be divided into several subnets, or they can be nested in a hierarchy in which case a subnet is represented as a transition of its supernet, providing better abstraction and decomposition mechanisms. Such a subnet corresponds to our notion of a task (if the PML does not support subnet constructs, tasks are implicit, like in basic rule-based PMLs).

A transition along with its attached predicates and input and output places correspond to a process step, and is necessarily the minimal unit of commonality for Treaties, since it is impossible to alter the input or output places of a transition without having to modify the transition itself (this would be analogous to allowing to change the number or types of the parameters to rules). Also, the predicate is a local constraint on the transition and therefore conceptually part of it.

The *import* operation in a Treaty is more complicated than in the case of rules, mainly due to the explicit topology of the net. That is, while in the case of rules the relationships between the imported and the existing rule sets can be inferred automatically, in Petri-nets there must be manual modification of the net to integrate the imported process steps. The integration of a process step into an existing net involves: (1) merging (or adding new) output places of local steps with input places of the imported step; and (2) merging output places of the imported step with (possibly newly created) input places of local steps. These operations effectively merge the imported (common) step with the local process (net). It is not mandatory, however, to connect an imported step to the net. There might not be opportunities to do so, just as it is possible that in rule-based PMLs an imported rule will not match with any local rule, leaving it isolated, in which case pre and Post-Summit become trivial.

The Summit protocol starts when a common transition is attempted, and the input places contain some tokens representing remote objects (again, we assume remote binding capabilities which are provided by the underlying PCE):

Summit initialization — The unusual aspect of this phase is the binding procedure.

While only the coordinating SubEnv actually binds data arguments to its input places, all involved SubEnvs mark their nets like the coordinating SubEnv, except the tokens in the non-coordinating SubEnvs are merely stubs.

Pre-Summit — The transition's predicate (if any) is evaluated at the coordinating site, and if not satisfied, the involved SubEnvs are notified. Since Petri-net based PMLs are usually not extended to support the equivalent of backward chaining in rules, pre-Summit might not exist or degenerate to condition evaluation if needed to be performed in a distributed manner.

Summit — The transition is fired in the coordinating SubEnv, invoking an activity on the data arguments. When the activity finishes, all involved remote SubEnvs fire the transition *without executing the activity.* If there is a conditional branching that depends on the result of applying the activity, then the same "return code" is used in all SubEnvs to properly direct the flow of tokens to the output places.

Post-Summit — All associated SubEnvs transfer the appropriate tokens from their input to their output places. This can lead to firing of local transitions depending on the local nets. When local firing of transitions that were triggered by the Summit transition completes, the remote SubEnv notifies the coordinating SubEnv.

Summit-Completion — The coordinating SubEnv checks if new Summits can be derived from the previous Summit, based on further connections in the coordinating SubEnv's net. If none exist, the Summit is complete.

To summarize, one way to look at a Treaty and a corresponding Summit in Petri-nets is as an "intersection" subnet which is shared by the participating local nets (although possibly with different usage privileges), whereby each local net has its own private connections to the subnet, and its own "role" in the shared subnet, in terms of sending the data that is necessary for enacting the Treaty subnet.

3.5.2.1 An Example

The following example, depicted in Figure 3.7, illustrates how Treaties and Summits can be applied in Petri-nets. This is a multi-process extension of an example which was originally given in [4] describing SLANG.

In the example, there are two processes, CODE and TEST, used by two separate groups that are responsible for coding and testing the application, respectively. In order

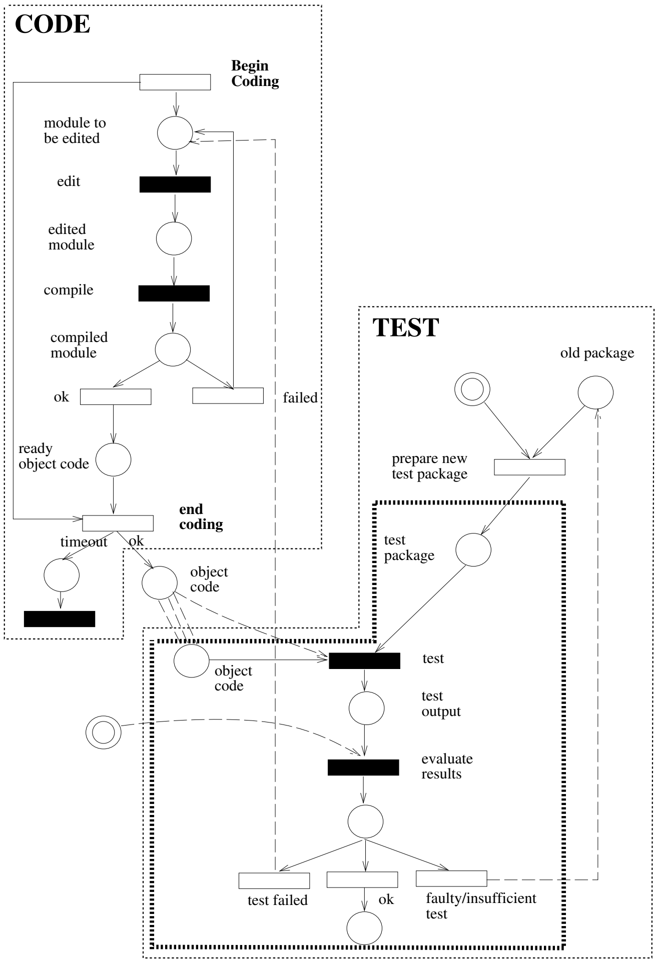


Figure 3.7: Example Multi-Process Petri-net

to increase productivity and consistency, the two teams, previously not connected in any way by their processes, decide to collaborate. The main collaborative step involves a joint evaluation of the test results by representatives from both groups that will lead to better understanding of the errors. In addition, implications of this step should provide local feedback to both groups. Finally, the necessary data transfer among the groups (e.g., object code, reports, etc.), previously done outside the process, should be modeled and handled through the inter-process modeling and binding mechanisms, respectively, thereby enabling automatic yet consistent transfer of the artifacts between the collaborating groups.

The dashed sub-process within the TEST process is then identified as the future shared sub-process. The main modifications made to that sub-process before turning it to a Treaty sub-process are in the addition of an interface input place (depicted by a circle with an inner-circle, representing in SLANG an end-user interacting with an activity) from the CODE group for purposes of the evaluation of the test results, and two new transitions with cross-process implications: (1) if the test fails, the CODE group is notified to fix the problems indicated by the test; (2) if the test is recognized as faulty, or insufficient, the TEST group is notified and modifies its package according to the recommendations made in the evaluation. Finally, the input place holding the object code is now transferred by the CODE group through the Summit mechanism, whereas before it was implicitly supplied to the TEST group. This, however, does not require a change in the sub-process, since when the Treaty is established, the object-code output place in the CODE process is merged with the corresponding input place in TEST.

Once the Treaty is established, all coding and test package preparations are still done independently and autonomously as before, but the processes synchronize for the actual testing phase when both groups are ready, as indicated by the presence of their respective tokens in the input places of the shared activities.

When the shared activities (Summit) are complete, a "fan-out" (or Post-Summit) occurs, involving passing the relevant evaluation results to each team, possibly affecting their (local) state. At a later point, when both teams are ready for a second test, a second Summit activity is initiated.

3.5.3 Grammar-Based PMLs

The grammar hierarchy [22] and the corresponding automata provide another powerful formalism for modeling a wide variety of systems, although they may have been less frequently applied to software process modeling than the other paradigms mentioned. There is a spectrum of approaches to employing grammars in process enactment, analogous to sentence generation at one end (what Heimbigner calls a prescriptive process [42]) to sentence recognition (parsing) at the other (proscriptive). The PDL project employed the former for context-free grammars [56], while the implementation of the Activity Structures Language on top of Marvel follows the latter approach [60]. One group experimented with both in the context of attribute grammars, for HFSP [61] and ObjectiveAttributeGrammars [96], respectively.

Considering the grammar-based PMLs, a *terminal symbol* corresponds to an activity in our context hierarchy, a *non-terminal symbol* to a task, and a *production* to a process step. Grammar-based PMLs usually associate some kind of condition with each production, or possibly with each symbol in a production, to specify when it could be selected. For example, in the PDL-based system these are called restriction conditions, in the Activity Structures Language they are rule skeletons, and in HFSP they are decomposition conditions. Symbols are associated with formal and actual parameters in some fashion specific to the PML and PCE.

The symbol (along with its possible condition) seems the best candidate for the unit of commonality. But it doesn't have to be a terminal symbol. This reflects the hierarchical decomposition property of grammar-based PMLs, since it essentially allows to define any sub-process as common. However, any sub-tree that can be possibly generated at execution from that symbol must be identical in both processes (otherwise it will not be common). Thus, the *import* of a symbol is necessarily recursive, i.e., when a symbol is imported, all of its possible productions are imported recursively. Of course, a cyclic import must be detected as part of the *import* procedure. As with Petri-nets, the importing site must also explicitly augment its grammar with the new symbol, and use it in its production(s).

An issue that comes up in all PMLs but is particularly eminent here is the issue of (sub)task naming. The newly imported symbol must not conflict with the name of any other local symbol, and at the same time it (and in fact all the derived symbols in a Treaty) must be identified as the common symbol when the Summit is enacted, eliminating simple local renaming as an option. The general approach recommended here, and the one actually taken in OZ (see Chapter 5) consists of separation of logical and physical names combined with unique physical name generation. This approach enables both private (logical) naming of subtasks, as well as a global name space for running Summits.

The Summit protocol works as follows (we skip the first and last phases):

1. *Pre-Summit* — This phase begins when an activity represented by a common symbol is invoked in one process with data from multiple processes. The remote SubEnvs are notified, and any prerequisites of enacting that symbol are checked in each of the participating SubEnvs, each according to their own local process. In principle, a recognition-oriented PCE might now recursively enact any symbols immediately preceding the common symbol in the current production in an attempt to fulfill the prerequisites, analogous to backward-

chaining for rule-based PCEs. This could be regarded as a form of sentence generation.

2. *Summit* — Assuming all SubEnvs ultimately agree, the symbol is enacted in the coordinating SubEnv. If, however, this is a non-terminal symbol representing composite subtask, it is "parsed" recursively, possibly involving multiple multi-site activities. This is in fact a "natural" instance of composite Summits mentioned in the generic model. This is also why non-terminal Treaty symbols are imported recursively: a common sub-task must be literally common so that all involved sites know (and trust) what exactly is taking place when their data is accessed.

3. *Post-Summit* — All the participating SubEnvs are notified by the coordinator to complete the symbol. For example, in the case of a generation-oriented PCE, each local process might automate control flow through its local production within which the symbol was embedded. Once again, the productions including a common symbol might be completely different in different local processes, and enacted independently and autonomously.

3.5.4 APPL/A

We conclude this section with an attempt to apply the decentralzied model to APPL/A [104]. APPL/A is an imperative PML that extends Ada [40] with several constructs that support modeling and execution of processes, such as persistent data types (relations), triggers for reactive control, and predicates for specification of local constraints. APPL/A takes an unusual approach to process modeling in that it models processes as actual programs that execute directly on the computer, unlike the more common approach whereby process models are interpreted by and executed on a process engine. In that sense, it is lower-level than the declarative PMLs mentioned above.

Hence, it is hard to imagine how to support the Treaty model in APPL/A. For example, it is not clear how a common sub-process could be defined and integrated with local sub-processes, nor is it clear how Treaties can be formed incrementally and over possibly pre-existing processes (short of recoding and recompiling the programs).

However, Ada (and hence APPL/A) has some built-in constructs for concurrent and distributed programming that might be used to model Summits. In particular, the Ada

Rendezvous [12] seems suitable at first glance. Figure 3.8 shows a partial implementation of the motivating example using Rendezvous. The coordinating process encodes the composite Summit as an Ada task (`Change_Summit`) and defines two synchronization points (represented as task entries) used by the local processes to signal that they have completed their work. The task entries are also the shared interfaces used for exchanging data between the tasks. The coordinator task first waits for the local tasks to complete their local reviews (the `local_review_done` entry), then if the reviews are approved, the coordinator performs local code modifications. When the body of the `accept` completes, the results are transferred back to the waiting local processes, which in turn proceed with the unit testing. When done, they invoke the `unit_test_done` entry sending the information that is needed by the coordinator; the coordinator then performs integration test, and sends the new binaries to the local processes. For simplicity, we ignore various exceptions to the process.

While powerful for certain kinds of concurrent tasks, the Rendezvous mechanism does not seem satisfactory for modeling Summits. The main problem is with the alternating "activeness" of the local and coordinating entities. That is, Summits inherently require both that the coordinator will wait (i.e., `accept`) for the local processes, as well as that the local processes will wait for the coordinator. For example, the initiation of a Summit from the coordinating site (missing form the solution, as the reader might have noticed) requires that the local processes will be waiting for a Rendezvous. However, if the local task executing the body of such an `accept` would invoke a task entry back to the coordinator, a deadlock would occur, since the coordinator is blocking on the entry. Although the `accept` blocks could be made smaller, it seems in general that Rendezvous is geared towards a single acceptor and multiple senders, as opposed to multiple tasks, each of which alternates between being a sender and an acceptor.

Another problem is that Rendezvous is "too synchronizing" for our purposes, and limits the potential concurrency. For example, when the local process completes its review, there is no reason for it to block. Instead it should be able to perform other tasks and be notified asynchronously (again, by its own accept block) when the coordinator has completed its task. Finally, another problem is support for multi-process (not only binary) Rendezvous. Here again, a solution might be to "collect" all the information from the individual binary Rendezvous and then call the "global" procedure, but the coding involved in implementing this would be complicated. In fact, this is the way the Oz process engine implements Summits. But this implementation is totally hidden from the process engineers; they use the Summit abstraction to model multi-site interoperability. Thus it is possible to realize Summits with Rendezvous, but perhaps would be easier for the process engineer if some enhancements could be made to APPL/A.

3.5.5 Summary

An interesting outcome of investigating the application of the generic model on different PMLs is that each PML illuminates different aspects of the model, due to its distinguished characteristics. For example, the explicit process structure of Petri-nets shows the Treaty as an intersection-net, and grammars bring the naming issue to the forefront and also emphasize composite Summits due to their decomposition nature. Moreover, actual implementations of the model to different PMLs are likely to raise new requirements and subsequently modifications to the model. Yet, overall it seems that having a language-independent model enables on one hand to investigate problems inherent to process-interoperability regardless of the PML used to define it, and on the other hand it makes it potentially applicable to a wide range of formalisms. The particular model presented in this chapter seems to be capable of adjusting itself to the different concerns, abstractions, and granularities of the different PMLs.

3.6 Groupware Tools and Delegation in Summits

The model presented so far in this chapter lacks consideration of two important generic issues. The first has to do with integrating groupware technology, mainly integration of synchronous multi-user tools (e.g., multi-user editors) that enable multiple human users, possibly physically dispersed, to collaborate (support for enactment of asynchronous multi-user tools in PCEs has been investigated in [108]). While in some aspects the introduction of this technology obviously implies additional architectural support from the PCE (for example, connecting multiple human users from multiple sites to the same conceptual activity), the extension of this aspect to our model seems both natural and simple. Recall that the Summit phase is always executed in the coordinating site, while the other sites behave "as if" the activity executed at their site, in order to carry out any implications of that activity locally. Integrating multi-user tools simply implies that when such tools are invoked as part of a Summit, *all* sites (not just the coordinating site) actually execute the Summit activity, instead of just "pretending" to execute it, so the Summit phase still exe-

```
task Change_Summit is
    entry local_review_done(sources: in  files; in_reviews: in  docs;
                            out_results: out res);

    entry    unit_test_done(sources: in files; in_results: in res;
                            out_results: out res; bins: out binaries)
end Summit;

task body Change_Summit is
  begin
    loop
      accept local_review_done(sources: in files; in_reviews: in  docs;
                               out_results: out  res) do
          if (in_reviews = OK) then
                out_results =  modify_code(sources);
          end if;
      end local_review_done;

      accept  unit_test_done(sources: in files; in_results: in res;
                        out_results: out res; bins: out binaries) do
         if(in_results = OK) then
            out_results = integration_test(sources, bins);
         end if;
      end unit_test_done;

     end loop;
  end Summit;

task type local_proc;

task body local_proc is

   do_local_review(doc);
   local_review_done(source_files, review_results, global_results);
   if (global_results = OK) then
     do_local_test(files, result);
     unit_test_done(files, local_results, global_results, binaries);
   end if;
   if(global_results = OK) then
    install_new_binaries(global)
   end if;
end local_proc;
```

Figure 3.8: Summits in Ada

cutes in one point in time, but not necessarily in one point in space. Consider, for example, Petri-nets. Whereas in the standard model the remote sites would normally fire the transition without actually carrying out the associated activity, in the case of multi-user tools the transitions would really fire the activities, except they might involve binding new users, one per site, and thus must be defined (either as "instances" or as "roles", analogous to the distinction between classes and instances in object-oriented programming) somewhere in the body of the activity or elsewhere in the Summit.

A second major aspect, which is related to but separate from groupware, is the issue of delegation. As soon as multiple users might be involved in the execution of related activities, the notion of delegating a task to a specific user (or a role instantiated by a human user) comes to the surface. Delegation can occur in our model in two places: within the Summit phase in a multi-user tool, as described above, and in either pre or post Summit phases. For example, a pre-Summit phase might involve firing an activity at a remote site by a user at that site, as opposed to the user that initiated the Summit, or in case of a non-interactive tool it could be fired "unmanned". The latter is considerably more efficient if the SubEnvs are physically distributed, since the execution occurs where the data resides. And the former might be more appropriate in many cases that involve interactive tools that have to be performed by specific remote users (see Section 4.5).

Support for delegation, while extremely important for effective execution of Summits, is largely a PML/PCE-specific issue, since it requires language extensions to enable binding of activities to human roles, or providing a "user-context" from which potential users can be selected for interactive execution of certain activities. Once such support exists, it can be extended to support multi-site activities. The main necessary extension is to enable binding of remote users to activities, analogous to the mechanisms which are necessary to enable binding of remote data to Summit activities. A particular approach to supporting delegation in the Oz framework is given in Section 4.5.1.

Finally, note that support for both delegation and multi-user tools is in some sense orthogonal to SubEnv-interoperability, in that the same functionality could be used by multiple users of the same SubEnv. It is only more evident in the case of multiple SubEnvs. Indeed, the implementation of delegation in Oz supports both intra- and inter-process delegation and activation of multi-user tools.

3.7 Extensions and Alternatives to the Summit Model

There are several directions in which the Summit algorithm could be altered and/or extended. Some of the changes actually divert from the basic model and are not compatible with it, while others could be augmented to the basic model with varying degrees of implementation efforts. We consider here only the conceptual changes to the model, and defer implementation issues to the next chapter.

3.7.1 Summit Branching Policy

The first alternative to consider is related to the execution order of local vs. global activities in composite Summits. Recall that in the Summit algorithm a Summit activity is followed by fan-out to local sites, and only when the fan-out completes, another Summit activity is enacted, and so forth. An alternative approach would have been to execute all related Summit activities consecutively, preceded and proceeded by local operations (This, in fact, was the initial composite-Summit model as presented in [16]). That is, when a summit activity completes, the coordinating SubEnv enacts (recursively) any further Summit activities emanating from the previous one, and fan-out begins only when the global execution completes. The main advantage of the former (our Summit approach) is in the fact that it subsumes the functionality of the latter.

A related problem at the implementation level has to do with carrying out the branching policy discussed above. The coordinating SubEnv has to be able to distinguish between Summit and local activities, and apply the right order of execution as described above. This problem occurs particularly in cases where the choice of activities to run is not explicit in the PML code but rather inferred, such as in rule-based and grammar-based PMLs, where the branching occurs dynamically and is not known a priori. This issue is addressed in Section 4.4.3.4.

3.7.2 Local Derivation of Summits

The basic Summit algorithm restricts Summit activities to occur only at the coordinating site. Of course, different Summits can occur at different sites, but once a Summit begins at the coordinating site, all subsequent Summit activities in the same composite Summit must occur at the coordinating site. A reasonable extension to the model would be to allow derivation of Summit activities off local executions. Consider, for example, the case where an initial Summit between sites E_1, E_2, and E_3 leads to another Summit between, say, E_3 and E_4, where E_4 is a sub-contractor of E_3 but is not concerned with the previous Summit. While this seems like a natural extension, there are several difficulties with it, both conceptual and technical.

Conceptually, the main problem is that there is no way to define clear boundaries for a Summit and identify each activity with its associated Summit. Thus, the notions of a Summit context and a coordinating site become blurry and ill-defined.

Technically, there are numerous problems to be resolved. The main one is concerned with concurrency: multiple "sub-Summits" can lead to circular dependency which might lead to a deadlock, as will be explained in Section 5.4. Another problem is if atomicity semantics were desired for some kinds of composite Summits (as is the case in Oz, see Section 4.4.5), maintaining atomicity across multiple coordinating SubEnvs requires complicated transaction facilities. Finally, the issue of dynamic binding comes up. The problem is how can a site executing in "local mode" derive a Summit with other sites. One way is to "inherit" the context of the parent coordinating site and use it, but this is still limited to forming Summits off sites that participated in previous Summits.

One restriction to the local derivation mechanism could make it more feasible: limit local derivation of Summits to involve new sites which did not participate in prior Summits, thereby forming acyclic hierarchical Summit structure that alleviates the concurrency problem, although it worsens the binding problem (a possible solution to the latter in Oz is discussed in Section 4.2).

To summarize, while local derivation of Summits might be an attractive extension, there are several implications to implementing it, and they have to be weighed against the potential gained benefits before attempting any implementation.

3.7.3 Multiple Global Environments

Another extension to consider is interaction between multiple (global) environments. One particularly attractive option would be to have a SubEnv that belongs to several environments, and executes in the context of one of the environments. Or it could even allow simultaneous execution with multiple environments (by the same or different users). The former approach is compatible with the general model since SubEnvs are self-contained and loosely coupled with each other. It might even have different Treaties with different

SubEnvs in the different environments. The main architectural requirements are (1) to support global name space for global environments and globally unique SubEnv identification scheme, and (2) to have a mechanism that enables to dynamically bind (and unbind) a SubEnv to a particular global environment. The more ambitious approach to allow simultaneous interaction with multiple environments imposes several architectural problems, such as addressing and selecting objects in different SubEnvs, but is also in principle compatible with the general model, so long as the SubEnv has one instantiated process.

4

Realization of the Decentralized Model in Oz

In this chapter we explore issues that are concerned with the realization of the decentralized model in a real PCE. Although most of the concepts introduced here have been fully implemented[1], the discussion in this chapter will try not to focus on the actual implementation. Thus, while detailed enough for understanding the wide range of issues concerned with such realization, the discussion will attempt to present the problems, ideas, and techniques that could be relevant to an implementation of the model in other PMLs/PCEs in general, and rule-based PMLs/PCEs in particular.

Before launching into the realization of the decentralized multi-process model, we begin with an overview of the concepts that were "inherited" from the single-process Marvel PCE, so the reader should be familiar by now with the main concepts and functionalities of Marvel as presented in Section 2.2.

Oz is a multi-process PCE as defined in Section 3.1, supporting modeling and enactment of autonomous multiple sub-environments. As such, Oz subsumes the definition of a single-process PCE. In particular, an Oz environment consisting of a single SubEnv behaves similarly to a Marvel environment (albeit an enhanced one, because of improvements that were made in Oz independent of support for multiple processes).

---

[1] The unimplemented features are summarized in Section 4.6.

As in Marvel, each *local* (sub)environment in Oz is tailored by a local administrator who provides the *data model*, *process model*, *tool envelopes*, and *coordination model* for its team. These definitions are translated into an internal format and then *loaded* into the environment by a special loader component.

The data modeling capabilities of Oz are identical to those of Marvel — object-oriented data definition including classification, multiple inheritance, composition hierarchy, horizontal bi-directional links, and file and state attributes. Similarly, the process modeling language of Oz is based on the Marvel Strategy Language (MSL), with some minor extensions (which will be pointed out as their functionalities are discussed). Most importantly, Oz extends the user-driven, rule-based paradigm to multi-process environments. Specifically, as far as local processes are concerned, Oz processes are defined in terms of Marvel-like rules, which correspond to the notion of process-steps in the generic context hierarchy. Process tasks are implicitly defined by matchings between effects and conditions of rules. The constraints defined in the process (in terms of rule conditions) are enforced, and automatic enactment is supported through backward and forward chaining. Finally, the single-site transactional semantics of Oz are like those of Marvel, supporting *atomicity* and *automation* chaining (more details are given in Section 4.4.5).

4.1 Operational Overview of Oz

In general, Oz has a two-level architecture: within a SubEnv, it has a client-server architecture, with multiple clients communicating with a single centralized process-server. Across SubEnvs, Oz has a "share-nothing" architecture, as advocated in the formal model. This means that the processes, schemas, and instantiated objectbases are kept separately and disjointly in each SubEnv, and that there is no global repository or "shared memory" of any sort (details of the architecture will be given in Chapter 5).

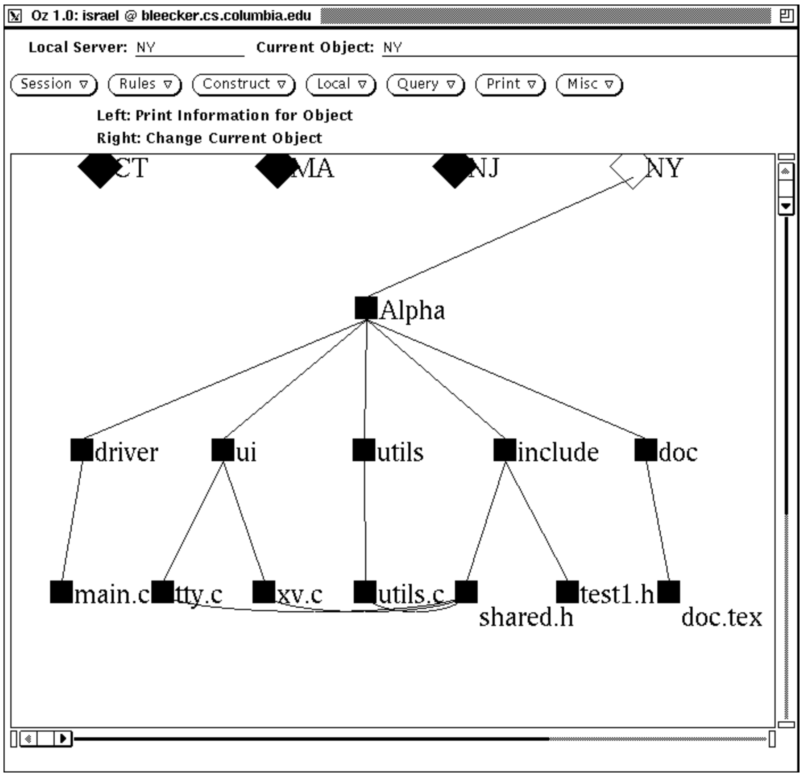
Human interaction with the environment is provided through a *client* that is connected primarily to its local *server*. Using the client's connection to its local server, users can operate with the local tools, on local data objects, and under the local process, much like in Marvel. In addition to the local server, however, Oz users can connect to remote servers. Each remote SubEnv is represented in each local objectbase by a "stub" object that is visible to the client. By issuing the built-in `open-remote` (`close-remote`) command with the appropriate stub object as parameter, a client can open (close) a connection to a remote SubEnv (again, implementation details are deferred until the next chapter). A remote connection provides limited access to the remote SubEnv. A remote client can browse through remote objectbases and get information about remote objects (subject to access control permissions). However, a client has no access to remote processes (i.e., rules, tools) and access to remote data can be made only by binding remote objects as parameters to Treaty rules (their realization is the subject of Section 4.3.1).

For example, figure 4.1 shows how the client for user `israel` (the user's name is shown in the upper left corner of the interface window) is connected to the local server of SubEnv `NY`, with a (default) view of the local objectbase[2] (parent-child relationships are depicted with straight lines and links by curved lines). Figure 4.2 shows `israel`'s view after an `open-remote` on site `CT` has been made, making `CT`'s remote objectbase available for browsing by `israel`. `israel`'s client has not connected to SubEnvs `MA` and `NJ`, and they may, or may not, be currently active (i.e., executing). `israel` interacts with the environment by selecting commands from the `rules` menu, which contains all the process-specific user-level commands (inference rules used internally do not appear in the menu), and he supplies arguments to the rules by clicking on objects from the objectbase. In particular, if a remote objectbase is open, he can initiate a Summit by selecting remote objects as arguments to Treaty rules. When the (local) server services the request to fire a rule, it checks its own process, and communicates with remote SubEnvs if the rule accesses remote data from their objectbases, and eventually determines whether an activity has to be executed. That activity could be either the one explicitly requested by the user, or another activity related to the requested one through a chained rule. The server then sends a message to the requesting client to execute the activity in its activity-manager component (except in cases of delegation, where an activity can be sent to a delegated client, see Section 4.5). During a Summit activity, remote objects are temporarily copied to the local SubEnv and passed to the client prior to the activity execution. Note that since a client has no explicit access to remote processes, it cannot invoke "remote Summits", thus all Summits are initiated by local clients.

A special *administrator* client has, in addition to the normal client functionality, an interface for updating the actual process definition. An administrator client can evolve the process by adding, removing, or loading a whole new or revised set of rules into the current

---

[2] For simplicity, only a small objectbase is shown, but in reality Oz can maintain thousands of objects with adequate browsing support.



Oz 1.0: israel @ bleecker.cs.columbia.edu

Local Server: NY

Current Object: NY

Left: Print Information for Object

Right: Change Current Object

Figure 4.1: An Oz Environment

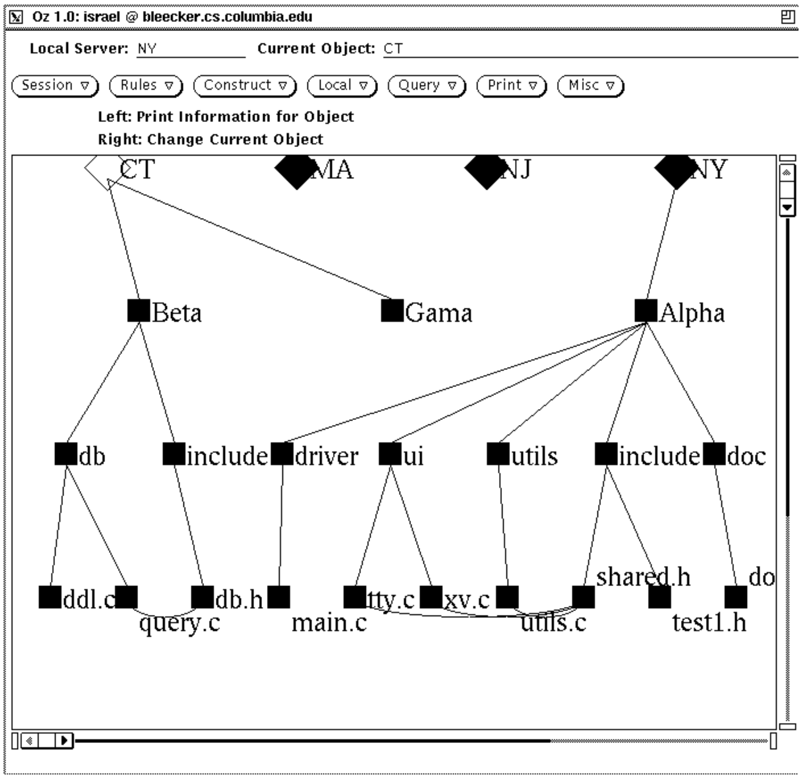
SubEnv. The interface to the `load` command is shown in figure 4.3, where the `analyze` strategy is about to be added to the local process. The administrator can also optionally specify a process configuration file that contains a list of strategies to load (The notion of an Oz strategy and its contents are explained later in Section 4.3.1).

4.2 Oz Objectbase

Whereas an instantiated objectbase in Marvel is a forest data structure, a local objectbase in Oz is a rooted tree with a special `SubEnv` root object that contains information pertaining to the local SubEnv (the details are given in Chapter 5). Thus, a typical forest-like Marvel objectbase is mapped to an Oz objectbase by connecting all the forest's roots as children of the `SubEnv` object[3].

---

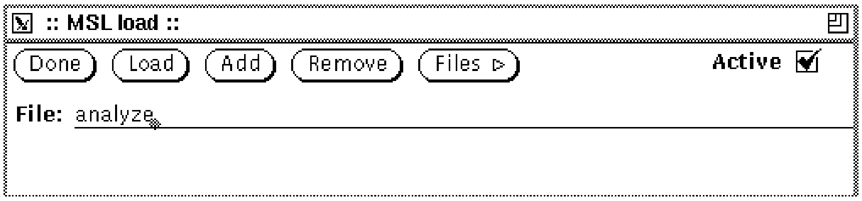
[3] This provides an easy migration path from Marvel to Oz objectbases. Marvel objectbases can be migrated to Oz local objectbases with a simple upgrade facility that essentially extends the original schema with the addition of the Oz built-in class that defines SubEnv objects, and connects all roots of the forest to the SubEnv object.



Local Server: NY
Current Object: CT

Left: Print Information for Object
Right: Change Current Object

Figure 4.2: Oz Environment with one open remote site



Active ☑

File: analyze

Figure 4.3: Load Interface in Oz

Since each local objectbase is maintained by a different Oz server, a "global" objectbase is merely the union of all local disjoint objectbases, although the schema may vary from site to site. This implies that composite objects cannot be partitioned across different objectbases, since this would violate the disjointness property, because a composite object contains its sub-objects. For reference links, the situation is different. Links could conceptually cross an objectbase boundary since they do not impose a containment relationship.

4.2.1 Cross-Site Links

There are tradeoffs in supporting cross-site links. The main advantage with having them is that they provide cross-site data modeling capabilities. Another advantage is that they enable remote objects to be bound as parameters to multi-process rules in an on-going chain. This seemingly obscure property of cross-site links stems from the fact that automatic (as opposed to user-invoked) derivation of the parameters of rules during chaining is based on their structural relationships to parameters in previously executed rules (for details of this "inversion" algorithm, see [50]). Thus, since cross-site links are the only way to structurally relate the otherwise disjoint objectbases, the lack of such a construct would eliminate the possibility of automatically deriving parameters from remote SubEnvs, unless those SubEnvs have already participated in earlier Summits in the on-going chain.

If however, cross-site links are allowed, there are several conceptual and technical problems:

1. The main conceptual problem is that a cross-site link *permanently* connects two local objectbases. With the navigational querying capabilities of MSL, cross-site links would allow a query to traverse an entire remote objectbases through a remote link. This might violate both the autonomy and independent operation requirements. While autonomy might be relaxed in some cases in favor of close cooperation purposes, independent operation should not be compromised. For example, if a site with remote links from/to other sites is "down" or just unreachable, then any rule that uses links to the disconnected objectbase will either fail or produce different results depending on which sites are reachable.

2. With cross-site links, it is no longer clear to users (as well as high-level modules in the system) what the origins of the involved data are, and how expensive it is to fetch, bind and execute an activity. This violates the non-transparency property.

3. Cross-site links create cross-site dependencies that might lead to communication deadlocks if not handled carefully. The gist of the problem is that when a remote query is requested from site `A` to site `B` (e.g., in the binding phase of a rule), site `B` might not be able to service the query without consulting other

servers that are connected through cross-site links, possibly creating a circular dependency. The general issue of communication deadlocks is covered separately in Section 5.4.

4. Since links in Oz are typed, cross-site links require implicit specification of common sub-schema. Thus, a remote object which is linked to a local object must correspond to the local schema type.
5. Implementation of cross-site links is both hard and expensive because these links would be "virtual", i.e., two objects linked by such a link do not share an (operating system) process address space or file-system space. They would have to be implemented by "stubs" at both ends containing information that allows queries to "follow" the link, and a corresponding protocol between the servers that enables efficient access to those remote objects.

To summarize, cross-site links could be viewed as an implicit, "data-oriented" approach to enabling access to remote data. Instead, the approach taken in this thesis favors explicit specification of the remote data to be accessed (through rule parameters), and in a more "process-oriented" fashion. Thus, regular cross-site links cannot be supported in Oz.

However, preliminary experience with using Oz revealed that for some situations, not having any means to model inter-site data modeling led to an unintuitive and awkward modeling of inter-process modeling, particularly in the cases where the sites were more tightly coupled. This led to the contemplation of "soft" links[4].

4.2.1.1 Soft Links

The main ideas in soft links are: (1) to distinguish cross site links from regular links, not only in the data definition language (different attribute types), but also in the process modeling language; and (2) to treat the invocation of rules with soft links as Summit rules, which implies that they must be "Treatified" before their use.

This design addresses most of the problems which were raised above: (1) queries would not cross sites unless they specifically contain soft links in their definition, and site autonomy would be preserved by the fact that only Treaty rules can use soft links, and they are regarded as Summit rules. (2) Non-transparency would still be preserved, since

---

[4] Borrowed from the Unix terminology for symbolic links.

the process must explicitly state when it accesses remote data, and normal links will only refer to local data. And any satisfactory solution to the sub-schema (3) and deadlock (4) problems for regular Treaties and Summits would also cover these soft link Treaties (these solutions are presented in Sections 4.3.3 and 5.4, respectively). Problem 5 is still valid, though, and is the main reason for not having at the time of this writing an implemented version of soft links.

4.2.1.2 Remote Derivation Without Soft Links

Even if cross-site links do exist, there is still a need for a complementary mechanism that addresses the automatic remote derivation problem mentioned earlier without using them (e.g., because of data privacy concerns). Recall the "sub-contractor" example given in Section 3.7.2; with no cross-site links, there is no way that the sub-contractor site could be (automatically) derived as a candidate for a Summit with the sub-contracting site, because it was not part of the previous Summit. The proposed solution extends the parameter inversion algorithm to allow logical inversions off associative (i.e., non-structural) queries. This capability, in conjunction with *global* associative queries, would enable one to model the example in the following manner: in the sub-contractor site E_4, a special attribute of the relevant object is set to denote that it is a sub-contractor of E_3. Then, upon completion of the first Summit, E_3 issues a global query that locates the sub-contracting object in E_4 and binds it to E_3's rule, thereby providing the necessary context to subsequently fire a Summit with E_4 as desired. Once again, this approach might be preferred over using cross-site links in cases where a "permanent" connection between the sites is not desired. Furthermore, the definition is more "process-oriented" as it is defined explicitly in the process and not "hidden" in the data modeling level.

4.3 Modeling Process Interoperability in Oz

As discussed in Section 3.2.1, modeling interoperability involves the following aspects:

1. Definition and evolution of common sub-processes.
2. Definition and evolution of common sub-schemas.
3. Definition of data instances to be used by common sub-processes.

Given the core requirements of autonomy, independent operation, and flexibility, effective modeling of interoperability imposes guidelines and constraints on how to implement modeling facilities for the above aspects, among them independent and dynamic evolutions of local processes, as well as non-global definition.

In the rest of this section we examine each aspect in detail. The first aspect, discussed at length in Sections 4.3.1 and 4.3.2, is an effective implementation of the Treaty protocol as outlined in the previous chapter. The second and third aspects, discussed in Sections 4.3.3 and 4.3.4, respectively, are PCE and DDL specific, and are presented here in a somewhat abbreviated form.

4.3.1 Defining Common Sub-Processes: the Treaty

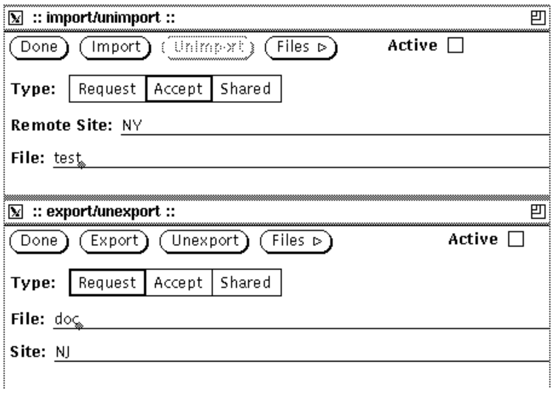
In general, Treaties are realized in Oz following the formal model outlined in Chapter 3. Treaties are formed by active participation of both sites of the Treaty, with one site being the requester and the other site the acceptor, as well as one importer and one exporter (all four combinations are possible).

The basic unit of commonality in Oz is the *rule*. However, mainly due to granularity issues, the unit that is exported and imported is the *strategy*. A strategy is a bundling construct for rules, somewhat analogous to a module consisting of functions in modular programming languages. Although the intent of a strategy is to bundle rules that are conceptually related — either by their functionality or by belonging to the same task— there is no enforcement of this policy. In particular, there are no chaining restrictions between rules in the same or in different strategies[5].

A strategy consists of two sections: a `tools` section that declares tools and defines the interface to them (from rules), and a `rules` section in which rules are defined. In addition, the schema required by the rules is provided by special *data* definition strategies defined separately. The separation between schema and process definitions (which did not exist in Marvel) has several benefits: (1) it facilitates the realization of Treaties that are formed among rule strategies only; (2) it eases local evolution, avoiding the need to perform schema evolution if only rule strategies are modified; and (3) it increases componentization by separating data and process modeling, thereby allowing the potential to use different data definition languages (and different OMSs) with the same rule language, and vice versa.

---

[5] An early Marvel paper [59] described a different approach with respect to restrictions and chaining across strategies.



Type: Accept

Remote Site: NY

File: test

Type: Request

File: doc

Site: NJ

Figure 4.4: Import and Export Interfaces in Oz

Oz provides the following five built-in commands for establishing Treaties: *export*, *import*, *unexport*, *unimport*, and *treaty*. Although there are no separate commands for *request*, *accept*, *deny* and *cancel*, they are specified as parameters to each of the above commands, making it possible to generate all possible combinations that were discussed in the formal model.

4.3.1.1 export

The export operation is defined as:

$$export(strategy(SrcSubEnv), DstSubEnv, [privileges])$$

This is an inexpensive operation in Oz. It executes locally at $SrcSubEnv$ and merely involves adding an entry with the specified *strategy* and $DstSubEnv$ to a persistent local export table. By default, Oz associates *request* privileges with *export*, i.e., it assumes that in most cases the the exporter wants to use the exported strategy on data from $DstSubEnv$. But the administrator can change the default by explicitly selecting *accept* privileges. In addition to *accept* and *request*, Oz provides a third option called *shared*. The semantics of the *shared* option are to export a strategy both as a requester and as an acceptor. The main use of this option is to facilitate convenient generation of full (i.e., bi-directional) Treaties: a *shared export* followed by the proper *shared import* establishes a full Treaty. The actual Oz interface is shown in the lower window in figure 4.4, where strategy `doc` is about to be exported to SubEnv `NJ` as a requester.

4.3.1.2 import

The import operation is defined as:

$$import(strategy(SrcSubEnv), DstSubEnv, [privileges])$$

import is the main operation in modeling Treaties, and is quite complicated both in terms of user interface support and the internal implementation. As usual, we assume the existence of the necessary underlying infrastructure to communicate with the remote SubEnv (which is explained in detail in Chapter 5). In particular, there must be an open connection to *SrcSubEnv*, since the operation is initiated at *DstSubEnv* but it involves both SubEnvs.

The actual interface is shown in the upper window in figure 4.4. It shows that strategy `test` is to be imported from SubEnv `NY` as an acceptor, i.e., it will allow rules defined in the `test` strategy to be fired from users at remote site `NY` on local data (which happens to be site `CT`). As with *export*, it is possible to associate either of the three privileges with *import*, with *accept* being the default.

1. The first issue to consider is how to select the strategy to import, given that it resides in a remote SubEnv and is thus not usually visible to the local process. The *import* interface must supply the administrator at *DstSubEnv* a list of the available strategies at *SrcSubEnv* that were explicitly exported from it to *DstSubEnv*. Further, this information must be generated dynamically, since the list of exported strategies at *SrcSubEnv* can change at any time as a result of issuing local *export* or *unexport* operations.

2. Once the importer at *DstSubEnv* selects the strategy to import, it has to be copied from *SrcSubEnv*, along with additional information needed for runtime verification (see Section 4.3.2). This is done in Oz by fetching a copy of the strategy, which is held for the duration of *import* and is removed immediately afterwards. (A physical copy of the strategy is required only when the SubEnvs do not share a file system. Otherwise, only a path to the strategy is sent, and the process translator at the importing site fetches the file from its original location.) There are two reasons for not keeping a local copy of the imported strategy: (1) only one physical copy of the strategy "source code" is maintained (and all importing SubEnvs point to that copy), thereby avoiding the need to keep multiple copies consistent; (2) this approach also facilitates dynamic verification of Treaties that might be violated by local evolutions, as will be explained shortly.

Note that *import* fetches only the rules, without the envelopes (and tools called from them) associated with the rules. While this is not a problem with the default *import-accept* option (since in this case the activity is not executed at the importing SubEnv, only its data is accessed by the activity, which executes at another SubEnv), the *import-request* combination implicitly assumes that the activity exists at the importing SubEnv, so if this is not the case it must be copied outside the environment (e.g., `cp`, or `ftp` across domains).

3. The third step is to verify that the strategy can be integrated with the local process at $DstSubEnv$, both syntactically and semantically. This includes sub-schema compatibility (discussed in Section 4.3.3) and process-consistency (discussed in Section 4.3.5).

4. The fourth step involves connecting the rules in the imported strategy to the local rule-network. This is done by *forward* connecting each new rule to all other rules (both imported and local) whose conditions match the rule's effect, and *backward* connecting it to all rules whose effects matches the rule's condition[6]. At the end of this procedure, the imported strategies are fully integrated with the local process. When executed as part of a Summit, local prerequisites and consequences (in addition to "global" Summit implications) of the imported rules would be automatically enacted.

Figure 4.5 illustrates the integration phase (using rules that correspond to some of the steps in the motivating example). Suppose the `modify` rule is imported by two different processes residing at `SiteA` and `SiteB`, respectively. In `siteA`, `modify` is backward connected to rule `review` through the matching between `modify`'s condition and `review`'s effects, and it is forward connected to rule `manual_test` through the matching between `modify`'s effects and `manual_test`'s condition. Similarly, in `siteB`, the rule `modify` is backward connected to `analyze` and forward connected to `auto_test`. Thus, `modify` becomes an integral part of both processes, and may trigger, or be triggered by, invocation of related rules during enactment.

The ease with which process integration can be achieved reveals the strength of the declarative nature of the rule paradigm: process fragments can be incrementally added (or incrementally removed) and automatically *integrated without user intervention. The contextless rules, as well as the fine granularity of rules as process building blocks, also pay off handsomely.*

Since *accept/request* operations can only be invoked by *import/export*, it is neces-

---

[6] Actually it is more complicated because chaining directives filter some of the connections.

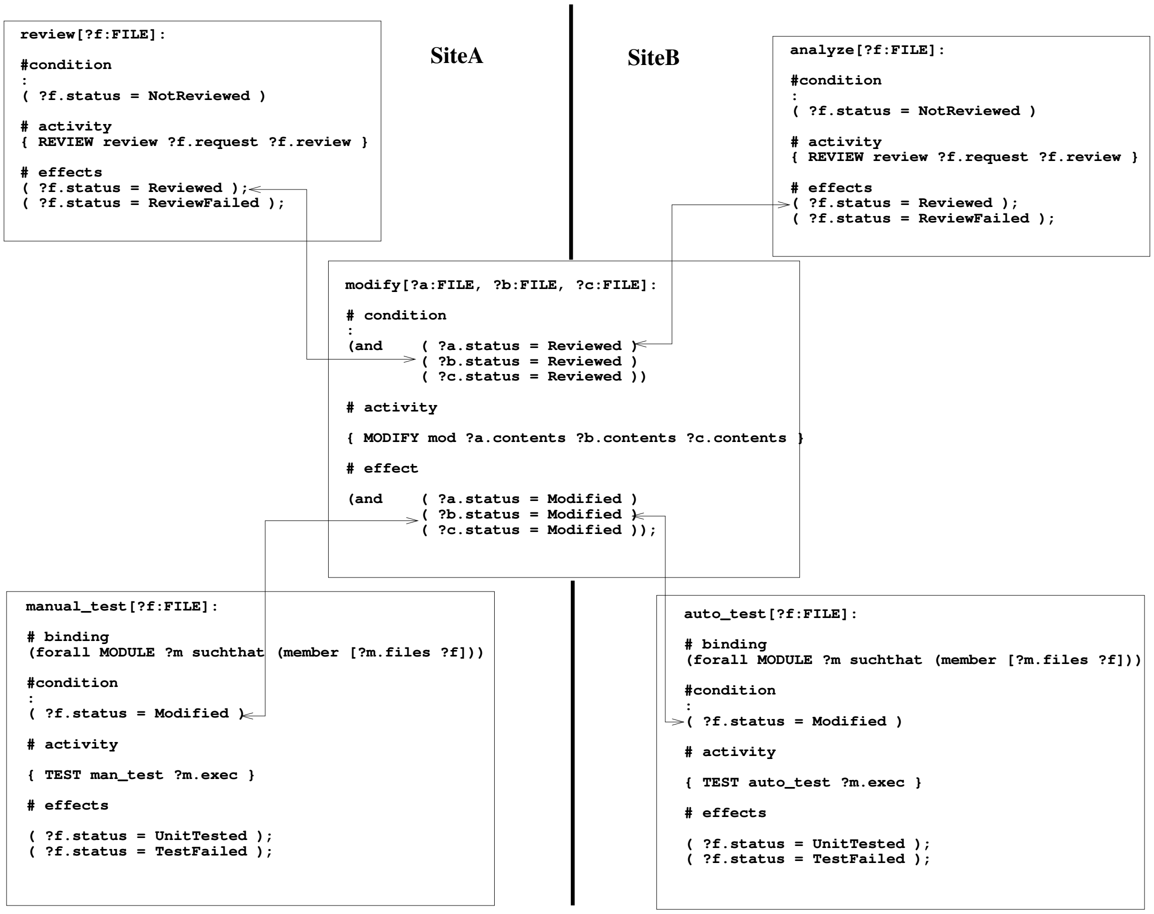


Figure 4.5: Integration of Imported Rules

sary for *import* to be idempotent with respect to the compilation mentioned above. This is particularly important for Treaties that involve more than two sites. For example, suppose site E_1 imports strategy S_2 from site E_2 and site E_3 also imported S_2 from E_2. Now site E_1 wants to grant *accept* privileges to E_3, so it issues an *import-accept* command, but this time compilation of the process model is not necessary so only the execution-privileges flag is modified. When an *import* is requested on an already imported strategy (or alternatively, if it is a local strategy which was exported and is now imported, possibly to form a full Treaty), only the process privileges are updated, and the compilation part is ignored. We will refer to such *import* operation as a "faked" import.

5. The last step of *import* involves sending an acknowledgement to *SrcSubEnv*. This acknowledgement is not critical, however, since runtime checks are performed anyways

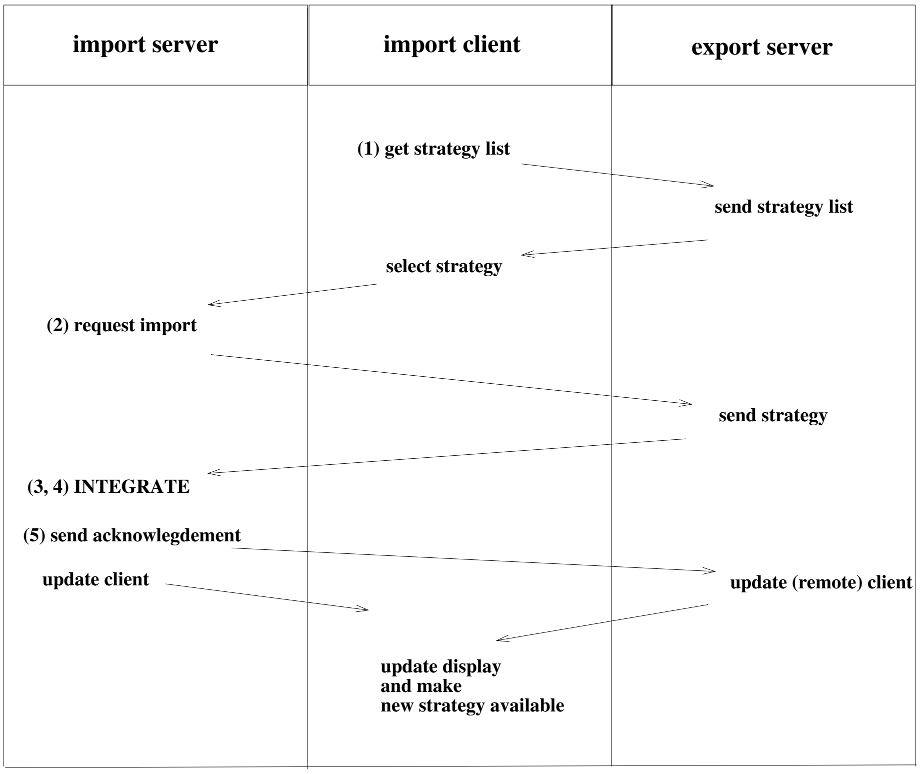


Figure 4.6: The Import Algorithm in Oz

to verify the validity of Treaties. Its sole purpose is to notify users at $SrcSubEnv$ of those Treaties that are possibly available to them.

Figure 4.6 illustrates the *import* algorithm and the interaction between the SubEnvs at the execution of the *import*, with numbers corresponding to the steps listed above. Notice that menu generation in the first step is done by direct communication between the remote client and the export server. Once the Oz client selects which strategy to import, it requests its local server to perform the *import*. From then on, the two servers communicate in a client-server fashion — with the import server acting as a “client” and the export server acting as a “server” — until the service is completed, in which case both servers update the client: the import server sends to the client the revised set of rules and rule-network, and the export server notifies the client that a Treaty from the import server to the export server was formed.

It is worth mentioning here a problem that manifests itself in this implementation of *import*, but is only a private case of the more general problem regarding server-to-server communication. Basically, the problem with this client-server-like interaction between the import server and the export server is that as a "client", the import server needs to *wait* for the service to be provided. This means that if the export server is single threaded and non-context switchable at step (2), then the import server might block indefinitely, potentially "starving" its own clients, or even worse, deadlocking with other waiting servers acting as clients. In the specific case of *import*, since it involves changing the local process, the import server must execute in single-user mode. This alleviates the import problem since there are no potential starving clients, but it doesn't solve the deadlock problem. And in the general case multiple clients interact concurrently with a server, adding the starvation problem as well. This problem is addressed separately in Section 5.4.

There are two more properties that the *import* operation must have. One is *atomicity*; clearly, the *import* operation has several potential failure points, meaning that it must be accompanied by a context-sensitive rollback mechanism that preserves the integrity of the server in case of failures. However, there is no need to guarantee cross-site atomicity for *import*, which fits well with the general decentralized requirements; the acknowledgement is optional, as mentioned above. The atomicity of the operation has to be preserved only in the importing server.

The second property is *persistence*. The imported strategy, along with the necessary information used for runtime validation, must be stored permanently with the local process since it outlives an execution of the server, and needs to be reloaded in subsequent evolutions.

4.3.1.3 unexport

The unexport operation is defined as:

$$unexport(strategy(SrcSubEnv), DstSubEnv, [privileges])$$

Like *export*, this is a local operation that executes at $SrcSubEnv$. It removes $DstSubEnv$ from the list of SubEnvs that are entitled to further import *strategy*. In addition, the execution privileges are undone based on the specified *privileges* argument — when coupled with *accept* the effect is *deny*, coupled with *request* results in *cancel*, and coupled with *shared* revokes both. Note that if, for example, the exported strategy was previously *shared* (i.e., both requested and accepted), then unexporting with *request* (*accept*) retains the *accept* (*request*) privileges intact.

4.3.1.4 unimport

The unimport operation is defined as:

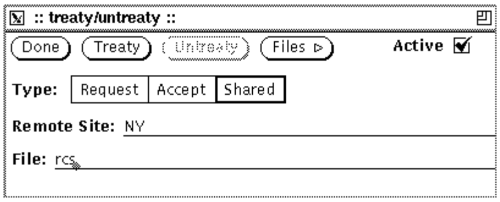
$$unimport(strategy(SrcSubEnv), DstSubEnv, [privileges])$$

Unlike its *import* counterpart, *unimport* is a local operation, as it should be according to the formal model. However, unlike *unexport*, it might involve some non-trivial amount of work at the server. The algorithm is as follows: if *strategy* is marked as imported from more than one SubEnvs, or if *strategy* is a local strategy (which was faked imported for full Treaty purposes), then *unimport* does not modify the process, and only updates the privileges similar to the way it is done in *unexport*. If, however, $DstSubEnv$ is the only strategy from which *strategy* is marked as imported, then *unimport* removes *strategy* from $SrcSubEnv$'s process. This requires "decremental" recompilation and regeneration of the rule network. Such a physical *unimport* also revokes all privileges from all remote SubEnvs regardless of the parameters that were specified with the operation, since the strategy is removed from $SrcSubEnv$ and cannot be used in any manner there.

As can be seen, not having the four execution-privileges commands (*request*, *accept*, *cancel*, and *deny*) available separately from the four strategy-transfer commands (*export*, *unexport*, *import*, *unimport*) introduces some technical and conceptual difficulties. On the other hand, preliminary experiments showed that easing the procedure of forming Treaties is pragmatically important, and that most of the Treaties can be formed using the default privileges, while more proficient administrators can still select other options in order to get the desired behavior. In any case, this is mainly a user-interface issue; the important issue is that the equivalent semantics of the formal model are fully obtainable in OZ.

4.3.1.5 Operational Overview of Forming Treaties

Going back to the formal model, a simple binary Treaty between two SubEnvs is formed by an *export* operation at the source SubEnv, followed by a matching *import* operation at the target SubEnv. But these operations do not have to be synchronized, and in particular, the *import* can occur at anytime after the *export*, or never occur at all. From the system's standpoint, Treaties are formed *implicitly*, and perhaps even without explicit intention. That is, Treaties can be inferred automatically, when the right combination of *export* and *import* occurs at the SubEnvs. In some sense, this is a continuation of the



Active ☑

Type: Shared

Remote Site: NY

File: rcs

Figure 4.7: The Treaty Interface in Oz

context-less rule-based model that fits well with autonomy concerns. In particular, there is no need for a "global administrator" to form Treaties; they are formed by local administrators willing to collaborate in order to form the Treaties, and using the system to formalize their intentions as well as ensure that they are carried out as agreed.

In cases where SubEnvs are more tightly coupled, however, there might be a need to support (simple and full) Treaties as one operation, to simplify their formation. Indeed, early experience with Oz revealed the need for such an operation in cases where, for example, each SubEnv represented a single-user process, in which case a global administrator (and a corresponding global Treaty operation) was essential. Therefore, Oz supports the explicit $Treaty$ operation, which bundles an *export* and *import*, as explained below.

4.3.1.6 Treaty as one operation

In order to be eligible for executing the Treaty operation, a user has to have administrator privileges on both SubEnvs. Note, however, that in conformance with the "not-only-local-or-global" principle, the user does not need universal administrator privileges, only on the two sites of a given Treaty.

The treaty operation is defined as:

$$Treaty(strategy(SrcSubEnv), DstSubEnv, privileges)$$

and the actual Oz interface is shown in figure 4.7.

The semantics of the operation are as follows: $strategy$ is exported from $SrcSubEnv$ and subsequently imported by $DstSubEnv$. Treaty is atomic, meaning that both SubEnvs have to rollback in case of a failure. In addition, $DstSubEnv$ has to operate in single-user mode (i.e., only one client can be connected to it, although $SrcSubEnv$ and other SubEnvs might have arbitrary number of active clients). To simplify matters, Treaty is always initiated by the exporter. However, the exporter can be either a requester (default) or an acceptor, implying acceptor or requester privileges on $DstSubEnv$, respectively. Finally, as mentioned earlier, a *shared* privilege implements a full Treaty, i.e., either site can operate the rules in the strategy on the other site's data.

4.3.1.7 Rule Name Space

Back in Section 3.5.3 we identified the naming problem of process units, which appears here too. Since Treaties effectively implement common sub-processes, once a set of rules is imported and integrated in a local process, there must be a way to identify the very same rule across the multiple "members" of the Treaty. Rule names are obviously not sufficient, since multiple (overloaded) rules with the same name can co-exist even within a single local SubEnv (see [50] regarding rule overloading). So, some sort of unique rule id scheme is needed. In Oz, this is done by using the unique SubEnv id (multiplied by a large constant) as a prefix to the normal id generation of the rule translator. (In fact, all the necessary unique identifiers are derived from the uniqueness property of the SubEnv id which is guaranteed at site registration time, as will be seen in Section 5.3.) Thus, rule-ids of all rules of an imported strategy are guaranteed to be distinct from the rule-ids of any local rules, or rules from other imported strategies. When a Summit rule is invoked, its rule id enables remote sites to uniquely identify the invoked rule with their own copy of the rule, and service the various Summit requests that refer to that rule (e.g., verification, remote backward and forward chaining, etc.).

4.3.2 Local Evolutions and Dynamic Verification

In Chapter 3 we discussed the rationale for autonomous local evolution and the tradeoffs associated with it. The main point was that in order to comply with the autonomy principle, we want to allow independent modification of local processes, including operations that explicitly leave Treaties, as well as operations that might indirectly affect Treaties. To comply with the independent operation principle, we do not want to depend on other SubEnvs when such evolutions occur, and we want to minimize the communication overhead. On the other hand, we still want to make sure that Treaties are valid during relevant Summits. Another aspect of the dynamic verification is to protect a site from remote "invasions" of privacy, by ensuring that rules that were never included in a Treaty are not allowed to execute across SubEnvs.

Oz fully complies with this model of local evolution. With the exception of *import*, all operations that manipulate the local processes are local and involve no interaction with other SubEnvs. Therefore, some of the operations can potentially invalidate prior Treaties. In order to be able to detect invalid Treaties, we have to revisit the conditions that constitute a valid Treaty, analyze all the cases that might cause some of those conditions to not hold, and ensure that the proper checks are made at run time to detect invalid Treaties.

A (simple) Treaty from E_1 to E_2 on strategy S_1 is valid only if all three conditions below hold:

1. Either:
    (a) S_1 is marked at E_1 as exported to E_2, and is marked at E_2 as imported from E_1;
    (b) S_1 is marked at E_1 as imported from E_2, and is marked at E_2 as exported to E_1.
2. S_1 is marked at E_1 as a requester of E_2, and is marked at E_2 as an acceptor from E_1.
3. S_1 is identically defined in both SubEnvs. This is the "common sub-process" invariant discussed earlier in Section 3.2.6.

The first condition can be invalidated whenever *unexport* at the exporting site, or *unimport* at the importing site, is issued. *unexport* can be easily detected locally at the invoking site — the invocation is rejected if the issued rule is not (anymore) exported. *unimport* is also easily detectable since when a Summit rule is requested on the remote site, if it is part of a strategy which has been unimported it will simply not be found (the case of faked *import*, i.e., when the rule was already defined locally, is covered in the second condition since the sole purpose for faked imports is to affect execution privileges).

As for the second condition, both *request* and *accept* privileges have to be checked for their validity.

1. *request* — E_1 can lose its *request* privileges on S_1 if the equivalent of $cancel(S_1, E_1, E_2)$ was issued. This can occur in Oz in one of two ways, depending on the method by which the *request* privileges were originally assigned: (1) If through *export-request*, then an *unexport-request* on S_1 from

E_1 to E_2 revokes *request* privileges. This can be validated at E_1 locally when the Summit rule is invoked, at the same time the *export* privileges are checked. (2) If through *import-request*, then an *unimport* on S_1 at E_1 invalidates condition 2. Thus, validity checking is similar to that for condition 1. In case of a faked *import*, the *request* privileges are checked locally.

2. *accept*— E_2 can revoke *accept* privileges from E_1 on S_1 whenever the equivalent of *deny*(S_1, E_1, E_2) occurs at E_2. This can occur also in one of two ways, depending on the original commands issued to set up the privileges: (1) In case of *export-accept*, an *unexport-accept* command revokes the *accept* privileges. To validate this case, E_2 must explicitly check for proper *accept* privileges every time a rule in S_1 is issued from E_1 on data from E_2; (2) In case of *import-accept*, an *unimport* at E_2 invalidates the *accept* privileges. Again, in case of normal *unimport*, there is nothing to check, the rule will simply not be found. In case of a faked *unimport*, a check for *accept* privileges is required.

The third condition implies that all copies of a Summit rule must be identical in all involved SubEnvs. This condition can become unsatisfied as a result of (local) process evolutions, and is more complicated to check for. The key to the solution is *evolution timestamps*, explained below.

4.3.2.1 Evolution Timestamps

A set of strategies can be added, removed, or modified, effectively evolving the on-going process at a local site. There are several considerations with respect to Treaty verification. First, it is important to retain the validity of prior Treaties which are not affected by the evolution. Specifically, a Treaty is not affected by a local evolution if the evolved site is (only) the importing site in that Treaty. Since sites have no access to the "source code" of imported rules (as explained earlier in Section 4.3.1.2), this evolution does not violate the common-subprocess invariant.

The more severe problem is when a strategy which was imported into a remote SubEnv(s) is being evolved at the exporting ("source") SubEnv. Regardless of the process privileges attached to the exported strategy, such evolution violates the common-subprocess invariant.

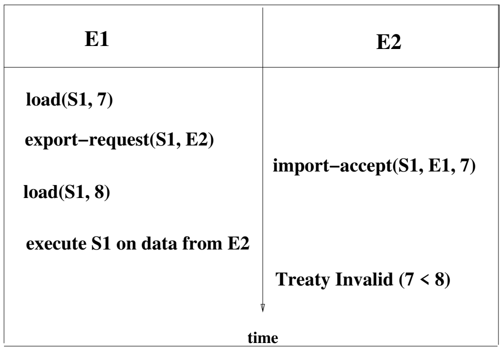
The idea is for the local SubEnv to assign a "timestamp" each time a strategy in its process is compiled and loaded locally. The term timestamp is a bit misleading, in that it is simply a local incrementing counter which does not depend on any real time or any other SubEnv's counter that would require "global time". When a strategy is imported, its timestamp is also shipped and stored at the importing SubEnv. At run-time, whenever a Summit rule is invoked for execution, the timestamp at the requesting SubEnv is compared to the one stored at the accepting SubEnv. If there is a mismatch, it means that local evolution took place at the exporting SubEnv, implying invalidation of the Treaty, and the execution is rejected. Re-activation of the Treaty can be made by either re-importing explicitly the (possibly modified) strategy, or by reloading the process, which also fetches the up-to-date versions of all imported strategies.

Under the above circumstances, the mismatch is guaranteed regardless of whether the exporter is an acceptor or a requester. But care must be taken that the check is for exact comparison. That is, if the exporter is a requester, its timestamp will be greater than the importer's, but if the exporter is an acceptor, its timestamp will be smaller than the importer's.

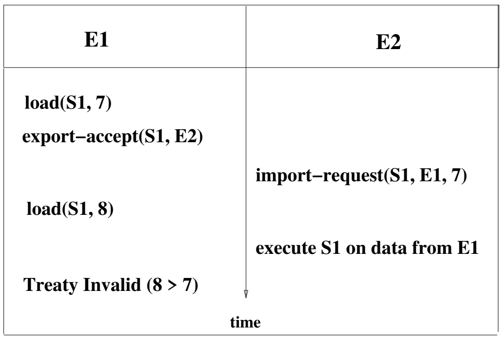
To illustrate this point and the timestamp mechanism in general, consider the two scenarios depicted in figure 4.8. In (a), strategy S_1 is loaded in E_1 with timestamp 7, then exported as a requester to E_2, and imported by E_2 as an acceptor. A subsequent (local) *load* at E_1 increases S_1's timestamp to 8, so when (a user operating in) E_1 requests execution of a rule defined in S_1 on data from E_2, the timestamp comparison at E_2 fails because the acceptor's (E_2) timestamp on S_1 is *smaller* than the requester's (E_1). In (b), E_1 is the exporter and E_2 the importer as before, but the privileges are switched: E_1 is the acceptor and E_2 is the requester. When an attempt is made to execute from E_2 a rule defined in S_1 on data from E_1, right after a (local) load that occurred at E_1 (thereby increasing S_1's timestamp) the timestamp comparison fails again, but this time because the acceptor (E_1) has a *larger* timestamp number than the requester (E_2).

This dynamic approach to Treaty verification eliminates the need to notify all related SubEnvs when a local process change occurs (some of them might not even be active at that time), and moves the responsibility of upgrading the imported rules to each remote SubEnv when it actually needs to use them. This "lazy update" approach fits well with the general decentralized philosophy.

Figure 4.9 summarizes this section by presenting the dynamic Treaty verification



(a) export–request and import–accept



(b) export–accept and import–request

Figure 4.8: Evolution Timestamp Example

algorithm (in pseudo-code), which is executed in the acceptor SubEnv prior to invocation of each Summit rule. Notice the three distinct levels of checking which correspond to the three conditions discussed above, and the possible actions that are required in order to reactivate old Treaties or establish new ones.

4.3.3 Common Sub-Schema

Since every data binding and reference in Oz rules is *typed*, rules implicitly require an underlying schema. For example, the `compile` rule in figure 4.10 operates on a formal parameter of type `CFILE` (line 1), which must have a `compile_status` attribute of `enumerated` type with at least three of the possible values in that type being `NotCompiled`, `Compiled`, and `ErrorCompiled`, as seen in lines 8, 14, and 15, respectively. The bindings of the rule also impose structural requirements on the schema. In the example, line 3 implies that `CFILE` has a link attribute named `hfiles` to the class `HFILE` (representing header files that are included by the `CFILE`).

```
verify-treaty(SrcSubEnv, DstSubEnv)

/* Executes at DstSubEnv */

      /* CONDITION 1 */
      if ( find rule with the given rule id)

           /* CONDITION 2 */
           if ( DstSubEnv is an acceptor of the rule's strategy
                for SrcSubEnv)

               /* CONDITION 3 */
               if (rule's remote timestamp == rule's local timestamp)
                  Treaty is valid, allow execution
               else

                    Treaty is invalid, reject execution
                    Reason: local evolution at the exporting SubEnv
                    Reactivation: re-import (or reload) at the
                                  importing SubEnv with proper privileges
           else

               There is no Treaty on that rule, reject execution
               Reason: an equivalent of cancel occurred
               Reactivation: DstSubEnv needs to accept the strategy

      else

           Requested rule does not exist in local SubEnv, cannot execute
          (Re)activation: DstSubEnv needs to (re)import the remote strategy
```

Figure 4.9: Run-time Treaty Verification Algorithm

When a rule is imported, the importing SubEnv must have the proper compatible sub-schema in order to be able to compile the rule and later execute it. Thus, the strong typing property rules out any hope for defining Treaties over totally disjoint and unknown schemas. The discussion of this issue in Section 3.2.4 advocated a solution that allows to specify arbitrary common and private sub-schemas, as opposed to requiring global schema or restricting the common sub-schema by some criteria. We now turn to the solution in Oz.

```
1) compile [?c:CFILE]:

2)  # Bindings: collect the header files which belong to this project

3) (forall HFILE ?h  suchthat (linkto [?c.hfiles ?h])):

4)  # condition:
5)  # if the C file has been analyzed successfully but not yet compiled,
6)  # you can compile it.

7)  (and ( ?c.analyze_status = Analyzed )
8)       no_backward ( ?c.compile_status = NotCompiled))

9)  # activity: invoke the activity with all necessary files/attributes

10) { COMPILER compile ?c.contents ?c.compile_log ?c.object_code ?h.contents }

11) # effects:
12) # mark the state of the ?c to compiled or error, depending on the
13) # return code from the compiler.

14) ( ?c.compile_status = Compiled );
15) [ ?c.compile_status = ErrorCompiled ];
```

Figure 4.10: Compile Rule

4.3.3.1 Solution in Oz

There are two problems with realizing common sub-schemas: (1) Static — how to test whether a remote sub-schema that is implied by a set of imported rules, is compatible with the local schema of the importing SubEnv. (2) Dynamic — how to uniformly manipulate objects with only partial common-subschema.

To illustrate the problems, consider the two definitions of the class `CFILE` at two SubEnvs, E_1 and E_2, given in figure 4.11. There are several differences between the class definitions. For example, the first definition has a `config` attribute (line 7) that is missing from the second definition (which has instead a set attribute `configs`, in line 25); the enumerated `analyze_status` attribute at line 3 has a different set of values than in its counterpart at line 17; and the `compile_options` attribute is of type `string` in the first definition (line 6) and an `enumerated` type in the second (line 19). Yet despite these differences, the `compile` rule that was shown earlier in figure 4.10 should be able to properly fire on objects from either definition, because the subset of attributes that is accessed by that rule is compatible. But in order to enable this (and disable non-compatible rules), the local process translator has to syntactically accept such rules at compile time, and the process engine must be able to accept objects from either class at runtime, even though they share only a sub-schema and are therefore not structurally identical.

In some cases the same symbol might be bound to a set of objects defined by different classes, making the dynamic problem more severe. Consider, for example, the multi-edit rule in figure 4.12, in conjunction with the definitions of the CFILE class of figure 4.11. Suppose that a user invokes the `multi-edit` rule with parameter objects `M1` bound to `?m1`, and `M2` bound to `?m2` where `M1` and `M2` are from SubEnvs E_1 and E_2, respectively. Then, the symbol `?c` (line 5) will be bound to all `CFILE` objects which are children of either `M1` or `M2`, meaning that they will be instantiated from different definitions of `CFILE`. Note that this rule should also be allowed to execute as far as sub-schema compatibility is concerned, because the accessed attributes in this rule are common to both definitions.

Addressing the Static Problem

OZ's solution to the static problem is as follows. First, there are no provisions for allowing isomorphic compatibility, i.e., structurally identical sub-schema with different names. Although a possibly useful feature, it is beyond the scope of this research. But only by-name type checking is not sufficient either. For example, the `compile_options` attribute has the same name in both definitions but the types are different, so such incompatibility, which cannot be tolerated, will not be detected. So both by-name and by-structure checks are required. The process translator first checks whether the *names* of classes and attributes that are referenced in the imported rules exist in the local schema. This is identical to the by-name checking that is performed for local rule compilation, so no special extensions are needed to obtain this functionality. This should be true for any strongly typed PML that must perform type checking to verify that the operands conform to the schema. The by-structure check is more complicated, however. Each attribute specified in the imported rules has to be checked in both schemas, to see that it corresponds to the same type. Note that there could be some complications due to inheritance, because an attribute of a class that

```
----------
SubEnv E1
----------

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

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

# Composite Attributes

# Reference Attributes
   hfiles : set_of link HFILE;
end

----------
SubEnv E2
----------

CFILE :: superclass FILE, PROTECTED_ENTITY;
# State Attributes
  analyze_status : (NotAnalyzed, ErrorAnalyzed, Analyzed) = NotAnalyzed;
  compile_status    : (NotCompiled, Compiled) = NotCompiled;
  compile_options   : (Debug, Optimize, Normal) = Normal;
  object_time_stamp : time;

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

# Composite Attributes
   configs  : set_of CONFIG_SRC_oz;

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

Figure 4.11: Two Definitions of class CFILE

```
multi-edit [?m1:MODULE, ?m2:MODULE]:

 (and

  # binding:
  # collect all source files contained within either module

  (forall CFILE ?c suchthat (or
                             (member [?m1.cfiles ?c])
                             (member [?m2.cfiles ?c])))

  # collect all the header files linked to the sources

 (forall  HFILE  ?h  suchthat (linkto [?c.ref ?h]))):

  # condition : check that sources are accessible to invoker

  (and
      (?c.reservation_status = CheckedOut)
      (?c.allowed_edit  = CurrentUser))

  { MULTI_EDIT multi_editor ?c.contents ?h.contents }

   #0. sources from both m1 and m2 changed
   (and  (?m1.status  = NotCompiled)
         (?m2.status  = NotCompiled);
   #1. only sources from m1 changed,
         (?m1.status  = NotCompiled);
   #2. only sources from m2 changed,
          (?m2.status  = NotCompiled);
   #3. no changes made, assert nothing
       no_assertion;
```

Figure 4.12: Multi-edit rule

is specified in a rule might actually be defined in one of its superclasses, implying that the process translator has to search through the class hierarchy. Moreover, checking for lattice compatibility (i.e., the composition hierarchies) might be required for the structural bindings in rules. Finally, a potential problem might be different ordering of the same attribute in different class definitions. This, however, is not a problem in Oz since attributes in Oz are accessed through their name, not through field offsets. In fact, this attribute-based access also facilitates the solution to the dynamic problem, as will be seen shortly.

The general solution to the static problem is then to compare the subschema specified in the imported strategy with the local schema, generate the structural "delta" between them, and determine if the attributes accessed by the rules in the imported strategy overlap with the "delta". If there is such overlap, the imported strategy is not schema-compatible with the local process, and the *import* fails, otherwise it succeeds.

An alternative solution would have been to merge schemas. That is, in case of, say, two `CFILE` definitions, both SubEnvs would end up with the same `CFILE` definition that is the union of the attributes, with some arbitration policy among conflicting attributes. However, this approach has several drawbacks, particularly with respect to the "core" requirements. The main problem with this approach is that it implies that the local instantiated objectbase has to be evolved to correspond to the new merged schema, and even the "source" SubEnv might possibly need to be evolved. But most of all, since objects are "mixed" only temporarily during execution of rules, but otherwise reside in their own private objectbase with their own schema, there is no justification to merge local schemas and evolve the objectbases just to satisfy the type restrictions.

The solution in OZ is based on the "delta" analysis front-end component of the Evolver, discussed in Section 2.2.5. The idea is to apply this delta analyzer selectively to the classes that are referenced in the imported strategy, and generate a "delta" between the importing and the exporting SubEnv (which therefore must send those class definitions at *import* time). If the delta is unacceptable (for example, because of mismatched types) the *import* is rejected.

It seems that the same approach could be applied to a wide range of implementations, so long as they have a schema evolution utility. The key observation is that the analysis step in schema evolution is similar to the one that is needed for *import* purposes.

Addressing the Dynamic Problem

The essence of the solution to the dynamic problem is *attribute-based* access mentioned above. That is, when an object is transferred from one site to another, it is treated merely as a set of attribute-value pairs (and an object-id). These pairs also include all the attributes that were inherited from any superclasses. Since all accesses to objects from the rule processor specify attributes, there is no need to carry the original class definition with the object. As long as the attributes that are accessed by the rule have been verified at Treaty definition time to be compatible with those defined in the remote schema, the corresponding remote objects can be accessed properly. Since all attributes that are accessed by the process are explicitly stated in the rules, there is no way to mistakenly access attributes that are not defined or have the wrong types.

The simplicity by which the set of accessed attributes is specified in rules enables us to rigorously analyze which object fragments will be accessed, and therefore enable co-existence of different schemas with sufficient common-ground for execution of multi-site activities. The object-based paradigm in itself also helps to support co-existence of multiple schemas, because the object identity allows a rule to have a direct handle on an object, without necessarily requiring to know its full schema. It is sufficient to require only that the values of the attributes which are part of the subschema accessed by the "method" (rule) will be valid.

4.3.4 Exporting Data Instances

In Section 3.2.5, it was realized that while in some cases the definition of a common sub-schema might imply that the *instances* of the classes in the common-subschema are also common, this is the exception, not the norm. In general, it would be impractical to assume that an *accepting* SubEnv implicitly "exports" all instances that belong to the common sub-schema. This observation certainly holds for OZ, where a typical project-database is built around a composition hierarchy that is orthogonal to the class hierarchy. Thus, the fact that a set of objects is instantiated from the same class is immaterial with respect to their semantic relationship; more often, a SubEnv would want to export a set of objects that are *structurally* related. Consider, for example, the case where `CFILE`, a class that abstracts C source files, is part of a Treaty from E_1 to E_2. Then the above approach would imply that users from E_1 can access with Treaty rules *any* `CFILE` objects in E_2. Not only is this granularity too coarse, it is the wrong kind of association. Instead, the export of data should be specified by selecting objects which are related structurally, e.g., a sub-project containing some `CFILE` objects, as well as libraries, binaries etc.

It is clear, then, that a separate *export_data* mechanism, as outlined in Section 3.2.5, is required. The generic specification of *export_data* ignored several practical issues concerning specific PCE implementations, however. The first major issue to discuss is the various granularities for specifying the export: (1) Is it affordable to control the data export in Oz on a SubEnv-basis (as suggested in the generic specification) or should it be global; (2) Is it affordable to further control the export on a per-strategy basis (as suggested in the generic specification); (3) What should be the objectbase granularity for exporting data (the generic specification did not address this PCE-specific aspect at all); and (4) What should be the granularity for specifying access modes on exported data, if different from (3). The second major issue concerns the interface to such a mechanism. The problem is to abstract the operation so that it can be performed in a relatively high-level, not requiring to specify each data element separately. We now discuss our solution.

4.3.4.1 Exporting Data in Oz

Clearly, controlling the export on a per-SubEnv basis must be supported as it is essential for autonomy and security purposes, and is in-line with the "not-only-local-or-global" approach adopted throughout the thesis. Given that the number of SubEnvs is relatively small, the overhead should be small. As for the data granularity for specification of export, a single object seems like the ideal granularity, but the overhead is much higher since the number of objects is typically large, and this scheme implies that each object must have an additional information concerning export status. Moreover, in conjunction with the per-SubEnv support, this information must be maintained as a list, as opposed to a binary flag. Similarly, specifications for access permissions and control on a per-SubEnv basis are desired but add both space and computation overhead.

The design of *export_data* in Oz is based on extensions to the basic access control mechanism inherited from Marvel (discussed earlier in Section 2.2.6). The idea is to utilize the already existing mechanism for specifying group permissions, and associate SubEnvs permissions with such groups. That is, an exported object has, in addition to its local user and group permissions, SubEnv permissions (that happen to be implemented as permission groups). When a SubEnv E_1 receives a remote request from E_2 to access a local object O_1, the server at E_1 checks whether O_1 is accessible to E_2 by inspecting O_1's group permissions for E_2. If no group permissions are defined in O_1 for E_2 or if the permissions are not compatible with the request, the access is denied.

Thus, using a straightforward extension of the general-purpose access control mechanism, we obtain: (1) object-level granularity for export; (2) per-SubEnv specification of export; and (3) object-level permission specification. Moreover, this extension does not seem to incur extra overhead beyond what is necessary by the local access control. However, just as with local access control, the number of groups attached to each object does affect performance, as each access to an object involves a larger search space. But given that the name space of SubEnvs is well known, a simple effective hashing algorithm can retain a constant search time. The only capability that cannot be supported in this approach is to be able to specify exports on a per-strategy basis. However, the problem with this feature is not with the actual specification, since it could be achieved by considering the Cartesian product of SubEnvs and strategies and creating a unique group for each pair. The main problem with this feature is that such a mechanism would imply that any evolution involving Treaty rules would require a global search in the database to update all objects that might have been affected by the evolution, making this feature intractable and unacceptable.

The next issue to consider is how to abstract the *export_data* operation so that it can be performed as a relatively high-level command. The idea here is to rely on the composition hierarchy as the abstraction for grouping objects. Thus, *export_data* takes as arguments a remote SubEnv, a local composite-object that is the root of the sub-tree to be exported, and a permission string. The operation then traverses all descendant objects of the root, and for each object it generates the proper SubEnv group (unless it already exists) with the specified permissions. This approach allows the administrator to apply the *export_data* operation on arbitrary level of granularity. In particular, it can be applied to single "leaf" objects.

Finally, an additional possible extension to the access-control mechanism could be to extend remote permissions on a per-user basis (as opposed to on a SubEnv-basis), similar to local access control. The main problem with this approach is that the set of possible (remote) users is not known at any time, and obtaining knowledge at each SubEnv about users from remote SubEnvs contradicts decentralization. In addition, this model assumes that in most cases the group-level permissions associated with SubEnvs is sufficient, just as Treaties are formed between groups of users on a SubEnv basis (if the group becomes too large to treat it coherently, perhaps it should be split into separate SubEnvs).

In the rare cases where remote user-level access control might still be highly desired, a possible approach might be to assign the remote user a (remote) `user` object, which entitles the permissions as specified in its `mask`, subject to the SubEnv permissions which receive first priority. One way to look at these permissions is that the remote user is treated as a *friend* user (borrowed from the concept of a friend function in C++), which, regardless of the origin SubEnv from which he/she operates, can access the remote data as a local user. In some cases, the "friend" user can be the same (mobile) person who logs in from several SubEnvs at different times.

4.3.5 Preserving Process Consistency

The general process consistency problem was defined in Section 3.2.7. Because of the decentralized nature of our model, there is no notion of "global" consistency. Just as sub-schema compatibility is a sub-problem of the more general schema evolution problem, sub-process "compatibility" (i.e., consistency) is a sub-problem of the more general process consistency problem. In fact, this is even more evident in the case of process than in the case of schema, since *import* implies only pure additions or deletion of whole rules. That is, it excludes the harder cases of allowing to modify existing rules. The reason is that OZ allows multiple rules with the same name (and even signature) to co-exist, and further, it does not have the notion of "merging" rules[7]. Thus, importing a strategy amounts to evolving a process by adding to it a set of new rules (or deleting rules in the case of *unimport*). This means that the process evolution algorithm employed in Marvel can be used "as-is" by the *import* operation to verify the consistency of the local process after a set of Treaty rules has been added to it.

4.4 Multi-Process Enactment in OZ

Process enactment in OZ can be roughly divided into *local* and *multi-SubEnv* enactment. The former involves only a single SubEnv and all interactions between the server and any of its local clients (and their corresponding users). The latter includes all operations that involve interactions among servers and clients from multiple SubEnvs.

Pure local enactment not related to multi-SubEnv enactment is largely the same as in Marvel, and is not discussed here any further. As for multi-SubEnv enactment, there are several types:

---

[7] This was supported in earlier versions of Marvel, but not in Marvel 3.x. and not in OZ.

1. Purely remote — a client interacts directly with a remote SubEnv without involving its own local server
2. Built-in cross-site commands — multi-SubEnv operations that are implemented as part of the kernel, and are not process-specific
3. Enactment of Treaty rules, following the Summit model.

The first two kinds of enactment are, for the most part, not process-specific and relatively minor, and are discussed briefly in Sections 4.4.1 and 4.4.2, respectively. The third type is the major type of enactment, and is discussed in depth in Section 4.4.3.

4.4.1 Direct Remote Interaction

First, we need to justify why a client would directly interact with a remote server without involving its local server (except for establishing the connection in the first place, explained later in Chapter 5). A simpler architecture would direct any cross-SubEnv interaction through the local server, reducing the types of interactions across SubEnvs. However, given that there are some "core" built-in operations that have identical well-defined semantics in all servers and involve data from only one remote site, it makes sense to allow direct communication between clients and remote servers for these operations, thereby eliminating unnecessary overhead at the local servers and reducing the number of message "hops".

Several direct remote services are provided in OZ, mostly those that correspond to the built-in objectbase access and manipulation operations, namely: `browse`, `print object`, `add`, `delete`, `link`, `unlink`, and the single-server versions of `move` and `copy`. Thus, a client that is connected to a remote SubEnv can potentially issue these commands directly to the remote server without involving the local server.

One potential negative implication of this approach might be due to overloading. Since OZ allows built-in operations to be overloaded with process-specific rules that specialize these operations on certain classes (the default built-in operations work on all classes), a request for remote operation might have various unanticipated implications. This could have been a serious problem if the executed rule was remote to the data (i.e., local to the remote user's SubEnv). However, this is an impossible scenario since the operation is executed at the remote SubEnv using its own (perhaps overloaded) built-in operations, which are *local* to its data (and therefore "owned" by it). Thus, this does not incur any violation of autonomy or privacy. In the worst case, the (remote) user might be surprised by some unanticipated behavior resulting from invoking an overloaded built-in operation.

4.4.2 Built-in Multi-SubEnv Operations

Oz extends Marvel's repertoire of built-in commands with: (1) cross-site `copy` and `move` objectbase operations; (2) *import* and *treaty* operations that support construction of Treaties; and (3) a set of built-in rules for SubEnv (de)registration.

The *import* and *treaty* operations were covered earlier, and site registration is covered separately in Section 5.3, so we cover here only `copy` and `move`.

The operational semantics of these operations are straightforward: they copy (move), a (possibly composite) object from one SubEnv to another. There are several technical problems, though: (1) The schemas at the two SubEnvs might differ in an incompatible manner, such that one or more of the objects at the source SubEnv belongs to classes that are either defined differently at the target SubEnv, or worse, not defined at all. A related problem, particularly with copying (moving) composite objects, is possible incompatibility in the composition hierarchies; (2) The overloading mechanism could be potentially dangerous here since, as described above, local processes might overload these operations, and unlike the single-SubEnv case, both SubEnvs are involved here. This means that it might be possible for one SubEnv to invoke an overloaded version of, say, the `copy` rule, on remote data, with the remote SubEnv (and its administrator) not knowing the contents of the rule (consider a worst case scenario where some malicious remote SubEnv overloads `copy` with `delete`).

Before addressing these problems, it is worth mentioning a useful application of cross-site `copy` and `move`: it can be used to effectively implement objectbase `split` and `merge` operations (provided that the merged objectbases have compatible schemas).

4.4.2.1 Addressing the Schema Compatibility Problem

The main reason that the schema compatibility problem reappears here is that we want to support these operations without the need to specify them as part of a Treaty, similar to the way single-server built-in operations are supported. Further, this problem differs from the schema compatibility problem addressed in Section 4.3.3.1 where objects are *temporarily* transferred across SubEnvs. Here, the copied objects become part of the local persistent objectbase, thus they must fully correspond to the local schema definition.

If we blindly allow `copy` (or `move`) to occur, we risk the possibility of acquiring objects which are completely or partially schema-incompatible, making them either inaccessible or even worse, corrupting the internal objectbase structure. An analogy to this situation is the well known "structure offset" problem in conventional programming languages, whereby a data structure with the same name is defined differently in two modules, and a pointer is passed from one module to another. The offset due to the different definition is likely to corrupt the program stack when the "receiving" module executes.

There are two possible solutions to the schema problem. The first solution simply avoids the problem by requiring an identical sub-schema, and rejecting the operation in case of incompatibility. This a simple but unnecessarily restrictive solution. The second solution, which was adopted in Oz, interprets an object copied to the target SubEnv according to the local schema's class definition. This in turn might result in some loss of data if some attributes in the class definition of the source SubEnv are missing from the definition in the target SubEnv. Alternatively, if the target class subsumes the source class, the default values (which can be determined optionally in the class definition) are assigned to the missing attributes. If attribute types conflict, the target SubEnv can coerce values if possible, or assign the local default values.

As for the composition hierarchy, a similar approach is applied. Here, however, entire (composite) sub-objects might be lost if the expected composition attributes are missing in the target SubEnv. To avoid possibly undesirable loss of data due to schema incompatibility, which is especially important in the case of `move`, a warning message listing the lost data should be presented to the user with the option for a possible retraction.

4.4.2.2 Addressing the Rule Overloading Problem

A cross-site `copy/move` operation can be executed in one of two possible ways: it can be executed either at the source SubEnv (i.e., the SubEnv from which the objects are copied/moved) or at the target SubEnv. In either case, only one SubEnv fires the actual rule, and the other SubEnv performs the remote operation derived from the actual built-in `copy/move` operation. (Note, however, that the low-level operations for `copy` and `move` as well as the other structural built-in operations cannot be overloaded in Oz — only the rules containing them can be overloaded. This design choice was made deliberately to avoid the possibility of arbitrarily changing the semantics of primitive operations.)

Thus, if the SubEnv that executes the `copy` (`move`) happens to have an overloaded rule which is unknown to the other SubEnv, such execution (without a Treaty) presents a clear violation of autonomy. Therefore, only the built-in (non-overloaded) versions of `copy/move` can be used for cross-site execution, and an overloaded version could be used only if it is part of a Treaty.

4.4.3 The Summit Model in Oz

Implementation of Summits in Oz could be considered as the most important and most comprehensive aspect of the implementation effort. Summits are the main means by which multiple SubEnvs interoperate, and as such, they encompass all the support that is required to enable multi-process activities among the pre-defined common sub-processes while still preserving the autonomy and privacy of the private sub-processes. While Treaty support is also dynamic, it is conceptually static or "meta-enactment" since it deals with *definitional* aspects of the process. Consequently, infrastructure support for the realization of the Treaty protocol is much less complicated. Summit is the realization of "real" enactment of multi-user and multi-process activities, rules, and rule chains (i.e., automatic enactment of tasks). Thus, whereas Treaties refer to static properties of rules and data (e.g., formal parameters and types), Summits are concerned with dynamic properties of rules under execution, such as the runtime objects that are bound to an executing rule, the chaining context in which they execute, and so forth.

To make our discussion more clear, we define a *Summit rule* to be a rule that contains *actual* parameters from at least one remote SubEnv. This is a dynamic property of rules. While it is true that every Summit rule must have been defined in some Treaty, the converse is not always true, since a Treaty rule can at times execute only with local data, in which case it is not acting as a Summit rule. A *Summit task* refers to an entire rule chain that involves at least one Summit rule.

We now describe in detail the realization of the Summit protocol, covering all five phases of the formal model, with focus on inter-process aspects. To simplify the discussion, we defer the discussion of two important aspects: (1) transactional semantics of Summits, which are discussed in Section 4.4.5; and (2) Context switching requirements that enable concurrent execution of rules and Summits, discussed in Section 5.4. An efficient caching mechanism for accessing remote objects during Summits is covered in Section 5.5.

To better illustrate how Summits work, the explanations below are accompanied by an example, involving the `multi-edit` rule from figure 4.12, and the simple objectbases shown in figure 4.2.

4.4.3.1 Summit Initialization and Verification

A Summit task is initialized as a result of an explicit request from a user. From the user's point of view, the only difference between invoking a Summit rule and a normal rule is that at least one of the parameter objects specified by the user is remote (recall from Section 2.2.3 that MSL supports late binding, allowing the user to select different objects at different times as parameters to rules, which is essential to the understanding of Summits). The local server from which the user invoked the rule is called the *coordinating server*. For example, assume that user *israel* operates in site *NY* and invokes the `multi-edit` rule (from figure 4.12) with one local `MODULE` object named *ui*, and one remote `MODULE` object from *CT* named *db*, corresponding to the objectbases of figure 4.2. (To simplify the example, this rule operates with only one remote SubEnv, but in general Summit rules can operate with multiple remote SubEnvs.)

The first action taken by the coordinating server is to fetch copies of the remote objects from their original SubEnvs, and bind them to the parameters of the rule[8].

This fetching is necessary because the client only holds an objectbase image that enables the user to select objects as parameters to rules. The client sends to the server object-ids, which are resolved by the server(s) to real objects.

The reader might wonder at this point why is it necessary to fetch the remote objects before doing Treaty verification. The reason is somewhat pragmatic, and has to do with the rule-overloading mechanism. Recall that Oz allows multiple rules with the same name to co-exist, and determines which rule to execute based on the types and number of actual parameters supplied by the client [50]. Therefore, when the local server receives a request to execute a rule, it has to find the "closest" rule that matches the types of the parameters, so only after the remote objects (and their type information) are fetched, can the server determine which rule is intended for the Summit. (Actually, this could have been optimized, if the client maintained type information in its image and had sent it along with the remote

---

[8] This is in addition to the obvious binding of local objects, but as we focus on inter-site issues, we will ignore from now on purely local aspects of the rule processor.

object-ids — but this is not implemented in Oz.) Overloading of rules appears to introduce another problem: when an object is fetched from the remote site, its class definition might differ from the one in the local schema, or might not even exist. However, if the class is not in the local schema, then overloading would never find a proper rule. Alternatively, if the class is identified in the local schema and a Treaty rule that matches the types of the parameter list is found, the corresponding sub-schema is guaranteed to be compatible, as described earlier in Section 4.3.3.

Once the rule is identified, the second step involves Treaty verification. First, the coordinating server checks locally whether the rule could be invoked as a Summit rule, by checking that the rule has *request* privileges on the remote participating SubEnvs (i.e., those SubEnvs that have objects bound to parameters of the rule). If this is not the case, the rule cannot be executed in a Summit. But, as explained earlier, this is only a necessary condition, not a sufficient one, because the Treaty might have been invalidated unilaterally by one or more of the participating remote SubEnvs. So, after local verification, the coordinating server requests each participant SubEnv to execute the verification algorithm of figure 4.9 (covered in Section 4.3.2).

After the rule has shown to be a valid Summit rule, the third step binds remote *derived* parameters. In our example, the bindings of the `multi-edit` rule collect all objects of type `CFILE` that are children of either *ui* or *db*, and bind them to the symbol `?c` (lines 5-7)[9]. This results in a binding set $\{ddl.c, query.c, tty.c, xv.c\}$. It also binds to the symbol `?h` all the `HFILE` objects which are linked to all `?c` objects (line 9). This results in the binding set $\{db.h, shared.h\}$. As with regular parameters, sub-schema compatibility among derived parameters is assumed to have been checked at Treaty definition time. This concludes the initialization phase of a Summit in Oz.

4.4.3.2 Pre-Summit

This phase consists of two parts. First, the coordinating server evaluates the rule's condition. In our example, the condition is a simple conjunction of two predicates that evaluates to True if <u>all</u> objects bound to `?c` have been properly checked out (line 12) and that the current user invoking the rule is allowed to edit those objects (line 13)[10].

---

[9] The oddity with respect to universally quantifying `?c` in line 5 was explained in Section 2.2.2.

[10] `CurrentUser` is a built-in value in Oz that denotes the user who fired the rule, or on whose behalf the server is firing the rule automatically.

The second part of Pre-Summit is required if the condition is not satisfied (i.e., it evaluates to False). The coordinating server attempts to satisfy the condition by fanning out to the participating sites and triggering local *backward* chaining at each site in an attempt to update the objects so that they satisfy the condition. Backward chaining is private, i.e., each process performs this step according to its autonomously defined sub-process. Some optimizations could be made here. For example, in cases where it is possible to satisfy a condition only based on backward chaining at the coordinating site, it should be attempted first, before any remote chaining is spawned. And in the cases where remote chaining is necessary, it should be spawned and performed simultaneously in all sites (including the coordinating site in "local" mode). Note this can be done only if there are no data dependencies across sites (another good reason to avoid cross-site links).

The backward chaining algorithm is iterative in the following sense: After spawning the remote backward chains, the coordinating server "waits" (in practice it actually saves the context in a data structure called the "rule stack" and performs a context-switch to service other requests, but for the purposes of this discussion we can assume that it logically waits) for the remote servers to return with the results, which possibly contain some modified objects. The coordinating server then re-evaluates the condition. If backward chaining has not satisfied the condition, the rule is denied execution. However, even if a particular predicate became satisfied during backward chaining, the changes to the objects could have made other parts of the condition unsatisfied. Thus, the entire condition has to be re-evaluated each time, and backward chaining may be iteratively spawned several times during this phase until either the condition is satisfied, or all possibilities have been exhausted and the rule is not satisfiable and cannot be executed. Note that the potential to enter an infinite loop while evaluating a condition exists, but it merely indicates a flaw in the process model.

In our example, suppose that the object *xv.c* from `NY` and *query.c* from `CT` are not checked-out. `NY`'s process backward chains to its local configuration manager, say RCS [106], and issues a `check-out` rule on the object. At the same time, `CT` backward chains locally to its private configuration manager, say SCCS [94], and issues its own `check-out` request. This rule could further have a condition that implies firing other rules recursively, independent of any other site's process knowledge or interest. To illustrate the need for re-evaluating the entire condition, it could be the case that some `check-out` rule satisfied the first predicate in `multi-edit`'s conjunction (line 12), but at the same it might have also unsatisfied the second predicate (line 13), which could have been satisfied prior to the backward chaining. Although perhaps not a likely situation in this particular case, we can see how this could happen in general.

Execution of Remote Activities in Pre-Summit

One important aspect of remote backward (and also forward) chaining involves execution of (remote) activities. In Oz, both backward and forward chaining can lead to the execution of further activities. That is, chaining is not limited to inference rules, and can involve the same kinds of activities contained within user-invoked rules. In particular, some of those activities might be interactive, requiring input from a user. This presents both conceptual as well as technical problems that do not come up in local backward chaining: conceptually, the remote server must determine which user's client should execute the remote activities; technically, it should be able to redirect the activity to the specified user's client.

The solution in Oz is to direct all activities to the initiating user, by default. An optimization could be to direct only interactive activities to the remote client and execute non-interactive activities with a local "proxy" client. To provide a full solution, however, Oz allows remote activities to be delegated to (remote) users by extending its modeling language to specify delegation, and by providing a delegation mechanism that redirects activities. This is explained separately in Section 4.5.

In addition to directing activities to clients, an Oz server also sends *process animation* messages to the client that inform the user visually about the task being executed. In order to extend this useful capability to Summits, the server redirects all animation messages to the "coordinating-client" (the client that issued the Summit request), including messages that are executed on behalf of remote sites, and possibly by remote delegated users, thereby providing a "global" view of the process for monitoring purposes.

4.4.3.3 Summit Activity

If the condition of the rule is satisfied, the activity can execute at the coordinating client. Activity execution usually involves dereferencing file attributes, which map to files in Oz's "hidden" file system (inherited from Marvel), and handing them to the file-based tools defined in the activities. If all servers share the same file system (e.g., via NFS), then as in the case of a single server, only path names of the associated files (both local and remote) need to be sent to the client, and the tool can access the files through the path names. If servers do not share a file system, however, the actual files have to be physically transferred to the coordinating server's file system before handing them to the client (clients and their local server must have a shared file system, by definition), and when the activity completes the files have to be transferred back to their original location in the file system. This implies a remote file transfer mechanism separate from the object transfer mechanism. Note that while Oz objects are "light"— containing only state attributes, pointers to other objects, and "pointers" to files— files are arbitrarily large. In all the various cases mentioned previously, the remote objects being transferred contained no files. The remote file transfer mechanism, including prefetching and file caching considerations, is a separate mechanism outlined in Section 5.6.1.1.

Back to our example, the bound `CFILE` objects ($\{ddl.c, query.c, tty.c, xv.c\}$) and the related `HFILE` objects ($\{db.h, shared.h\}$) are passed to the `editor` activity, which in turn invokes a multi-buffer editor tool with one buffer per file (line 14). Although access modes for objects are discussed later in Section 4.4.5, it is worth addressing here the issue of access modes for (file) attributes. Each strategy file has a tool definition section that enables the process administrator to specify in which mode the files could be accessed. This information also affects transaction management and lock assignment. For example, here is the tool definition for `multi_edit`:

```
MULTI_EDIT :: superclass TOOL;
    multi_edit : string = "multi_edit CFILE.contents X HFILE.contents S";
end
```

This definition specifies that the `multi_edit` activity requires to access the `CFILE` objects in eXclusive mode (denoted by the X lock request) and the `HFILE` objects in Shared mode (denoted by the S lock request).

4.4.3.4 Post-Summit

The first step in Post-Summit asserts the appropriate effects of the Summit rule, depending on the return code from the activity, including remote assertions. Since the executed rule is identical at all participating sites (because of the common-subprocess invariant), this phase can be carried out in one of two ways: either the coordinating server sends a message to the remote servers to assert the effect of the rule on the (remote) objects, or the coordinating server itself asserts the effects on the replicated objects and sends the updated objects to the remote servers. Actually, a similar tradeoff exists with respect to the implementation of the binding phase: bindings could be either evaluated by requesting the remote server(s) to carry out entire binding queries, or by sending the remote servers primitive requests to fulfill parts of a composite binding query. The prototype rule-based approach to applying the general model (Section 3.5.1) suggested the former approach, and seems to be the natural solution given that the definitions of both bindings and assertions are guaranteed to be identical in all participating SubEnvs. Moreover, this approach has the potential to perform better since the communication overhead is reduced. However, the latter approach simplifies rule processing in that the Summit rule executes as a whole at the coordinating server and there is no need to invoke remote rule processors to execute rule "fragments". In addition, the replicated remote objects must be updated in the coordinating server anyways as part of the cache management (explained in Section 5.5). Therefore, Oz employs the latter approach both in bindings and in effects.

In the formal Summit protocol, the next step following the assertions is the "forward fan-out", in which each SubEnv (including the coordinating SubEnv) fires rules locally based on their local and private (sub)processes. Oz deviates slightly from this order, mainly due to low-level implementation details not discussed here[11]. The coordinating server first derives both the local and the global (i.e., further Summit) rules to be executed in the forward chain. The inferred forward Summit rules are held in a separate *Summit stack* and are invoked in the Summit completion phase only after all local forward chains complete in all sites. The inference of Summit rules is an important topic, explained separately below.

Following the derivation phase, forward fan-out takes place. Each SubEnv then determines which rules to execute based on its local process, and carries out the chains locally until all possible forward chains have completed. At this point, they return to the coordinating server.

Inference of (forward chaining) Summit Rules

Multi-step (or composite) Summits are crucial for modeling and enactment of multi-process tasks, simply because a Summit task may consist of several steps. There are several

---

[11] They have to do with the notion of atomicity chaining, discussed in Section 4.4.5.

approaches to modeling and enacting multi-step Summits. Technically, the coordinating server must distinguish chains which are part of the local fan-out from these which are "global" Summit rules. One alternative is to add modeling primitives (e.g., in the form of effect directives, similar to MSL chaining directives) that explicitly annotate effect predicates in rules as "Summit" predicates. These annotations could be used to determine which chains are local, and which are global. In fact, the initial implementation in Oz was done that way. However, this alternative both limits the power of the rule inference engine and proves to be unnecessary.

Given that a Summit rule is syntactically a "normal" rule that just happens to have remote objects bound to it, then by extending the standard inversion mechanism to handle inversion of remote bindings in addition to local bindings, the basic rule-inference mechanism could infer Summit rules — these are simply the rules that happen to have been instantiated with (some) remote objects as parameters. This, of course, could only happen in our Summit model if the triggering rule had some remote object parameters in the first place, or in other words, if it was a Summit rule. Thus, inference of Summits is done exactly in the same way that local inference is done.

The main advantage of this approach is that, as a natural extension of the rule processor for handling derivation of Summits, it is no more (and no less) implicit that derivation of rules, and it has the potential for *automatically* inferring multi-step Summits which could not have been formed in the explicit notation unless they were pre-determined. Another advantage is that Summit rules are formed only as needed, whereas the annotation approach would force the administrator to consider Summits even when no remote data is involved. Finally, adding annotations would have added an (apparently unnecessary) burden on process administrators in forming Treaties.

4.4.3.5 Summit Completion

Once local forward chaining completes in all involved SubEnvs, they notify the coordinating server, which in turn checks its Summit stack to see if there are any pending Summit rules. If there are no such rules, it completes the task, releasing resources that were allocated for the Summit, and commits the associated transactions (see Section 4.4.5).

If there are pending Summit rules, the coordinating SubEnv essentially starts with the Summit initialization phase, except it bypasses the manual parameter binding phase, which was (automatically) performed. Recall that binding must occur before the initiation of forward Summits, because it is the binding phase that actually recognizes which rules are Summit rules.

Finally, there is one more important difference between the invocation of the first Summit and subsequently derived Summit rules: The pre-Summit phase in derived Summits consists only of condition evaluation, without the fan-out for backward chaining. This stems from the fact that the rule processor in Oz (as in Marvel) is not capable of backward chaining during forward chaining. Thus, the realization is limited in that respect compared to the generic model.

4.4.4 A Composite Summit Example

The following is an execution trace of a composite Summit example in Oz that realizes the motivating example (which was presented in Section 1.3 and revisited in Section 3.4). The example is best illustrated in figure 4.13, while figures 4.14 and 4.15 show two snapshots of the actual Oz animator that were taken as the Summit was enacted.

The environment consists of three SubEnvs, `siteA,` `siteB`, and `siteC`. To simplify the example, each SubEnv has two objects which are relevant to this Summit, a parent object of class `MODULE` and a child object of class `FILE` (shown in the top of figure 4.13).

The `Change` rule is fired at `siteB` (which then becomes the coordinating SubEnv) with `FILE` objects `fA,` `fB` and `fC` from `siteA,` `siteB` and `SiteC`, respectively, effectively initiating a three-site Summit. Pre-Summit is carried out by fanning-out and performing local backward chaining. At `siteB`, the `review` rule is preceded by the `SCCS_co` rule that checks-out `fB`; at `siteA`, `review` is preceded by `RCS_co` on `fA`; and at `siteC`, `review` is preceded by an `analyze` activity. Then the actual `Change` activity (Summit rules are annotated in figure 4.13 with **S**, and their enactment relationships with bold lines; in the Oz animator, Summit rules are depicted by a special "mountain-summit" icon, to distinguish them from local rules) is carried out, followed by forward fan-out Post-Summit, which in turn produces local chaining only at `siteC`, to the `update_log` rule. When Post-Summit completes, `siteB` triggers `approve`, the next Summit rule. Note that `approve` takes `MODULE` objects as parameters, which are the respective parents of the `FILE` objects bound in `Change`, so remote derivation of parameters is necessary here. `approve` produces no local chaining in any SubEnv and leads directly to the next Summit rule, `modify`. (Here again, remote deriva-

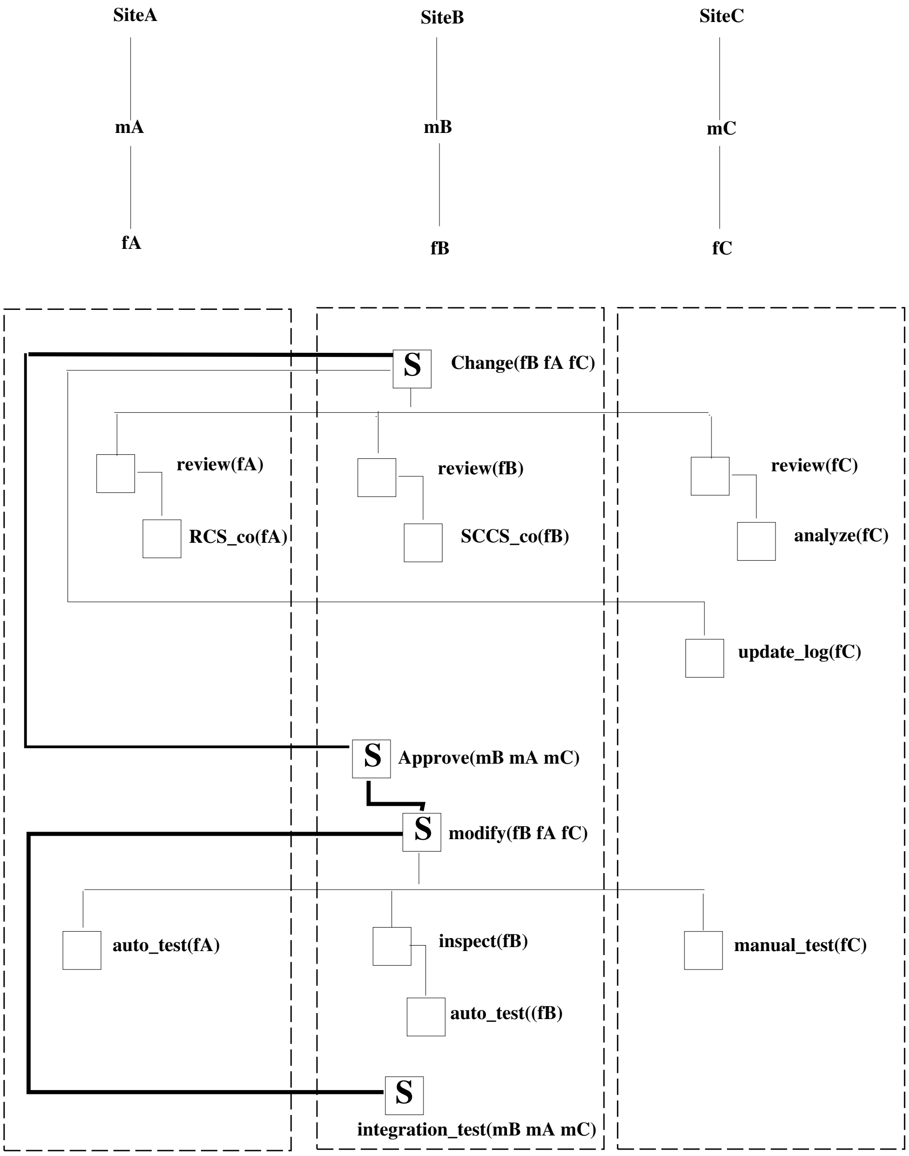
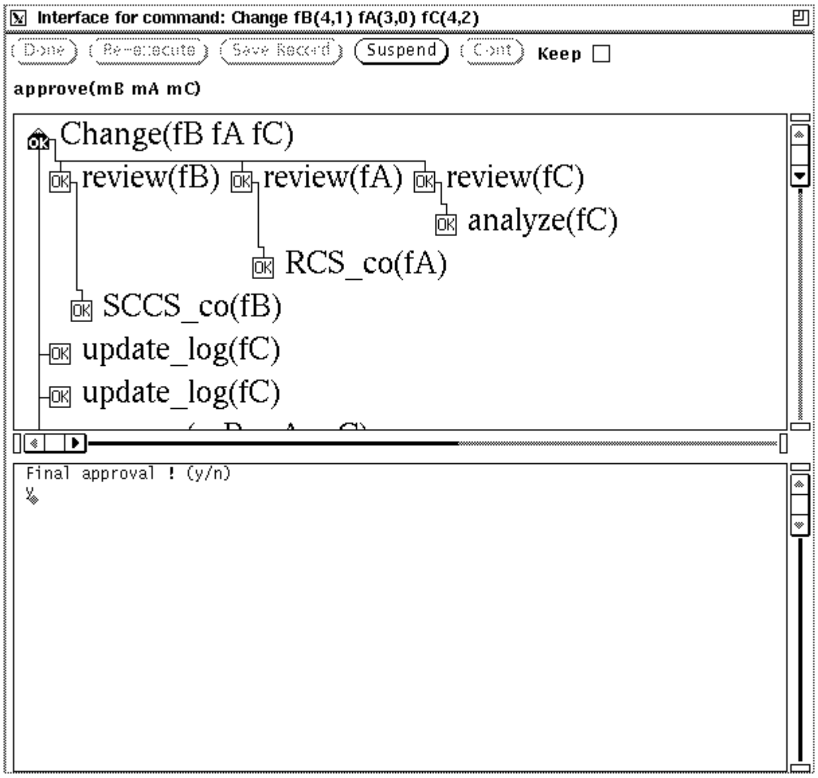


Figure 4.13: Execution Trace of Summit Example

tion of parameters seamlessly takes place as `modify` operates on the children of `approve`'s parameters.) When `modify` completes, the forward fan-out leads to the local testing phase, whereby `siteB` fires `inspect` followed by `auto_test` rules, `siteA` performs only `auto_test`, and `siteC` does `manual_test`. At the end of this local chaining, the final `integration_test` Summit rule is fired, to complete the composite-Summit.

As presented in Section 3.4, it is possible that at any point during the enactment of such a process, some of the local operations do not succeed and a totally different ex-



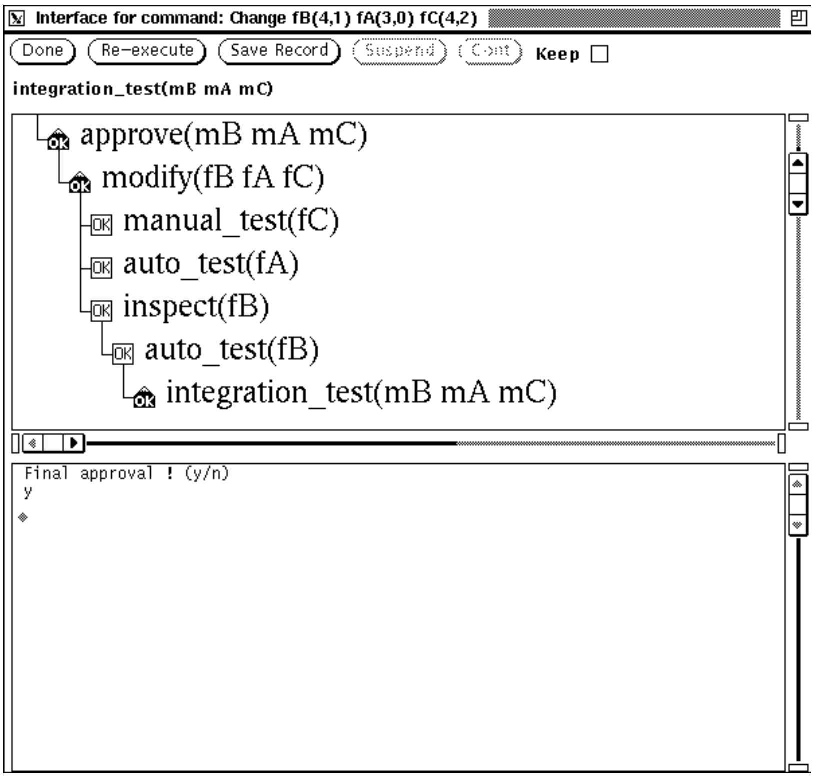
```
Interface for command: Change fB(4,1) fA(3,0) fC(4,2)
approve(mB mA mC)
Final approval ! (y/n)
y
```

Figure 4.14: Oz Animation of Summit Example (a)

ecution trace is produced. For example, if any of the local reviews fails (i.e., the change is not approved at one site), then the `Approve` rule should lead to a revision session that leads to a second review (such a scenario is shown in Chapter 6). And it could also be the case that local or global unanticipated exceptions in the process lead to dead-ends from which the process cannot proceed. But this again indicates merely that there is a flaw in the process model. While system exceptions should be handled by Oz, process exceptions and mismatches between the different processes are the responsibility of the SubEnv administrators.

4.4.5 Transactional Semantics of Summit

This is an important aspect of the Summit model, which was deferred until now mainly due to lack of detailed-enough context in which to discuss it. Clearly, some transactional properties are desirable in the execution of Summits. We are concerned here mainly with the *atomicity* property of transactions, which can be stated as a grouping of operations



```
integration_test(mB mA mC)
└ approve(mB mA mC)
  └ modify(fB fA fC)
    ├ manual_test(fC)
    ├ auto_test(fA)
    └ inspect(fB)
      └ auto_test(fB)
        └ integration_test(mB mA mC)
```

```
Final approval ! (y/n)
y
```

Figure 4.15: Oz Animation of Summit Example (b)

such that the outcome of their execution has all-or-nothing semantics. At the very least, the execution of a single Summit rule should be atomic. This by itself requires support for distributed transactions. The situation becomes more complicated if some degree of atomicity is desired during a Summit task including pre- and Post-Summits, where there is a need to handle simultaneous or overlapping execution of subparts of the atomic task at multiple sites. Finally, supporting atomicity of composite Summits, i.e., across Summit rules, might also be desired in some cases, and requires yet more transactional facilities.

This section focuses on defining the desired transactional semantics of the Summit model. The general solution of *how* to build a transaction mechanism that addresses the needs described here is beyond the scope of this thesis and is presented by Heineman in [49]. The transactional semantics in Oz are tied to the notion of *atomicity* vs. *automation* chaining which were inherited from Marvel and extended in Oz. Thus, we begin with a summary of this model in Marvel, and proceed with the extensions made in Oz.

4.4.5.1 Atomicity vs. Automation in Marvel

Both in Marvel and in Oz, a single rule is always an atomic unit. It is the smallest unit for which the atomicity property holds. This, however, does not mean that the rule actually executes atomically, only that the outcome of its execution is all-or-nothing. In fact, rules with activities never really execute atomically, because the server sends the activity for execution at the client and switches its context to service other clients (see Section 5.4).

The interesting issues are with respect to the transactional semantics of *chains* of executing rules. What makes this form of execution interesting from a transaction perspective is that the set of rules being executed is discovered dynamically, and is not known a priori.

atomicity chains support all-or-nothing execution of a chain of rules[12]. In contrast, automation chains support tasks of activities that are logically related to each other, but do not require atomicity. That is, if a rule is aborted during automation chaining, only that rule is rolled-back, not the whole chain of previously executed rules. Atomicity chaining is usually associated with propagation of values that retain some complex consistency constraints in the process (as opposed to the simple constraint embodied in each rule's condition) whereas the automation chaining supports the execution of long-duration tasks, when it is not reasonable to rollback work that was done over large period of time (hours, or even days) just because a rule along the chain has aborted. The transaction support for this model in Marvel is described in [17, 6, 68].

Recall that the implementation of this model in Marvel was based on annotations made to predicates of rules. Atomicity chaining is realized by ensuring that chaining from an atomicity predicate in an asserted effect to rules with satisfied conditions and empty activities[13] is mandatory; if it fails, the corresponding transaction rolls back. In contrast, chaining from an automation predicate (or into any rule with a non-empty activity) is optional and can be explicitly restricted through `no_forward`, `no_backward` or `no_chain` directives on individual automation predicates. If, during execution, automation chaining fails, only the updates of the failed rule are rolled-back, without affecting the outcome of rules that completed execution previously in that chain. However, since the type of chaining is based on attributes, it is possible that a certain rule in a chain will trigger both automation and atomicity chains from the same or from different attributes. This poses a

---

[12] In some of Marvel publications they are called consistency chains, but this is a misnomer.

[13] The restriction to rules without activities is not inherent, it is only a limitation in the implementation.

problem because there could be some overlap in the data accessed by the rules in the chain. This means that if atomicity "wins", then in cases of roll-back it will erase automation effects, thus violating the semantics of automation chaining. And if automation "wins", atomicity is simply not preserved.

The solution employed in Marvel is to execute all atomicity chains (also termed consistency implications) first, and any automation rules which are encountered during the atomicity chaining are queued (first-in-first-out). Once all immediate atomicity chains complete and commit their work, the queued rules are inserted into the execution rule stack (initially in the same order in which they were queued), and automation chaining commences. Note that automation chaining can lead to further atomicity chains, in which case they are again executed atomically and queue all encountered automation rules, and so forth. One way to look at this form of execution is as a chain of automation rules, with occasional "bursts" of atomicity chaining.

4.4.5.2 Support for Atomicity and Automation in Oz

The goal in the design of transaction support for Oz was to preserve the transactional semantics of automation and atomicity as in Marvel, and to properly extend them to Summits. Moreover, just as Marvel supported the ability to define the granularity of atomicity on a per-task basis using annotations in rules, so does Oz allow to define either of the three possible granularities — a single Summit rule, a Summit rule enclosed with pre- and post- Summit rules, and composite Summits — using similar annotations.

The execution of a single Summit rule is modeled as a distributed transaction, preserving the atomicity of a rule. This includes support for two-phase commit protocol and distributed abort, and involves interaction between local transaction and lock managers (the details are in [49]).

The Summit model introduces few new kinds of chains:

1. Automation chain from a Summit rule to a local rule ("local" in this context means non-Summit, it does not mean that it necessarily executes at the coordinating SubEnv). We will refer to this as Summit-to-Local-AUtomation chain (SLAU).

2. Automation chain from a Summit rule to another Summit rule. (Summit-to-Summit-AUtomation, or SSAU)

3. Atomicity chain from a Summit rule to a local rule (SLAT).

4. Atomicity chain from a Summit rule to another Summit rule (SSAT).

The semantics that are associated with the above new kinds of chains are as follows: The first two automation cases, SLAU and SSAU, are handled similarly to the local case, i.e., the chained rules are transactionally independent of the Summit rule that triggered them. The interesting cases are those that involve atomicity, namely SLAT and SSAT.

Summit-Local-ATomicity (SLAT)

One way to view SLAT chains is as a direct extension of local atomicity chains, and therefore to treat SLAT chains as part of a standard distributed transaction. That is, the global transaction commits only if all local sub-transactions commit (using the 2 phase commit protocol), and any local abort leads to a roll-back of sub-transaction at all sites and the global transaction. However, this "standard" approach has a serious flaw with respect to the semantics of Treaties and Summits. Since the local rules (to which the Summit rule chains) are not necessarily part of any Treaty, thus not explicitly "signed" or even known to exist in other sites, their effect on the Summit must be limited (consider, for example, a local "malicious" rule that always aborts and therefore causes the Summit rule to be rolled back). An alternative approach, therefore, is to make the local rules abort-independent from the global Summit transaction. That is, if a local rule aborts in the midst of SLAT chaining, the effects of the local chain are rolled back, but the state of the Summit transaction and other local sub-transactions remain in general intact. An abort at the Summit transaction, however, still entails local aborts. Moreover, local sub-transactions are still commit-dependent — they cannot commit unless the Summit as a whole commits successfully. This model seems to fit well with the semantics of Summits, but it also introduces a problem: Since a Summit rule made an assertion on local data as part of its effect and the local atomicity chaining has aborted, the local SubEnv might be regarded as being in an inconsistent state. One solution to this approach is to follow the nested transaction model [79], and replace the failed local transaction by a sub-transaction that commits (e.g., retrying the local transaction) in order for the Summit transaction to commit. But this approach might hold up the Summit and is complicated to realize. Another solution is to roll-back all updates that were made on the local data of the aborting site, including the updates of the Summit rule (which are maintained locally anyway). This approach retains local process consistency, although it might produce global process "inconsistency", in that the effects of a Summit rule are completely undone in the aborting site and are completely done in all other sites. However, autonomy concerns outweigh global concerns in SLAT, and therefore global process consistency is not considered to be preserved under SLAT chaining. In order to obtain global process consistency, SSAT chaining should be used.

Summit-Summit-ATomicity (SSAT)

SSAT has the strongest notion of global atomicity. It indicates that several Summit rules, all of which have been Treatified and thus known at all participating SubEnvs, are bonded to each other atomically. Note how this is different from the SLAT case, where the chained-to local rules are not part of a Treaty. The semantics of SSAT chains are that all operations made during SSAT chaining are fully atomic. Thus, we can distinguish "local" atomicity (this includes pure single-server execution that is unrelated to any Summit) in which local transactions are affected by the Summit transaction, but cannot affect it, and a stronger "global" atomicity that ties several SubEnvs and ensures true atomicity in all involves SubEnvs.

Order of Execution

The last issue concerns the order of execution. The ordering between local and Summit rules regardless of atomicity/automation concerns was already discussed in the context of the "Summit branching" policy in Section 3.7.1. As for automation vs. atomicity, arbitrary interleaving of automation and atomicity chains across SubEnvs would violate the corresponding semantics, similar to the problem in the single-server case which was covered in Section 4.4.5.1.

Therefore, the order of rule execution combines both concerns, consisting of an atomicity phase followed by an automation phase, where each phase alternates between global and local modes following the standard Summit branching policy. More specifically, when a Summit rule completes, all local atomicity rules first execute in the participating SubEnvs, queueing (locally) any local automation rules encountered. When all SubEnvs complete their local atomicity, the next Summit atomicity rule (if any) is fired, followed by all local atomicity, and so forth. When the Summit atomicity phase completes, a global commit occurs. The next step is to fire all automation chains. Again, the Summit automation rules fire first, followed by the local automation chains, followed by the next Summit automation, and so forth. Just like in the single-server case, any local or Summit automation can trigger various local atomicity chains which are executed as they are encountered, recursively, and Summit automation chains can also trigger atomicity Summits.

The idea is the same, and applies to both the single-server and the extended multi-server models: While in automation mode, any encountered atomicity chains (and all of their atomicity implications) are executed immediately, and while in atomicity mode, any automation chains encountered are queued for later execution.

As an example, reconsider the enactment of the motivating example which was given earlier in Section 4.4.4. There, the `update_log` rule was actually spawned off an atomicity predicate in the `Change` rule, to enforce an invariant in `siteC` that states that every change to a source file must be logged (so, if there is no log record the file has not changed). This is an example of SLAT chaining. Thus, if `update_log` aborts (e.g., due to a conflict), `siteC` rolls-back the updates which were made to `fC` by both `update_log` and the Summit `change` rule. However, the other effects of `change` are not undone. This might lead to a different execution trace — for example, if the condition of the Summit `approve` rule requires that all the relevant source files will be modified — but it doesn't violate any process consistency. In particular, a similar execution trace could occur if the user that invoked the `change` rule simply didn't update `fC`.

In the enactment of the motivating example there is no need for real SSAT chaining that would bind atomically a set of Summit rules (and all their implications). Indeed, it seems that the decentralized nature of modeling and enactment does not lead to many occasions where such modeling is needed. One extension might have been to add an atomicity chain from `approve` to a `notify_managers` Summit rule, so that failure in the latter rule would roll-back the effects of `approve` and all other SLAT chains (if any) which fired off `approve`. In that case, all SSAT and SLAT chains would have fire before `modify` which is connected through SSAU chain.

4.4.5.3 Local Tailoring of Rule Annotations

This final aspect of atomicity in conjunction with autonomy has to do with local control over rule annotations. Recall that MSL supports two kinds of annotations on rules. One kind is the atomicity and automation annotations discussed above. The second kind is *chaining directives* (discussed in Section 2.2.3) — the annotations that control backward/forward chaining to/from rules. While we discuss here only the first kind of annotations, similar arguments, problems, and solutions apply equally well to chaining directives.

The general motivation should be to allow local tailoring of rule annotations. Since they are an orthogonal dimension to the rules themselves, used for specifying chaining *among* rules, such tailoring does not conceptually violate the "common-subprocess invariant". Moreover, such tailoring is important for autonomy concerns. The main reason is that a local process should control the local impact of a Summit. For example, if a local process imports a Summit rule that has an atomicity effect predicate, it might be desirable to "weaken" the atomicity of the predicate in the local version (i.e., to turn it into an automation predicate) to avoid *local* implications that are undesirable to that site. Similarly, it might be desirable at times to "strengthen" the local process consistency by replacing some automation predicates in the local version of a Summit rule with atomicity predicates. For example, this would allow to get the desired SLAT behavior from `change` to `update_log` even if `change` did not have originally an atomicity predicate.

However, there are some obstacles to that approach: (1) A conceptual problem is with respect to SSAT chains. There are really two reasons for annotating a Summit rule with an atomicity predicate: to specify connections to other Summit rules (SSAT), and to specify connections to local rules (SLAT). While the latter should be controlled autonomously, the former is a global constraint that is inherently part of the common sub-process. The only solution to this problem is to extend the lexicon of the annotations to distinguish between global and local atomicity annotations. (2) Technically, the Treaty mechanism must be able to distinguish between alterations made to annotations and other alterations which constitute violation of the "common sub-process" invariant. (3) Finally, an implication of this requirement is that local SubEnvs would need to maintain their own (possibly slightly altered) copies of the original Treaty strategies, which would in turn require to address the associated problems that do not exist with the "single-source" approach (as explained in Section 4.3.1.2. Once these technical issues are resolved, the support could and should be added to enhance the autonomy of local processes.

4.5 Modeling and Enactment of Delegation and Multi-user Tools

Delegation and groupware support were discussed at the generic level in Section 3.6, focusing on how they could fit in the Summit model. We discuss here general issues regarding modeling and enactment of delegation, not necessarily across sites, and the particular realization in Oz. Note, however, that we have conducted only preliminary investigation of this subject, which is a major topic for future research (see Section 7.2).

In general, our main interest is neither in inventing actual multi-user tools, nor in generic human-computer interaction support. It is specifically about *process support* for modeling and assisting in the interactions between multiple users of the PCE. Further, we restrict here the multi-user tools discussion to *synchronous* tools, i.e., tools that require simultaneous participation of multiple users at the same time (or with bounded delay), such as multi-user editors, virtual white-boards, and so on. Without discounting the importance of integrating all kinds of groupware activities and user interactions discussed into PCEs, it seems that activities that are synchronous in their nature provide more opportunities for enactment support (mainly automation), and thus they are the subject of this section. For related work on infrastructure support for multi-user asynchronous tools (e.g., "large" tools that are themselves systems, like databases) see [108].

4.5.1 Modeling and Enacting Delegation

The need for delegation is clear: since delegation of tasks is commonplace in multi-person organizations, multi-user PCEs should enable users to delegate certain tasks to other users and/or other machines. Moreover, in multi-site PCEs, local activities that are executed as part of Pre- or Post-Summits must often be operated by local users and/or local machines to preserve autonomy/security of their process. Consider, for instance, part of the motivating example: A multi-site change activity is preceded by a local review phase which must be performed at each site by the local person who is responsible for the document being reviewed. Clearly, the review task has to be delegated to the proper reviewer, and run in his/her local machine. At other times the combination of special resources (e.g., a special-purpose computer) and special users at remote sites might require the delegation of an activity. Another case is when non-interactive activities are "delegated" to a proxy

	Local	Remote	Type
1		Machine	"Proxi" delegation (non-interactive)
2	User	Machine	Machine delegation (export display)
3	Machine	User	User delegation (export display to remote user)
4		User, Machine	Full delegation

Table 4.1: Delegation Types

client[14] at a remote site that is nonetheless local to its data, thereby saving the overhead of transferring the data to the site where the original request was made. Finally, interactive activities could still be delegated to a remote machine for any of the above reasons with the display being exported to the user. The various user/machine delegation combinations are summarized in table 4.5.1 (Local vs. Remote is with respect to the Initiating User). We focus in this section on user and full delegation (cases 3 and 4 in the table); machine-only delegation has been explored by Valetto in [108].

Delegation can be ad hoc or process-based. Ad hoc delegation assumes that a unit of work has been pre-assigned to a user (or a set of users) and that he can manually assign (parts of) it to other user(s). This approach is relatively straightforward, and can be realized "outside the process" by built-in commands that allow to transfer tasks across users, provided that there exist a persistent mechanism in which to store these requests (e.g., a user agenda); such approach was taken by Tong et al. in [107][15], and we do not discuss it here any further. In contrast, process-based delegation assumes that delegation has been modeled in the process, and therefore the PCE with such instantiated process model can assist in determining which tasks are delegated under specific circumstances, and who are the possible delegatees, depending on the current state of the project, the particular artifacts being manipulated, and the available users. Moreover, the process engine can assist in setting up the delegation, recover from failures (e.g., refusal to accept the delegation, see below), and so on. The main point is that the delegation is captured in the process model, and consequently it can be supported in various ways.

Process-based Support for delegation consists of two aspects: (1) modeling the work itself, delegatee(s), tools, and artifacts involved; and (2) supporting the enactment of the

---

[14] A proxy client mechanism in Oz was implemented by Peter Skopp originally to support low-bandwidth clients, but was later generalized for other purposes including delegation; see [98] for details.

[15] Tong also describes support for hybrid form of process-based delegation, where the delegation is modeled in the process but the delegated work is still entered into an agenda rather than executed immediately by the delegatee.

defined delegation. This includes locating delegatees, redirecting tasks to them, notification mechanisms (for both delegators and delegatees), and setting up the environment to enable invocation of tools on local or remote machines.

Regarding modeling, there are several issues to explore: (1) what is the granularity of delegation that the process should support; (2) how to model users in general, and how to specify and determine the delegatees; and (3) what are the operational semantics of delegation.

Interestingly, the same set of issues come up in supporting multi-user tools, except they require different solutions there, as will be seen in Section 4.5.2.

4.5.1.1 Granularity of Delegation

Granularity of delegation can range from a single activity, or rule, to a complex rule-chain that consists of many related activities, some of which may themselves be delegated. Delegation of whole sub-chains may be particularly attractive for pre- and post- Summits, where whole local chains could be delegated to local users. However, this subject is yet to be explored (for a partial treatment of task delegation see [107]); here we focus on the simpler case of activity delegation.

In modeling delegation of individual activities, the specifications can be made within rules, and as such, all the modeling power of Oz rules can be used for delegation — rule bindings can be used for *dynamic binding of delegatees*, provided that users are represented as objects in the objectbase; rule conditions could be used to check if delegation is possible; rule effects could be used to update information regarding the delegation; and backward chaining could be used to automatically search for delegatees by invoking other logically related rules that notify inactive delegatees.

Essentially, modeling delegation involved the addition of a new `delegate` operator that accepts representations of user(s) as operands. The `delegate` operator is defined and evaluated after the binding section and before the condition section of a rule, for reasons that will be explained below. At run time, the rule processor tries to establish the delegation (explained below) and if the condition of the rule is satisfied, the (non-empty) activity is redirected to a qualified delegatee, provided that there exists one with an active client (otherwise, the activity might be stored in an agenda, as suggested earlier.). When the activity completes, the rule processor switches back to the delegator, and subsequent activities from chained rules are directed to him, unless further delegation operations occur, and so on.

4.5.1.2 Modeling and Binding Delegatees

The main goal in the design of delegation in Oz was to not hardwire the delegatees in the instantiated process, but rather to bind them dynamically to rules. The main advantage of this approach is that different (sets of) users can be bound to a rule depending on the context in which the rule is invoked, and particularly depending on the specific set of objects that the rule manipulates. For example, a review rule could specify that its activity should be delegated to the owner of the document; then, depending on the document, each time the rule fires, it will be delegated to the appropriate owner. Moreover, if the owner of a document changes over time, subsequent invocations of the rule on the same document will automatically bind the rule to (one of) the new owner(s), because the delegation is specified to the owner(s) of the document, not to a specific user.

Dynamic user binding was achieved in Oz relatively easily, using the normal data binding facilities. Delegatees are represented in binding predicates by an `object.attribute` pair, with the attribute restricted to being of type `user` — a primitive Oz attribute type that accepts as correct values only valid (operating system) user-ids. Note that there is no restriction on the kinds of objects that the `delegate` operator accepts. An alternative approach would be to rely on some well known user repository. That is, users would be represented by objects in the objectbase and instantiated from a `USER` class, and the user repository could store information that could be used for various purposes, not only for delegation binding. This user repository approach is superior in terms of user modeling, and also solves the problem of relying on the operating-system's user-id which might not be unique across domains. In fact, such a user repository has already been used (optionally) for access control purposes [68], but has not been been applied to delegation yet and is a topic for future work (see Section 7.2.3).

Regarding bindings, it is desirable to be able to specify a set of delegatees (not only one) for two reasons: to enable simultaneous delegation of an activity to multiple delegatees (which is exactly the mechanism used for multi-user tools), and to provide for multiple potential delegatees to select from. Binding to a set of users is also achieved "for free" when using the normal binding facilities, since they allow to bind a set of objects to

```
1) analyze_bug[?tr:TEST_RUN, ?c:CFILE]:

2) (and
3)   (forall MODULE ?m suchthat (member [?m.cfiles ?c]))
4)   (exists WORKSPACE ?w suchthat (linkto [?w.module ?m])))
5)  :
6)  delegate[?w.owner]:

7) (?c.bug_status = Suspected)

8)  # Prompt the user whether the bug is here (so return 0) or not (Return 1)
9)  # also, generate a change request in the CFILE
10) # For the demo should have a small request already written
11) # ---------------------------------------------------------
12) { ANALYZE_TOOLS analyze_cfile_bug ?tr.report  ?c.change_request
13)                                   ?c.contents ?c.bug_report }

14) (and no_chain (?c.bug_status = Defected)
15)               (?tr.report_status = Confirmed));
16) (?c.bug_status = Clean);
```

Figure 4.16: Delegation Example

a symbol.

Figure 4.16 shows a rule with a delegation specification[16]. The activity involves analysis of a source file that is suspected of having a bug, so it is delegated to the person who "owns" that file (i.e., the author of the file). This is expressed in the rule by issuing a `delegate` operator (line 6) and binding to it the owner of the `WORKSPACE` (`?w`) that is linked to the `MODULE` (line 4) that contains the "suspected" `CFILE` object (lines 3). Note how the same rule would be delegated to different users, depending on the `CFILE` object on which the rule was invoked.

4.5.1.3 Semantics of Delegation

Having established the binding to users still leaves some aspects of the delegation open:

---

[16] This rule is taken from the "ISPW example" process, see Chapter 6.

1. How to determine the single delegatee in the case of a set of candidates (the problem of simultaneous delegation to multiple users is addressed separately in Section 4.5.2)
2. how to react to, and recover from, cases where delegation fails.

The first issue comes up only if there is more than one delegatee. The case where there are no candidates at all (i.e., the symbol is bound to the empty set) is treated as a failure (see below); another special case is when one of the delegatees is also the delegator, in which case the delegation is void, and the rule is treated normally. In the general case we identified two methods to direct the selection:

1. `random` — choose an arbitrary candidate. A slight alternative is to choose the "first" candidate in an implementation-specific order; this still allows process engineers to predict the order in which delegation will be attempted.
2. `interactive` — allow the user to choose the candidate, from the (sub)set of users that are active in OZ.

In either case, the process engine should filter out inactive candidates (i.e., users with no active clients). In addition, the interactive mode which is designed to refine the control over delegation, must have an option to not choose any of the available active users in the binding set. Currently, OZ implements only the `random` method, and consequently does not support the explicit specification of method choice in the definition of delegation.

The second issue is the failure semantics for delegation. A delegation operation is considered "failed" if either there is no active user among the candidate set, or none of the available users is willing to perform (that is, immediately) the delegated activity (the mechanism to determine the latter is described in Section 4.5.1.4). Failure is indicated by treating the `delegate` operator not only as a binding but also as a boolean predicate that returns `true` if delegation succeeded, and `false` if it failed. Thus, as far as the process is concerned, delegation failure is equivalent to a failure to satisfy the condition of a rule, and thus prevents the process engine from executing the activity of the rule.

At first glance, one might be inclined to associate no further semantics with failure beyond the normal rule failure semantics. However, as we discovered, delegation failure may require further actions. We identified three, non-mutually exclusive actions:

1. `store` the delegation in the delegated user's agenda. The delegatee(s) can be chosen using similar techniques as above. This action applies to both kinds of failure (i.e., no users vs. declining users).
2. `notify` the delegated user(s), for instance by e-mail.
3. `delay` the delegation and retry later.

To distinguish between the various options, they should be supplied as either environment variables or as PML directives to delegation, and in any case optional.

The "compensating" operations above are limited in the sense that they do not make the failed rule satisfiable. There are two ways in which the PCE could still attempt to proceed. The first and simple method involves a "programming trick" — the process engineer could write an alternative rule that would be triggered if the delegated rule failed. The idea is to match the conditions of both the delegated and the compensating rules and order them so that the delegated rule is evaluated first, and if it fails, the non-delegated rule is fired alternatively. (OZ allows for ordering multiple rules with different conditions whose signatures match the same user command.) Obviously, this option only makes sense in cases where there is an alternative path to follow; if the failed step is an "articulation point" in the possible "execution graph" (i.e., the process must pass through this rule), then nothing can be done. An example where this technique is used is given in Chapter 6.

The second and more natural approach to address delegation failures in a rule-based PML like OZ is to attempt to satisfy them by *backward chaining* to other rules that could potentially satisfy the delegation operation, analogous to normal backward chaining. The fact that delegation is modeled as a boolean predicate facilitates this approach, since it could logically match with a "delegation" effect of another rule, thereby possibly chaining to it. For example, a generic "wakeup" rule could be chained off a rule with failed delegation and activate inactive users by firing an activity that would notify the relevant users (if they're logged in at all, of course) of the requested delegation.

Another useful extension to the PML builds on the fact that delegation is represented as a predicate in the PML. The delegation construct could be extended to support complex logical clauses. For example, a disjunction of delegation predicates would allow to bind a set of delegatees from several variables instead of one, and would ease the specification of delegation that is otherwise limited to a single variable; conjunction of delegation predicates could be used to bind users bound to all predicates, and so forth. This and the "chaining" extensions to delegation have not been realized yet, and in general require further investigation.

4.5.1.4 Infrastructure Support for Delegation

The above discussion made some implicit assumptions about the capability of the system to support several operations: (1) how to locate users specified in delegation, and how to identify if they are "active"; (2) how to redirect activities across clients; (3) how to notify users (both delegators and delegatees) and in general support the interface to delegation. We briefly discuss these issues here.

A user is considered active if he has at least one client associated with his user-id that is connected to the process server from which delegation is issued. That client can be either local or remote to the delegated server (see Chapter 5 for more on the architecture), but it must be connected to it. In this scheme, checking whether a user is active is simply done by maintaining in the server an internal client-table, and searching clients with the specified user-id. Note that with the `user` attribute method (currently employed in Oz), Unix user ids may not be unique across sites, in which case site prefixes could be used to provide uniqueness. An extension to this scheme is to consider any client that is connected to *any* server in a multi-site environment as a potential delegatee. However, this entails significant overhead in discovering potential candidates, and has not been explored yet.

As for activity redirection, this is mostly a low-level implementation-specific issue. There are two important points to mention here. First, the thread in the delegator's client that issued the delegation blocks until the delegatee completes the execution of the activity; this, however, does not prevent the delegator's client from issuing other activities from other threads. Second, when the delegatee completes execution of the activity, the return code of the activity is redirected as if it came from the delegator client. Thus, most components of the rule processor are shielded from this redirection; the rule processor is "fooled" and for the most part is not even aware of the delegation.

The next issue is the general user interface support for delegation. Delegation is somewhat unusual in terms of user interface, in the sense that the delegatee is asynchronously (and perhaps might unexpectedly be) notified about the activity. Thus, the user interface must then have a way to (1) attract the delegatee's attention to the delegated activity; and

(2) enable the delegatee to reject or at least defer the execution of the activity, with optional directions to store them for later execution, as discussed above.

The last issue concerns the dynamic visualization of the process; since Oz supports dynamic animation of the executed process, the question is how to redirect animation messages. Our approach is to retain all process animation with the delegator, and redirect only the activity (and its enclosing user interface) to the delegatee(s). The rationale is that in activity delegation the "ownership" is only temporarily transferred to the delegatee for the execution of that activity, but in general control returns to the delegator when the delegation terminates. Note how in this case delegation of coarser granularity (e.g., task delegation) would require a different solution, probably to redirect the animation as well as the activities to the delegatees.

4.5.2 Modeling and Enactment of Synchronous Multi-User Tools

Considering process support for synchronous multi-user tools, there are two main issues to explore:

1. Specification and parameterization of multi-user activities besides user binding considerations; and
2. Execution semantics, including selection of a user-subset.

Basically, the invocation of a multi-user tool is initiated by the invocation of a rule (either manually by a client or through chaining) that encapsulates a multi-user activity. An activity is denoted as multi-user in the tool definition section of the strategy[17](see Section B.2.3.3 for an example). That rule must also contain a user-binding specification for binding the participants. If proper binding is made, and the rule's condition is satisfied, the activity is invoked. When the activity finishes, the rule proceeds regularly and continues to be associated with the client that initiated the activity. We now discuss bindings and invocation in more detail.

4.5.2.1 Semantics of User Binding

As already mentioned, the same mechanism used for delegation can also be used for binding users to a multi-user activity. However, there are several differences regarding the

---

[17] Borrowed from the language extensions which were made in [108].

policy of an acceptable binding set.

First, the activity must be delegated to a set of at least two users and the initiator may or may not be treated as an implicit participant (regardless of whether he is specified in the binding set). Second, both the `random` and the `interactive` options for user selection must be implemented in a different manner here: the latter has to allow selection of a subset of participants (not just one), and the former has to choose a subset of participants. To further assist in both the automatic and the interactive binding procedures, the modeling of user binding should also be extended to allow the specification of `minimum`, `exact`, or `maximum` number of participants required for a certain multi-user activity, with the default being all the users in the binding set.

4.5.2.2 Invocation of Multi-user Activities

When a multi-user activity is about to execute (assuming the condition of the rule is satisfied and the proper users are bound to the activity) OZ conceptually replicates the activities in all participating clients[18], and uses a similar user interface as for delegation in order to notify the participants of the activity. Then, each client invokes the activity in its own address space. However, OZ does not interfere or otherwise support the communication at the tool level, which is considered the responsibility of the (multi-user) tool itself. In order to "tie" the tools to the process(es), the standard tool envelopes [38] can be used to bind information from the process to the tools (see for example, the `white_board` envelope in Appendix B.2.5.4), or our new enveloping mechanism for asynchronous tools can be employed.

When the activity completes, all return codes except the return code from the initiator clients are ignored. In that respect, multi-user tool support is simpler than delegation. An improved implementation should investigate how to incorporate return codes from all participants in order to form a single representative return code (e.g., majority vote, negotiate).

One of the important characteristics of process support for multi-user tools is its invocation point. In our approach, it was invoked from a single rule, associated with a single client; this is the simplest form. There are at least two alternatives which should be investigated: (1) simultaneous invocation of a Summit activity in the participating sites

---

[18] In practice, if all clients share the file system it only needs to send them a path of the envelope to execute.

(as opposed to one replicated activity), as outlined in Section 3.6; and (2) invocation from multiple independently executing rules — for example by local concurrently executing chains in post Summits.

Finally, another open issue is support for *roles*. That is, if a tool has the notion of different roles (e.g., a multi-user inspection tool with a moderator vs. participant, see Appendix B.2.3.3 for an example), then the process should be able to model this, and subsequently support it by, for example, invoking different envelopes for different roles. Currently, this has to be manually coded within the envelopes.

4.6 Implementation Status

Version 1.0 of Oz is fully operational, and most of the features which were discussed throughout this chapter are fully implemented. To summarize, Oz supports, over multiple sites, each with a private process model and a private objectbase, the following:

- Treaties — strategy-sharing operations (e.g., *import*) and their intersection with the execution privileges operations (e.g., *request*), global Treaty operation, local evolutions and dynamic verification.
- Summits — Direct remote interaction, all built-in cross-site commands, including support for sub-schema compatibility as described in Section 4.4.2.1, and most importantly, a full blown support for Summits as described in in Section 4.4.
- Delegation and Groupware — an effective (although still preliminary) implementation of both modeling and enactment of delegation and multi-user tools. See Chapter 6 for actual examples of using these mechanisms.

The aspects which were discussed in this chapter and are only designed or not fully implemented at the time of this writing are (a similar section appears in Chapter 5 to summarize features discussed there): global associative queries and soft links are not fully implemented yet; the `export_data` mechanism discussed in Section 4.3.4 has not been implemented yet, although the underlying access-control on top of which this facility should be constructed is fully implemented and operational; the solution to the "static problem" in common sub-schemas, which was contemplated in Section 4.3.3.1 — involving the Evolver's front-end to verify by-structure type equivalence among the common sub-schema — has not been implemented yet; checking for process consistency in Treaties using the Evolver as described in Section 2.2.5, is not implemented, mainly because some essential features in the Evolver itself are still incomplete; backward chaining during fan-out in Pre-Summits is carried out serially, and not in parallel, as suggested in Section 4.4.3.2 (forward chaining during Post-Summits is carried out in parallel, though); the various language extensions to support "delegation-chaining" and the extended logic in evaluating delegation, as well as the delegation directives to control the assignment of users and the failure semantics, have not been implemented yet.

5

Architectural Support for Decentralization in Oz

In the previous chapter we discussed the interpretation of the generic decentralized model into a specific PML and PCE; in this chapter we focus on the underlying *infrastructure* that supports such a realization. One way to distinguish the material in this chapter from the previous one is that the previous chapter discussed *interoperability* at the software-process level, whereas here we discuss mainly *interconnectivity* at the system level, on which interoperability is founded. Despite being "low-level", the discussion in this chapter is, for the most part, conceptual, focusing on the research issues and ideas that are concerned with the design of a decentralized architecture that supports the interoperability model and meets the requirements set forth in Section 1.5.

The chapter is organized as a collection of loosely-coupled issues which are concerned with different aspects of the system's characteristics. Section 5.1 describes an overall architectural overview. Sections 5.2 and 5.3 are closely related: the former discusses the communication infrastructure, and the latter discusses the process for (dynamic) configuration of the database that maintains connectivity information. Sections 5.4 and 5.5 are also somewhat related: the former discusses the underlying support for Summits in Oz with emphasis on the context switching mechanism and handling communication deadlocks, and the latter discusses the remote object cache that enhances the performance of Summits. Section 5.6 discusses the extensions to the Oz architecture for operation over the Internet, supporting geographically dispersed SubEnvs. Finally, Section 5.7 summarizes the current state of the implementation.

5.1 Architectural Overview

The external view of the Oz architecture matches the generic description as seen in figure 3.3, and has already been outlined in Section 4.1. It is a multi- client-server architecture, whereby each SubEnv follows a standard client-server architecture and is essentially self-sufficient for local work, and multiple SubEnvs are inter-connected through a communication layer that enables process-interoperability, with no shared-memory. As many aspects of the single-server architecture are similar to the Marvel architecture [17], they are therefore not discussed here any further.

The internal architecture of Oz is illustrated in figure 5.1. We use the following graphical lexicon, partially adopted from [63]: 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 which are part of an operating-system process but are relatively independent from other components; squared boxes with narrow solid lines are computational sub-components; dashed-line separators within sub-components further modularize a (sub)component into the its various functionalities; shaded ovals represent data repositories; and arrows represent data and control flow.

Oz consists of three main runtime computational entities: the Environment Server (or simply, the *Server*), the *Connection Server*, and the *Client*[1]. In addition, there are several entities that convert the various project-specific definitions into an internal format which is understood and loaded by the server, and some objectbase utilities for checking, repairing, and converting (across platforms) Oz objectbases.

There are three kinds of inter-connections: (1) client-to-local-server; (2) client-to-remote-server; and (3) 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. (An extension of this model, in which clients can be disconnected temporarily from their server, is investigated separately by Skopp [99].) In contrast, the two other connections can be regarded as "temporary", since they are

---

[1] There are actually three kinds of clients: XView, Motif, and a command-line client.

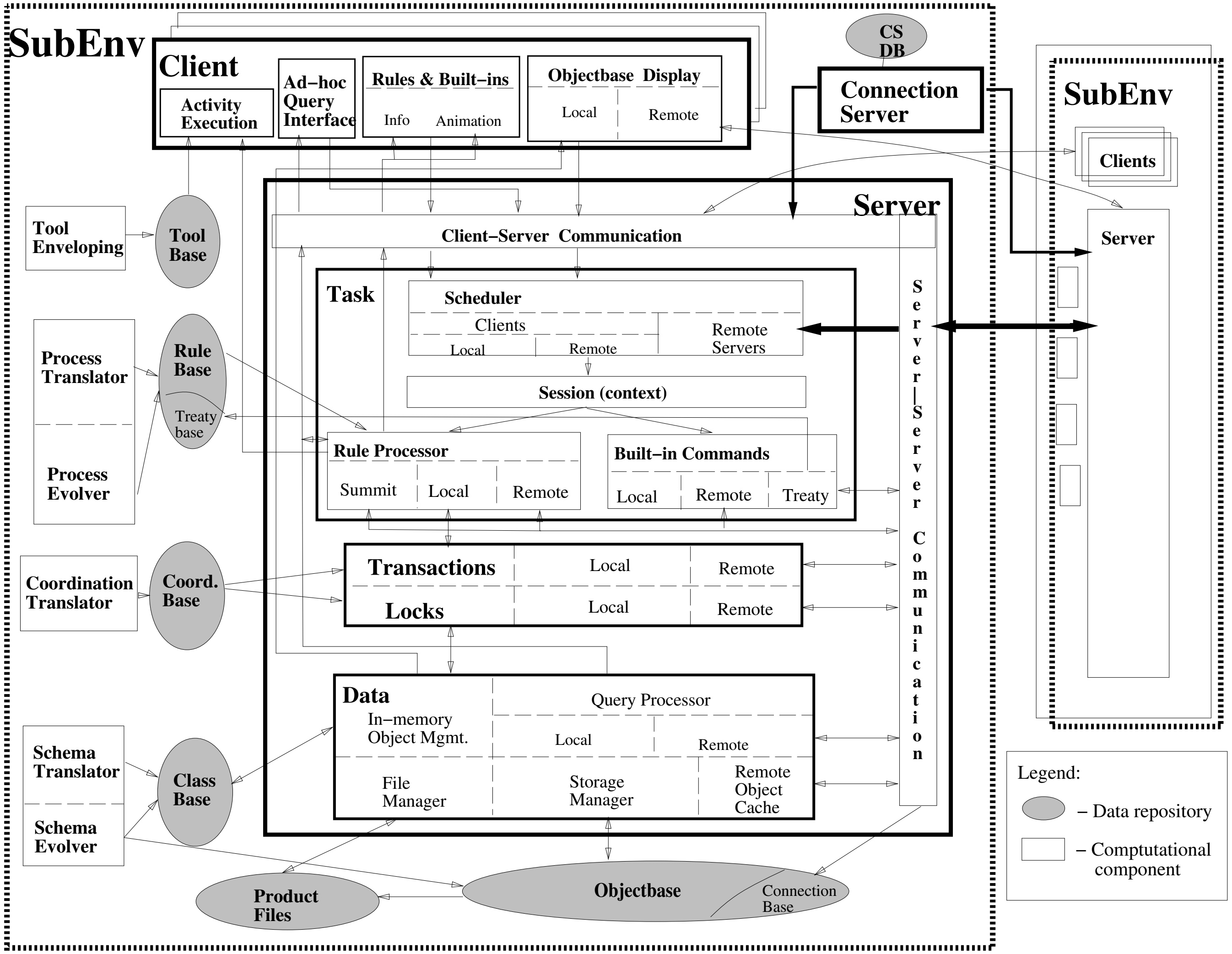


Figure 5.1: Oz Architecture

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 domains. Both kinds of connections are implemented as `tcp` connections, but with different operational semantics. When a permanent connection gets disconnected (either voluntarily or involuntarily due to some failure) the client ceases to exist and is removed from the local server's state. In contrast, when a temporary connection is disconnected (again, voluntarily or involuntarily), the system should still enable continuation of local work and other unaffected remote work, and in case of an unexpected failure, should recover gracefully.

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 the persistent product data and process state) is under the control of the server's object management system. Typically, an active environment has also at least one local active (i.e., executing) client connected to its local server, because the server automatically shuts itself down when there are no more active clients (and is automatically started up on demand by the Connection Server, as will be explained shortly). In the rest of this section we will interpret the architecture figure, so the reader is advised to refer to it throughout the section.

5.1.1 The Oz Environment Server

The server is the "brain" of Oz. It consists of three main distinct components: Task (or process), transaction, and data managers, each of which can be separately tailored externally.

5.1.1.1 Task Manager

The **task manager** is the main component in the server. Its front-end component is the **scheduler**. The scheduler 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 exceptions are explained in Section 5.4). The server is non-preemptive, i.e., it relinquishes control and context-switches to other tasks only voluntarily. The next layer below the scheduler is the **session**, or context, layer. Each interaction with a server is enclosed within a context containing information that enables to switch and restore contexts (again, see Section 5.4). The most common case of a multi-step task in Oz is a chain of rules. The context of a local rule-chain is kept in a data structure called the **rule-stack** and the context of composite Summits is maintained in a **Summit-stack** (mentioned earlier in Section 4.4.3). Most of the services provided by the server are handled either by the rule processor or by the built-in command processor.

The **rule processor** is the heart of task processing. It contains the necessary functionality for processing both Summit and non-Summit rules, including parameter binding and rule overloading, activity execution preparations (including converting the object-based arguments to their file-based counterparts used by external tools), and backward and forward chaining with either direction in one of three possible modes: (pure) local, Summit, and local but spawned off a Summit rule. In addition, it controls the bindings, condition evaluation, and assertion in rules (all preformed by the query processor) and all access to remote objects in either of the above phases (performed by the data and transaction managers).

The rule processor has very few "system" built-in rules (e.g., registration rules, see Section 5.3.1), so the behavior of a particular instantiated SubEnv is mostly determined by the external rule-base repository that it reads upon initialization. The rule-base contains the internal representation of the parsed administrator-defined rules, their inter-connections (i.e., the rule-network), their interface to envelopes, and all the Treaty information — imported, exported, requested, and accepted rules, as well as the corresponding sites with which those relationships hold.

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

5.1.1.2 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 database, and interacts with remote transaction managers to manage access to remote objects. Thus, only local locks are maintained at each local transaction manager. The transaction manager can be configured by one or more of the following mechanisms: (1) an external lock table, containing compatibility matrix, power matrix, and inheritance tables; (2) a transaction table that associates lock modes (from the lock table) with operations which are carried out during process execution; and (3) a set of control-rules that tailor the default two phase locking protocol for concurrency control (see [17] for (1) and (2). (3) is due to Barghouti [6]). As with rules, an instance of the transaction manager without proper lock and transaction tables is useless, but unlike rules, Oz provides default tables which are suitable in most cases (the control-rule base is entirely optional).

As outlined in Chapter 1, the implementation of this component is in general outside the scope of this thesis and is treated separately by Heineman [46].

5.1.1.3 Data Manager

This is the lowermost component in the server, consisting of several sub-components. The main sub-component is the in-memory **object manager** that provides a uniform object-based access to data from any system component. Objects can be looked up in one of three ways: by structural navigation, by testing class membership, and by their object-id. Thus, three different data structures are superimposed on the objectbase: a directed graph that represents the structure of the objectbase with edges labeled as parent, child, or link; a linked-list that contains all objects of a given class; and a hash-table keyed by the 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 as parameters to rules.

The second major sub-component is the **query processor**. It has a language interface, and is called from both the rule processor 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 manages access to file attributes (recall that file attributes in objects are merely paths to files which reside in the "hidden" file system); and an **object cache** that holds transient copies of remote objects when Summits take place — discussed separately in Section 5.5.

As far as modeling facilities, the data manager is defined by the project-specific schema which is tied to the instantiated objectbase, including both class- and composition-hierarchies. As in the case of rules, without a schema the data manager is useless since it cannot instantiate any objects (the built-in classes SUB_ENV, TOOL, and ENTITY are not suitable for general use).

Finally, one of the major research topics in the Oz project that is for the most part orthogonal to the work described in this thesis, is concerned with componentizing the server's main three functionalities (task, transaction, and data management) into independent and replaceable components, as well as integrating each component in other frameworks. Componentization, among other things, is important to support architectural design autonomy, where SubEnvs can be built with different components (e.g., different OMS, or different transaction management). While preliminary work towards componentization has been done by the author in the Marvel project (see [17]), it is in general outside the scope of this thesis, and is addressed in the theses of Popovich [88] and Heineman [46].

5.1.2 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 multi-threaded, in that a single client supports 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 consists of rule- and built-in menus, utilities for displaying rules, and the (local) rule-network, all of which are stored at the client's address space and can be dynamically refreshed when a new process is (re)loaded. Another informative utility is the Treaty information menu, which prints the state of the various active Treaties, import, export, request, and accept information, as it is known to the local site. But recall that Treaties can be invalidated unilaterally, which means that the Treaty information regarding remote sites represents only an approximation of their current state.

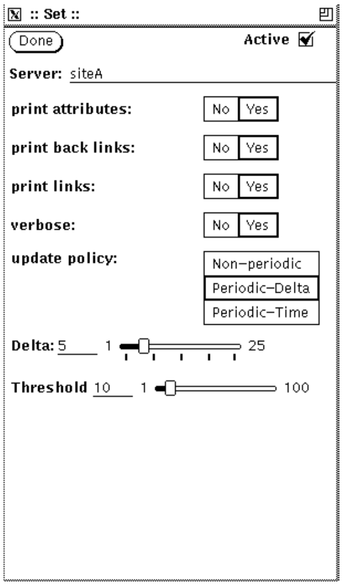
In addition to the display of the static rule-network, the client has a dynamic **rule animator** that animates the enactment of rules, including backward and forward chaining.

In Oz, the animator has been extended to support animation of Summits. The idea is that when local execution of rules at the remote sites takes place, the animation of these rules is still directed towards the client who initiated the Summit, so that a complete picture of the Summit is presented at the coordinating client. This occurs even when an activity is delegated to other clients. There are several architectural implications to this design: (1) There must be a complete separation between control and animation messages which are sent from a server to client(s); (2) when a rule executes as part of remote backward or forward chaining, (i.e., during Pre-Summit or Post-Summit, respectively), it must carry with it the identification of the coordinating client (which operates in a different SubEnv) in order to direct the animation messages to it; and (3) the remote server on whose behalf Pre- or Post-Summit takes place must be able to communicate with the coordinating client to direct to it the proper animation messages.

The **objectbase display** is the central component of the user interface, particularly with respect to multi-site interactions. In Oz, the client supports the display, browsing, and parameter selection from both local and remote objectbases, as was seen earlier in figure 4.2. (But recall that the client maintains an image of only the structural information for browsing and selection, not the full contents of the objectbase. Actually, this is also true for schema and rules: only their "names" are passed to clients for selection and display purposes, not their contents.) This implies that the client has to maintain multiple simultaneous connections to the remote servers, and be able to direct different requests to different servers. In addition, decentralization concerns imply that the policy concerning the refresh of the various images 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, and frequency of interactions with, servers. Thus, Oz supports a SubEnv-specific tailorable **refresh-policy**[2]. That is, a user can determine for each objectbase the frequency for refreshing the local image, thereby controlling the communication overhead. The policy itself can be based on time, or on number of updates made to the objectbase (by other clients, since a client that affects the state of the objectbase receives always the updated image immediately). The default policy, as with other aspects of communication in Oz, follows the "lazy" approach — the updates are deferred until users actively request services from the server, after which the updates are piggy-backed to the reply. Figure 5.2

---

[2] This feature did not exist in Marvel even for the single-SubEnv; instead, the refresh policy was hard-coded in the kernel.



Active ☑

Server: siteA

print attributes: No [✓ Yes]

print back links: No [✓ Yes]

print links: No [✓ Yes]

verbose: No [✓ Yes]

update policy: Non–periodic [✓ Periodic–Delta] Periodic–Time

Delta: 5 (1–25)

Threshold 10 (1–100)

Figure 5.2: Refresh Policy in Oz

shows the client interface to the refresh policy, where the `Periodic-Delta` mode is selected on SubEnv `siteA` with value 5, meaning that after each 5 updates that are made to the structure of the objectbase of `siteA` (e.g., adding or deleting objects) the changes are propagated to the client. The threshold value denotes the number of changes after which instead of sending the "delta" the entire objectbase should be sent. This is usually a function of the size of the objectbase.

The two remaining components of the client, namely activity execution and an ad-hoc query interface, are similar to their counterparts in Marvel, and even when remote objects are accessed in the activities their transfer is transparent to these components.

5.1.3 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 always active[3]. Thus, each configured host has its own (logical) Connection Server that supports all SubEnvs (of the same or different global environments) that reside in that host. The actual invocation and functionality of Connection Servers is discussed in Section 5.2.3.

5.1.4 Summit from the Architecture Standpoint

We summarize the section by an overview of Summit execution, describing how the system's components interact during the course of multi-SubEnv enactment, leaving out the details of two important topics which are discussed later separately, namely context-switching and caching of remote objects.

A user interacts with the environment through a client. He/she initiates a Summit by selecting a rule from the rule-menu, and by selecting objects for the rule's parameters, with at least one object from a remote objectbase (recall that only object-ids are known to the client, the real objects are resolved at the server). Prerequisites to such firing are: (1) The rule must have gone through a Treaty definition, and must have *request* privileges on the sites from which remote objects were selected; (2) An `open-remote` command was issued by the client on all remote objectbases from which objects were selected, or otherwise the remote objects would have been invisible to the invoking client.

The request is then sent to the local server (the details of the actual communication are deferred to Section 5.2) and enqueued for execution. At some point the scheduler dequeues the request for service and directs it to the session manager which creates a new context for the requested task. From there, the rule processor takes control. The first operation involves resolving the object-ids to real objects. The local objects are resolved through search-by-id, and the remote objects are fetched by invoking a server-to-server request, and are stored in the local cache. Once the objects are resolved, the overloading module determines which rule(s) should be invoked, followed by a server-to-server interaction to perform dynamic Treaty verification. The binding phase follows, calling the query processor, which in turn might submit queries to remote query processors. At this point, the rule processor calls the transaction manager to acquire locks on the binding set, and requests for locks on

---

[3] Actually, it is implemented as a daemon invokable from the Unix `inetd` mechanism.

remote objects are directed to the proper (remote) transaction managers. The condition evaluation follows, and if it fails, backward chaining is invoked, involving server-to-server interaction to notify remote servers to perform local chaining. If/when the server determines that the client has to execute an activity, it sends to the client the necessary data and animation messages. Then, the client's activity manager spawns an operating system process that executes the activity; when finished, the client returns to the server with the return code and output from the activity. The server's rule processor enters the Post-Summit phase, initiates remote forward chaining, and when all sites notify completion it proceeds with subsequent Summit rules, if any, and eventually completes the task, releasing all the resources associated with the task and removing it from the session manager.

5.2 Communication Infrastructure

The communication infrastructure is the cornerstone of the inter-connectivity mechanism and is, therefore, very 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 core research requirements impose several constraints on the design of the infrastructure:

1. Decentralization and independent operation requirements (which in turn entail a "shared nothing" architecture) imply that the communication information cannot 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 certain points in time, imply that the architecture should be designed to tolerate temporary disconnections between SubEnvs as a built-in normal scenario, not only as an exception. Moreover, since the communication address of the entities might change dynamically (due to the "tempo-

rary" 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. The flexibility requirement suggests that there should be some degree of freedom in modeling the communication on a per-project basis.

We begin with a high-level outline of the approach taken to address these issues, followed by the actual realization.

5.2.1 Approach

The key to addressing the two major issues given above requirements is in the proper design of: (1) a decentralized connection database and (2) a proper communication protocol that manipulates the 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 becomes simply 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 be at times 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 5.2.3).

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 modeling procedure. Second, communication involves low-level system calls and mechanisms that are hard to expose to the high-level modeling language.

The solution here is 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. And manipulation of the database is performed in part by low-level components of the kernel, and in part by (built-in) rules. The idea is to define a built-in structure of the database, but expose it and its contents to all levels of the system (and to users), and in particular make it modifiable via the PML, 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 Section 5.3). And even the class definition of the connection database can be augmented with additional attributes, so long as the default required attributes are intact. We now turn to the actual solution employed in Oz.

5.2.2 The Oz Connection Database

The implementation of the connection database in Oz follows the rationale given above. Each SubEnv maintains a private connection database consisting of a set of objects of the built-in class `SUB_ENV`, each of which represents a distinct SubEnv in the global 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 5.3. The static attributes contain information which is determined at site configuration time, and is modified only by subsequent configurations (see Section 5.3). It contains values that enable to always locate the SubEnv and connect to it (through the Connection Server), like the `subenv_name` and `subenv_id` fields for identifying the SubEnv, and the `site_name` and `site_ip_addr` which specify the location of the Connection Server. Note that the value of the `site_name` attribute

```
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;           # dotted format, 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;           # dotted format, 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
# Project Specific

end
```

Figure 5.3: The built-in class SUB_ENV

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 in Section 5.2.3.1.

Unlike the static attributes, the dynamic attributes are frequently modified by the 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 attribute). The rest of the `SUB_ENV` objects are "stubs" which are used to connect to other SubEnvs. For example, in an environment consisting of four SubEnvs, each SubEnv will have four distinct SubEnv objects (i.e., the total number of SubEnv objects in an environment is 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.

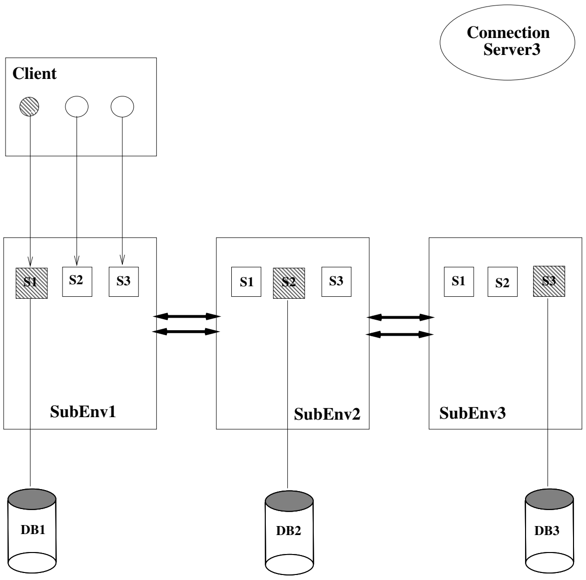


Figure 5.4: Connection Database

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 (see, for example, figure 4.1). When the client issues the (built-in) `open-remote` command on a remote SubEnv stub (shown in figure 4.2), the client switches 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 best illustrated in figures 5.4 and 5.5: In 5.4 the client has no open remote connections so its image of the connection database is directly mapped to the local connection database; and 5.5 shows client's image after an `open-remote` was issued on `SubEnv3` (ignore for now 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

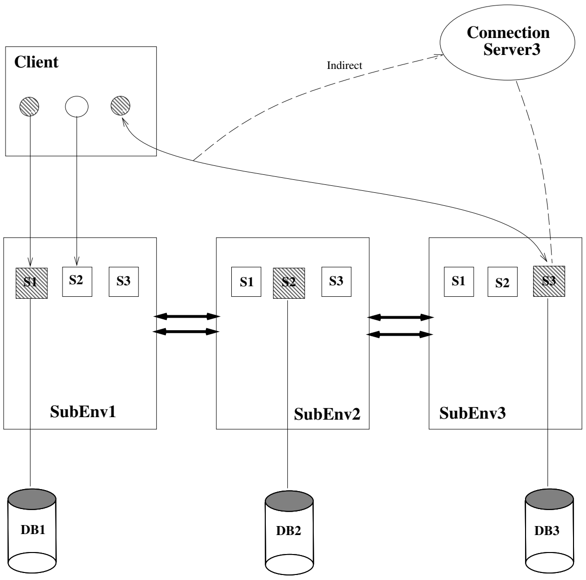


Figure 5.5: Connection Database with remote connection to SubEnv3

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 client, including possibly contacting the Connection Server. The main advantage of this 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 increases significantly 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 (note that access control can still protect sites from unauthorized remote access just as it does so for local clients).

5.2.3 The Communication Protocol

As mentioned earlier, Oz supports two modes of communication: the *direct* mode which uses the (possibly invalid) dynamic information at 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 connect through 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 out of date (e.g., due to the fact that the target server terminated its execution).

In either case, indirect communication is followed by updating 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 5.5 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 the `SubEnv3`'s address is not known to the client, 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 which used to run on a SubEnv 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 site. The communication protocol is summarized in figure 5.6. Note how all the necessary information can be obtained from the SubEnv objects in the local connection database.

This 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 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

```
if (remote-server is marked as Active)
then
    try to connect directly using the dynamic host information
    if connection is successful
    then
        communicate
    end

if ((remote-server is marked as NotActive)
                  OR
    (direct communication failed due to invalid dynamic information)
   then
  contact the Connection Server through the static information
   if connection is successful
   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
```

Figure 5.6: server-to-server communication

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 (so 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.

5.2.3.1 Remote Invocation of Environment Servers

We hinted earlier on the possibility to enable a Connection Server invoked in one host to spawn an Environment Server on a different host within the same domain[4]. This feature decouples the SubEnv's static "contact" host from the actual host in which the Environment

---

[4] Recall that we use domain to explicitly denote a physical Internet domain and assume a shared files system, unlike our use of the term site, which may or may not map to a domain.

Server executes, thereby allowing to determine the execution node *dynamically.* There are several benefits to this design. First, the execution node can be determined based on various considerations such as proximity to the SubEnv's data, load balancing, and fault tolerance issues. Second, if logical SubEnv names are supported (as explained below), SubEnvs can be migrated without affecting the Connection database, and in this case the Connection Server can choose the proper host based on its local site information.

Thus, the Connection Server maintains information on, and can be configured to operate based on, local domain considerations, which are shielded from Environment Servers and Clients that operate over a logical "site-based" name space (see below).

Typically, the SubEnv's static address (as denoted in the `site_name` attribute of the `SUB_ENV` objects) should be that of the host in which the SubEnv's data physically resides, and that local host is then the default host for spawning the Environment Server. However, in some cases it is desired to have a static address of a "public" node in the domain, in which case the Connection Server should spawn an Environment Server on the "private" node in which the SubEnv physically resides. This scenario is typical in "firewalled" sites, where only a single node (the "firewall") can communicate with the outside world.

To facilitate this feature, the Connection Server maintains a Domain SubEnv Table that contains invocation information for each SubEnv in its domain. This table is modified only at site configuration time (covered in 5.3) and is used only by the Connection Server. The main two pieces of information stored there are:

1. Mapping of the logical name of a SubEnv to a physical host and path in which the SubEnv repositories reside.
2. A priority list of hosts in the domain in which to invoke Environment Servers on the SubEnv.

5.2.4 Decentralized Naming Schemes

The obvious difficulty in decentralized naming is ensuring uniqueness and proper identification and location of elements without a centralized repository or global control. Further, autonomy considerations should lead us to strive to enable SubEnvs to handle their own naming without depending in any way on other SubEnvs. And as with other naming schemes, there might be a need to provide logical names which are mapped to internal physical ids. Oz employs separate name spaces for SubEnvs, object-ids, rule-ids, and client-ids.

5.2.4.1 SubEnv Name Space

SubEnv naming is the central naming scheme in Oz. It is used for both inter-site communication purposes and as a basis for guaranteeing uniqueness in all other name spaces in Oz. Most of the necessary information resides in the connection database.

One possible approach to naming, termed here *site-based*, is to bind each SubEnv with an Internet host *as part of its identification.* The main advantage of this approach is that it eliminates the need for an Oz-specific name space for SubEnvs, and in order to distinguish between multiple SubEnvs within the same site, the file system could be used. However, this approach implies a "hard" binding of a SubEnv to a specific site, which may not be desirable. Our motivation is to enable SubEnvs to migrate across hosts with minimum configuration overhead, as well as to enable different hosts (in the same domain) to execute on a given SubEnv regardless of the host that holds their data, as explained in Section 5.2.3.1. The approach in Oz is to maintain a logical naming scheme independent of the underlying physical hosts, thereby enabling to move SubEnvs across hosts/file system while retaining their same unique id, and invoke different hosts on SubEnvs regardless of their physical location. Thus, in this *SubEnv-based* approach, the network address is only a regular attribute of the SubEnv object used for *location* purposes, but is not used as part of its identification. In particular, its value can change if the SubEnv migrates.

However, we are still faced with the problem of ensuring uniqueness. One approach might be to reuse the native object-ids of the SubEnv objects as the SubEnv-ids. But this approach would require global control in assigning object-ids (or at least stub objects) to ensure uniqueness, which is far from desired for autonomy concerns, particularly given that assigning object-ids is a frequent operation (see Section 5.2.4.2).

The preferred solution is then to base the naming on some sort of consensus. The actual naming scheme in Oz is as follows. When the registration process (discussed in Section 5.3) initializes a new SubEnv in an environment, it queries all existing SubEnv objects (by looking at the connection database) and assigns a number that differs from all others. This SubEnv-id is guaranteed to be unique within the (global) environment (although not across global environments). And as part of the static attributes of the

SubEnv object, all stubs pointing to the same object get the same SubEnv-ids although they may or may not have the same *object-ids*, which are determined autonomously. Joining a pre-existing SubEnv (as opposed to registering a new one) requires then to reassign the SubEnv-id.

In order to actually communicate across SubEnvs, their physical network address is extracted from the proper SubEnv stub object, consisting of the file system path name of the environment directory that contains the SubEnv's persistent repositories (i.e., objectbase, rule-base, etc.), coupled with the Internet address of the host that controls that repository and the port number.

Finally, in order to enable movement of SubEnvs across global environments, a global assignment is in general unavoidable. This functionality has not been realized in Oz yet, as evidenced by the first two unused fields in the `SUB_ENV` class.

5.2.4.2 Object, Client, and Rule Ids

Distributed object naming schemes have been thoroughly investigated in the distributed object-oriented database community (see, for example, Orion-2 [65]). This is in general outside the scope of this thesis as a research topic, and we only present here a simple solution. The main goal in the design of the object-id management is to reconcile the conflict between allowing autonomy in id assignment and still providing uniqueness. The solution here 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 id as explained earlier. Note that this is not a "long" id split into two fields, but rather two different ids. In particular, unless cross site operation takes place (e.g., Summits) only the `obj_id` field is used. This reflects the decentralized nature of the architecture. However, the client's image treats the pair effectively as a single id, since it might contain images from multiple objectbases. This approach enables each local object management to 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.

Two other entities that require global uniqueness are client- and rule-ids. As with object-ids, their uniqueness is derived from the uniqueness of the SubEnv-id. Rule-ids were already discussed in Section 4.3.1.7. The reason why client-ids need to be unique across SubEnvs is that an `open-remote` operation is effectively treated by the remote server as a remote-login operation, meaning that the remote server makes an entry for the client, and it uses the client id in order to identify the client, so the ids of all clients originating from all sites must be distinct. The implementation of client-ids is similar to rule-id assignment, i.e., each local server maintains a private counter, to which it adds the SubEnv-id multiplied by large constant (the same constant is used in all SubEnvs, of course), thereby ensuring global uniqueness.

5.3 A Process for Site Configuration

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 5.2.2). 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 PML, and to exploit the concept of process enactment by executing the configuration process using the 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 1.5, 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. A detailed description of the configuration process is also given in [15].

5.3.1 Configuration Facilities

The registration process presented here is mostly similar to any other Oz (sub)process. The differences are in that (1) it was written by the environment kernel implementors rather than by 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 on all remote SubEnvs, since it manipulates the connection database in all 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; and (3) Moving a SubEnv to a different location and/or host within the same global environment. All tasks are modeled as rules which are invoked interactively inside any one of the existing SubEnvs, with the same user interface normally employed for regular process enactment.

5.3.1.1 SubEnv Registration

A multi-site environment is populated by means of a *registration* task which 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 special utility. The registration task consists of two steps: (1) adding a new stub object (representing the new SubEnv) to all existing SubEnvs (modeled by the `register_subenv` rule); and (2) physically creating and initializing a new SubEnv or joining a pre-existing one (modeled by the `send_connection_db` rule). Both rules are listed in Appendix A.1.

The `register_subenv` rule may be evaluated from *any* site already participating in the relevant global environment. It binds the SubEnv objects of all existing SubEnvs ("real" object for the local SubEnv and stub objects for reomte SubEnvs), and executes the `register_subenv` tool envelope (listed in Appendix A.2) with the SubEnv objects as the activity's parameters.

This "tool" prompts the administrator for the new site's static information, and creates in all existing SubEnvs (including the local one) a replicated SubEnv stub object instantiated with the specified static information. If the envelope 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 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 which was specified in `register_subenv`, 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 to contact remote SubEnvs and update 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 envelope generates a script of commands that contain invocation of the `init_remote_subenv` rule (listed in Appendix A.1). This rule[5] simply adds a new (SubEnv) object with some specified values. Then, it traverses the local connection database (i.e., the set of SubEnv objects) and for each remote stub object it spawns a batch client that operates on the proper remote SubEnv with the generated script as the input command batch file.

5.3.1.2 SubEnv Deregistration

This task is modeled by the `deregister_subenv` rule (shown in Appendix A.1). It removes a site 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 site'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. For autonomy reasons, this step can only be performed locally, i.e., at the site that is about to be de-registered.

---

[5] The `hide` keyword preceding the definition of that rule prevents that rule from being displayed at the client's rule-menu, since it is not intended to be executed interactively.

5.3.1.3 SubEnv Migration

The last supported step in the configuration process is migration of a SubEnv to another physical location, modeled as the `change_subenv_location` rule (shown in Appendix A.1). This rule prompts the user for the new location (host and file system path), physically moves the environment directory's contents to the new location, and updates all the stubs in the remote SubEnvs (again, using the remote batch facility). As with deregistration and for similar reasons, this rule can be fired only locally, i.e., at the SubEnv that is actually moved.

5.3.2 Summary

The main research contribution with respect to configuration is that it is treated as a fully integrated process, benefiting from most of the advantages that come with process. 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, provided that the required parts of the data and rules are protected from modifications.

There are several additional important advantages of this approach, as opposed to hard-wiring the (re)configuration mechanism internally in Oz:

1. It is likely to prove substantially easier to modify, mainly for the parts that do not require kernel changes. In particular, it can be modified by system administrators on a per global environment basis.

2. It was much easier 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.

3. 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.

5.4 Context Switching in Summit

5.4.1 The Problem

In a conventional client-server architecture, it is clear that if some requests take 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) rule-chain, 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, as a chain consists of several rules, it is even more unacceptable to assume that a whole chain (and its associated activities) will actually execute atomically, even if the "all or nothing" atomicity property is required for the execution. Thus, it is necessary for the server to keep a context for each chain, and switch among the contexts to service multiple clients concurrently. A mechanism for context-switching among multiple executing clients was part of Marvel (due to the author, see [13]) and was upgraded to Oz.

Considering the multi-server architecture of Oz, there is an additional problem. In cases where a server has to communicate with other servers in order to service a client request, two things can happen: (1) the server might wait arbitrarily long until the remote servers complete to service the request, thereby reintroducing the starvation problem more vigorously than in the single server case; and (2) if a server has to wait for other servers, then the servers might deadlock. Moreover, since servers might wait arbitrarily long, the chances for getting into a deadlock situation in a naive implementation are pretty high.

To illustrate the problem, consider the following example of a communication deadlock between two servers (illustrated in figure 5.7): Suppose a client `C1` is requesting to fire a Summit rule at its local server, `S1`, involving some remote objects managed by server `S2`. `S1` requests from `S2` the remote objects, and waits. The request is enqueued in `S2`'s service queue, which already contains several requests. At some point after the request from `S1` was made and before it was serviced, a client `c2` requests from its local server `S2` to perform another Summit rule, involving objects from `S1`. `S1`'s request is enqueued at `S2`, but since both servers are waiting for each other, we have a deadlock. Note that this deadlock problem is completely orthogonal to the notion of transaction deadlock. In particular, the objects or rules which were accessed by the servers can be totally unrelated to each other.

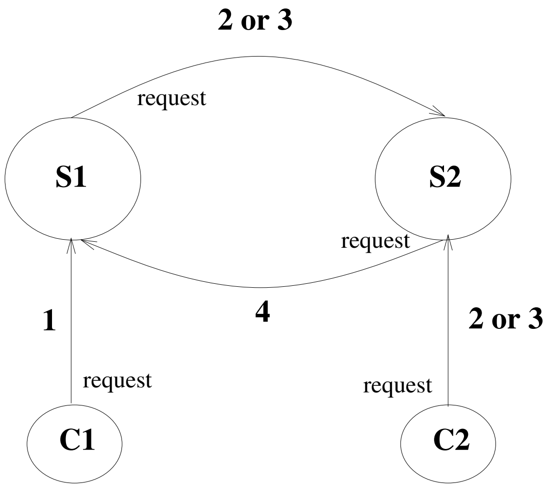


Figure 5.7: A Communication Deadlock Example

In order to realize the magnitude of this problem and the importance of solving it in Oz, we analyze the types of synchronous server-to-server and server-to-client interactions which require the requesting server to conceptually wait for the response before continuing with the underlying task.

The majority of these cases occur during the normal execution of an ordinary Summit rule:

1. binding of remote objects to the Summit rule's parameters
2. Treaty verification - all involved remote sites validate the eligibility of executing the Summit rule
3. binding of remote objects to the Summit rule's derived parameters
4. remote backward chaining
5. activity execution (server waits for client)
6. remote assertions
7. binding of remote objects as part of the inversion algorithm for new Summits (discussed earlier in Sections 2.2.3 and 4.4.3.4)
8. remote forward chaining

Other places that require the server to wait include the *import* and *treaty* commands, cross-site built-in operations (particularly *move*, where the local copy should not be removed unless the copy was successful) which were discussed earlier, remote file transfer, and transactions.

The key problem in such interaction is that a server, acting as a client, can block indefinitely waiting to be serviced by another server due to circular waiting.

5.4.2 The Solution

One possible solution to the problem 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 (i.e., regions of code that have to execute atomically) might need to be defined in order to protect the integrity of the data in the servers, reintroducing the deadlock problem.

The pragmatic solution in Oz [6] consists of three different methods which are applied at different "break points" as defined above. The first one is full context switching, that is, 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. In addition to the context switch point for activity execution at the client (which already existed in Marvel), Oz employs two other context switch points in the major natural breaks of the Summit algorithm, namely remote backward chaining and remote forward chaining.

The second method, which we shall refer to as "busy-service-wait" loop, is applied to services that are characterized by being simple and consisting of a single step, but are called from deep within a complex context that makes it highly undesirable to switch contexts there. Examples of such break points include binding of parameters, Treaty verification in Summits, and some other non-Summit services. The idea in 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 server, never needs to communicate with other servers or clients[7].

---

[6] done with Peter Skopp

[7] This would not be true if arbitrary cross-site links were implemented. Another exception to this is

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 it handles while waiting for the reply do not depend on any other computational entity.

The third and last method, that we shall 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, but are still hard to fully context switch. In Oz these are the direct and inverse binding phases in Summits (steps 3 and 7 above), which might require several requests to multiple remote servers to complete the binding (recall from Chapter 4 that bindings in Oz are realized as a set of individual sub-binding requests). 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 update 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 "binding" mode and were 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 5.8 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, and table 5.4.2 summarizes the method applied in each of the break points presented earlier in the previous section.

5.5 The Remote Object Cache

5.5.1 The Problems

While a composite Summit is being executed, the same object might be accessed by the same remote server multiple times and for various purposes, including:

1. Binding of user-selected remote objects as parameters for Summit rules

---

coordination rules, see [47].

```
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
         else /* this is the reply */
           waiting_for_reply = FALSE
         end
   end
```

Figure 5.8: The Extended Busy-wait-service Algorithm

	Remote Request	Method
1	Parameter binding	busy-wait-service
2	Treaty verification	busy-wait-service
3	Derived parameter binding	extended-busy-wait-service
4	Remote backward chaining	full context switch
5	Activity execution	full context switch
6	Remote assertions	busy-wait-service
7	Inversion binding	extended-busy-wait-service
8	Remote forward chaining	full context switch

Table 5.1: Context Switch Summary

2. Binding of remote objects to derived parameters in Summit rules
3. Automatic derivation of remote objects as parameters to Summit rules in an ongoing chain (using the inversion algorithm)
4. Firing of different unrelated Summit rules, i.e., across different chains, whose respective durations and object "working set" overlap

In a naive implementation of Summits (and the implementation that predated the current version of OZ), a fresh transient copy of the remote object had to be fetched for every request to access that object regardless of whether it had been already fetched. If a local copy of a remote object could be maintained consistently such that subsequent access to the object from the local server would only need to refer to that local copy instead of re-fetching it, and if the overhead for maintaining this consistency is small enough, a great improvement in performance would result. The idea of establishing remote object cache came up aiming at this purpose.

There are several problems in implementing a cache in a system like Oz:

1. Cache invalidation and update — This is the most crucial aspect of the cache. Clearly, a copy of a remote object should be invalidated if the original object has been updated, or else the access to the (stale) copy is inconsistent.
2. Structure Validity — The arbitrary complex relationships among objects in an object-oriented database complicate cache management, since in addition to the validity of the objects, the validity of the links has to be maintained, particularly due to the various structural and associative querying capabilities which are used to access objects.
3. Pre-fetching — A related issue is to assess whether to pre-fetch related objects along with the requested one, and to what degree. For example, prefetching could range from immediate "relatives" (children, parent, links) at one end of the spectrum, to the transitive closure at the other end.

Cache and replica management as well as pre-fetching methods in distributed systems in general and in distributed database and file systems in particular, is a wide topic on its own and has been widely explored (e.g., Coda [95]). However, special characteristics of Oz make it a "special case" worth discussing the solution to the first two problems mentioned above. Prefetching, while a promising direction, is beyond the scope of this thesis and is not further addressed here.

5.5.2 The Solution

The cache in Oz is part of the data manager component (see figure 5.1), and can be viewed conceptually as a non-persistent extension of the in-core local objectbase. Each server maintains a single cache that is shared by all clients. An alternative to this design would have been to maintain a temporary cache on a per-task basis that is initialized upon fetching the first remote copy and is destroyed when the task is complete[8]. The main advantage of this approach is that it simplifies the invalidation scheme since no other tasks can access the (private) copies, and at the end of the task all the copies are simply removed. However, the limited scope of such a cache and the large overhead associated with it, make this approach not worth pursuing. Thus, the cache can be accessed by any local task (from any client) that involves access to remote objects over arbitrary periods of time, as long as the cache objects are valid.

The cache is implemented as a hash table that can be looked up by the object's id similar to the hash table used in the main in-core objectbase, except the lookup is also by the SubEnv-id, not only the object-id.

However, unlike the regular objectbase, it does not maintain the by-class list or the structural graph. Instead, each cached object is enclosed within a "cache entry" that contains information regarding its relationships to other objects, termed here the *relationship lists* (the cache entry contains also validity information which is discussed later). More specifically, the cache entry contains four lists that cover the two directions in the two kinds of relationships among Oz objects— children, parent, forward links, and backward links. Each list is further decomposed to sub-lists that correspond to the attributes within their category[9]. The individual sub-lists, however, contain only object-ids, not pointers to objects. Thus, at any point in time, none, some, or all of the replicas of the objects that correspond to an id-list may reside in the cache.

An important invariant in the design of the cache can be defined as follows: if a cache object is marked as valid, then its relationship lists are also valid, i.e., it is indeed connected in the original objectbase to all objects whose ids are stored in those lists. One implication of this *validity invariant* is that an object needs to be invalidated not only when its content changes, but also when its relationships with other objects change. More specifically:

- When a new object is added, any cache copy of its parent should be invalidated.
- When an object is deleted, any cache copy of itself, its parent, and objects that previously linked to the deleted object or were linked-to by the deleted object, become invalid.

---

[8] Recall that a task corresponds to a single chain of rules.

[9] Recall that a class in Oz can have several attributes of the same type.

- When an object is linked to, or unlinked from, another object, any cache copy of both objects should become invalid.

Other implications of this invariant will be discussed shortly.

Finally, the cache in Oz is a “write-through” cache. That is, any updates to a remote copy (through remote assertions in the effects of rules) are immediately propagated to the original copy. However, such updates do not invalidate the remote site from which the update was made since its copy is still up-to-date. Only remote copies of that objects from other sites are invalid. Technically, the cache should handle similarly structural changes to a remote objectbase such as addition and deletion of objects. However, at present only changes to the contents of individual objects (not to the object lattice) are supported in the cache; the structural operations are performed directly in the remote objectbase (and any relevant entries in the cache are invalidated).

5.5.2.1 Maintaining Structure and Connectivity

When a remote object is accessed directly by its id, the local object manager first looks in its cache, and if an entry is found and is marked as valid, the proper pointer to the object is returned to the caller. Otherwise, the object is fetched from the remote site, along with its relationship lists (but none of the related objects is actually accessed).

The second way to access remote objects is through structural queries. (Associative queries that appear in conjunction with structural queries are handled by evaluating the structural query first, and then filtering it through the associative query. The infrequent purely associative queries are not supported in the cache and require remote fetch.) In Oz, exactly one of the two symbols in a structural query is always already bound, and the second symbol is bound as a result of the query. For example, in the query:

```
(forall CFILE ?c suchthat (member [?m.cfiles ?c]))
```

the symbol `?m` is already bound to a set of objects, and the query binds to the symbol `?c` all objects that are members (i.e., children) of the `cfiles` attribute of the objects in the binding set of `?m`.

Thus, when a structural query on remote objects is made (e.g., as part of a Summit rule), all remote objects associated with the bound symbol (`?m` in the above example) must have already been fetched previously (either directly during the binding or through a previous query) and thus exist in the cache. To evaluate the query, first the bound objects must be valid, otherwise they are re-fetched. Thereafter, the proper relationship list is scanned (for example, in the above query the `children.cfiles` sub-list would be scanned) and each id is looked up in the cache. In the ideal case where all remote objects are found in the cache and are all valid, the query completes without accessing remote servers at all. Otherwise, the missing or invalid objects are (re)fetched from the remote servers (along with their relationship-lists). Thus, the relationship-lists play a major role in reducing the amount of remote fetching while incurring a relatively small overhead. Notice how the validity invariant mentioned above is necessary for the correctness of this scheme.

5.5.2.2 Cache Invalidation Scheme

The cache invalidation problem is clear: given that objects can be updated over time, there must be a way to know if the the replica of an object in the cache is up-to-date with respect to the original copy. If the replica is out-of-date, it must be invalidated so that subsequent access requests would result in re-fetching of a new copy.

One solution to this problem is to maintain in the original object a list of sites that hold a replica in their cache, and whenever the primary copy is modified, all remote caches are asynchronously notified to invalidate their outdated copies of the object. The list of SubEnvs that hold a cache copy would have to be maintained in each object, adding an entry upon remote fetch and deleting it when the associated replica becomes invalid. Several minor variations can be made to this basic approach. For example, the server might send the modified object (or the changes from the original copy) along with the invalidation message. And another alternative to maintaining the per-object SubEnv list would be to simply broadcast the invalidation to all sites in the environment.

However, this "instant notification" approach does not jibe well with autonomy and independent operation requirements, due to the overwhelming cross-site communication involved, and due to the fact that (possibly long) delays might easily lead to race conditions in which a cache object is accessed before it has been invalidated.

There are other requirements that make the invalidation problem hard in Oz. The fact that object management is decoupled from transaction and task management implies that the cache and the invalidation scheme should also be decoupled from transactions and tasks, with transaction and tasks only invoking invalidation requests and the cache implementing them. This is a clear example of the need to separate (invalidation) mechanisms and (invalidation) policies. Moreover, an instance of Oz might employ an advanced concurrency control method that facilitates collaboration by means of allowing, for example, multiple simultaneous writers. Embedding a built-in invalidation policy in the cache is likely to conflict and effectively prevent such tolerance by, for example, repeatedly fetching new copies and restarting tasks due to the interleaved updates. This reinforces the need to determine the invalidation policy independent of the mechanism. The cache in Oz takes into considerations all of the above arguments.

The invalidation *mechanism* is relatively straightforward. Each cache entry contains a `valid` flag that indicates the validity of the object in its entry. When a cache object is invalidated, its `valid` flag it turned off, but its memory is not freed. The reason is that the same cache object might be pointed to by multiple (possibly sleeping) tasks, or even by multiple symbols in the same active task, so freeing the memory would result in stale pointers (unless the cache maintains a list of accessors and notifies them to unlink their pointer, but this is a costly procedure). Therefore, whenever a new copy is fetched, its contents are copied into the contents of the old object, so that all references continue to point to the right object. The only problem with this approach is that the memory occupied by a cache never shrinks (although it grows slowly because multiple fetches of the same object do not require additional memory allocation), but a simple garbage collection could be launched when the server is idle. Then, whenever a cache object is accessed, either directly or via a relationship list of another (valid) cache object, its `valid` flag is checked. If it is invalid, then a fresh copy is requested to be fetched. Note that following the analysis in Section 5.4, there is no need to abort the task to avoid deadlock. Once again, the validity invariant mentioned above is necessary here for the correctness of queries involving cache objects.

The invalidation *policy* in Oz is not as simple, however, as it is tied to the semantics of the operations that use the cache, in order to optimize it with respect to the requirements set above. Given that instant notification is impractical, we try to find ways to apply the lazy approach without sacrificing correctness. This is done by enforcing the following constraints:

1. The first constraint made is that no two *executing* (i.e., active) tasks can update the same object (or its replica) simultaneously. Note that while this

scenario cannot physically occur within a single-threaded server, it might very well occur if two tasks execute each at a different server and update different copies of the same object. This constraint can be enforced by the transaction manager, since we assume that each request for a remote fetch involves also a lock request. Although limiting, this scheme still allows for *interleaved* updates of an object by multiple tasks, as will seen shortly. Also, it is still adequate for the current support for groupware multi-user tools, because from the server's point of view, only the initiating task is updating the objects; multiple updates to parts of these objects (e.g. files) are handled by the multi-user tool and are shielded from the server. If, however, the support for multi-user tools is modified in Oz and requires to allow multiple decentralized tasks to explicitly update the same objects (for example, along the lines of [108]), this constraint might need to be relaxed.

Given the above constraint, the implication is that if we could find a way to properly update a ready-to-run task with the relevant update information just before activating it, its view of the cache would still be valid.

2. The second constraint that we impose is that no two *unrelated* tasks executing in *different* servers can operate on an object with conflicting modes, that is, with at least one of them being a writer. While apparently restrictive, it still allows for *related* (sub)-tasks to conflict, and also allows multiple unrelated tasks executing in the *same* server to conflict. A typical example of the former case is remote backward chaining in Summit, whereby the object that "failed" to satisfy the condition must be updated in order to make the condition satisfiable, even-though the (non-active) Summit task at the coordinating site has been accessing the same object, at least for reading purposes. An example of the need for the latter case is for supporting advanced concurrency-control mechanisms such as in [6].

   The rationale behind this constraint is to keep the update privileges under the control of, and within the boundaries of the (global) Summit task (and its associated distributed transaction), so that all relevant update information can be passed upon reactivation of a Summit task at the coordinating site.

3. The third and last constraint states that an object in the cache is valid only while there is at least one Summit task that references it (note that objects reside in a cache of a server only if that server is the coordinating server of an executing Summit task). This implies that at the completion of a Summit, all the cache objects which were accessed by the Summit task are invalidated, unless some other Summit task is referencing them. This constraint appears to reduce the utilization of the cache most significantly, but is nevertheless necessary with lack of asynchronous instant notification, since we need to keep each cache object under the control of (at least one) active task that maintains its validity.

5.5.2.3 Cache Operation During Summit

The above constraints effectively define the invalidation policy. To illustrate invalidation, consider the operation of the cache manager at the coordinating site in a typical execution of a Summit, along its major phases:

1. *Summit Initialization and Verification* — This phase binds all objects that are accessed by this rule (but not necessarily all objects accessed by subsequent Summit rules in a composite Summit). Binding of the user-selected remote objects to the rule parameters and binding of remote objects to derived parameters through queries is done as explained earlier in Section 5.5.2.1, and these objects are put in the cache.

2. *Pre-Summit* — If remote backward chaining takes place, the Summit task becomes inactive, and gets reactivated only when all remote sites complete their backward chain. When a remote backward chain completes, it sends along fresh copies of the modified objects to the coordinating site, and when the Summit task becomes active it updates the cache properly. The actual update can be implemented either by sending a "delta" (reducing the communication overhead) or by replacing whole objects (reducing the computation invovled in calculating the delta). For small sized objects the latter approach was adopted in Oz, but for objects with file attributes that had to be transferred to remote domains a different approach was taken (see Section 5.6).

It is important to note that during backward chaining the cached objects in the coordinating site cannot be updated by the coordinating proces because it is effectively blocking.

Notice that besides the local backward chains, no other subtask in other SubEnvs can update objects which were accessed by the Summit task, due to the second constraint.

3. *Summit Activity* — This phase causes no problems in terms of invalidation since it executes only at the coordinating site. Still, any access to cached objects has to be validated as usual.

4. *Post-Summit* — This case is similar to Pre-Summit in terms of the cache. While the coordinating task is not active, the remote sites might update their original copies of the objects and when they complete their work in "local" mode, they send to the coordinating server the updates so that it can update its cache.

5. *Inference of (forward) Summit Rules* — This case is similar to the binding step in phase 1. Note that this phase might extend the "working set" of objects accessed by this (composite) task, since it derives new parameter objects for subsequent Summit rules.

6. *Summit Completion* — The cache manager invalidates all objects accessed during the task that are are not accessed by an on-going Summit in the same coordinator site.

5.5.3 Results and Summary

In order to assess the improvement in performance, several experiments were conducted. The nature of the experiments was basically to run identical Summits on identical objectbase states twice, one time on a server with cache, and a second time on a server without cache, and compare their performance. Since the cache is not intended to add or remove functionality, the execution trace should be identical in both cases, the only difference being in the performance. Aside from the absolute execution times, the main basis for comparison was the number of messages which were exchanged between the servers

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 verification
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 5.2: Performance comparison with and without cache

during the Summit, and their type. Since the major delays are incurred by the communication overhead and their growth is inversely proportional to the available bandwidth, and given that all objects in the experiment were roughly of same size, it is a valid measure of performance improvement.

We show here the results of one specific example on a three-site environment, involving a small number of objects and two remote derivations, one for parents and one for children. The results are summarized in table 5.2. Obviously, the improvement is significant due to the reduction in the number of calls to get the remote parents and children, and there is no additional communication overhead due to the operation of the cache. It is interesting to note that the reason for the large number of requests for `GET_REMOTE_PARENTS` and for `GET_REMOTE_CHILDREN` stems from the fact that when the rule processor evaluates which rules to forward-chain to, there are many possibilities to instantiate each such rule (i.e., to select objects as parameters), of which only a small fragment really gets executed, because the rule's condition is not satisfied on most of the instantiations[10].

As for overall performance improvement, it was expected that the communication overhead will outweigh computation overhead across hosts, particularly as the available bandwidth between the hosts decreases. However, even across (operating-system) processes in the same machine the improvement was significant. For example, a simulation of the above Summit with no interactive activities (thus mostly involving operation at the server host) with the three "sites" running as (operating-system) processes in the same physical host, took 29 seconds on a non-cache server versus 9 seconds on a server with cache. As can be seen in table 5.2, the time ratio is very close to the ratio of the number of messages

---

[10] This spurious instantiation is typical of Oz rules with multiple parameters, since the rule processor generates a cross product of the sets of objects for each parameter in the chained-to rule.

between the non-cache and the cache-server, confirming that the computation overhead is negligible relative to the communication overhead, and consequently clearly showing the performance improvement of the cache server.

5.5.3.1 Cache and Transaction Management

There is one caveat to the above results: they were taken ignoring the overhead of the locking-based decentralized transaction manager. In principle, the transaction manager should not impact the ratio in performance between the cache- and the non-cache server, since when optimized, its locking overhead should be proportional to the number of objects obtained, and the non-locking-related overhead should be independent of the cache. In practice, however, if the locks are maintained separately from the objects, (as in the case of OZ), this requires implementation of *cache locks* in addition to the cache objects. With lack of such a cache, each request for a remote object would incur a locking request message to the remote transaction manager even if the server has a local replica of that object, thus decreasing the improvement of the cache-server. This is a subject of future work.

5.6 OZ Over the Internet

Although the OZ model and its architecture are conceptually geared towards operation across sites, and although most of the implementation was done with geographical dispersion in mind, extending the system to operate over the Internet introduces several new problems:

1. No shared file system across sites
2. Security, authorization, and access control issues
3. Different administrative network domains
4. Variable bandwidth and time shifting

This section provides a brief summary of our preliminary exploration of only the first two problems, and solutions to them. The discussion in this section is by no means comprehensive or complete, and by-and-large, it is a subject for further investigation (see 7.2). In particular, the last problem dealing with the general performance, optimization, and synchronization in the operation of the system as a function of the (perhaps dynamically changing) bandwidth, load, and frequency of interaction between the sites, is outside the scope of this thesis.

5.6.1 No Shared File System

The fact that SubEnvs have no means to share any data, either physically or through the underlying operating system (e.g., NFS), complicates several aspects of the implementation:

1. The biggest problem is with respect to sharing bulk data, (e.g., files), due to the large volume of data involved.
2. With lack of shared data, the file system cannot be used for communication purposes, for example to store and retrieve addresses of remote ports. Moreover, pathnames are no longer necessarily unique.

5.6.1.1 File Transfer

The main reasons for sharing files across sites in Oz are:

- For execution of activities in Summit rules that involve remote objects (and their associated files).
- In built-in cross-site `copy` and `move` operations
- The *import* operation as part of Treaty involves receiving a list of available strategies from the exporting SubEnv and copying them (see Section 4.3.1.2).
- The configuration process which was discussed earlier requires invocation of batch clients, meaning that at least the generated batch files must be transferred. Another problem with registration is the initialization of a new remote site (as opposed to joining an existing one, which is fine).

Of the above, the file transfer during the activity execution on behalf of Summit rules is the most critical, and the only one discussed here. The other cases are simpler and can be realized on top of the general purpose file transfer mechanism (e.g., `ftp`).

With a shared file system, when an Oz server transfers objects across sites for rule execution purposes, it physically transfers only the "light" status attributes (along with other information such as its id, class, etc.)[11], but for the "heavy" file attributes only their pathname is sent, and when a remote client needs them (e.g., for executing a rule activity) they are accessed through the provided pathname (recall that we still assume that clients and their local servers share the file system[12]). Thus, with no shared file system there must be an underlying file transfer mechanism that physically transfers the files. Moreover, this mechanism needs to transfer the files back if they were changed during their use.

The main technical problem with implementing a file transfer mechanism is that it cannot be executed synchronously, otherwise it would effectively block both servers (the sender and the receiver) which, as seen earlier in Section 5.4, cannot be allowed. Thus, files must be sent asynchronously, "in the background", and the receiver server notified when transfer is complete. Since the light objects arrive quickly at the receiving end, another optimization would be to start the activity and wait only when the files are really needed. Finally, care must be taken that no files are transferred unnecessarily (either back or forth).

The general design of the file transfer mechanism[13] is as follows. All file transfers are initiated by the server that executes the (Summit) rule, i.e., the receiving server, and only when they are needed (as usual, this lazy approach seems most appropriate for this purpose). At this point, the server performs a context-switch and the executing rule enters a sleep state. In order to not occupy the server for a long duration, files are transferred in small chunks (4kbytes) and only when the server is idle. An alternative and perhaps more appropriate approach would have been to send the files over a completely separate channel, but for practical considerations the former approach was chosen.

When the transfer is complete, the sleeping rule is woken up and continues its execution (but recall that it has to wait for the next context-switching opportunity since the server cannot be preempted) by sending the necessary arguments to the client on whose behalf the file(s) were requested, for executing the rule activity. When the client finishes its activity, the rule enters a second context-switch point, this time for copying back files if they were updated by the activity. Both the receiver and the sender servers create and

---

[11] The size of an Oz object depends on the schema defined in its process, but on average it is about 100 bytes long.

[12] An extension of this model that deals with low-bandwidth clients where there is no shared file system with their server (mainly to address remote clients that are connected through a modem to the server) is dealt with separately by Skopp [98].

[13] This is largely due to Peter Skopp and Shelley Tselepis

maintain information on the transfer. However, information regarding the halted rule is held only by the receiver.

In order to reduce unnecessary transfer, the following methods could be employed (in practice, only the first two have been implemented as of this writing):

1. File caching — When a server imports a remote file, the copy is placed in a special area of the SubEnv that identifies it uniquely. Then, upon completion of an activity involving a remote file, it is not destroyed. Instead, if the same file has to be sent to the same SubEnv for execution of another activity and it hasn't been changed since, the transfer is not necessary and the client can use the cached file.

2. Checksum — Each time a transfer is requested, the sending server first sends the receiving server a "magic" number that represents the file. If the receiving server happens to have that file, and their "magic" numbers are identical, there is no need to perform the transfer. There are two frequent situations in which no transfer is necessary: 1) in the copy-back stage, in case the executing activity accessed the files in "read-only" mode; and 2) if the file was recently updated by the same client or another client in the same SubEnv, and is sent again for another activity— this is possible due to file caching.

3. Semantic prefetching — Prefetching of files during idle time can lead to substantial improvement in performance. The idea is to anticipate future use of some files, and prefetch beforehand, so that when the user wants to use them they are already local. The key issue is to use the right criteria to determine what files are likely to be needed soon, and prefetch them. For example, one method might be to use history of access patterns (for more work in that area see [105]). The interesting aspect from the Oz perspective is that the process model contains semantic knowledge which might help in predicting future references. For example, by observing the rule network and anticipating a path in a chain of rules, files that are intended to be used in (forward-chained) rules in the near future can be prefetched. Some preliminary work in this area has been done by Skopp, see [98].

5.6.1.2 Extensions to Connection Server and Database

The basic requirement here is to ensure that all the information which is necessary for connectivity, without exceptions, can be obtained from the connection database or through the Connection Server. To achieve that, the following enhancements to the Connection Server were made:

- The SUB_ENV class has been extended with a boolean attribute called `has_nfs`, that indicates for each remote SubEnv, whether it is sharing its file system with the local SubEnv or not. This is important information that enables to determine, for example, whether to send pathnames and rely on the underlying shared file system, or actually send the files. Note that the attribute might be set to `false` even if there exists an underlying shared file system, in cases that the performance of the shared file system degrades, or is temporarily not operational. The `has_nfs` attribute has the unique property that even though it is part of the static information of the SubEnv objects (and its value is defined at registration time), it is not replicated, since for example, the value of this attribute in each SubEnv object at its local site is always `true`, but the value of its stub in remote SubEnvs may or may not be `true` depending on whether the two SubEnvs share a file system. Thus, the value should be determined at each remote site individually.

  Incidentally, this example shows how the tailorability of the connection database has paid off: Modifying the SUB_ENV class was done simply by adding the attribute and evolving the objectbases, with no code changes in the kernel.

- To uniquely resolve pathnames as locations of environments, the simplest solution is to prepend them with the full host name.

- The Connection Server has been extended with the capability to respond to queries about port numbers of active servers in its domain. (Previously, servers were "peeking" to each other's port-files which contained the port number.)

- Finally, since intermittent disconnections and noise are more likely to interfere with communication between geographically dispersed SubEnvs, the inter-process-communication layer in clients and servers should be more fault tolerant and anticipate them.

5.6.2 Security Firewalls

In order to address the desire of private corporations to be connected to some networked services but at the same time isolated (which is in a sense similar to the tension between autonomy and interoperability at the process level), some security mechanisms have been invented with an option to control the level and kind of "openness". One common security mechanism that is intended to isolate private networks from public networks (i.e., Internet) is the "firewall". A firewall host is a *dually homed* machine, meaning that it contains two network interfaces — one attached to the secure, private network and the other one is attached to the public, insecure network. The firewall machine can then be customized to allow or deny certain network packets to pass through, depending on source address, destination address, port number and in more advanced software, even by user identity. For example, it is common in companies to allow only incoming and outgoing mail and block any other service.

In order to provide more flexibility and programmable control over the allowable communication through secured networks, the SOCKS package was chosen[14]. SOCKS is a public-domain package that allows hosts behind a firewall to gain full access to the Internet without requiring direct IP reachability. It works by redirecting requests to talk to Internet sites to a server, which authorizes connections and passes IP packets back and forth. The SOCKS package also allows external hosts to access a defined set of internal machines. The SOCKS daemon runs on a firewall machine and serves requests from other SOCKS clients. The SOCKS daemon receives packets on a designated SOCKS port, and decides whether to forward the packet to the other interface. By using the SOCKS library of replacement socket calls, OZ clients and servers can communicate with each other, thereby enabling interoperability through firewalled sites. For more details on the SOCKS integration, see [72].

---

[14] This work was done by Andrew Lih.

5.7 Implementation Status

As already mentioned in Chapter 4, at the time of this writing, Oz version 1.0 has been completed and is fully operational[15], with most of the features discussed in this chapter fully implemented. The following is a summary of the aspects which were discussed in this chapter and are only designed or not fully implemented at the time of this writing.

Connection Server — Automatically reverting from direct to indirect mode during a remote session when the dynamic information turns out to be out-of-date, is not supported. Instead, a client has to manually close its remote connection and re-open it using the indirect mechanism. The domain SubEnv table and its associated feature — to anble the Connection Server to invoke remote Environment Servers on remote hosts in the same domain as described in Section 5.2.3.1 — is not operational yet (an earlier prototype developed by Will Chou, has to be upgraded).

Environment Server — Global environment naming scheme is not supported, as explained earlier. Lock management is not optimized for operation with the object cache as explained in Section 5.5.3.1, and lock cache is not implemented yet. There is no garbage collection to clean up the cache. The file transfer mechanism has not been generalized yet. Therefore, the Treaty and registration processes are not operational over the Internet. Instead, they both have to be performed manually using external scripts. But once defined, Summits are fully operational across physical sites.

---

[15] We are using Marvel 3.1.1 to produce Oz, employing a process based on code re-engineering and componentization (see [48]), but plan to start using Oz for its own further development as soon as it is sufficiently robust.

6

The ISPW Example: Validation and Methodology Issues

The purpose of this chapter is twofold: (1) To validate the feasibility and effectiveness of the ideas and their implementations as presented in this thesis; and (2) to explore methodology issues regarding decentralized process modeling in a decentralized environment. Both objectives are attained by discussing the modeling and enactment of an instantiated Oz environment that supports the so-called ISPW-9 Example[1].

The ISPW example was first introduced at the 6th International Software Process Workshop [62] in an attempt to provide a canonical "benchmark" process scenario, and ".. as a common framework for understanding and evaluating various approaches to software process modeling and enactment". Since then, the example has evolved several times ([45, 31] and the latest version [83]), adding or revising (sub)scenarios that require more advanced modeling and enactment capabilities, and removing some of the rigidity of earlier versions.

The advantages of using this example for validation purposes are not different than the case for using benchmarks in general, namely: they tend to be objective and not (suspected to be) contrived by the implementors of the solution; they are written by experts in the field and therefore expected to be comprehensive in their coverage of the issues that need to be addressed; therefore, they are well accepted within the community as a valid criteria for evaluating the technologies and their underlying concepts.

---

[1] This Oz environment was actually demonstrated at the 9th ISPW.

The remainder of this section is organized as follows: Section 6.1 describes a brief overview of the Scenario (a full description copied from [83] is given in Appendix B). Section 6.2 discuses in detail the solution to the Scenario with focus on design issues and rationale, as opposed to actual codification of the process (which is given separately in Appendix B). Finally, Section 6.3 discusses methodology issues based on, but not only on, examples from the ISPW solution.

6.1 Overview of the Scenario

The scenario involves a software system that is under development and is in a relatively advanced phase, at a point where at least some parts of the system can be tested outside the development team. The process involves *test* and *change* tasks. Briefly, the process[2] is initiated at the testing phase, where a tester finds a problem, and reports it. The next step is the analysis of the problem, which produces a proposed solution, or a change request, identifying the source module(s) which might need to be modified in order to fix the problem. Thereafter, the change task starts, ".. according to pre-established change procedures (which entail assignment of resources, code and/or documentation modification, analysis/testing/review, approval/rejection..)", followed by actual modification of the code and reiteration to the testing phase. In addition to the base-scenario, the example suggests additional optional sub-scenarios (e.g., problem reporting/analysis, approval/rejection procedures) and recommends to demonstrate support for some specific capabilities (e.g., multi-user coordination, dynamic process changes).

In order to demonstrate the full modeling and enactment capabilities of Oz, we extended the Scenario along two dimensions (which may also be regarded as an extended solution to the original problem):

1. We discerned three teams, each responsible for a subset of the overall process and treated as a "site".
   (a) *Quality Assurance (QA)* — In charge of testing the system
   (b) *Coding* — The code development team

---

[2] In reality, the process described in the Scenario is really a small sub-process of the overall software process, but for the sake of brevity and clarity we will refer to it as a process, and will refer to smaller units within it as sub-processes.

   (c) *Design* — The design team, also supervising code development

2. We added steps to the process that require multi-user and/or cross-team collaborations.

6.2 Solution in Oz

The solution environment consists of three autonomous yet cooperating sub-processes that correspond to the three teams specified above. That is, each process can perform some tasks locally and independent of the other processes, while some sub-tasks might be dependent on other processes, or require interoperability with other sub-processes. Some of those interoperabilities require both modeling and enactment support for multi-user collaborations via synchronous multi-user tools, which may or may not cross sub-process boundaries.

The solution process defines four major tasks:

1. *Test* — Performed locally at QA.
2. *Analyze* — Performed by QA and Coding teams.
3. *Review* — Performed by Design and Coding teams.
4. *Change* — Performed mostly locally at Coding, but with small extensions to both Design and QA.

Thus, with the exception of the Review task — which was defined in the problem specification as a sub-task of the Change task and is modeled in our solution as a distinct task — all tasks map directly to the original problem specification.

Figure 6.1 depicts the high-level design of the solution. The ovals represent tasks, bold links represent general control-flow dependencies, and arrows represent artifacts that get generated in one task and used in a subsequent task. The dashed "clouds" in the figure represent other tasks that are not relevant to the Scenario and might be executed independently and concurrently.

We now turn to the description of the specific processes and their participation in the various multi-site tasks, which are modeled as Treaties and enacted as Summits. But first, we start with a description of the "product" we chose for the example.

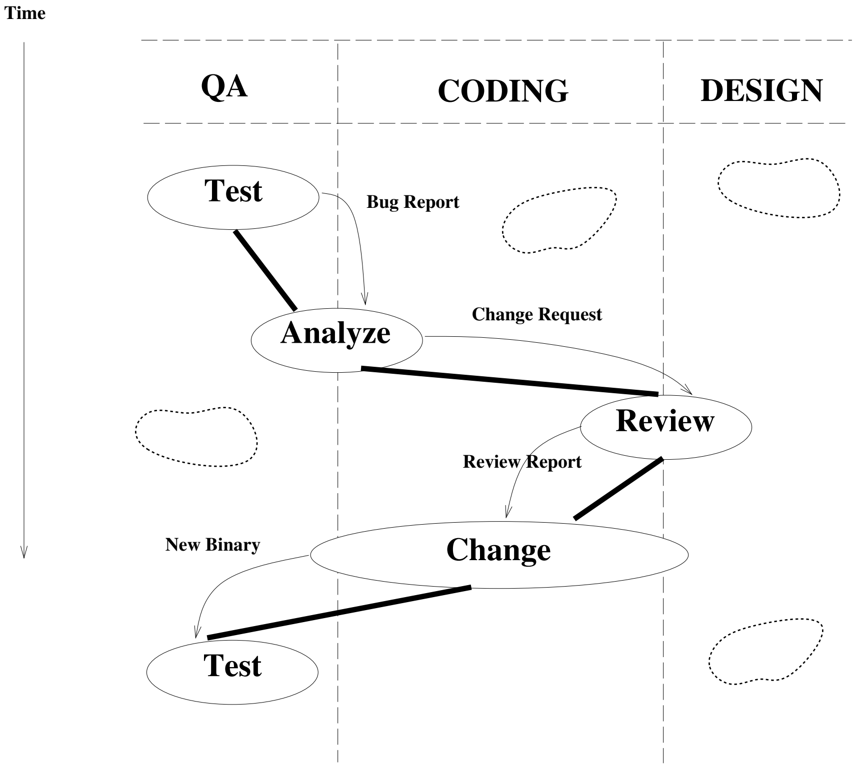


Figure 6.1: Process Design for ISPW-9 Example Scenario

6.2.1 The Product

It is important to outline the product being developed, since clearly, the characteristics of the intended product impact the process. Our choice for the product was the query processor for the Darkover Object Management System[3].

Briefly, the query processor receives as input a query in a format that is equivalent to the condition sub-language of MSL, applies the query to an object-oriented objectbase with class definitions that are equivalent to MSL's Data Definition Language, and returns the result of the query.

The reasons for choosing the query processor as our example product were: (1) It was still under (real) development, so readily available as a "real" example; (2) It was complex enough to justify a team of developers, but small enough to enable easy migration into Oz[4]; (3) It has a text-based user interface with well defined input and (expected)

---

[3] Darkover is being developed as part of the componentization of Oz; one of near-term future goals is to make Oz's OMS replaceable by an external system.

[4] 11 header files with 1375 lines of code, 11 source files with 2884 lines of code.

output. This characteristic enabled to construct an automatic testing sub-process. This is a typical example where the characteristics of the product affect the process. Obviously, a testing sub-process for a product that involves an interactive user-interface could not be fully automatic, since the essence of the testing is in the human interaction with the product.

6.2.2 The QA Process

The highly automatic testing (sub) task in our solution is *black-box*, i.e., we assume that the testers have neither access to, nor knowledge of, the source code.

The schema defines two main kinds of artifacts: a `TEST_SUITE` that represents test cases (both input and expected output), and can be grouped under a `TEST_SUITE_SET` object; and a `TEST_RUN` entity that represents the summary of the results of running a particular test (its instances can also be grouped under a `TEST_RUN_SET` object). Each `TEST_RUN` object is associated with a particular `TEST_SUITE_SET` object that contains a set of `TEST_SUITE` objects. `TEST_SUITE` objects are generated manually, and are relatively static, whereas `TEST_RUN` objects are generated automatically, and are relatively dynamic.

The process starts when a tester invokes the `start_test_run` rule (see strategy `test` in Appendix B), selecting the executable program to test, and a `TEST_SUITE_SET` object to use. This rule generates a `TEST_RUN` object, and spawns a sequence of executions of the `run_test` rule, one for each individual test suite. This rule runs the program with the input specified in the test suite, and stores the output of the execution in the `TEST_RUN` object. If any of the individual tests fails (which is indicated by a mismatch between the expected and the actual output) the rule automatically chains to the `report_bug` rule which forks a report generation tool that inserts all the relevant information such as the input and expected output of the test and the differences between them, and so forth. The `report_bug` rule then chains to the `notify_bug` rule which has an activity that is delegated to a user who is the manager of the Coding group. Figure 6.2 shows `notify_bug`. Note (lines 2-4) how the delegatee is determined dynamically by fulfilling the desired values for the `role` and `group` attributes, rather then hardwiring its value somewhere in the rule. If the delegatee is not logged-in at the moment of delegation, an asynchronous overloaded version of `notify_bug` is triggered, which notifies the delegatee about the pending activity (this is an example of the "programming trick" described earlier in Section 4.5.1.3). This completes the mostly

```
#########
#
#  notify_bug:
#
#  chained off report_bug, delegated to a user which is notified
#  of the problem
#
#########

notify_bug[?tr:TEST_RUN]:
  (exists PROGRAMMER ?p suchthat (and
                                     (?p.group = "CODING")
                                     (?p.role  = "Manager"))):

  delegate[?p.user_id]:
  (?tr.report_status = Reported)

#  Generate a notification message with all the necessary information
  { TEST_TOOLS notify_bug ?tr.report }

  (?tr.report_status = Notified);
```

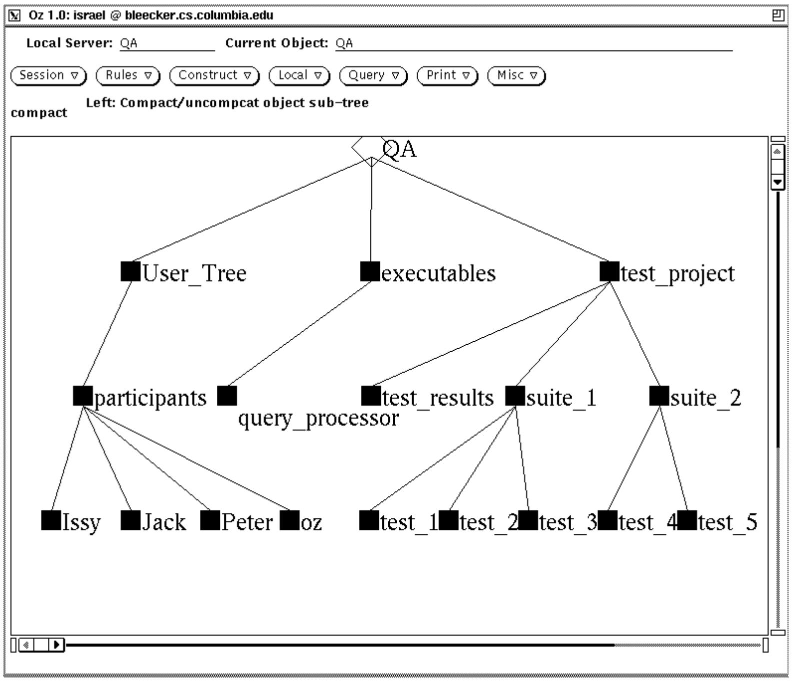
Figure 6.2: notify_bug Rule

local testing task.

Figure 6.3 shows a screen dump of the objectbase display at the testing site before any tests were performed (the client's objectbase display is zoomed at the QA site, thus the other SubEnv objects are not seen in the figure, as they are seen in figure 6.4); and figure 6.4 shows the objectbase after the execution of two test runs with the newly generated `TEST_RUN` objects (called `run_1` and `run_2`) properly linked to their respective `TEST_SUITE` object (`suite_1`) and to the tested executable (`query_processor`). Finally, figure 6.5 shows a trace of the execution of the automatic test task. Note how this execution invoked the asynchronous version of `notify_bug`, as evidenced by the message that was generated by the rule (shown in the bottom half of the window).

6.2.3 The Coding Process

Coding is the central process in the global environment, since the entire Scenario revolves around code changes. The main artifact in this process is naturally the source code,

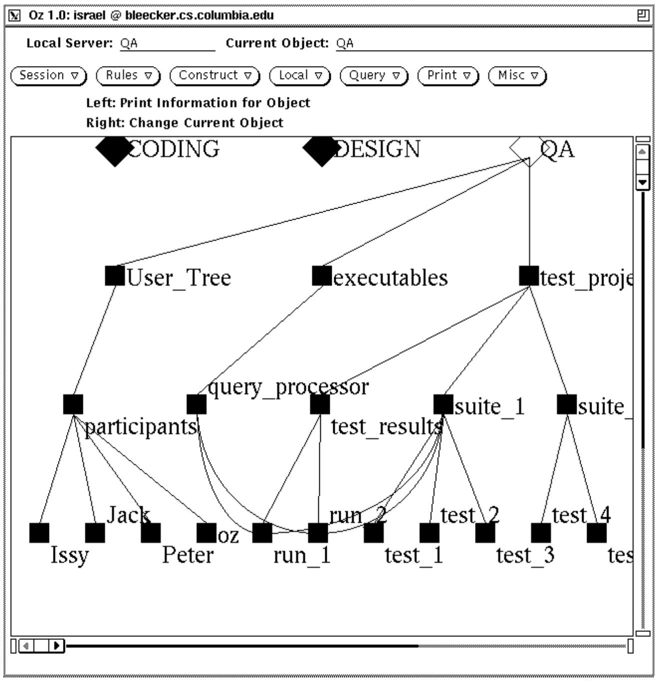


Oz 1.0: israel @ bleecker.cs.columbia.edu

Local Server: QA
Current Object: QA

Left: Compact/uncompcat object sub-tree
compact

Figure 6.3: The QA Objectbase

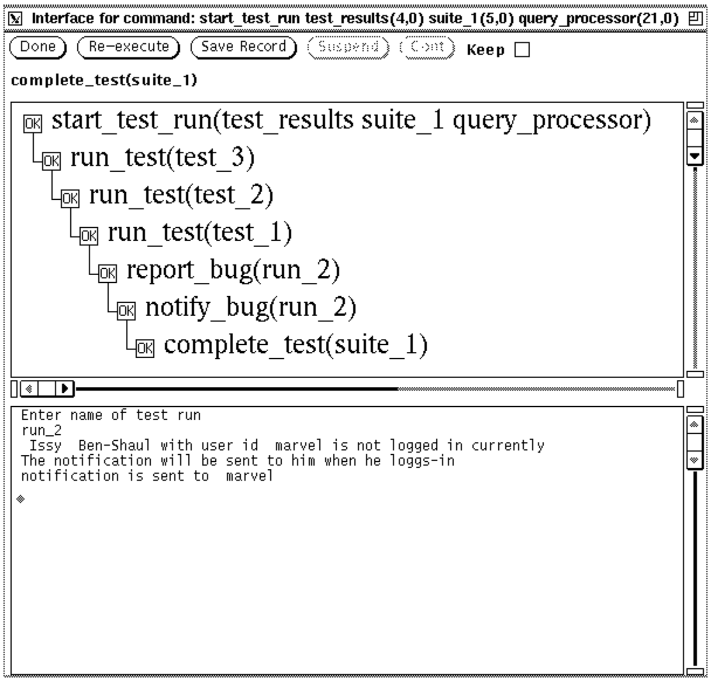


Oz 1.0: israel @ bleecker.cs.columbia.edu

Local Server: QA
Current Object: QA

Left: Print Information for Object
Right: Change Current Object

Figure 6.4: The QA Objectbase with New Test Runs



```
Enter name of test run
run_2
 Issy  Ben-Shaul with user id  marvel is not logged in currently
The notification will be sent to him when he loggs-in
notification is sent to  marvel
```

Figure 6.5: Rule Animation of the Testing Task

along with its derivatives and relatives (e.g., object code, libraries, documentation, configuration management). They are organized in a project hierarchy, with the `PROJECT` class at top, containing `MODULE`s, `BIN`aries, `LIB`raries, and so forth. The Schema further distinguishes between a shared stable "master" area, and private workspaces in which developers make their code development/modifications.

The Coding group performs the Analysis and Change tasks, and participates in the Review task. The Analyze task, depicted in Figure 6.6, consists of two phases. The general analysis phase (carried out by the `analyze_bug` rule) is performed by the manager of the Coding group (who was previously notified by QA), and he uses the bug report generated by the QA team along with its own source tree (thus, it is a Summit rule because it contains a parameter from a remote site). The manager tries to assess where the problem is (if there is any), ending up selecting a set of "suspected" source files. The second phase of the Analyze task consists of local forward implications at both the QA and the Coding groups. At the Coding group, each suspected file triggers an instance of a local overloaded version of the `analyze_bug` rule which is delegated to the file's "owner" (ownership is determined by the value of an attribute of the `CFILE` class). This rule provides the developer with all the

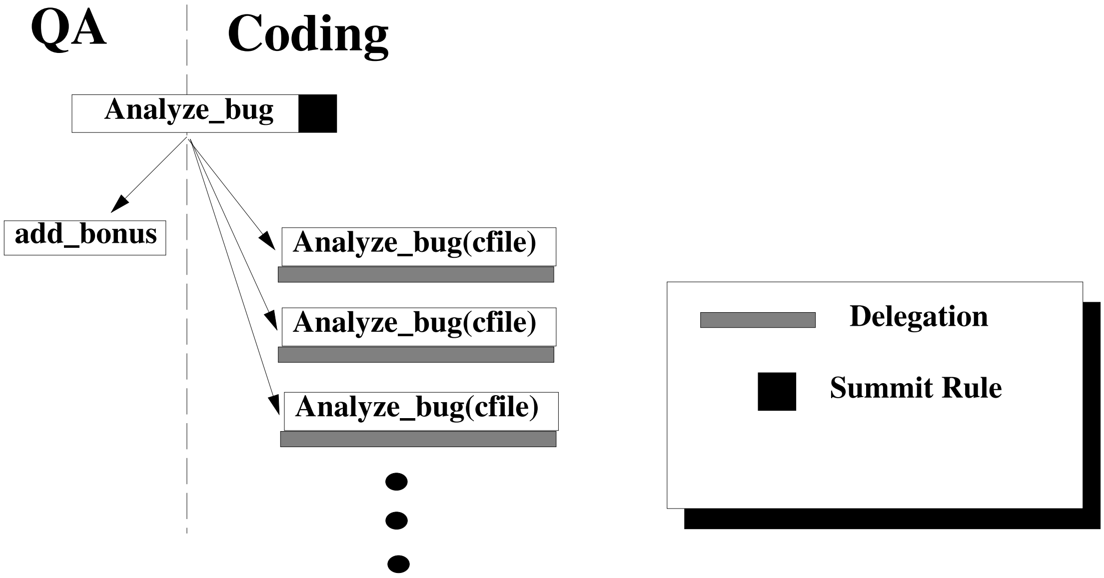


Figure 6.6: The Analyze Summit Task

past information that is relevant to the bug (e.g., reports from the QA group and from the manager), and the developer has to produce a change request report. At the QA group, the Summit triggers local chaining to the `add_bonus` rule which credits the person who found the bug. This is an example of a simple multi-site Summit rule that is integrated in both processes and triggers different local activities.

Analyze is followed by the Review task, which is performed jointly by the Coding and the Design groups. This is the most interesting task from the process-interoperability standpoint, and its discussion is deferred to Section 6.2.4.

The Change task is most comprehensive in terms of activities involved, but is essentially a local one. It is a scaled-down version of the real Marvel process which has been employed for the production of Oz, based on a check-out model, where developers check-out (check-in) artifacts from (to) the stable master area to (from) their private workspaces. Although mostly local, this task contains a single multi-user multi-process step, namely code inspection[5]. This step involves one participant from Design and one from Coding. Other steps in the Change task include: Interaction with the configuration management

---

[5] The actual synchronous multi-user inspection tool, developed in-house by Heineman and Skopp, defines one moderator and other participants, all of which are virtually sharing the same `emacs` buffer, in the sense that when the moderator moves its cursor it also moves it in the other participants. The tool also incorporates audio for verbal communication during the inspection process, and video stills to identify the participants.

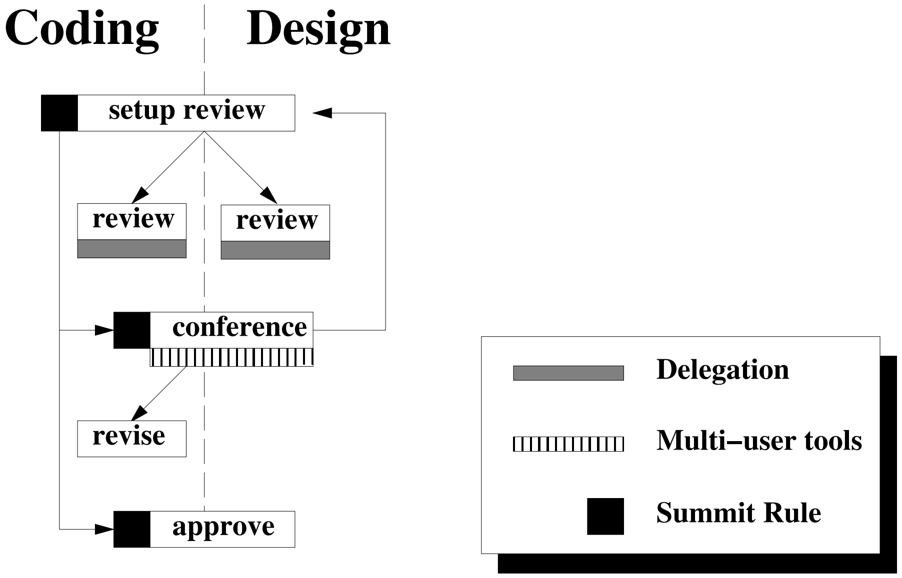


Figure 6.7: The Review Summit Task

sub-system, including sophisticated support for version branching and merging which was built on top of RCS [106]; Tools for editing, compiling, building and debugging the code fix, automation support for parts of the process, consistency propagations, and more (see Section B.2.3 for details).

The final step in the Change process is the `install_bin` Summit step that involves the QA and Coding processes, providing the QA team with the newly created binaries for (re)testing. Thus, the overlap of the Change Task with Design and QA as shown in figure 6.1, is due to the "isolated" inspection and binary-installation steps, respectively. The task is, however, mostly local.

6.2.4 The Design Process

The main artifacts in the Design process are of class `DESIGN_DOCUMENT` (which is a specialization of the `FILE` class), which can be grouped under objects of class `DESIGN_DOC`.

The Review task (shown in figure 6.7) is the only Summit task in which the Design process participates It is initiated with the invocation of the `setup_review` Summit rule. This rule collects the artifacts which are necessary for the review, including the change proposal produced by the Analyze phase. Then, the Summit fans-out to both sites for local (and delegated) reviews which are performed in parallel. Each reviewer summarizes his/her results and recommends whether to approve the proposed change, revise it, or reject it altogether. Once both reviews complete, then if both succeeded, the `approve` Summit rule is invoked automatically and completes the task (see below). Otherwise, if at least one of the reviews failed, the multi-user, multi-process `conference` rule is invoked, setting up both reviewers with a groupware tool called `white_board`[6] that allows them to discuss and reconcile their conflicts with respect to the proposed change. At the end of this step, there are two possibilities: either a recommendation for revision is made, or the change is rejected. In the former case, the `conference` rule leads to a local (delegated) `revise` step at the Coding process, in which the responsible developer(s) try to generate an improved change request. The `conference` rule is shown in Figure 6.8. Note how the rule binds the activity to two users based on objectbase information that relates them to the reviewed documents (lines 7-9); also note how the rule's condition "requires" at least one failed review (line 14-16); finally, note the two effects that correspond to the two possible outcomes mentioned above (lines 19-25).

The Review task is iterative, in that following the local `revise` step, the `setup_review` Summit rule is automatically invoked to start a second round of reviews, and so forth. If the Review task completes successfully, the `approve` rule automatically triggers the Change task, by automatically checking-out the faulty modules that need repair to the workspaces of the respective developer(s). Thus, this is a case of an automatic transition between high-level tasks. This example of a composite Summit shows the versatility of the interoperability mechanism, and particularly how it can be used to model "process negotiations", where the processes essentially interact with each other with the intention to reach an "agreement".

6.2.5 Treaty Definitions

We now turn to discuss Treaty definitions for the ISPW example. In general, the decentralized model was followed and its associated mechanisms were used, both for the initial definition as well as for incremental evolution of Treaties. Following the model, then, it is clear that any Summit rule discussed above must have been defined by a Treaty before it could be properly executed across sites. However, it is not immediately clear where should the Treaty rules originally be defined, nor is it clear how to assign execution privileges (i.e., who can run Summits and on what data) to the Treaty rules. Finally, another classification

---

[6] This is a public-domain tool that enables multiple remote users to share a virtual white-board on their screens.

```
#########
#
#  conference:
#
#  This is a multi-user Summit rule. It is invoked when some local
#  reviews fail. It forward chains to *local* revise at the CODING
#  process, and it forward chain to *Summit* set_up_review and approve
#  rules.
#########

  conference[?c:CFILE, ?design_doc:DESIGN_DOCUMENT]:
   (and
    (forall GROUP   ?coding suchthat (ancestor [?coding ?c]))
    (forall GROUP   ?design suchthat (ancestor [?design ?design_doc]))
    (forall MODULE  ?m      suchthat (member   [?m.cfiles ?c]))
    (forall DOCFILE ?d suchthat (member [?design_doc.docfiles ?d]))
    # bind the relevant participants to ?p
    (forall WORKSPACE ?p suchthat (or
                                  (linkto [?p.module ?m])
                                  (linkto [?p.doc  ?design_doc]))))
   :
delegate[?p.owner]:
    (and
      no_backward (?d.review_status = ReviewRequested)
      no_backward (?c.review_status = ReviewRequested)
      (or
        no_backward (?d.review_rc = Failed)
        no_backward (?c.review_rc = Failed)))

    # invoke the multi-user white_board tool

   {  MU_TOOLS confer ?c.contents ?d.contents
               ?coding.site_ip_addr ?design.site_ip_addr}

    # 0. ok, go to revise , and enable setup_review if revise succeeds
     (and
      (?c.review_status   = RevisionRequested)
    # this double assertion enables to chain to setup_review
      (?c.bug_status      = Suspected)
      (?c.bug_status      = Defected));
    # 1. no hope, go to reject. needs to start all over again.
     (and
     (?d.review_status = ChangeRejected)
     (?c.review_status = Rejected));
```

Figure 6.8: Conference Rule

	QA	CODING	DESIGN	interoperability	composite
Analyze	imp-acc	exp-req		p,d	no
Review		imp-acc	exp-req	p,d,u	yes
Inspection		exp-req	imp-acc	p,d,u	no
Install_bin	imp-acc	exp-req		p,d	no

Table 6.1: Treaties in the ISPW Process

of Treaties could be made along the lines of the kind of interoperability obtained by the Treaty. By definition, any Treaty provides for interoperability at the process and data levels. However, some Treaties also provide for "user-level" interoperability by means of multi-user tools. In fact, our Treaty and Summit mechanisms enabled us to implement sophisticated groupware tools with relatively small overhead, which could be regarded as one of the important by-products of this research (see more in Chapter 7).

The Analyze task should be executed only from the Coding process, since analyzing the bug is the responsibility of the developers. In addition, most of the involved artifacts reside in the Coding process, so the multi-site `analyze_bug` rule should also be defined at the Coding site. Thus, the (simple) Treaty is defined from Coding to QA, with Coding as the requester and QA the acceptor. Analyze involves only "standard" interoperability, i.e., at the process and data levels.

The Review task is more symmetric, and could potentially be both defined and executed from either the Coding or the Design process, with slight preference to Design on both accounts, due to the fact that Design has more impact on the outcome of the Review. Thus, it was defined with Design as the exporter-requester and Coding as the importer-acceptor. Recall that not all the steps shown in the Review Summit were actually defined as Treaty rules, only the multi-site rules, namely `setup_review`, `conference` and `approve`. Finally, Review is an example of a task with a "user-level" interoperability, obtained through the `conference` rule.

The other two Summit steps, namely code-inspection and binary installation, are isolated within the otherwise local Change task, and so should be defined in Coding as exporter and requester, with the other sites being importers and acceptors.

Table 6.2.5 summarizes the multi-process steps in the solution, along with information about the execution privileges set in the corresponding Treaties, and whether the interoperability involves process, data, and users (denoted by `p`, `d` and `u` in the `interoperability` column). The last column indicates whether the Summit is composite or not.

Several observations can be made with respect to Treaties and Summits in the process:

- All Treaties in the process were binary (i.e., among two sites). This, however, is a somewhat misleading observation in that one of the main reasons for avoiding definition of ternary Treaties was due to fact that they could not be shown at the demonstration[7]. Still, in retrospect it seems that for the most part binary Treaties were indeed sufficient here, and while nothing prevented technically from creating ternary treaties, it didn't seem necessary in this particular process.

- With the exception of the Review task, all Treaties in the example were defined in the Coding process, and more importantly, Coding was the sole requester. This observation is a bit misleading too. It stems mainly from the lack of people who could code Treaties in a timely manner. While one might argue that this might be the case in a real-world example, this example would have been defined differently under normal circumstances, with Treaty definitions and execution privileges spread more uniformly over the local processes.

- With the exception of the Review task, there was no need for full Treaties. Moreover, there was actually a need to explicitly distinguish execution privileges (for example, in the case of the Analyze task). This validates our approach to enabling a refined definition of Treaties as opposed to allowing only definition of Full Treaties. A related observation is that all Treaties were defined with the import-accept and export-request combination, which verifies our intuition towards specifying the default modes for Treaty.

- An interesting observation regarding Treaties is that they were used to model task *transitions*. This provides a new perspective on the role of Treaties, namely as an active mediation between (sub)processes, whereby both processes "meet" each other with their interfaces to facilitate the transition from one process to another. Task transitions are not the only reason for Summits, however (as evidenced by the code-inspection tool, for example), so one

---

[7] The demonstration at ISPW-9 was restricted to only two physical machines.

	QA	CODING	DESIGN	BUILT-IN	TOTAL
SEL	360	1787	163	346	2656
MSL	270	1233	58	233	1794
SCHEMA	46	256	30	95	427

Table 6.2: Summary of Lines of Code for the ISPW problem

should not draw the conclusion that interoperability is used only for task or process interface purposes.

6.2.6 Statistics and Summary of Solution

The entire solution was designed and coded by four people[8] over nine days. Although it appears that the process has been outlined top-down, it was mainly because the Scenario was defined without consideration of autonomy; most of the subtasks were defined autonomously and in a decentralized manner, with Treaties defined in most cases *after* the local sub-processes have been established. Each process was developed by a different person with the Treaties designed and implemented by the designers of the relevant SubEnvs. A summary of the total lines of code per process (including comments, which on the average account for about half of the lines of code in strategies) and their breakdown to schema, rules (MSL) and envelopes (SEL), is given in table 6.2.6.

The amount of Treaty code (i.e., strategies and envelopes that were part of Treaties) was 375 lines of MSL code and 344 lines of SEL code, accounting for about 15% of the overall code. The actual execution time of Treaty rules was small relative to local executions, too. Thus, in general most of the work was performed locally. Another interesting observation is that large portions of the Change task in the Coding process reused fragments from an already existing single-site process[9], and were adjusted to the requirements of the ISPW example (the main enhancement was to support version branching and merging in the configuration management task).

To make the process realistic, it was fully instantiated and enacted with a real product, which was "ozified" into an Oz environment[10]. Similarly, all the test cases were valid, as

---

[8] The author, George Heineman, Peter Skopp and Jack Yang.

[9] C/Marvel , a process that supports general purpose code development in the C programming language.

[10] A tool that aids in migrating file-system-based environments into Oz is currently under development, based on the Marvelizer [100] tool which migrated artifacts from the file system into a Marvel environment. We already employ a separate tool for upgrading Marvel environments into (single-site) Oz environments, and intend to combine both functionalities into the *ozify* tool.

well as the other artifacts that were maintained in the objectbases. All processes underwent numerous evolutions during the development cycle, both local and inter-site.

The following is a summary of the highlights of the ISPW solution:

- Three independent yet cooperating processes were constructed, assisting in the development of a multi-team software project, with control and data interleaving from being private for local work and shared for "global" work, as needed.
- Enforcement and automation were the main forms of enactment support. For example, enforcement of the constraint that prevents from depositing code before undergoing code-inspection[11]. An example for automation support was the automatic testing task.
- Support for both engineering tasks (e.g., the local Change task) and managerial tasks (e.g., assignment through delegation).
- Support for modeling and executing user-delegation, with emphasis on dynamic user binding as explained in Section 4.5.
- Support for modeling, executing, and integrating multi-user tools, including both in-house and off-the-shelf tools.
- Several additions, removal, and modifications to Treaties on the fly were made for evolution purposes (not described in this chapter).
- The modeling itself was decentralized, which contributed significantly to the effectiveness of the process-modeling process.

To conclude, the main validation was in the very fact that it was possible to fully implement in OZ an effective and comprehensive solution to the ISPW-9 example, both in terms of modeling as well as enactment, and including the multi-process and groupware extensions. This should be regarded as an important acceptance criteria by itself (both conceptual and technical), given that there are very few (if any at all) other PCEs with similar capabilities. A detailed evaluation is given in Chapter 7.

---

[11] This constraint was also implemented in our production Marvel process that supports the development of OZ, after it was realized that code was routinely deposited prematurely.

6.3 Methodology Issues

The main issue to explore here is, what is a recommended way to define decentralized processes. One of the objectives of this whole experiment was to observe not only the resultant environment, but also to observe the modeling meta-process and deduce from it a general methodology for defining processes. By-and-large, this is a topic for future work, as the experience we have had so far in modeling multi-site processes is obviously limited, and in enacting them even less so. Nevertheless, based on the conceptual framework of our approach, our rich experience with modeling single-site processes in Marvel[12], and combined with the few experiments we have conducted in modeling and enacting decentralized processes — several observations can be made, as well as recommendations on how to approach modeling.

The focus of this section is on methodology for the *design* and implementation phases. Although requirement specifications and analysis of the process are important phases (as in any other software engineering undertaking), methodology for these phases is beyond the scope of this section and is left unspecified. More specifically, the emphasis here is on modeling interoperability in multi-site environments and the impact on the overall modeling, as opposed to modeling stand-alone processes in a single-site environment. The following is a set of issues that have to be addressed when considering to build a multi-site environment instance:

1. The first issue to consider is whether a multi-site environment is at all necessary. In cases where scale and heterogeneity are reasonably contained, and physical and organizational boundaries do not exist (or can be somehow eliminated) there is probably no good reason for dividing an environment, in which case a single-site, single-process environment may suffice.

2. The next issue to consider is *how to divide an environment* into sub-processes. Several factors impact this division. As pointed out back in Chapter 1, in some cases the division is a given constraint, not a design consideration. For example, when some or all of the the SubEnvs already pre-exist, or when physical and/or organizational boundaries pre-define the division.

---

[12] e.g., CMarvel (mentioned earlier), PMarvel for process evolution and DocMarvel for document preparation, to name a few.

However, in cases where the designer(s) have some control over the division, two main axes for division can be identified:

- *Project-based decomposition* — The scale of the project and its complexity require to decompose it into a set of sub-projects and a corresponding set of groups, whereby each group is internally heterogeneous and is responsible for the complete development of a sub-project.
- *Task-based decomposition* — Here the project personnel is divided into groups such that each group is more homogeneous internally, has a designated role and is responsible for certain *tasks* within the overall global project. An example of such a division is the ISPW solution, where the processes corresponded to the QA, Coding, and Design tasks in the overall process.

While both decompositions are motivated by scale and heterogeneity, the former is slightly more associated with scale and the latter with heterogeneity. In any case, the two approaches are not mutually exclusive and some environments might employ a combination of both[13].

3. Once the division to SubEnvs exists (or if pre-existed), the next issue to address is *how to actually model* the processes. There are several non-functional requirements that might constrain or otherwise impact inter-process modeling:

    - A major constraint is bandwidth. The effective bandwidth between every two SubEnvs should be considered when designing Summits, and should be minimized when the bandwidth is low.
    - Another related consideration is whether the SubEnvs share a file system. In the case they do not, Summits should be defined so that the amount of transfer of bulk data (e.g., files) is minimized.
    - Time shifting (across geographically dispersed teams) should also be considered when modeling processes, particularly regarding process steps

---

[13] The natural extension to a hierarchy of SubEnvs was originally proposed in [13], but is beyond the scope of this thesis.

that depend on timing constraints. For example, synchronous multi-user activities should be restricted to times of day when there is overlap in working hours.

4. Another dimension of the actual modeling is related to the external requirements which were partially mentioned above. For example, pre-existing SubEnvs necessarily imply a bottom-up approach to modeling, with inter-site connections defined on top of the pre-existing (sub)processes, and with a high degree of process autonomy. Another consideration that might impact modeling is the nature of the organization. For example, a centralized organization might require a top-down approach to modeling multi-site environments, and might also limit site autonomy.

With all this in mind, we turn now to discussing a methodology for building an instance of what we consider a “mainstream environment”, where the external constraints are minimal and most of the major modeling aspects can be determined by the designer(s).

6.3.1 Approach to Modeling

Our recommended approach to modeling multi-site processes can be classified as a hybrid between bottom-up and top-down. The approach is also consistent with the general decentralized philosophy, and resembles the enactment of a Summit. That is, modeling is carried out by alternating between local mode — where the local processes are defined — and global mode — where Treaties are defined (and integrated within each local process), generally oriented towards maximizing locality.

More specifically, an Oz environment should be built by alternating between two orthogonal and interleaved iterations:

1. *data-process* — First, an initial schema should be defined, mostly covering the definition of the product data. Next, the process definition comes, which in turn requires enhancements to the schema, mostly for adding state attributes. This iteration continues until the process stabilizes, and can also be made after the process is enacted, as part of process evolution.

2. *local-global* — First, the private data and process should be defined (unless they pre-existed), followed by the definitions of the data and process to

support site-interoperability, which in turn lead to enhancing the private schema and process to integrate these steps, and so on. Again, this iteration continues until both the local processes and their interactions stabilize, and can also be refined later when the enacted process is evolved.

The ordering of steps to perform across these two axes depends on several factors, such as whether some of the local processes were pre-defined, the degree of interactions between the process, and so forth. Under minimum constraints, though, the recommended order is:

1. Define the local schema (if not already exists)
2. Define the shared sub-schema
3. Define the local process (if not already exists)
4. Define the shared tasks across processes (Treaties)

An important recommendation here that might be perceived as a divergence from the bottom-up decentralized approach is that if possible, the definition of shared sub-schema should come before the definition of local processes. Based on our experience, it proved much easier to lay the shared schema foundation before working on the local processes, as it facilitated a smoother composition of Treaties over the local processes. More discussion on the “globality” of data definition is given in Chapter 7.

To summarize, this section attempted to provide some methodology for modeling multi-site processes. However, the importance of this section goes beyond providing implementation tips and guidelines for designing specific Oz processes. The main research point was to assert that the approach for the (decentralized) modeling process (or meta-process) is very much tied to the approach for (decentralized) process modeling, and that similar arguments, models, and techniques, are applicable to both. And this assertion was verified through the ISPW example.

7

Summary, Evaluation, and Future Work

The main purpose of this research was to investigate the wide range of issues that are concerned with introducing decentralization (both inherent and by choice) into the process modeling and enactment research domain. To achieve this goal:

- A conceptual framework was built in the form of a generic (i.e., language- and system-independent) model that supports the definition, evolution, and execution of multiple autonomous and heterogeneous yet interoperating processes.
- A technological framework was constructed to investigate the application of the model, as well as to validate, evaluate, and provide feedback to improve the conceptual model.

Two key concerns guided this research: (1) maximization of local autonomy, both physically and logically, so as not to force a priori any global constraint on the definition, execution and operation of local sites, unless explicitly specified in a particular environment instance; and (2) flexibility and fine-grained control over the degree of interoperability. As would be expected, these two issues are central in the domains of decentralized systems and process modeling, respectively.

The essence of the approach to address decentralization was to extend the notions of process modeling and process enactment to inter-process modeling and inter-process enactment, respectively. The former was achieved by the *Treaty* model. In essence, Treaties are abstraction mechanisms that allow to specify shared sub-processes for interoperability purposes while retaining the privacy of the local sub-processes. Treaties have several unique characteristics: First, they require explicit and active participation of the involved entities to mutually agree on the nature of the interoperability, thereby balancing autonomy and global specification. Second, the definition of Treaties is fine-grained, in two respects: (1) they are defined pairwise, between every two sites that need to interoperate, as opposed to being global and known in all sites of a multi-site environment; and (2) each Treaty is formed over a single and small sub-process unit. Still, complex Treaties can be formed (and subsequently executed) between an arbitrary number of sites (not only two) and involve arbitrarily large sub-processes, by successive invocations of simple Treaties (which could be optimized from user interface perspective, as discussed earlier). The third characteristic of Treaties is that they are superimposed on top of pre-existing processes as opposed to being specified as part of each individual process; this enables gradual and incremental establishment of interoperability and supports the decentralized bottom-up approach. Fourth, they are designed to support local evolutions including unilateral retraction from Treaties, on demand.

Inter-process enactment was achieved by the complementary *Summit* model. Summits are execution abstraction mechanisms for Treaty-defined sub-processes. They support "global" execution of shared sub-processes involving artifacts and/or users from multiple sites, while maximizing local execution of related private sub-processes. This is done by successively alternating between shared and private execution modes: The former is used for the synchronous execution of the (fine-grained) shared activities, involving artifacts and/or users from multiple sites. The latter is used for the autonomous execution of any private subtasks emanating from prerequisites and consequences of the shared activities, thereby enabling to maintain process consistency according to the local processes, which is unknown to the the "global" executing task. And depending on the state and "willingness" of the local processes involved in the Summit, Summits may "re-convene", and so on. The decentralized model is the primary contribution of this research.

The second major contribution of this research is the application of the model to Oz. This work filled gaps left by the formal model, by discussing solutions to various issues that are lower-level but are nevertheless still applicable to a wide range of PMLs and PCEs, such as specification of common sub-schemas in an object-oriented database, and associating transaction semantics with the model. In addition, the realization contributed significantly to the design of the model itself. Indeed, it would be naive to assume that the choice of the rule-based PML and the basic architecture of the PCE had no impact on the general model, and it is no coincidence that the rule paradigm fits comfortably with this model. However, the rule-based modeling paradigm seems to be well-suited for decentralized modeling in its own right, regardless of our particular model, since: rules do not require top-down definition and more often are modeled in bottom-up fashion; they are loosely and implicitly interrelated (like decentralized processes); and they are context-less and fine-grained, allowing to minimize the impact of any interoperability mechanism (e.g., Summit) on the local processes.

Finally, the fact that the system was fully implemented enabled us to implement real environment instances with real processes, and to begin to explore "meta" issues such as methodology for defining processes and the viability of process modeling in general. The Oz project and system are far from complete, but they are both "alive".

The third major contribution of this research was in the investigation of infrastructure (or architectural) facilities to support a high degree of interconnectivity despite the "no-sharing" and non-transparency requirements that are essential for enabling physical decentralization. The general direction in most cases was to follow the "lazy-update" approach and tolerate a possibly temporary skewed view of the global state but add facilities that allow to both indicate this inconsistency (with some additional overhead) and repair it, on demand. Such was the case, for example, with the semi-replicated connection database and the remote object cache management.

There are two contributions which can be viewed as by-products of this research, but are nevertheless extremely important and form the basis for future research (see Section 7.2). The first is preliminary integration of *groupware* and process technologies. The concept of formally defining (modeling) collaboration as process steps and subsequently assisting in the execution of collaborative tools can be investigated regardless of decentralization. However, this research brought groupware to the forefront as the need for its use in processes became evident (as in the ISPW example). And at the same time it was realized that groupware "as-is", not integrated in a framework, and without modeling and enactment support, is limited too. The fact that groupware is particularly attractive for supporting physically dispersed users, means that decentralized and heterogeneous environments are a realistic setting in which to consider such integration. Moreover, proper infrastructure facilities for environment decentralization can be utilized to enable support for the integration of such tools into the environment framework. In other words, it seems that (system) interconnectivity enables (process) interoperability which in turn enables (human) collaboration. Thus, the marriage of process modeling and CSCW with the "blessing" of decentralization seems particularly attractive.

The second by-product was the use of a process model for site configuration purposes. There were several interesting aspects to this approach. First, it was an example of modeling and enacting an other-than-software process. Second, it required to push the PML to its limits with respect to its capabilities to access and manipulate low-level system facilities such as communication protocols; the investigation of the relationship between a PML and its underlying PCE (the "process machine") is in itself an interesting research topic. And third, it gave an opportunity to explore the possible domain of modeling decentralized systems (see Section 7.2).

7.1 Evaluation

We now turn to specific evaluation of the results of this research by considering how the decentralized model fulfills the requirements from Section 1.5[1]. Most of these issues have already come up in one way or another, so this is mainly a consolidation of them. We consider here both the formal model and the realization.

1. *Locality* — To a large degree, this requirement was met, both in the generic model and in Oz. The model was specifically designed to minimize the impact on local work. In particular, the approach of gradually superimposing interoperability on top of the underlying (possibly pre-existing and enactable) local processes, maximizes locality. As far as the impact of decentralization on the quality and performance of local work — this issue seems to have been successfully met, too. The overhead imposed by Oz on local work in a SubEnv compared with, say, work in an equivalent Marvel environment, is negligible. This is because the infrastructure overhead is directly

---

[1] Some of the requirements are coalesced here.

proportional to the degree of interoperability, so with no interoperability there is no overhead.

Another aspect that promoted locality both in the model and in the realization was the general evolutionary research approach, which, in analogy to decentralized systems, was built on top of pre-existing conceptual and technological foundations (i.e., the Marvel research and system, respectively). Indeed, one of the big pay-offs of this approach was the ability to thoroughly investigate and evaluate decentralization relatively quickly.

2. *Autonomy and independent operation* — Throughout the thesis, we have seen numerous cases where autonomy played a major role in determining the design of the model and the system. Perhaps the major aspect that fulfills this requirement is that site autonomy is the default and is guaranteed unless explicit specification of interoperability is made. This is in contrast to (typically distributed) systems in which the shareability is the default, and some work has to be done to protect the privacy of individual sites. Autonomy-by-default is closely related to enabling independent operation, but includes also definitional and execution aspects. Regarding definition, the schema, process, and database are all by default autonomous. The fine-grained modeling of Treaties contributes also to autonomy since each site can control precisely what is shared and what is not. The loose commitment to a Treaty that enables unilateral retraction further supports autonomy, even though it incurs some performance overhead in dynamically verifying Treaties at runtime. Regarding execution, the general idea in supporting autonomy was to minimize the impact of interoperability beyond what was explicitly defined as shared, and to maximize local execution. Most of these arguments hold equally well to the model as well as to OZ.

   The tension between supporting autonomy and enabling facilities for interoperability have led to some oversights regarding autonomy, however, mostly in the realization in OZ. One of them is the global objectbase browsing facility. While powerful in its ability to visualize whole remote objectbases and control their refresh policies, this mechanism is provided by default and does not require explicit definition. Global browsing has two functionalities —

the ability to view the contents of individual objects, and the ability to view and browse through the objectbase structure. The latter service cannot be disabled in Oz, and an improvement of this design might consider facilities for controlling the degree of browsing. The former can be controlled by specifying proper access-control permissions, but given that these permissions are optional, this too might be viewed as a violation of autonomy. Other built-in services provided by the kernel might also need to be improved in similar ways (e.g., cross site `copy`).

3. *Interoperability* — Given that autonomy was a crucial requirement, this "competing" requirement seems to also have been adequately addressed. Specifically, the execution semantics (and the corresponding infrastructure in the realization) support well the enactment of activities involving data from multiple sites. The additional facilities for modeling and enacting delegation and groupware tools facilitate collaboration, too. And finally, the infrastructure support and the built-in services enable interoperability.

   The areas where some improvements might be needed are: (1) Not having any means to associate data artifacts across sites, proved to be a limitation. A solution that addresses this issue with minimal impact on autonomy (e.g., the soft link proposal in Section 4.2.1.1) would be useful. (2) More operations that facilitate global definition (for both process and schema) may prove useful, especially in tightly-coupled environments, where autonomy can be compromised. The Treaty operation as a single command (with the issuer being administrator in both sites) was a step in that direction. Other possible improvements in process modeling include commands for defining multi-site Treaties, more selective Treaty invalidation procedures that do not invalidate Treaties unnecessarily, automatic updates of strategies without requiring to re-establish Treaties, and so forth.

   Finally, Summits represent one model of execution, in which interoperability is attained through shared activities. Other models should be considered as supplementary alternatives, e.g., message passing between local activities with no data sharing.

4. *Support for pre-existing and heterogeneous processes* — This is an extension of the previous requirement, since the requirement is to support interoperability over such heterogeneous processes. As already outlined above, as far as process is concerned, both Summits and Treaties were designed with this requirement in mind, and proved to be quite effective. It is of course possible and even likely that two pre-existing and unrelated processes will have no common sub-process a priori. But "bridges" of interoperability can be incrementally added, with minimal distractions to local work. The situation with pre-existing *schemas* is less satisfactory, however, particularly with respect to strongly typed PMLs. Such PMLs should provide facilities that enable to superimpose new shared sub-schemas on top of the pre-existing ones (perhaps along the lines of what is done in Pegasus [29]). Alternatively, PMLs might need to sacrifice some of their typing restrictions, at least for Summit rules, to accommodate heterogeneous schemas.

5. *Infrastructure support* — The comprehensive infrastructure that was built in Oz to support interconnectivity seems to have addressed the "conflicting" independent operation requirement. For example, the Connection Server as an auxiliary entity for (re)establishing connections across sites and for (re)activating servers enabled sites to operate independently but acquire the necessary information when they needed to communicate with other sites. The main open issue here is proper modeling support for operation over a wide-area network (see Section 7.2).

7.2 Future Directions

7.2.1 Modeling of Decentralized Systems

This thesis explored how decentralization impacts process modeling. An interesting topic to explore is the opposite direction, namely, how modeling and enactment can be applied to decentralized systems. It seems that the idea of describing the behavior of autonomous entities formally, as a basis for constructing consistent and trustworthy interoperability among them, and operating within an environment that supports their execution, goes beyond process modeling and can be applied to general distributed and decentralized system design. For example, this could be used to model and subsequently support interoperability among autonomous Internet repositories, making them more active and responsive to other objects on the network.

Some related work in this direction has already been done in the area of interconnection languages (e.g., Conic [75]), and in the more general emerging field of software architecture [36], where researchers have been looking at formalisms to define the structure of complex (distributed) systems (e.g. [1]).

7.2.2 Wide Area Modeling

This direction has been addressed in Oz only preliminarily at best (in Section 5.6). The main research issue here is to explore constructs for modeling things like physical distance, (perhaps dynamically varying) bandwidth, time differences and other temporal constraints, security level, location in case of mobile sites, frequency of inter-site interactions and more — in order to support the optimization, synchronization and in general the operation of sites collaborating over a wide area network. The key issue is to use the semantics of the explicit specifications to improve the the enactment support. For example, a PCE might employ data prefetching when the bandwidth is low, and use further information in the process model to improve the hit/miss ratio.

7.2.3 User Modeling, Groupware and Process

This topic has been already discussed above, and some preliminary work in this direction has been already done in this thesis (Sections 3.6, 4.5 and 6.2), as well as by other members in our research group [14, 108, 107]. The research path to follow here is to continue to increase the modeling capabilities and abstractions that define various aspects of collaboration, use the semantics of the model to enhance the enactment and the integration of CSCW tools, and characterize the necessary infrastructure support for supporting CSCW in the process, such as advanced transaction facilities and object caching. One particularly neglected aspect of modeling is user modeling. This has several implications such as unique identification of users across sites, aliasing of "users" from multiple sites to the same nomadic user, security issues, assignment of roles, end so on.

7.2.4 System and Language Heterogeneity

To re-quote Heimbigner [43], "environments will move to looser, federated, architectures... address inter-operability between partial-environments of varying degrees of openness". We certainly agree with this assessment.

This thesis focused on heterogeneity at the process level, and for the most part assumed a homogeneous underlying framework and modelling language. Multi-formalism and system interoperability, are still, by-and-large, open research issues, particularly when coupled with decentralization. One promising approach to follow is to construct a "virtual process machine" (i.e., generic PCE) with low-level services (both centralized and distributed) that is decoupled from a particular formalism in which process models are specified, and thus supports any formalism that can be translated into the machine's "assembly" language. Such machines should be capable of communicating with other heterogeneous machines using underlying standards (e.g., CORBA [80]) and service explicit as well as implicit (i.e., not-coded) processes.

Bibliography

[1] Robert Allen and David Garlan. Formalizing architectural connections. In *16th International Conference on Software Engineering*, pages 71–80, Sorrento, Italy, May 1994. IEEE Computer Society Press. Panel Introduction.

[2] Robert Balzer. A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering*, SE-11(11):1257–1268, November 1985.

[3] 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.

[4] Sergio Bandinelli, Alfonso Fuggetta, Carlo Ghezzi, and Sandro Grigolli. Process enactment in SLANG. In J.C. Derniame, editor, *Software Process Technology Second European Workshop*, number 635 in Lecture Notes in Computer Science. Springer-Verlag, Trondheim, Norway, September 1992.

[5] Jay Banerjee and Won Kim. Semantics and implementation of schema evolution in object-oriented databases. In *ACM SIGMOD Annual Conference on the Management of Data*, pages 311–322, San Francisco CA, May 1987. Special issue of *SIGMOD Record*, 16(3), December 1987.

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

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

[8] Naser S. Barghouti and Gail E. Kaiser. Scaling up rule-based development environments. *International Journal on Software Engineering & Knowledge Engineering*, 2(1):59–78, March 1992.

[9] David R. Barstow, Howard E. Shrobe, and Erik Sandewall (editors). *Interactive Programming Environments*. McGraw-Hill, New York, 1984.

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

[11] Noureddine Belkhatir, Jacky Estublier, and Walcelio L. Melo. Software process model and work space control in the Adele system. In *2nd International Conference on the Software Process: Continuous Software Process Improvement*, pages 2–11, Berlin Germany, February 1993. IEEE Computer Society Press.

[12] M. Ben-Ari. *The Ada Rendezvous*, chapter 6, pages 93–105. Prentice Hall International, Englewood Cliffs, NJ, 1982.

[13] 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.

[14] Israel Z. Ben-Shaul, George T. Heineman, Steve S. Popovich, Peter D. Skopp, Andrew Z. Tong, and Giuseppe Valetto. Integrating groupware and process technologies in the Oz environment. In Carlo Ghezzi, editor, *9th International Software Process Workshop*, Airlie VA, October 1994. IEEE Computer Society Press. In press.

[15] 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. IEEE Computer Society Press.

[16] 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.

[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. IEEE Computer Society Press.

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

[20] Omran A. Bukhres, Jiansan Chen, Weimin Du, and Ahmed K. Elmagarmid. Interbase: An execution environment for heterogeneous software systems. *Computer*, 26(8):57–69, August 1993.

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

[22] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2), 1959.

[23] Donald Cohen. Automatic compilation of logical specifications into efficient programs. In *5th National Conference on Artificial Intelligence*, volume Science, pages 20–25, Philadelphia, PA, August 1986. AAAI.

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

[25] Prasun Dewan and Rajiv Choudhary. A high-level and flexible framework for implementing multiuser user interfaces. *ACM Transactions on Information Systems*, 10(4):345–380, October 1992.

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

[27] Mark Dowson. Istar — an integrated project support environment. In Peter Henderson, editor, *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 27–33, Palo Alto, CA, December 1986. Special issue of *SIGPLAN Notices*, 22(1), January 1987.

[28] Anthony Earl. Principles of a reference model for computer aided software engineering environments. In Fred Long, editor, *Software Engineering Environments International Workshop on Environments*, volume 467 of *Lecture Notes in Computer Science*, pages 115–129, Chinon, France, September 1989. Springer-Verlag.

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

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

[31] Marc I. Kellner *et al.* Software process modeling example problem. In *1st International Conference on the Software Process: Manufacturing Complex Systems*, pages 176–186, Redondo Beach CA, October 1991.

[32] 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.

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

[34] Pankaj K. Garg, Peiwei Mi, Thuan Pham, Walt Scacchi, and Gary Thunquest. The SMART approach for software process engineering. In *16th International Conference on Software Engineering*, pages 341–350, Sorrento, Italy, May 1994. IEEE Computer Society Press.

[35] P.K. Garg, T. Pham, B. Beach, A. Deshpande, A. Ishizaki, K. Wentzel, and W. Fong. Matisse: A knowldge-based team programming environment. *International Journal of Software Engineering and Knowledge Engineering*, 4(1):15–59, 1994.

[36] David Garlan and Dewayne Perry. Software architecture: Practice, potential, and pitfalls. In *16th International Conference on Software Engineering*, pages 363–364, Sorrento, Italy, May 1994. IEEE Computer Society Press. Panel Introduction.

[37] Carlo Ghezzi, editor. *9th International Software Process Workshop*, Airlie VA, October 1994. IEEE Computer Society Press.

[38] 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.

[39] Volker Gruhn and Rudiger Jegelka. An evaluation of FUNSOFT nets. In J.C. Derniame, editor, *Software Process Technology Second European Workshop*, number 635 in Lecture Notes in Computer Science, pages 196–214. Springer-Verlag, Trondheim, Norway, September 1992.

[40] Nico Haberman and Dewayne Perry. *Ada for Experienced Programmers.* Addison-Wesley, Reading, MA, 1983.

[41] A.N. Habermann and D. Notkin. Gandalf: Software development environments. *IEEE Transactions on Software Engineering*, SE-12(12):1117–1127, December 1986.

[42] Dennis Heimbigner. Proscription versus Prescription in process-centered environments. In Takuya Katayama, editor, *6th International Software Process Workshop: Support for the Software Process*, pages 99–102, Hakodate, Japan, October 1990. IEEE Computer Society Press.

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

[44] 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.

[45] Dennis Heimbigner and Marc Kellner. Software process example for ISPW-7, August 1991.

[46] George T. Heineman. A transaction manager component for cooperative transaction models. In Ann Gawman, W. Morven Gentleman, Evelyn Kidd, Perke Larson, and Jacob Slonim, editors, *1993 CASCON Conference*, pages 910–918, Toronto Ontario, Canada, October 1993. IBM Canada Ltd. Laboratory and National Research Council Canada.

[47] George T. Heineman. A transaction manager component for cooperative transaction models. Technical Report CUCS-017-93, Columbia University Department of Computer Science, July 1993. PhD Thesis Proposal.

[48] George T. Heineman and Gail E. Kaiser. Incremental process support for code reengineering. In *International Conference on Software Maintenance*, pages 282–290, Victoria BC, Canada, September 1994.

[49] George T. Heineman and Gail E. Kaiser. An architecture for integrating concurrency control into environment frameworks. In *17th International Conference on Software Engineering*, Seattle WA, April 1995. In press.

[50] 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.

[51] SynerVision for SoftBench: A Process Engine for Teams, 1992. Marketing literature.

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

[53] M. Honda. Support for parallel development in the sun network software environment. In *2nd International Workshop on Computer-Aided Software Engineering*, pages 5–5 – 5–7, 1988.

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

[55] Watts S. Humphrey. *Managing the Software Process*. Addison-Wesley, Reading MA, 1989.

[56] Hajimu Iida, Takeshi Ogihara, Katsuro Inoue, and Koji Torii. Generating a menu-oriented navigation system from formal description of software development activity sequence. In Mark Dowson, editor, *1st International Conference on the Software Process: Manufacturing Complex Systems*, pages 45–57, Redondo Beach CA, October 1991. IEEE Computer Society Press.

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

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

[59] 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.

[60] Gail E. Kaiser, Steven S. Popovich, and Israel Z. Ben-Shaul. A bi-level language for software process modeling. In *15th International Conference on Software Engineering*, pages 132–143, Baltimore MD, May 1993. IEEE Computer Society Press.

[61] Takuya Katayama. A hierarchical and functional software process description and its enaction. In *11th International Conference on Software Engineering*, pages 343–352, Pittsburgh PA, May 1989. IEEE Computer Science Press.

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

[63] Rick Kazman, Len Bass, Gregory Abowd, and Mike Webb. SAAM: A method for analyzing the properties of software architectures. In *16th International Conference on Software Engineering*, pages 81–90, Sorrento, Italy, May 1994. IEEE Computer Society Press.

[64] Marc I. Kellner and H. Dieter Rombach. Session summary: Comparisons of software process descriptions. In Takuya Katayama, editor, *6th International Software Process Workshop: Support for the Software Process*, pages 7–18, Hakodate, Japan, October 1990. IEEE Computer Society Press.

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

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

[67] Balachander Krishnamurthy and Naser S. Barghouti. Provence: A process visualization and enactment environment. In Ian Sommerville and Manfred Paul, editors, *4th European Software Engineering Conference*, number 717 in Lecture Notes in Computer Science, pages 451–465. Springer-Verlag, Garmisch-Partenkirchen, Germany, September 1993.

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

[69] David B. Leblang and Robert P. Chase, Jr. Computer-aided software engineering in a distributed workstation environment. In Peter Henderson, editor, *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 104–112. ACM Press, April 1984. Special issue of *SIGPLAN Notices*, 19(5), May 1984.

[70] Manny Lehman, editor. *Software Process Workshop*, Egham, Surrey, UK, February 1984. IEEE Computer Society Press.

[71] Manny M. Lehman. Process models, process programs, programming support. In *9th International Conference on Software Engineering*, pages 14–16, Monterey, CA, March 1987.

[72] Andrew Lih. Oz firewall support, February 1994. Project-course E6998y.

[73] Lion Engineering Enviornment, 1994. Marketing literature.

[74] Jeff Magee, Naranker Dulay, and Jeff Kramer. Sturcturing parallel and distributed progrmas. *Software Engineering Journal*, 8(2):73–82, March 1993.

[75] Jeff Magee, Jeff Kramer, and Morris Sloman. Constructing distributed systems in Conic. *IEEE Transactions on Software Engineering*, 15(6):663–675, June 1989.

[76] Peiwei Mi and Walt Scacchi. Modeling articulation work in software engineering processes. In Mark Dowson, editor, *1st International Conference on the Software Process: Manufacturing Complex Systems*, pages 188–201, Redondo Beach CA, October 1991. IEEE Computer Society Press.

[77] Peiwei Mi and Walt Scacchi. Process integration in CASE environments. *IEEE Software*, 9(2):45–53, March 1992.

[78] Naftaly H. Minsky and David Rozenshtein. A software development environment for law-governed systems. In Peter Henderson, editor, *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 65–75, Boston MA, November 1988. ACM Press. Special issue of *SIGPLAN Notices*, 24(2), February 1989 and of *Software Engineering Notes*, 13(5), November 1988.

[79] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* Information Systems. The MIT Press, Cambridge MA, 1985.

[80] John R. Nicol, C. Thomas Wilkes, and Frank A. Manola. Object orientation in heterogeneous distributed computing systems. *Computer*, 26(6):57–67, June 1993.

[81] Leon Osterweil. Software processes are software too. In *9th International Conference on Software Engineering*, pages 1–13, Monterey CA, March 1987. IEEE Computer Society Press.

[82] James D. Palmer and N. Ann Fields. Computer supported cooperative work. *Computer*, 27, May 1994.

[83] Maria H. Penedo. Life-cycle (sub) process demonstration scenario, March 1994. 9th International Software Process Workshop (ISPW9).

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

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

[86] James L. Peterson. *Petri Net Theory and The Modeling of Systems.* Prentice-Hall, Englewood Cliffs NJ, 1981.

[87] Burkhard Peuschel and Stefan Wolf. Architectural support for distributed process centered software development environments. In Wilhelm Schäfer, editor, *8th International Software Process Workshop*, pages 126–128, Wadern, Germany, March 1993. IEEE Computer Society Press.

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

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

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

[91] Steven P. Reiss. An approach to incremental compilation. In *SIGPLAN '84 Symposium on Compiler Construction*, pages 144–156, Montreal, Canada, June 1984. Special issue of @i[SIGPLAN Notices], 19(6), June 1984.

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

[93] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors.* Texts and Monographs in Computer Science. Springer-Verlag, New York, 1989.

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

[95] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990.

[96] Yoichi Shinoda and Takuya Katayama. Towards formal description and automatic generation of programming environments. In Fred Long, editor, *Software Engineering Environments International Workshop on Environments*, number 467 in Lecture Notes in Computer Science, pages 261–270. Springer-Verlag, Chinon, France, September 1989.

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

[98] Peter D. Skopp. Process centered software development on mobile hosts. Technical Report CUCS-035-93, Columbia University Department of Computer Science, October 1993. MS Thesis Proposal.

[99] 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.

[100] Michael H. Sokolsky and Gail E. Kaiser. A framework for immigrating existing software into new software development environments. *Software Engineering Journal*, 6(6):435–453, November 1991.

[101] Ian Sommerville. *Software Engineering.* Addison-Wesley, Reading, MA, 1989.

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

[103] Robert E. Strom, David F. Bacon, Arthur P. Goldberg, Andy Lowry, Daniel M. Yellin, and Shaula Alexander Yemini. *Hermes A Language for Distributed Computing.* Prentice-Hall, Englewood Cliffs NJ, 1991.

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

[105] Carl D. Tait and Dan Duchamp. Detection and exploitation of file working sets. In *11th International Conference on Distributed Computing Systems*, pages 2–9, Arlington TX, May 1991. IEEE Computer Society Press.

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

[107] 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.

[108] Giuseppe Valetto. Expanding the repertoire of process-based tool integration. Master's thesis, Columbia University, Department of Computer Science, December 1994. CUCS-027-94.

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

Appendix A

Configuration Process Sources

A.1 Registration Strategy

```
#
#                        Oz Process Centered Environment
#
#                                 Copyright 1994
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
#  Registration process. This is a system built-in strategy

strategy register

imports data_model;
exports all;

objectbase

#
# Tool Declarations
#

REGISTER :: superclass TOOL;
    register_subenv    : string = register_subenv;
    deregister_subenv  : string = deregister_subenv;
    send_connection_db : string = send_connection_db;
    change_subenv_name : string = change_subenv_name;
end

end_objectbase

rules
```

```
#########
#
# register_subenv:
#
# collect static information about the new subenv,
# and replicate it in all existing subenvs
#
#########

register_subenv [?new_name:LITERAL]:

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

# this envelope actually does the replication in remote subenvs
    { REGISTER register_subenv ?new_name ?se.subenv_name ?se.site_name
                               ?se.has_nfs  ?lse.subenv_name
                               return ?new_subenv_id
                                      ?new_subenv_name
                                      ?new_site_name
                                      ?new_ip_addr}
    # now add the object locally with the information entered
    # by the administrator
   (and
      no_chain (?new:GROUP = add [NULL NULL ?new_name GROUP])
               (?new.subenv_id    = ?new_subenv_id)
               (?new.subenv_name  = ?new_subenv_name)
               (?new.site_name    = ?new_site_name)
               (?new.site_ip_addr = ?new_ip_addr)
               (?new.local        = false)
               (?new.state        = New));

#########
#
# init_remote_subenv:
#
# called from within the resister_subenv envelope (in batch mode),
# from all remote SubEnvs, to assign the proper values to the
# objects which were just added by register_subenv
#
#########

hide init_remote_subenv [?env_obj_name:LITERAL, ?env_name:LITERAL,
                         ?env_id:LITERAL,       ?subenv_id:LITERAL,
                         ?subenv_name:LITERAL,  ?site_name:LITERAL,
```

```
                          ?site_ip_addr:LITERAL]:
    :
    {}

    (and
      no_chain (?new:GROUP = add [NULL  NULL ?env_obj_name GROUP])
               (?new.env_name     = ?env_name)
               (?new.env_id       =  ?env_id)
               (?new.subenv_id    = ?subenv_id)
               (?new.subenv_name  = ?subenv_name)
               (?new.site_name    = ?site_name)
               (?new.site_ip_addr = ?site_ip_addr)
               (?new.state        = New)
               (?new.local        = false));

#########
#
# send_connection_db:
#
# initialize the newly created subenv, by sending to it the
# connection database
#
#########

send_connection_db [?nse:GROUP]:

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

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

    (?nse.state = Initialized);

#########
#
# init_subenv_map:
#
```

```
#
# called from within the envelope (by invoking a batch client)
# of send_connection_db, this assigns to all objects of the
# connection db the proper values.
#
#########

hide init_connection_db [?new:GROUP, ?env_name:LITERAL, ?env_id:LITERAL,
            ?subenv_id:LITERAL, ?subenv_name:LITERAL,
    ?site_name:LITERAL, ?site_ip_addr:LITERAL, ?local:LITERAL]:
    # no condition
    :
    # no activity
    { }

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

#########
#
# deregister_subenv:
#
# Remove a subenv, by removing its root object from all remote
# subenvs and removing all subenvs from the subenv that is removed.
# Leave the subenv "disconnected", but don't remove it.
#
#########

deregister_subenv [?lse:GROUP]:

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

    { REGISTER deregister_subenv ?se.Name ?se.subenv_name ?lse.Name}
    (delete [?se NULL] );


#########
#
# change_subenv_location:
#
# Move the physical location of a subenv.
```

```
# ASSUMES NFS !
#
#########

change_subenv_location [?se:GROUP, ?new_se_name:LITERAL]:
    :
    no_chain (?se.active = false)

    { }

    (?se.subenv_name = ?new_se_name);

change_subenv_location [?se:GROUP]:

    :
    no_chain (?se.active = false)

    { REGISTER change_subenv_name return ?new_se_name }

    (?se.subenv_name = ?new_se_name);
```

A.2 A Sample Oz Envelope

```
#
#                         Oz Process Centered Environment
#
#                                  Copyright 1994
#                       The Trustees of Columbia University
#                              in the City of New York
#                                All Rights Reserved
#
# add a site


ENVELOPE register_subenv;

SHELL ksh;

 INPUT
# the input matches the paramters specified in the register_subenv rule
  string            : new_name;
  set_of string     : subenv_name;
  set_of string     : site_name;
  set_of integer    : has_nfs;
  set_of string     : local_subenv_name;

 OUTPUT
  string : ?new_subenv_id;
```

```
  string : ?new_subenv_name;
  string : ?new_site_name;
  string : ?new_ip_addr;
 BEGIN

# get the new subenv_name

trap '/bin/rm -f /tmp/ping.out /tmp/oz_batch /tmp/host.out' 0 1 2 3 9 15

echo "Enter New SubEnv Name (Currently path name)"
read new_subenv_name

if [ "x$new_subenv_name" = "x" ]
then
    echo "Must enter new path name"
    RETURN "1":"", "", "", "";
fi

echo "Enter SubEnv ID (increment the largest subenv_id in the environment)"
read new_subenv_id

if [ "x$new_subenv_id" = "x" ]
then
    echo "Must enter new subenv id"
    RETURN "1":"", "", "", "";
fi

echo "Enter the name of the primary host (usually where the SubEnv resides)"
read new_site_name

/usr/etc/ping $new_site_name > /tmp/ping.out 2>&1

if [ $? != 0 ]
then
   cat /tmp/ping.out
   RETURN "1":"", "", "", "";

fi

host $new_site_name > /tmp/host.out

if [ $? != 0 ]
then
  cat /tmp/host.out
  RETURN "1":"", "", "", "";
fi

awk '{ if (NR = 1) {print $4}}' /tmp/host.out | read new_ip_addr

if [ "x$ip_addr" = "x" ]
```

```
then
   echo "Ip address could not be resolved."
   echo "Please enter the address or <return> to abort"
   read new_ip_addr
   if [ "x$new_ip_addr" = "x" ]
   then
        echo "Registration process aborted"
        RETURN "1":"", "", "", "";
        exit
   fi
fi

echo "#!oz script" > /tmp/oz_batch
echo "init_remote_subenv \"$new_name\" \"oz1\" \"0\" \"$new_subenv_id\" \"$new_subenv_name\" \"$

rm /tmp/subenv_name  > /dev/null 2>&1
rm /tmp/site_name    > /dev/null 2>&1
rm /tmp/has_nfs      > /dev/null 2>&1
rm /tmp/combined     > /dev/null 2>&1

for i in $subenv_name
do
  echo $i >> /tmp/subenv_name
done

for i in $site_name
do
  echo $i >> /tmp/site_name
done

for i in $has_nfs
do
  echo $i >> /tmp/has_nfs
done

# if only the local site exists skip this, because shuffle will
# bark on this

if [ "x$subenv_name" != "x" ]
then
  shuffle /tmp/subenv_name /tmp/site_name /tmp/has_nfs > /tmp/combined
  if [ $? != 0 ]
  then
    echo "register_subenv failed during shuffle"
    RETURN "1":"", "", "", "";
  fi
  nr=`awk 'END {print NR}' /tmp/combined`
else
  nr=0
fi
```

```
ctr=1

# add the new root object to all other SubEnvs

while [ $ctr -le $nr ]
do
  cur_subenv_name=‘awk ’{if (NR == ’$ctr’) print $1}   ’ /tmp/combined‘
  cur_site_name=‘awk ’{if (NR == ’$ctr’) print $2}   ’ /tmp/combined‘
  cur_has_nfs=‘awk ’{if (NR == ’$ctr’) print $3}   ’ /tmp/combined‘

  if [ $cur_has_nfs = "1" ]
  then
    oz_tty -a -b /tmp/oz_batch $cur_subenv_name
  else
   oz_tty -a -r $cur_site_name -b /tmp/oz_batch $cur_subenv_name
  fi

  ctr=‘expr $ctr + 1‘
done


rm /tmp/ping.out /tmp/oz_batch /tmp/host.out

echo "*********************************************************************"
echo "               First step of registration succeeded!"
echo " The registration process does not set the has_nfs attribute."
echo " Set its value in all SubEnvs using the set_has_nfs rule."
echo " Then proceed with the send_subenv_map rule"
echo "*********************************************************************"

RETURN "0":"$new_subenv_id", "$new_subenv_name", "$new_site_name","$new_ip_addr";
END
```

Appendix B

The ISPW-9 Problem: Definition and Solution in Oz

B.1 The ISPW-9 Example

Life-cycle (Sub) Process Demonstration Scenario
9th International Software Process Workshop (ISPW9)
March 1994
Maria H. Penedo (ISPW9 Example Chair)
Base Scenario for Demonstration
Problem Reporting and Change Process

- A software project is on-going, with "parts" of the system already designed, codified, tested and baselined (i.e., under configuration management control).
- A problem is reported by a tester on the testing of a piece of the system under development. The project's problem reporting and analysis procedures are then followed and a person is assigned the task of the analysis of the problem. (Note: these procedures can be formal or informal, depending on the type of project. Notification can be effected by mail, by forms, by a tool. The procedures may include rules or guidelines telling who assigns people resources to study which problems and what kind of steps need to be followed.)
- A developer/analyst analyzes the problem and proposes a solution. After the analysis (which can be illustrated via automated process support or assumed to have been done manually), the developer identifies that the problem affects one software module which has been coded, tested and baselined, and possibly also affects some documentation (e.g., design and/or testing documents). (Note: the related documentation can be identified explicitly with help from the system, or implicitly via existing predefined rules in the system).

- After some analysis, it is noted that the module to be fixed is currently being (re-)used by two separate users or teams (again how this is accomplished may vary, i.e., the system may flag this issue or this fact may be found explicitly by inspection by a configuration manager or the developer). Those users are notified of the problem and that the module will be changed.

- The change process starts according to pre-established change procedures (which entail assignment of resources, code and/or documentation modification, analysis/testing/review, approval/rejection and new baseline of the module and associated documentation).

- The module is checked out of the baseline according to the configuration management procedures for change but reuse of the old version continues.

- The module is changed to fix the problem. (Optionally, the fix could be done by two or more separate developers and their cooperation may be illustrated via process support).

- The module is tested (formally or informally). Once the problem is fixed, procedures for acceptance/rejection are followed. Once the module is accepted (i.e., the change does fix the problem and it does not violate any of the requirements), appropriate regression testing on the modules/systems which reuse a prior version of this module can be performed.

- Once all is done, the change process is finalized.

B.1.1 Sub-scenarios

1. Specify and demonstrate one or more specific procedures/policies to complement the scenario (preferably performed with automated process support):

    - problem reporting and/or analysis
    - testing procedure/method
    - analysis of a problem using data in system
    - configuration control: retrieval, storage
    - code fix
    - problem approval/rejection
    - resource allocation

2. *User role support* — Demonstrate implicit/explicit support for project user roles, (multiple)user-to-(multiple) role assignment (static/dynamic), the impact of actions of one role upon another (i.e., automated cooperation among roles based on process definition), and how roles affect the interaction styles and other aspects of the process.

3. *Individual Support* — Demonstrate how individuals are guided about what task to do next, how users are made aware of the state of the process, or how the system performs actions as a result of the users' actions. Demonstrations should clearly illustrate how users are aware of the process, how the environment and individuals interact, and what variables control the different modes of interaction.

4. *Multi-user Coordination* — Demonstrate coordination of multiple people, including any support for resolution and tolerance of inconsistency. In particular, demonstrations can illustrate which aspects of these policies, if any, are hard-wired into their systems, and which can be altered by the particular model, and when the policy selections are made.

5. *Configuration Management* — Demonstrate how software and/or documents are controlled for the purpose of change, and how individuals using a module in their development are made aware of problems and/or changes to that module.

6. *Project/Central vs individual coordination* — Demonstrate how the executing process supports both individual and project activities, and how the interactions of those activities are supported/mediated by the system.

7. *Process changes while in execution* — Dynamically demonstrate changing any of the process definitions supporting the scenario and points 1-5 above, and the effects of those changes.

B.2 Solution in Oz

The following section lists the source code of the solution processes. It is organized as follows: Section B.2.1 contains the data definitions for all processes (to avoid duplication it is presented only once, with annotations for the local definitions); Section B.2.2 lists the QA process; Section B.2.3 lists the Coding process, starting with the Treaty strategies followed by the local strategies; Section B.2.4 lists the local strategies in Design; and Section B.2.5 lists a few selected envelopes.

B.2.1 The Schema

```
#
#                       Oz Software Development Environment
#
#                                Copyright 1994
#                     The Trustees of Columbia University
#                            in the City of New York
#                              All Rights Reserved
#

strategy data_model

imports protection;
exports all;

objectbase

######  THE CODING TREE  #######

# PROJECT is an entity that defines much of the structure of a typical
# software project.  PROJECTs can contain libraries, binaries
# documents header files and source code modules.

PROJECT :: superclass ENTITY;
    archive_status : (Archived, NotArchived, Initialized) = Initialized;
    build_status : (Built, NotBuilt, Initialized) = Initialized;
    build_log : text;
    libs  : set_of LIBS;
    bins  : BINS;
    docs  : set_of DOC;
    incs  : set_of INC;
    srcs  : set_of MODULE;
end

LIBS :: superclass ENTITY;
    archive_status : (Archived, NotArchived, Initialized) = Initialized;
    libs  : set_of LIB;
end
```

```
# LIB is a shared archive type library. The ultimate representation of a
# library is a .a file, that is, an archive format file.

LIB :: superclass ENTITY;
    archive_status : (Archived, NotArchived, Initialized) = Initialized;
    afile : binary = ".a";
end

# MODULE organizes CFILES based upon some higher order.  Each module knows
# which library (possibly more than one) it will be archived to.  MODULES
# can recursively contain other MODULES, and sets of CFILES.

MODULE :: superclass ENTITY;
    library         : set_of link LIB;
    archive_status  : (Archived, NotArchived, Initialized) = Initialized;
    modules         : set_of MODULE;
    cfiles          : set_of CFILE;
    external_doc    : link DOCFILE;
end

# FILE is the generic class for anything that is represented as a unix
# file.  There are specializations (subtypes) for CFILE, HFILE and DOCFILE.
# the reservation status is for configuration managemnet purposes

FILE :: superclass ENTITY;
    owner : user;
    timestamp : time;
    reservation_status : (CheckedOutShared, CheckedOut, Available,
                          Initialized) = Initialized;
    contents : text;
    locker   : user;
end


# This is a specialization of FILE to C source files.
# Several status attributes are added to record the status of
# compilation, analysis, change, and review.
# And additional product-related artifacts are added to contain
# object code, change requests, bug reports, etc.
# finally a CFILE contains links to various HFILEs that it includes,
# and to branching information.

CFILE :: superclass FILE;
    compile_status : (Compiled, NotCompiled, Initialized) = Initialized;
    compile_log    : text;
    analyze_status : (Analyzed, NotAnalyzed, Initialized) = Initialized;
    analyze_log    : text;
    change_status  : (Idle, StartChange, CompleteChange, Inspected) = Idle;
    change_request : text = ".chg";
    modified_change_request : text = ".mod_chg";
```

```
    bug_status     : (Clean, Suspected, Defected) = Clean;
    bug_report : text = ".bug";
    review_status  : (NotReviewed, ReviewRequested, Approved, Rejected,
                      RevisionRequested, Revised) = NotReviewed;
    review_rc      : (Succeeded, Failed, None) = None;
    contents : text = ".c";
    object_code : binary = ".o";
    ref : set_of link HFILE;
    maintained_by : link PROGRAMMER;
    branches : set_of BRANCH;
end

# For different rcs versions
BRANCH :: superclass ENTITY;
  locker : user;
  file : link CFILE;
end

# For HFILEs, we only want to know if they have been modified recently,
# which will cause a global recompilation.

HFILE :: superclass FILE;
    recompile_mod : boolean = false;
    contents : text = ".h";
end


# DOCFILEs specialized FILEs with their artifacts that contain the
# various files which are used by latex, and some status attributes for
# monitoring the state of the change, review etc. are added.

DOCFILE :: superclass FILE;
    reformat_doc : boolean = false;
    plain    : text   = ".txt";
    tex_file : binary = ".tex";
    dvi_file : binary = ".dvi";
    ps_file  : binary = ".ps";
    review_status   : (Idle, ReviewRequested, ChangeRejected,
                       ChangeApproved) = Idle;
    review_rc : (Succeeded, Failed, None) = None;
    change_request : text = ".chg";
    modified_change_request : text = ".mod_chg";
    maintained_by  : link PROGRAMMER;
    bug_report : text = ".bug";
end


# DOC is a class that represents an entire set of documents, typically for
# a PROJECT or PROGRAM.  A DOC can contain individual documents, and files
# of it's own.
```

```
DOC :: superclass ENTITY;
    documents : set_of DOCUMENT;
    files : set_of DOCFILE;
end

# DOCUMENT represents a complete individual document, such as a
# user's manual or technical report.

DOCUMENT :: superclass ENTITY;
    docfiles : set_of DOCFILE;
end


# INC represents a set of include (.h) files.

INC :: superclass ENTITY;
    archive_status : (Archived, NotArchived, Initialized) = Initialized;
    hfiles : set_of HFILE;
end


BINS :: superclass ENTITY;
    build_status : (Built, NotBuilt, Initialized) = Initialized;
    bins : set_of BIN;
end

# BIN represents a place where binaries for PROGRAMs (parts of a
# PROJECT) are kept.

BIN :: superclass ENTITY;
    build_status : (Built, NotBuilt, Initialized) = Initialized;
    executable : binary;
end


######  THE TEST TREE  #######

TEST_PROJECT :: superclass ENTITY;
  test_status  : (Tested, NotTested, Initialized) = Initialized;
  test_suites  : set_of TEST_SUITE;
  test_results : set_of TEST_RUN_SET;
end

TEST_SUITE :: superclass ENTITY;
  test_status : (Tested, NotTested, Initialized) = Initialized;
  test_cases  : set_of TEST_CASE;
  shared_data : binary;
end
```

```
TEST_CASE :: superclass FILE;
    test_input      : text = ".test_in";
    expected_output : text = ".test_out";
    maintained_by   : set_of link PROGRAMMER;
    number          : integer = 0;
    run_status      : boolean = false;
    test_failed     : boolean = false;
end

TEST_RUN_SET :: superclass ENTITY;
   test_runs     : set_of TEST_RUN;
   number        : integer = 0;
end

TEST_RUN  :: superclass ENTITY;
    performed_by  : user;
    timestamp     : time;
    test_output   : text;
    report        : text;
    report_status : (NotReported, Reported, Notified, Confirmed)
                     = NotReported;
    number        : integer;
    test_status   : integer;   #  0  - successful
                               #  1  - minor errors
                               #  2  - moderate errors
                               #  3  - quite serious errors
                               #  4  - severe errors
    new           : boolean = true;
    test_suites   : link TEST_SUITE;
    bin           : link BIN;
end

######  THE USER TREE  #######
## modification of the default user tree

PROGRAMMER  :: superclass USER;
   fname        : string;
   lname        : string;
   role         : string;
   group        : string;
   at_office    : boolean;
   user_id      : user;
   bonus        : integer;
end

######  THE DESIGN TREE  #######


DESIGN_PROJECT :: superclass ENTITY;
    documents : set_of DESIGN_DOCUMENT;
```

```
end

DESIGN_DOCUMENT :: superclass ENTITY;
    docfiles : set_of DOCFILE;
end

#####  THE CODING TREE (george) #####


LOCAL_AREA :: superclass ENTITY;
  workspaces : set_of WORKSPACE;
  project    : link PROJECT;
end

WORKSPACE :: superclass BIN, PROGRAMMER;
  files : set_of FILE;
  module : link MODULE;    ## link to its MODULE
  doc    : link DESIGN_DOCUMENT;  ## link to its DESIGN
  owner : user;
end

end_objectbase
```

B.2.2 The QA Process

B.2.2.1 test

```
#
#                     Oz Software Development Environment
#
#                                Copyright 1994
#                     The Trustees of Columbia University
#                            in the City of New York
#                              All Rights Reserved
#
#
#
#
#
#


strategy test

imports data_model;
exports all;

objectbase
```

```
# Tool definitions

TEST_TOOLS  :: superclass TOOL;
    create_test_run : string = create_test_run;
    run_test        : string = run_test;
    notify_bug      : string = notify_bug;
    report_bug      : string = report_bug;
    add_bonus       : string = add_bonus;
    edit_test       : string = edit_test;
    view_result     : string = view_report;
    generate_report : string = generate_report;
    store_notify_bug : string = store_notify_bug;
end

end_objectbase

rules


#########
#
#  start_test_run:
#
#  add a test run, link it to test_suite and the binary and trigger
#  chain of tests.
#
#########

start_test_run[?tr:TEST_RUN_SET,?ts:TEST_SUITE,?b:BIN]:
 (forall TEST_CASE ?tc suchthat (member [?ts.test_cases ?tc]))

  :
# prompt the user for the name of the test run and return
# it in the envelope
 { TEST_TOOLS create_test_run return ?name }


# the link effects trigger the run_test rules.

 (and
     (?new:TEST_RUN = add [?tr test_runs ?name TEST_RUN])
      # assign run number and increment the "global" counter at the parent
      (?new.performed_by = CurrentUser)
      (?new.number = ?tr.number)
      (?tc.test_failed = false)
      (link [?new test_suites ?ts])
      (link [?new bin ?b])
      (?tr.number  += 1));

  no_assertion;
```

```
#########
#
#  run_test:
#
#  run automatically test on a test_case.
#
#########


run_test[?tc:TEST_CASE]:
  (and
      (exists TEST_SUITE ?ts  suchthat (member [?ts.test_cases ?tc]))
      (exists TEST_RUN ?tr suchthat (and (linkto [?tr.test_suites ?ts])
                                         (?tr.new = true)))
      (exists BIN      ?b  suchthat (linkto [?tr.bin         ?b])))
  :

  { TEST_TOOLS run_test ?tc.test_input ?tc.expected_output ?b.executable
?ts.shared_data ?tr.test_output  return ?stat}
  (and
          (?tr.timestamp = CurrentTime)
          (?tr.performed_by = CurrentUser)
          (?tc.run_status = true)
          (?tc.test_failed = false));

  (and
          (?tr.timestamp = CurrentTime)
          (?tr.performed_by = CurrentUser)
          (?tc.run_status = true)
          (?tr.test_status  += 1)
          (?tc.test_failed = true));

#########
#
#  complete_test
#
# this rule is fired when all indivial runs have finished, either
# sucessfully or not
#
#########

complete_test[?ts:TEST_SUITE]:
 (and
 (forall TEST_CASE ?tc suchthat (member [?ts.test_cases ?tc]))
 (forall TEST_RUN  ?tr suchthat (and (linkto [?tr.test_suites ?ts])
                                     (?tr.new = true)))):

  (?tc.run_status = true)
```

```
  { }
  (and
    (?tc.run_status = false)
    (?tr.new = false));


#########
#
#  report_bug:
#
#  when a test run fails, chain to this rule
#
#########

report_bug[?tr:TEST_RUN]:
  :
  (?tr.test_status > 0)

  { TEST_TOOLS generate_report ?tr.performed_by ?tr.timestamp
                               ?tr.test_output  ?tr.report }

  (?tr.report_status = Reported);


#########
#
#  notify_bug:
#
#  chained off report_bug, delegated to a user which is notified
#  of the problem
#
#########

notify_bug[?tr:TEST_RUN]:
   (forall PROGRAMMER ?p suchthat (and
                                   (?p.group = "CODING")
                                   (?p.role  = "Manager")))
   :

   delegate[?p.user_id]:
   (?tr.report_status = Reported)

# this tool should generate a message with the input parameters as
# "fill-in blanks" style

   { TEST_TOOLS notify_bug ?tr.report }

   (?tr.report_status = Notified);
```

```
#########
#
#  store_notify_bug:
#
#  chained off report_bug, in case notification is impossible
#
#########

notify_bug[?tr:TEST_RUN]:
   (forall PROGRAMMER ?p suchthat (and
                                   (?p.group = "CODING")
                                   (?p.role  = "Manager")))
   :

   (?tr.report_status = Reported)

# this tool should generate a message with the input parameters as
# "fill-in blanks" style

   { TEST_TOOLS store_notify_bug ?p.fname ?p.lname ?p.user_id }

   no_assertion;


# Local chain off summit rule analyze_bug - add bonus to tester,
#  depending on the severity of the bug


#########
#
#  bug_found:
#
#  chained off summit analyze_bug rule
#
#########


bug_found[?tr:TEST_RUN]:

(forall PROGRAMMER ?p suchthat (?p.user_id = CurrentUser))
:
  (?tr.report_status = Confirmed)

  {TEST_TOOLS add_bonus ?tr.performed_by return ?bonus }

  (?p.bonus += ?bonus);


#########
```

```
#
#  edit_test:
#
#########


edit_test[?tc:TEST_CASE]:
  (exists TEST_SUITE ?ts  suchthat (member [?ts.test_cases ?tc]))
  :

  { TEST_TOOLS edit_test ?tc.test_input ?tc.expected_output }

  (?tc.timestamp = CurrentTime);


#########
#
#  view_result:
#
#########


view_result[?tr:TEST_RUN]:
:
 {TEST_TOOLS view_result ?tr.test_output}
;

#########
#
#  find_unnotified_bug:
#
#  if notification was impossible, this rule is invoked by a batch
#  file of the notifiee
#########

find_unnotified_bugs[]:
   (exists TEST_RUN ?tr suchthat (?tr.report_status = Reported))
   :
   (?tr.report_status = Reported)

   { TEST_TOOLS notify_bug ?tr.report }

   (?tr.report_status = Notified);
```

B.2.3 The CODING Process

Treaty Strategies

B.2.3.1 Analyze

```
#
#                     Oz Software Development Environment
#
#                                Copyright 1994
#                     The Trustees of Columbia University
#                            in the City of New York
#                              All Rights Reserved
#

strategy analyze (Treaty)

imports data_model;
exports all;

objectbase


#########
#
#  TOOL Definitions
#
#########

ANALYZE_TOOLS  :: superclass TOOL;
      analyze_bug       : string = analyze_bug;
      analyze_cfile_bug : string = analyze_cfile_bug;
end

end_objectbase

rules


#########
#
#  analyze_bug:
#
#  This Summit rule starts up the analyze_task, done in CODING and QA.
#  It is performed at the CODING process, and is intended to find
#  "suspected" cfiles that might be the reason for the bug found at
#  the QA process
#
#########

analyze_bug[?tr:TEST_RUN, ?p:PROJECT]:

  (and (forall TEST_SUITE ?ts suchthat (linkto [?tr.test_suites ?ts]))
       (forall  CFILE ?c suchthat (ancestor[?p ?c]))):
```

```
  (and
     no_chain (?tr.test_status > 0)
     no_chain (?tr.report_status = Notified))

  # This envelope returns a subset, which is the "bad"
  # sources need to send more ?tr attributes
  # ----------------------------------------
  { ANALYZE_TOOLS analyze_bug ?tr.report ?tr.test_status
                              ?tr.performed_by ?tr.timestamp
                              ?c.contents return subset ?bad_c:CFILE}

  (?bad_c.bug_status = Suspected);

#########
#
#  analyze_bug (on a cfile)
#
#  This Summit rule chains off the above rule, and is delegated to the
#  owner of the cfile. The owner determines whether the bug was realy in
#  that CFILE, or not. In the former case, the bug_status becomes
#  defected, meaning that a  change task should be performed on it.  The
#  latter means this file is "clean". Also, generate a change request.
#########

analyze_bug[?tr:TEST_RUN, ?c:CFILE]:

 (and
  (forall MODULE ?m suchthat (member [?m.cfiles ?c]))
  (exists WORKSPACE ?w suchthat (linkto [?w.module ?m])))
  :
  delegate[?w.owner]:

 (?c.bug_status = Suspected)

  # Prompt the user whether the bug is here (so return 0) or not (Return 1)
  # also, generate a change request in the CFILE.
 { ANALYZE_TOOLS analyze_cfile_bug ?tr.report ?c.change_request ?c.contents
                                   ?c.bug_report }


 (and no_chain (?c.bug_status = Defected)
                # chain locally at TEST site
              (?tr.report_status = Confirmed));
 (?c.bug_status = Clean);


#########
#
#  reset_analyze_atts:
#  This rule is for debugging purposes only.
```

```
#  initialize attributes to start the analyze phase
#
#########

hide reset_analyze_atts[?tr:TEST_RUN, ?c:CFILE]:
:
 { }

(and
   no_chain (?c.bug_status = Clean)
   no_chain (?tr.report_status = Notified));
```

B.2.3.2 review

```
#
#                      Oz Software Development Environment
#
#                                 Copyright 1994
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#

strategy review

imports data_model;
exports all;

objectbase

#########
#
#  TOOL Definitions
#
#########


REVIEW_TOOLS  :: superclass TOOL;
  init_review : string = init_review;
  revise_doc  : string = revise_doc;
end



MU_TOOLS  :: superclass TOOL;
   [ protocol : SEL;
   multi_flag : MULTI_QUEUE ; ]
  confer     : string = confer;
end
```

```
end_objectbase

rules

#########
#
#  setup_review:
#
#  This Summit rule starts up the review task, done in CODING and DESIGN.
#  It forward chains to local review at both CODING and DESIGN processes
#
#########


setup_review[?p:PROJECT, ?design_doc:DESIGN_DOCUMENT]:

 (and (forall MODULE ?m suchthat (member [?p.srcs ?m]))
      (forall CFILE  ?c suchthat (and (member [?m.cfiles ?c])
                                      (?c.bug_status = Defected)))
      (forall DOCFILE ?d suchthat (member [?design_doc.docfiles ?d])))
  :

  (or
   (?c.review_status = Revised)
   (?c.bug_status = Defected))

 # this envelope simply copies the contents of the change request
 # and the bug_report to the same fields
 # in the design doc object, and the bug_report
 # could possibly use the copy operation instead

 { REVIEW_TOOLS init_review ?c.change_request ?c.bug_report
                            ?d.change_request ?d.bug_report }


  (and
   (?d.review_status = ReviewRequested)
   (?c.review_status = ReviewRequested));


#########
#
# approve:
#
#  This Summit rule  completes the Review Phase
#
#########

approve[?c:CFILE, ?design_doc:DESIGN_DOCUMENT]:
```

```
 (forall DOCFILE ?d suchthat (member [?design_doc.docfiles ?d]))
 :
  (and
    no_backward (?d.review_status = ReviewRequested)
    no_backward (?c.review_status = ReviewRequested)
    no_backward (?d.review_rc = Succeeded)
    no_backward (?c.review_rc = Succeeded))

 {  }

  (and
   (?d.review_status = ChangeApproved)
   (?c.review_status = Approved));


#########
#
#  confer:
#
#  This Summit rule  is invoked when local reviews fail.
#  it is a multi-user conference rule
#  it forward chains to local revise at CODING
#
#########

confer[?c:CFILE, ?design_doc:DESIGN_DOCUMENT]:
  # collect multiple users to same variable to get
  # delegation to multiple people for multi-user tool invocation
 (and
  (forall GROUP   ?coding suchthat (ancestor [?coding ?c]))
  (forall GROUP   ?design suchthat (ancestor [?design ?design_doc]))
  (forall MODULE  ?m      suchthat (member   [?m.cfiles ?c]))
  (forall DOCFILE ?d suchthat (member [?design_doc.docfiles ?d]))
  (forall WORKSPACE ?p suchthat (or
                                 (linkto [?p.module ?m])
                                 (linkto [?p.doc  ?design_doc]))))
  :
  delegate[?p.owner]:
  (and
    no_backward (?d.review_status = ReviewRequested)
    no_backward (?c.review_status = ReviewRequested)
    (or
      no_backward (?d.review_rc = Failed)
      no_backward (?c.review_rc = Failed)))

# start white_board here to discuss design. prepare working files
# this is a multi user activity

 {  MU_TOOLS confer ?c.contents ?d.contents
            ?coding.site_ip_addr ?design.site_ip_addr}
```

```
# 0. ok, go to revise, and enable setup_review if revise succeeds
#  (and
   (and
    (?c.review_status   = RevisionRequested)
# this double assertion enables to chain back to setup_review
    (?c.bug_status      = Suspected)
    (?c.bug_status      = Defected));

# 1. no hope, go to reject. needs to start all over again.
  (and
   (?d.review_status = ChangeRejected)
   (?c.review_status = Rejected));


#########
#
# reset_atts:
#
#  This Summit rule is for debugging, initializing attributes
#  for starting the review on a single cfile (from CODING) and a single
#  docfile (from DESIGN)
#
#########

reset_atts[?c:CFILE, ?d:DOCFILE]:
:
 { }

(and
   no_chain (?c.bug_status = Defected)
   no_chain (?c.review_rc = None)
   no_chain (?c.review_status = NotReviewed)
   no_chain (?d.review_status = Idle)
   no_chain (?d.review_rc = None));
```

B.2.3.3 change

```
#
#                     Oz Software Development Environment
#
#                                Copyright 1994
#                     The Trustees of Columbia University
#                            in the City of New York
#                              All Rights Reserved
#

strategy change
```

```
imports data_model;
exports all;

objectbase

#########
#
#  TOOL Definitions
#
#########

CHANGE_TOOLS :: superclass TOOL;
  install_bin : string = install_bin;
end

# Synchronous Multi-user tool
MU_INSPECTION :: superclass TOOL;
  [ protocol:SEL; multi_flag:MULTI_QUEUE ; ]
  code_inspect  : string = code_inspect;
end

end_objectbase

rules

#########
#
#  code_inspect:
#
#  This multi-user Summit rule is called manually, i.e., it is not
#  chained off any other rule. and at the moment doens't chain
#  to any other rule either.
#  it has no condition, so it can be called at any time.
#  but deposit cannot occur unless the file has been inspected
#  It calls the multi-media inspection program developed in-house.
#
#########

code_inspect[?c:CFILE, ?d:DOCFILE]:

  (and
  (forall GROUP   ?c_site suchthat (ancestor [?c_site ?c]))
  (forall GROUP   ?d_site suchthat (ancestor [?d_site ?d]))
  (forall MODULE  ?m  suchthat (member   [?m.cfiles ?c]))
  (forall DESIGN_DOCUMENT ?dm  suchthat (member  [?dm.docfiles ?d]))
  (forall WORKSPACE ?w suchthat (or
                                (linkto [?w.module ?m])
                                (linkto [?w.doc  ?dm]))))
   :
```

```
  delegate[?w.owner]:

  { MU_INSPECTION code_inspect ?c.contents ?c.change_request ?d.contents
                               ?c_site.site_ip_addr ?d_site.site_ip_addr
                               ?w.owner }

  (?c.change_status = Inspected);
  (?c.change_status = StartChange);

#########
#
#  install_bin:
#
#  A simple rule that copies the binary from the CODING site to the
#  QA site. it does so by copying contents of object, as opposed to
#  copying the whole object.
#
#########

install_bin[?coding_bin:BIN, ?test_bin:BIN]:

  # Enforce to allow only a coding bin to be installed.
  # ------------------------------------------------
  (exists PROJECT ?p suchthat no_chain(ancestor [?p ?coding_bin]))
  :
  { CHANGE_TOOLS install_bin ?test_bin.executable ?coding_bin.executable }
  (?test_bin.build_status = Built);
```

Local Strategies

B.2.3.4 Build

```
#
#                     Oz Process Centered Environment
#
#                             Copyright 1994
#                   The Trustees of Columbia University
#                         in the City of New York
#                           All Rights Reserved
#

strategy build

# This strategy provides a rule to build a PROGRAM.  It also provides two
# inferences rules that will build an entire PROJECT or GROUP.

imports data_model;
exports all;


objectbase
```

```
#########
#
#  TOOL Definitions
#
#########

BUILD :: superclass TOOL;
    build_program : string = build;
    build_workspace : string = build_workspace;
    archive : string = archive;
    archive_module : string = archive_module;
    list_archive : string = list_archive;
end

end_objectbase

rules

#
# Build the project when all BINS are built.
#
build [?proj:PROJECT]:
  (exists BINS  ?b  suchthat (member   [?proj.bins ?b])):

  (?b.build_status = Built)
  { }
  (?proj.build_status = Built);

#
# Force each BIN to be built.
#
build [?bs:BINS]:
  (exists BIN ?b suchthat (member [?bs.bins ?b])):
  (?b.build_status = Built)
  { }
  (?bs.build_status = Built);

#
# Build the BIN from libraries
#
build[?b:BIN]:
  (and (exists PROJECT ?p  suchthat (ancestor [?p ?b]))
       (forall LIB     ?l  suchthat (ancestor [?p ?l]))):

  (?l.archive_status = Archived)

  { BUILD build_program ?b.executable ?p.build_log ?l.afile }

  (?b.build_status = Built);
```

```
  (?b.build_status = NotBuilt);
#
# Build a WORKSPACE
#
build[?w:WORKSPACE]:
  (and
    (exists LOCAL_AREA ?la suchthat no_chain (member [?la.workspaces ?w]))
    (exists PROJECT  ?p  suchthat no_chain (linkto   [?la.project ?p]))
    (forall CFILE    ?c  suchthat no_chain (member   [?w.files ?c]))
    (forall LIB      ?l  suchthat no_chain (ancestor [?p ?l]))):

  (and (?l.archive_status = Archived)
       (?c.compile_status = Compiled))

  { BUILD build_workspace ?w.executable ?l.afile ?c.object_code }

  (?w.build_status = Built);
  (?w.build_status = NotBuilt);

#
# Archive a MODULE
#
archive [?m:MODULE]:

  (and (forall CFILE ?c suchthat (member [?m.cfiles ?c]))
       (exists LIB   ?l suchthat (linkto [?m.library ?l]))):

  (?c.compile_status = Compiled)

  { BUILD archive ?l.afile ?c.object_code }

  (?m.archive_status = Archived);

#
# Archive a LIB
#
archive [?l:LIB]:
  (and (forall MODULE ?m suchthat (linkto [?m.library ?l]))
       (forall CFILE  ?c suchthat (ancestor [?m ?c]))):

  (?m.archive_status = Archived)

  { }

  [?l.archive_status = Archived];

#
# Archive the LIBS
#
archive [?ls:LIBS]:
```

```
  (forall LIB ?l suchthat (member [?ls.libs ?l])):
  [?l.archive_status = Archived]
  { }
  [?ls.archive_status = Archived];

view [?l:LIB]:
  :
  { BUILD list_archive ?l.afile }
  ;

# -------------------- CONSISTENCY CHAINS --------------------

touch[?l:LIB]:
  (exists MODULE ?m suchthat (linkto [?m.library ?l])):

  [?m.archive_status = NotArchived]
  { }
  [?l.archive_status = NotArchived];

touch[?ls:LIBS]:
  (exists LIB ?l suchthat (member [?ls.libs ?l])):
  [?l.archive_status = NotArchived]
  { }
  [?ls.archive_status = NotArchived];

touch[?p:PROJECT]:
  (exists LIBS ?ls suchthat (member [?p.libs ?ls])):
  [?ls.archive_status = NotArchived]
  { }
  [?p.build_status = NotBuilt];

touch[?bs:BINS]:
  (exists PROJECT ?p suchthat (member [?p.bins ?bs])):
  [?p.build_status = NotBuilt]
  { }
  [?bs.build_status = NotBuilt];

touch[?b:BIN]:
  (exists BINS ?bs suchthat (member [?bs.bins ?b])):
  [?bs.build_status = NotBuilt]
  { }
  [?b.build_status = NotBuilt];

touch[?h:HFILE]:
  (exists INC ?i suchthat (member [?i.hfiles ?h])):

  [?h.recompile_mod = true]
  { }
  (and [?h.recompile_mod = false]
       [?i.archive_status = NotArchived]);
```

```
touch[?i:INC]:
  (and (exists PROJECT ?p suchthat (member [?p.incs ?i]))
       (forall CFILE ?c suchthat (ancestor [?p ?c]))):

  [?i.archive_status = NotArchived]
  { }
  (and [?c.compile_status = NotCompiled]
       no_chain (?i.archive_status = Archived));

touch[?m:MODULE]:
  (exists CFILE ?c suchthat (member [?m.cfiles ?c])):

  [?c.compile_status = NotCompiled]
  { }
  [?m.archive_status = NotArchived];
```

B.2.3.5 coding_review

```
#
#                      Oz Software Development Environment
#
#                                 Copyright 1994
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
#
#   Local review at the CODING group
#

strategy coding_review

imports data_model;
exports all;

objectbase

DESIGN_REVIEW_TOOLS  :: superclass TOOL;
  review_doc  : string = review_doc;
end

end_objectbase

rules
```

```
#########
#
#  review:
#
#  local review at coding site, chained off setup_review summit
#
#########


review[?c:CFILE]:
 :

 (?c.review_status = ReviewRequested)
 { DESIGN_REVIEW_TOOLS review_doc "review" ?c.contents
                                 ?c.change_request}

   (?c.review_rc         = Succeeded);
   (?c.review_rc         = Failed);


#########
#
#  revise:
#
#  local revision of change at coding site, chained off confer summit
#
#########


revise[?c:CFILE]:
    :
   (?c.review_status = RevisionRequested)
   { DESIGN_REVIEW_TOOLS review_doc  "revise" ?c.contents
                                     ?c.change_request  }

   (?c.review_status = Revised);
```

B.2.3.6 compile

```
#
#       Oz Process Centered Environment
#
#       Copyright 1994
#     The Trustees of Columbia University
#   in the City of New York
#     All Rights Reserved
#
```

```
strategy compile

# This strategy contains rules to compile and analyze CFILE type objects.
# Compilation is done with cc, and analysis with lint.  In our example,
# a file must successfully be analyzed before it is compiled.

imports data_model;
exports all;

objectbase

COMPILER :: superclass TOOL;
    compile : string = "compile CFILE.contents S CFILE.compile_log S
                                CFILE.object_code S HFILE.contents S";
    analyze : string = "analyze CFILE.contents S CFILE.analyze_log S
                                HFILE.contents S HFILE.contents S";
end

end_objectbase


rules

# Compile a file in the master area
# ---------------------------------------------
compile [?c:CFILE]:

 (and (exists PROJECT ?p   suchthat (ancestor [?p ?c]))
      (forall INC     ?inc suchthat (member   [?p.incs ?inc]))
      (forall HFILE   ?h   suchthat (member   [?inc.hfiles ?h]))):


  # If the CFILE has been analyzed successfuly but not yet compiled, it can
  # be compiled.
  # ----------------------------------------------------------------------
  (and          ( ?c.analyze_status = Analyzed )
       no_chain ( ?c.compile_status = NotCompiled))

  { COMPILER compile ?c.contents ?c.compile_log ?c.object_code
                     ?h.contents emptyset }

  ( ?c.compile_status = Compiled );
  [ ?c.compile_status = NotCompiled ];


# Compile a file in the local area
# ---------------------------------------------
compile [?c:CFILE]:

 (and (exists LOCAL_AREA ?l    suchthat (ancestor [?l ?c]))
```

```
      (forall PROJECT    ?p   suchthat (linkto   [?l.project ?p]))
      (forall INC        ?inc suchthat (member   [?p.incs ?inc]))
      (forall HFILE      ?rh  suchthat (member   [?inc.hfiles ?rh]))
      (exists WORKSPACE  ?w   suchthat (member   [?w.files ?c]))
      (forall HFILE      ?lh  suchthat (member   [?w.files ?lh]))):

  # If the CFILE has been analyzed successfuly but not yet compiled, it can
  # be compiled.
  # ----------------------------------------------------------------------
  (and          ( ?c.analyze_status = Analyzed )
       no_chain ( ?c.compile_status = NotCompiled))

  { COMPILER compile ?c.contents ?c.compile_log ?c.object_code
                     ?rh.contents ?lh.contents }

  ( ?c.compile_status = Compiled );
  [ ?c.compile_status = NotCompiled ];


# Analyze a file in the local area.
# -----------------------------------
analyze[?c:CFILE]:

 (and
   (exists LOCAL_AREA ?l suchthat no_chain (ancestor [?l ?c]))
   (forall PROJECT  ?p   suchthat no_chain (linkto   [?l.project ?p]))
   (forall INC      ?inc suchthat no_chain (member   [?p.incs ?inc]))
   (forall HFILE    ?rh  suchthat no_chain (member   [?inc.hfiles ?rh]))
   (exists WORKSPACE ?w  suchthat no_chain (member   [?w.files ?c]))
   (forall HFILE     ?lh suchthat no_chain (member   [?w.files ?lh]))):

  # If the CFILE has been Not yet been analyzed it can be analyzed.
  # ---------------------------------------------------------------
  ( ?c.analyze_status = NotAnalyzed )

  { COMPILER analyze ?c.contents ?c.analyze_log ?rh.contents ?lh.contents }

  ( ?c.analyze_status = Analyzed);
  [ ?c.analyze_status = NotAnalyzed ];
```

B.2.3.7 edit

```
#
#                      Oz Software Development Environment
#
#                                 Copyright 1994
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
```

```
#

strategy edit

# This strategy defines the editor tool and a viewer tool which displays
# the errors associated with a particular C file.  The rules for editing
# are overloaded, they set appropriate attributes depending upon the
# type of object being edited.

imports data_model;
exports all;

objectbase

EDITOR :: superclass TOOL;
    edit   : string = editor;
    edit_h : string = editor_h;
end

VIEWER :: superclass TOOL;
    viewErr      : string = viewErr;
    viewBuildErr : string = viewBuildErr;
    view         : string = view;
end

end_objectbase


rules

# this edit rule is for editing c files.  Note that all these rules have
# the same activities, but different postconditions.  If there were
# special editors, they could be invoked by calling edit rules with
# different activities.

edit[?c:CFILE]:

  # Only allow this rule to fire in the local area.
  (exists LOCAL_AREA ?l suchthat (ancestor [?l ?c]))

  # Documents can only be edited if they have been approved to be changed
  # by the init_change [?t:TEST_PROJECT, ?p:PROJECT, ?d:DESIGN_PROJECT]
  # rule.
  :

  # what about the requirement to be checked out ?
  no_chain (?c.review_status = Approved)

  { EDITOR edit ?c.contents ?c.analyze_status ?c.analyze_log
                ?c.compile_status ?c.compile_log}
```

```
  (and (?c.analyze_status = NotAnalyzed)
       no_chain (?c.compile_status = NotCompiled)
       no_chain (?c.timestamp = CurrentTime));
  no_chain ( ?c.reservation_status = CheckedOut );

# this edit rule is for editing document files.

edit[?f:DOCFILE]:

  # if the file has been reserved, you can go ahead and edit it
  :
  (and ( ?f.owner = CurrentUser )
       ( ?f.reservation_status = CheckedOut))

  { EDITOR edit ?f.contents }

  (and (?f.reformat_doc = true)
       (?f.timestamp = CurrentTime));
  no_chain ( ?f.reservation_status = CheckedOut );


# this edit rule is for editing include files.

edit[?h:HFILE]:

  # Only allow this rule to fire in the local area.
  (exists LOCAL_AREA ?l suchthat (ancestor [?l ?h]))

  # if the file has been reserved, you can go ahead and edit it
  :
  (and ( ?h.owner = CurrentUser )
       ( ?h.reservation_status = CheckedOut))

  { EDITOR edit_h ?h.contents }

  (and (?h.recompile_mod = true)
       (?h.timestamp = CurrentTime));
  no_chain ( ?h.reservation_status = CheckedOut );


# The following rule views output from the compiler and analyzer for a
# particular file.

viewErr[?f:CFILE]:
    :
    { VIEWER viewErr ?f.analyze_log ?f.compile_log }
    ;

view[?f:FILE]:
```

```
    :
    { VIEWER view ?f.contents}
    ;
```

B.2.3.8 rcs

```
#
#       Oz Process Centered Environment
#
#       Copyright 1994
#     The Trustees of Columbia University
#   in the City of New York
#     All Rights Reserved
#

# This strategy contains rules for doing revision control on FILE
# type objects.

strategy rcs

imports data_model;
exports all;

objectbase

RCS  :: superclass TOOL;
    reserve : string = check_out;
    deposit : string = check_in;
    deposit_first : string = check_in_first;
    view_rcs : string = view_rcs;
    branch : string = branch;
    merge_code : string = merge_code;
    create_rcs : string = create_rcs;
    move_file : string = move_file;
    start_chain : string = start_chain;
end

end_objectbase


rules

# Implication of summit rule. When the change is initiated (i.e. the
# change_status attribute of a CFILE is set to StartChange), a forward
# chain commences to reserve the appropriate files into the appropriate
# WORKSPACES.
#
hide setup_reserve[?mc:CFILE]:
  (and (exists MODULE ?m suchthat (member [?m.cfiles ?mc]))
```

```
      (exists WORKSPACE ?w suchthat (linkto [?w.module ?m]))):

  # Delegate to appropriate person
  # ------------------------------
  delegate[?w.owner]:


  # (1) This predicate causes chaining.
  # (2) Simple case of already available.
  # --------------------------------------
  (and          (?mc.change_status = StartChange)
                (?mc.review_status = Approved)
      no_chain (?mc.reservation_status = Available))

  { RCS reserve ?mc.contents }

  (and no_forward ( ?mc.reservation_status = CheckedOut )
      no_forward ( ?mc.owner = CurrentUser )
                 ( ?new:CFILE = copy [?mc ?w files ]));
  no_assertion;


hide setup_branch[?mc:CFILE]:
  (and (exists MODULE ?m suchthat (member [?m.cfiles ?mc]))
      (exists WORKSPACE ?w suchthat (linkto [?w.module ?m]))):

  # Delegate to appropriate person
  # ------------------------------
  delegate[?w.owner]:

  # (1) This predicate causes chaining
  # (2) More complex case of already checkedout by someone else
  # -----------------------------------------------------------
  (and      ( ?mc.change_status = StartChange )
            ( ?mc.review_status = Approved )
            (or
               no_chain ( ?mc.reservation_status = CheckedOut )
               no_chain ( ?mc.reservation_status = CheckedOutShared )))

  { RCS branch ?mc.contents "new" return ?rev }

  # Create duplicate copy of CFILE in local WORKSPACE and link from BRANCH
  # ----------------------------------------------------------------------
  (and no_forward (?mc.reservation_status = CheckedOutShared)
                 (?new:CFILE = copy [?mc ?w files ])
                 (?b:BRANCH = add [?mc branches ?rev BRANCH])
                 (link [?b.file ?new]));
  no_assertion;
```

```
# Reserve an HFILE into WORKSPACE
# -------------------------------
hide reserve[?h:HFILE, ?w:WORKSPACE]:
  :
  (or (?h.reservation_status = Available )
      (?h.reservation_status = Initialized ))

  { RCS reserve ?h.contents }

  (and no_forward ( ?h.reservation_status = CheckedOut )
       no_forward ( ?h.owner = CurrentUser )
                  ( ?new:HFILE = copy [?h ?w files ]));
  no_assertion;

# Make a version of a HFILE available (first time)
# -------------------------------------------------------------
deposit[?h:HFILE, ?i:INC]:

  (and (forall HFILE ?mh suchthat (and (member [?i.hfiles ?mh])
                                       (?mh.Name = ?h.Name)))
       (exists WORKSPACE ?w  suchthat no_chain (member [?w.files ?h]))):

  (and no_chain ( ?mh.Name = "" )
       no_chain ( ?h.owner = CurrentUser )
       no_chain ( ?h.reservation_status = CheckedOut ))

  { RCS deposit_first ?h.contents ?i.Name }

  (and no_forward ( ?h.reservation_status = Available )
       no_chain   ( move [?h ?i hfiles ?w] )
       no_chain   ( ?h.recompile_mod = false)
                  ( ?h.recompile_mod = true));
  no_assertion;


# Make a version of a HFILE available (second and future times)
# -------------------------------------------------------------
deposit[?h:HFILE, ?i:INC]:

  (and (exists HFILE ?mh suchthat (and (member [?i.hfiles ?mh])
                                       (?mh.Name = ?h.Name)))
       (exists WORKSPACE ?w  suchthat no_chain (member [?w.files ?h])))
  :
  (and no_chain ( ?h.owner = CurrentUser )
       no_chain ( ?h.reservation_status = CheckedOut ))

  { RCS deposit ?h.contents ?mh.contents ?mh.reservation_status
                ?i.Name empty }

  (and no_forward ( ?mh.reservation_status = Available )
```

```
                  ( delete [?h ?w] )
       no_chain   ( ?mh.recompile_mod = false)
                  ( ?mh.recompile_mod = true));
  no_assertion;



# Reserve a CFILE to local WORKSPACE
# ----------------------------------
reserve[?c:CFILE, ?w:WORKSPACE]:
  :
  (or [ ?c.reservation_status = Available ]
      [ ?c.reservation_status = Initialized ])

  { RCS reserve ?c.contents }

  (and no_forward ( ?c.reservation_status = CheckedOut )
       no_forward ( ?c.owner = CurrentUser )
                  ( ?new:CFILE = copy [?c ?w files ]));
  no_assertion;


# Make branch on current RCS tree, but only if not current owner.
# ----------------------------------------------------------------
reserve[?c:CFILE, ?w:WORKSPACE]:
  :
  (and no_chain (?c.owner <> CurrentUser)
       (or no_chain (?c.reservation_status = CheckedOut)
           no_chain (?c.reservation_status = CheckedOutShared)))

  { RCS branch ?c.contents "new" return ?rev }

  # Create duplicate copy of CFILE in local WORKSPACE and link from BRANCH
  # ----------------------------------------------------------------------
  (and no_forward (?c.reservation_status = CheckedOutShared)
                  (?new:CFILE = copy [?c ?w files ])
                  (?b:BRANCH = add [?c branches ?rev BRANCH])
                  (?b.locker = CurrentUser)
                  (link [?b.file ?new]));
  no_assertion;


hide get_version[?c:CFILE]:
  (and (exists BRANCH    ?b suchthat          (linkto [?b.file ?c]))
       (exists WORKSPACE ?w suchthat no_chain (member [?w.files ?c])))
  :
  { RCS move_file ?c.contents ?w.Name }
  no_assertion;
```

```
# Make a version of a CFILE available (first time)
# -------------------------------------------------------------
deposit[?c:CFILE, ?m:MODULE]:

  (and (forall CFILE ?mc suchthat (and (member [?m.cfiles ?mc])
                                       (?mc.Name = ?c.Name)))
       (exists WORKSPACE ?w  suchthat no_chain (member [?w.files ?c]))):

  (and no_chain ( ?mc.Name = "" )
       no_chain ( ?c.owner = CurrentUser )
       no_chain ( ?c.change_status = Inspected )
       no_chain ( ?c.reservation_status = CheckedOut ))

  { RCS deposit_first ?c.contents ?m.Name }

  (and no_forward ( ?c.reservation_status = Available )
                  ( ?c.compile_status = NotCompiled )
       no_chain   ( move [?c ?m cfiles ?w] )
                  ( ?m.archive_status = NotArchived ));
  no_assertion;

# Make a version of a CFILE available (second and future times)
# -------------------------------------------------------------
deposit[?c:CFILE, ?m:MODULE]:

  (and (exists CFILE ?mc suchthat (and (member [?m.cfiles ?mc])
                                       (?mc.Name = ?c.Name)))
       (forall BRANCH ?b suchthat (member [?mc.branches ?b]))
       (exists WORKSPACE ?w  suchthat no_chain (member [?w.files ?c])))
  :
  ## No branches yet.

  (and no_chain ( ?b.Name = "" )
       no_chain ( ?c.owner = CurrentUser )
       no_chain ( ?mc.owner = CurrentUser )
       no_chain ( ?c.change_status = Inspected )
       no_chain ( ?mc.reservation_status = CheckedOut ))

  { RCS deposit ?c.contents ?mc.contents ?mc.reservation_status
                ?m.Name empty }

  (and  no_forward ( ?mc.reservation_status = Available )
                   ( ?mc.compile_status = NotCompiled )
                   ( delete [?c ?w] )
                   ( ?m.archive_status = NotArchived ));
  no_assertion;


# Deposit earlier version of code that someone had already locked.
# -----------------------------------------------------------------
```

```
deposit[?c:CFILE, ?m:MODULE]:

  (and (exists WORKSPACE ?w   suchthat no_chain (member [?w.files ?c]))
       (exists BRANCH    ?b   suchthat no_chain (linkto [?b.file ?c]))
       (exists CFILE     ?mc  suchthat
                                  (and no_chain (member [?mc.branches ?b])
                                       no_chain (?mc.Name = ?c.Name)))):

  (and no_chain ( ?c.owner = CurrentUser )
       no_chain ( ?c.change_status = Inspected )
       no_chain ( ?mc.reservation_status = CheckedOutShared ))

  { RCS deposit ?c.contents ?mc.contents ?mc.reservation_status
              ?m.Name ?b.Name }

  (and ( ?mc.compile_status = NotCompiled )
       ( delete [?c ?w] )   # this will also delete link from ?b->?c
       ( ?m.archive_status = NotArchived ));
  no_assertion;

# Deposit version of the code that someone has an existing branch from.
# ---------------------------------------------------------------------------
deposit[?c:CFILE, ?m:MODULE]:

  (and (exists WORKSPACE ?w  suchthat no_chain (member [?w.files ?c]))
       (exists CFILE     ?mc suchthat
                                 (and no_chain (member [?m.cfiles ?mc])
                                      no_chain (?mc.Name = ?c.Name)))
       (exists BRANCH    ?b  suchthat no_chain (member [?mc.branches ?b]))
       (exists CFILE     ?oc suchthat no_chain (linkto [?b.file ?oc]))):

  ## 1. Am owner of original lock on ?mc
  ## 2. Am not owner of branch file
  ## 3. CheckOutShared
  ## ----------------------------------------
  (and no_chain ( ?mc.owner = CurrentUser )
       no_chain ( ?oc.owner <> CurrentUser )
       no_chain ( ?c.change_status = Inspected )
       no_chain ( ?mc.reservation_status = CheckedOutShared ))

  { RCS deposit ?c.contents ?mc.contents ?mc.reservation_status
              ?m.Name ?b.Name }

  (and
       ( ?mc.compile_status = NotCompiled )
       ## Notify future depositer that they'll have to merge
       no_chain ( ?mc.reservation_status = CheckedOut )
      ( delete [?c ?w] )  # this will also delete link from ?b->?c
      ( ?m.archive_status = NotArchived ));
  no_assertion;
```

```
# Someone else has deposited early version.  Need to incorporate before
# allowing a deposit.
# ---------------------------------------------------------------------------
deposit[?c:CFILE, ?m:MODULE]:

  (and
     (exists WORKSPACE ?w suchthat no_chain (member [?w.files ?c]))
     (exists CFILE  ?mc suchthat (and
                                 no_chain (member [?m.cfiles ?mc])
                                 no_chain (?mc.Name = ?c.Name)))
     (exists BRANCH ?b suchthat no_chain (member [?mc.branches ?b]))
     (forall CFILE  ?none suchthat no_chain (linkto [?b.file ?none]))):

  ## 1. BRANCH object doesn't link to anything: Already been deposited.
  ## 2. Am owner of original lock on ?mc
  ## 3. CheckOutShared
  ## ----------------------------------------
  (and no_chain ( ?none.Name = "" )
       (or no_chain ( ?mc.owner = CurrentUser )   # hack for ispw
           no_chain ( ?w.owner = CurrentUser ))
       no_chain ( ?c.change_status = Inspected )
       no_chain ( ?mc.reservation_status = CheckedOutShared ))

  { RCS merge_code ?c.contents ?mc.contents ?m.Name ?b.Name }

  (and no_chain ( ?mc.reservation_status = Available )
                ( ?mc.compile_status = NotCompiled )
                ( delete [?c ?w] )
( delete [?b ?mc] )
                ( ?m.archive_status = NotArchived ));
  no_assertion;


# Original CheckOut has deposited early version.
# Need to incorporate before allowing a deposit.
# ------------------------------------------------
deposit[?c:CFILE, ?m:MODULE]:

  (and (exists WORKSPACE ?w    suchthat no_chain (member [?w.files ?c]))
       (exists CFILE     ?mc   suchthat
                                   (and
                                        no_chain (member [?m.cfiles ?mc])
                                        no_chain (?mc.Name = ?c.Name)))
       (exists BRANCH    ?b    suchthat no_chain (linkto [?b.file ?c]))):

  ## 1. Am owner of original lock on ?mc
  ## 2. CheckOutShared
```

```
  ## ----------------------------------------
  (and no_chain ( ?c.owner = CurrentUser )
       no_chain ( ?c.change_status = Inspected )
       no_chain ( ?mc.reservation_status = CheckedOut ))

  { RCS merge_code ?c.contents ?mc.contents ?m.Name ?b.Name }

  (and no_chain ( ?mc.reservation_status = Available )
                ( ?mc.compile_status = NotCompiled )
                ( delete [?c ?w] )
( delete [?b ?mc] )
                ( ?m.archive_status = NotArchived ));
  no_assertion;

# View the rcs history for a file

view_rcs[?f:FILE]:
  :
  { RCS view_rcs ?f.contents }
  ;

# overload the default add rule

hide
add_rule [?parent:WORKSPACE, ?att:LITERAL, ?name:LITERAL, ?class:LITERAL]:
    :
    { }
    (add [?parent ?att ?name ?class]);


hide create_rcs[?c:CFILE]:
  (exists WORKSPACE ?w suchthat (member [?w.files ?c])):

  (and no_chain (?c.reservation_status = Initialized)
       no_chain (?c.change_status = Idle))

  { RCS create_rcs ?c.contents }

  (and (?c.reservation_status = CheckedOut)
       (?c.change_status = StartChange));
  no_assertion;

hide create_rcs[?h:HFILE]:
  (exists WORKSPACE ?w suchthat (member [?w.files ?h])):

  no_chain (?h.reservation_status = Initialized)

  { RCS create_rcs ?h.contents }

  (?h.reservation_status = CheckedOut);
```

```
  no_assertion;
```

B.2.4 The DESIGN Process

B.2.4.1 design_review

```
#
#                     Oz Software Development Environment
#
#                                Copyright 1994
#                     The Trustees of Columbia University
#                            in the City of New York
#                              All Rights Reserved
#


strategy design_review

imports data_model;
exports all;

objectbase

DESIGN_REVIEW_TOOLS  :: superclass TOOL;
  review_doc  : string = "review_doc DOCFILE.change_request S
                                 DOCFILE.modified_change_request S
                                 DOCFILE.bug_report S";
  view_ps     : string = "view_ps DOCFILE.ps_file S";
  edit_doc    : string = "edit_doc DOCFILE.contents S";
end

end_objectbase

rules

# local review, there's one also in the CODING team.
# delegated rule

review[?doc:DOCFILE]:
 :
  delegate[?doc.owner]:

 (?doc.review_status = ReviewRequested)
 { DESIGN_REVIEW_TOOLS review_doc ?doc.change_request
                                  ?doc.modified_change_request
                                  ?doc.bug_report }

   (?doc.review_rc         = Succeeded);
   (?doc.review_rc         = Failed);
   (?doc.review_rc         = None);
```

```
edit[?doc:DOCFILE]:
:
  {DESIGN_REVIEW_TOOLS edit_doc ?doc.contents}
;


view_ps[?doc:DOCFILE]:
:
  {DESIGN_REVIEW_TOOLS view_ps ?doc.ps_file}
;
```

B.2.5 Selected Envelopes

B.2.5.1 store_notify_bug

```
#                      Oz Software Development Environment
#
#                                 Copyright 1994
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
#

ENVELOPE store_notify_bug;

SHELL ksh;

INPUT
        set_of string        : fname;
        set_of string        : lname;
        set_of string        : user_id;
OUTPUT
        none ;

BEGIN

echo "$fname $lname with user id $user_id is not logged in currently"
echo "The notification will be sent to him when he loggs-in"

echo "#!marvel script" > tmp.notify
echo "find_unnotified_bugs" >> tmp.notify
destination=‘echo ˜$user_id | tr -d ’ ’‘
eval cp tmp.notify $destination/.ozrc
echo "notification is sent to $user_id"
rm tmp.notify
```

```
RETURN "0";

END
```

B.2.5.2 analyze_bug

```
#                       Oz Software Development Environment
#
#                                  Copyright 1994
#                       The Trustees of Columbia University
#                              in the City of New York
#                                All Rights Reserved
#
#
# usage:
#

ENVELOPE template;

SHELL ksh;

INPUT
        text    : report;
        integer : test_status;
string  : performer;
string  : timestamp;
set_of text : cfiles;
OUTPUT
set_of object : bad_cfiles;
BEGIN


  rm -f FILE_LIST.$$
  for i in $cfiles
  do
    B=`basename $i`
    echo $B $i >> FILE_LIST.$$
  done

# call an in-house tool that aids in locating the bug
  bad_cfiles=`bug_report FILE_LIST.$$ $report "$timestamp" "$performer"`
  rm FILE_LIST.$$

# bad_cfiles is a subset of files which are pissibly faulty

RETURN "0" : $bad_cfiles;

END
```

B.2.5.3 branch

```
#
#                     Oz Process Centered Environment
#
#                             Copyright 1994
#                   The Trustees of Columbia University
#                         in the City of New York
#                           All Rights Reserved
#
# branch envelope
#
#

ENVELOPE branch;

SHELL sh;

INPUT
  text   : contents;
        string : new_or_extend;
OUTPUT
string : REV;

BEGIN

  BASENAME=`basename $contents`
  DIRNAME=`get_dirname $contents`
  rcsdirectory=$DIRNAME/RCS

  # If the current RCS directory doesn't exist, ERROR!!
  # ---------------------------------------------------
  if [ ! -d $rcsdirectory ]
  then
    echo "RCS file doesn't exist!"
    echo "Unable to make a branch"
    RETURN "1":"";
  fi

  # Have user select a locked version to branch on.
  # --------------------------------------------
  REV=`select-version.sh $rcsdirectory/${BASENAME},v`

  if [ -n "$REV" ]
  then
    # Check out (but don't lock) that particular branch into a special file
    # ------------------------------------------------------------------
    USER=`whoami`
    if [ ! -d $DIRNAME/.$USER ]
    then
```

```
      mkdir $DIRNAME/.$USER
    else
      rm -f $DIRNAME/.$USER/$BASENAME
    fi

    co -r$REV $rcsdirectory/${BASENAME},v $DIRNAME/.$USER/$BASENAME
  else
    echo "No branch made"
    RETURN "1":"";
  fi

RETURN "0":"$REV";
END
```

B.2.5.4 white_board

```
#                       Oz Software Development Environment
#
#                                  Copyright 1994
#                       The Trustees of Columbia University
#                              in the City of New York
#                                All Rights Reserved
#
#
# usage:
#

ENVELOPE white_board;

SHELL ksh;

INPUT
        text : cfile;
        set_of text : design;
        set_of string : coding_site_ip_address;
        set_of string : design_site_ip_address;

OUTPUT
        none ;

BEGIN

  role=`get_rule $PWD`

  if [ $role = "CODING" ]
  then

   FILE_LIST=/tmp/wb_file_list
```

```
   FILE=‘basename $design‘
   echo $FILE $design > $FILE_LIST
   wb.static $design_site_ip_address/7200 &

   sleep 3

   wbimport $FILE_LIST

   wait $!

   echo "What’s the verdict ? type 0 for Revise and 1 for Reject"
   read res

   if [ $res = "0" ]
   then
       RETURN "0";
   else
       RETURN "1";
    fi

# if DESIGN
  else

     sleep 3
     wb.static $coding_site_ip_address/7200
    RETURN "0";
  fi
END
```

Index