An Interoperability Model for Process-Centered Software Engineering Environments and its Implementation in Oz

Israel Z. Ben-Shaul
Technion-Israel Institute of Technology
Department of Electrical Engineering
Technion City, Haifa 32000
ISRAEL
issy@ee.technion.ac.il

Gail E. Kaiser
Columbia University
Department of Computer Science
500 West 120th Street
New York, NY 10027
UNITED STATES
kaiser@cs.columbia.edu

CUCS-034-95

Keywords: process modeling, software engineering environments, distributed systems, workflow interoperability, collaborative work.

This paper is based on work sponsored in part by Advanced Research Project Agency under ARPA Order B128 monitored by Air Force Rome Lab F30602-94-C-0197, in part by National Science Foundation CCR-9301092, and in part by the New York State Science and Technology Foundation Center for Advanced Technology in High Performance Computing and Communications in Healthcare NYSSTF-CAT-95013.

Abstract

A process-centered software engineering environment (PSEE) enables to model, evolve, and enact the process of software development and maintenance. This paper addresses the problem of process-interoperability among decentralized and autonomous PSEEs by presenting the generic International Alliance model, which consists of two elements, namely Treaty and Summit. The Treaty abstraction allows pairwise peer-peer definition of multi-site shared sub-processes that are integrated inside each of the participating sites, while retaining the definition- and evolution-autonomy of non-shared local sub-processes. Summits are the execution abstraction for Treaty-defined sub-processes. They enact Treaty sub-processes in multiple sites by successively alternating between shared and private execution modes: the former is used for the synchronous execution of the shared activities, and the latter is used for the autonomous execution of any private subtasks emanating from the shared activities. We describe the realization of the models in the Oz multi-site PSEE and evaluate the models and system based on experience gained from using Oz for production purposes. We also consider the application of the model to Petri net-based and grammar-based PSEEs.

1 Introduction

As software systems become more complex and larger in scale, their development and maintenance requires more people with various skills, often organized into groups. The decomposition into groups can be characterized by the level of intra-group vs. inter-group heterogeneity. For example, a project may be composed of separate teams for requirements elicitation, functional specification, design, coding, testing, documentation, and maintenance. This decomposition exhibits high intra-group homogeneity and inter-group heterogeneity. Alternatively, a project may be decomposed into teams that are each responsible for full development of a distinct component of the system, exhibiting intra-group heterogeneity. Another characteristic of the project organization is whether it is formed top-down or bottom-up. An example of the latter is when multiple independent organizations team up (perhaps for a limited period of time) to develop a system. Finally, project personnel may be divided into, or made up of pre-existing, physically dispersed teams (or even individuals, e.g. telecommuting from home), a scenario that becomes more frequent with the advances in networking technologies.

Although the various decompositions have their own specific requirements, a common desirable property in a multi-team development is to allow some degree of operational as well as managerial team *autonomy*. For example, it may be desirable to allow teams to use their own set of software tools and hardware, their own private files or databases, and their own development policies and workflow, or process. Furthermore, when the teams belong to different organizations, autonomy and privacy are "hard" constraints that cannot be compromised or a priori restricted. At the same time, the autonomous teams need to *collaborate* in order to develop the product. For example, they may need to share tools or employ multi-user tools across teams, they may need to exchange and/or share files and other data, and they may need to agree on some common policies and workflow, at least for the parts of the work that involve collaboration.

The concept of system-interoperability, which has been largely motivated by the emerging globalization of computing, has been increasingly gaining popularity in various domains such as workflow interoperability for business process re-engineering [19], multi-database interoperability [12], and general client-server interoperability [43, 37]).

In this paper we explore interoperability in the context of process-centered software engineering environments (PSEEs). PSEEs are systems that support large scale software development by providing: (1) mechanisms and notations for explicitly *modeling* the process of development and maintenance of software, including task definitions, control integration such as global task ordering and local constraints on their activation, tool integration, data modeling and integration, and user modeling; and (2) mechanisms for *enacting* the modeled process by the PSEEs process-engine, where forms of enactment include process automation (e.g., Marvel [30]), consistency (e.g., CLF [46]), monitoring (e.g., Provence [34]), enforcement (e.g., Darwin [42]), and guidance (e.g., Merlin [51]).

Thus, the PSEE-interoperability problem is to balance autonomy and collaboration among multiple *processes*, both in the modeling and the enactment phases, as a basis for collaboration among multiple groups.

1.1 Requirements and Scope

decentralized PSEEs

We deliberately use the term decentralization as opposed to distribution to emphasize that our focus in this work is on interconnecting environments that are independent and loosely-coupled, both physically and logically. This is in contrast to a "classical" distributed system in which a single and homogeneous logical perspective is given to its applications but is physically distributed into multiple computing units. Note that PSEE distribution (in the "classical" meaning) is a form of "vertical" scale-up, in that it allows for more users to work, but under the same process and typically with some bounded physical distance (typically a local-area network). Here we address mainly "horizontal" scale-up, where the number of users per group sharing the same process may not grow much (and in fact may degenerate to 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 other groups.

Another aspect that is derived from decentralization is *Independent Operation and Self-Containment*. This means that a sub-environment (henceforth 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 a "share-nothing" architecture. That is, no multi-site service, mechanism, or data in the environment can be centralized or physically shared and all interaction should be based solely on message passing.

Decentralization also implies that A multi-site PSEE should impose minimum overhead on the operation of local work in SubEnvs. The underlying assumption is that most of the work done by a SubEnv is local to that SubEnv, and therefore each SubEnv should still be optimized towards local work.

Finally, we make the distinction between inter- vs. intra- process coordination. The latter 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 (see, for example, [4]). In contrast, we focus in this paper on collaboration between users or teams with different processes, different schemas, and different project databases.

PSEE Autonomy

Each local SubEnv should have complete control over its process, tools and data, while allowing access by remote SubEnvs under restrictions that are solely determined by the local SubEnv. Access to a SubEnv has two perspectives: access to the local artifacts owned by the SubEnv through some process interface; and access to, and interaction with tools and actual process tasks. Autonomy constraints imply that a SubEnv's data, tools and process are by default private, and some work has to be done to allow sharing and remote use.

Moreover, once defined, sharing should be restricted to the minimum degree necessary for interoperability.

Process Heterogeneity

Heterogeneity in software systems in general (and PSEEs in particular) can be classified into four levels: the operating system, the runtime support (PSEE engine), the front-end language (Process Modeling Language (PML)), and the applications (specific processes). For example, a multi-PSEE can support heterogeneous processes written in the same PML (application heterogeneity), it can support heterogeneous PMLs but still require the same underlying (multi-lingual, in this case) process engine (language heterogeneity), or it can support interoperability across heterogeneous PSEE engines (system heterogeneity). Support for heterogeneity is, in general, an extremely difficult problem, particularly in the context of decentralization. This paper explores a limited aspect of heterogeneity by fixing the system and language levels (although not restricting to *a* particular system or language) and supporting heterogeneity at the process model level. This is in contrast to ProcessWall [24] for example, which focuses on language heterogeneity (see Section 6).

Allowing Pre-Existing Processes

In a typical top-down approach a system (process) is decomposed into sub-systems (sub-processes), usually by a global authority which dictates where and how the different parts of the system, both control and data, will be defined and executed. In contrast, the bottom-up approach, which is closely associated with decentralized systems, does not assume a global authority, and multi-site applications are constructed between the possibly pre-existing local (sub)systems, thereby avoiding the need to have any a priori knowledge of the "neighboring" subsystems before the time of construction. Our focus here is on the more decentralized bottom-up construction of multi-PSEE processes. This is in contrast to most other distributed PSEEs which decompose a single process in a top-down fashion into sub-processes with predefined and coordinated interfaces (as done in ProcessWEAVER [16], see Section 6). Notice that we do not exclude support for top-down methodology; we do not, however, enforce it.

Thus, it should be possible for pre-existing SubEnvs to "join" an on-going multi-site environment or to form a new one with minimal configuration overhead. Similarly, a "split" of a SubEnv from its currently configured multi-site environment should be supported.

The rest of this paper is organized as follows: Section 2 presents the interoperability model for definition and enactment of multi-site activities; Section 3 discusses an actual implementation of the model in the Oz rule-based multi-site PSEE; Section 4 outlines the application of the model to other non-rule-based PMLs; Section 5 evaluates the research based on experience gained by using Oz in a production environment; Section 6 compares to related work; and Section 7 summarizes the contributions of this research and outlines future directions.

Our earlier paper [9] introduced a preliminary version of the model and its implementation, focusing on enactment. Our book [7] presented a revised, comprehensive and formalized model with detailed coverage of both the definition and enactment aspects, and describes a mature implementation. This paper abridges the book, and adds new material in two areas: Section 5 is entirely new. It describes anecdotal experience and provides statistics on using one Oz environment for production purposes by up to 14 users over approximately 8 months (to date), and (re)evaluates the interoperability model based on this experience. This and other retrospective led to abstraction of local process evolution and dynamic Treaty verification out of the Oz realization and its generalization and reformulation as part of the International Alliance model in Section 2.2.

2 The Process Interoperability Model

At a high-level, our approach taken to meet the challenges described above is to exploit the fact that process models are encoded in a formal notation, and use it as a basis for formally modeling *interoperability* among process models. Furthermore, we extend the concept of process enactment to encompass enactment support for the actual multi-process activities that enable collaboration between the sites, in addition to the conventional single-site execution.

We begin with definitions of terms and a formalization of concepts that are used in the rest of the paper.

2.1 Basic Concepts and Definitions

2.1.1 General Process Terminology

As stated earlier, a process model defines a project-specific process and is encoded in some process modeling language (PML), and a process-centered software engineering environment (PSEE) is a system in which processes are modeled and enacted. A process model can be *instantiated* when it gets bound with real data artifacts, tools, users, and any other system bindings which are required by the PSEE. An instantiated environment is an executable process model. For brevity, we shall call an instantiated environment simply an *environment* (or SubEnv in the context of multi-site PSEEs). This term should not be confused with the term PSEE, which refers to the system on which (instantiated) environments run.

Note that the same process model can be instantiated in multiple environment instances. At some point during its enactment, the process model of an environment might need to be changed, e.g., because of feedback from the environment and/or new requirements, in which case it is *evolved*, i.e., its persistent process and product states are upgraded to comply with the new process definition.

We can identify a generic three-level context hierarchy in process models. A particular PML may have more or fewer levels, but we assume that there is some mapping into these core levels:

1. *Activity* — This is the PSEE's interface to actual tools, including input/output data

bindings, user bindings (i.e., who should execute the tool if it is interactive), and machine binding (i.e., on what machine should the tool execute).

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 [21] a process step corresponds to a transition along with its (optionally) attached predicates; in the Articulator task graphs [41] 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* — A set of logically related process steps that represent a coherent process fragment. Depending on the specific PML and PSEE (1) there usually are some ordering constraints 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) depending on the subtasks, a task might be partially carried out automatically by the PSEE 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.

2.1.2 A Multi-User, Single-Process PSEE

An (instantiated) environment E is defined as a quintuple

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

where:

- U is 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 an environment *administrator*, who defines and can modify each of the elements in E (analogous to the role of a database administrator).

- T is a set of *tools* being used in the environment. The tools can be off-the-shelf, or customized to work in the PSEE, but in either case the PSEE is assumed to have means to invoke those tools with process activities.

- S is a *schema* sub-language representing data types for modeling the process and product data. S could be part of an external database that is separate from the PML (as in SPADE [2]) or it could be part of the PML (as in Marvel [30]). In addition, the process data could be kept separately from the product data (which may reside in the native file system). In PMLs with no data modeling at all (e.g., Synervision [27]) this element degenerates to the empty language and all data elements are considered to belong to the single "universal" class.

- D is a *database* for storing the persistent objects, each belonging to a certain type (or class) from S.

- P is 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.

Based on the above definitions and requirements, a high-level view of a single-process PSEE with an instantiated environment is depicted in Figure 1. It consists of a data server managing the process schema and data, a tool server integrating the project's tools, a process server executing the defined process, a client-user interface, and a communication layer connecting all components. A typical interaction with the PSEE 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.

2.1.3 A Multi-Group, Multi-Process PSEE

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 above, maintaining its own data repository, tools, and process model. 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 PSEE has the necessary mechanisms to reference and bind remote data objects to local activities. 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 authorization from the owner process.

The high-level architectural view of a generic decentralized PSEE with a three-site decentralized environment is depicted in Figure 2. Each local SubEnv consists, in addition to the single-process components, of an inter-process server, a remote-data server, a remote-tool server and, a connectivity server that enables SubEnvs to connect to, and communicate

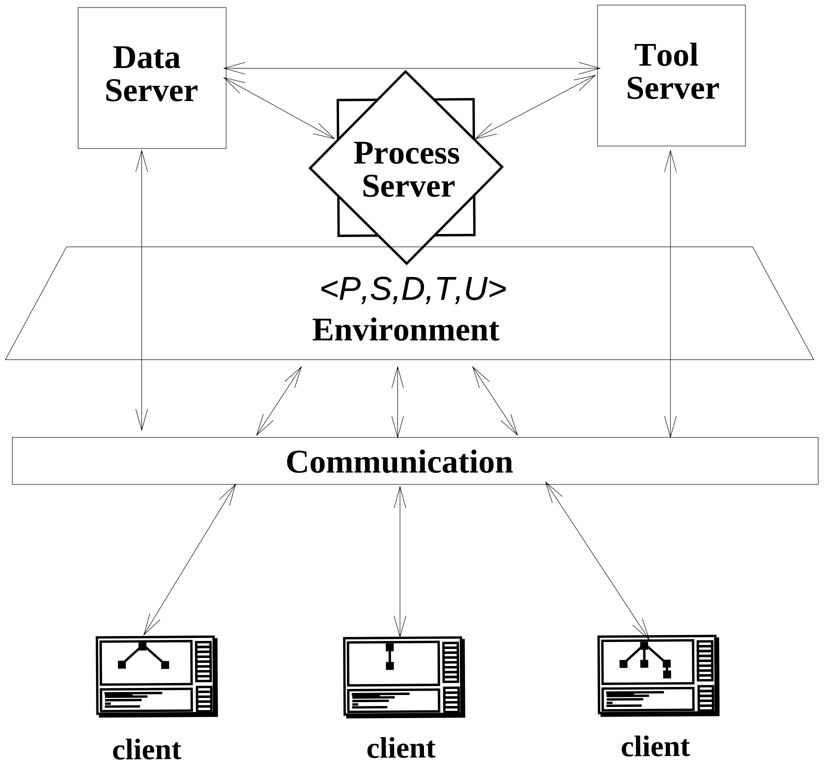


Figure 1: A Generic Multi-User Single-Process Environment

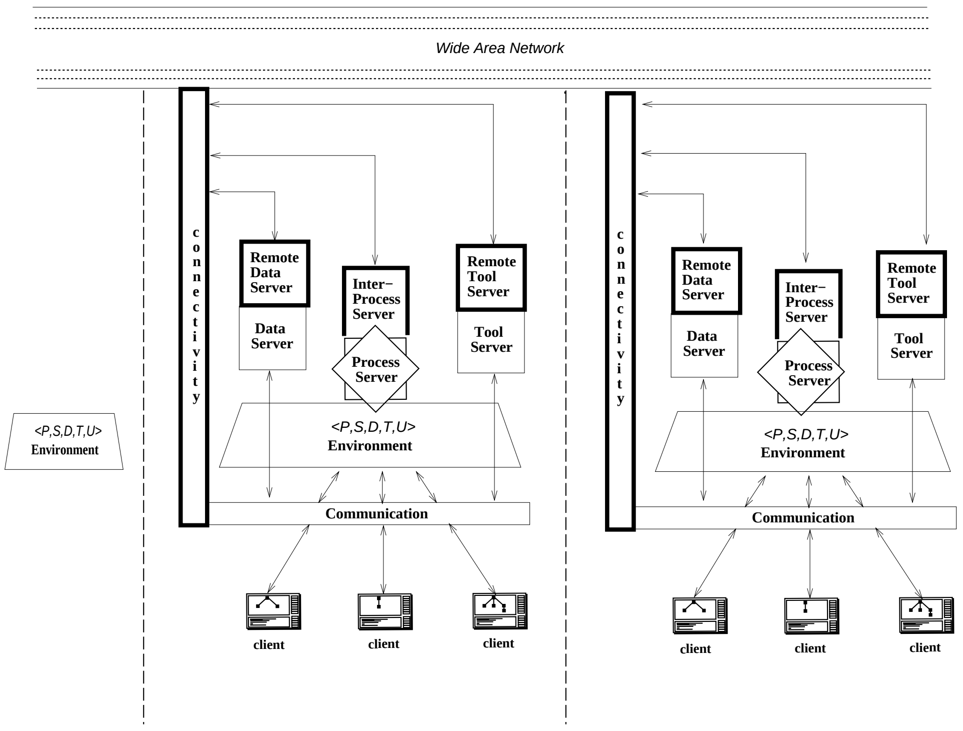


Figure 2: A Decentralized Environment

with, other SubEnvs participating in the same (global) environment. These elements together form the necessary infrastructure support needed for process-interoperability. Notice how the "no sharing" property allows normal operation of some sites when other sites (e.g., the leftmost site in the figure) are inactive or disconnected.

We define a *multi-site activity* as an activity that uses data objects, and optionally users and/or machines from one or more remote sites. Note that the activity (or tool) per-se, need not execute on multiple sites concurrently (as in groupware tools), it can execute at one site into which the remote data objects are transferred. Thus, a given activity may or may not be considered a multi-site activity during different invocations, depending on whether the resources bound to it include remote elements. Multi-site activities are the building blocks of our process-interoperability model.

Finally, referring back to the context-hierarchy described earlier, 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 our model 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.

2.2 Defining Process Interoperability: the Treaty

In general, the interoperability model is based on the idea that sites explicitly specify in what ways they are willing to participate in a multi-site operation, and the specifications are loaded into each site's local PSEE to establish all that is needed to enable those interactions. Some intuition to the model may be gained by the "international alliance" metaphor, whereby independent countries sign "treaties" that determine their collaboration but retain full control over their local laws. 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. In addition, Treaties have to be verified to make sure that they are being carried out as agreed.

2.2.1 Treaty Requirements

The following is a set of requirements specific to modeling interoperability, driven by the high-level requirements presented earlier in Section 1.1.

1. *Common sub-process* — 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 this "unit of commonality" might also be the process step, or even the task. In any case, this unit represents 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. In a Petri net formalism, for example, 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 [41]), 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. *Common sub-schema* — This requirement applies mainly to PSEEs with database and schema support. In such PMLs, invocation of multi-site activities (as part of a multi-site common sub-process) requires the involved SubEnvs to share a *common sub-schema*, so that the types of the parameters specified in the invocation are defined in the relevant 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 data types (with possible support for limited type coercion) 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. *Remote access control* — 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, but parts of them can be made accessible for remote access by multi-site activities.

4. *Locality of specifications* — 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 of a given global environment, not necessarily all of them. Furthermore, a SubEnv may contain multiple sub-processes, each of which is shared with different subsets of peer SubEnvs. 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. 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. The independent operation requirement further implies that the Treaty mechanism should minimize the inter-site dependencies that are required to maintain a consistent Treaty. This point is addressed in Section 2.2.4.

In the rest of this section we address requirements 1, 4 and 5. Requirements 2 and 3, which are more database oriented, are covered elsewhere [7].

2.2.2 Alternative Approaches

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 domain of distributed programming languages and systems, and investigate alternatives there:

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.

   The main problem with applying the above approach to our case is that it provides the wrong abstraction. Its prime motivation is to distinguish between a unit's external (public) interface and its internal (private) implementation, promoting modularity, encapsulation, and reuse. While this might be the case in process interoperability, 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. Another problem with this approach is that it is language-based, and thus static in nature, conflicting with the dynamic inclusion and evolution requirement stated earlier. 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.

2. Process interconnection language, separate from a specific PML — This is analogous to dynamic module interconnection languages, in which a separate notation is used to denote how modules are inter-connected. For example, the Darwin [38] configuration language[1] (the successor to Conic [39]) 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 is `require` and `provide`).

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

The abstraction here is closer to our needs, and it can also 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 [49], 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.

Our Treaty abstraction for defining process-interoperability is different than any of the above alternatives and is geared towards satisfying the requirements. It is defined pairwise between each two SubEnvs that intend to collaborate, reflecting the peer-peer nature of interoperability; it is defined inside each of the participating SubEnvs, to address decentralization; and it allows unilateral cancelation coupled with dynamic verification, to address autonomy. Finally, in contrast to the language-based approaches, we advocate a *system-based* approach, i.e., we extend the available PSEE's execution engine with "system calls" that support the definition of the interoperability model. As such, this approach does not require the invention of a whole new process-interoperability modeling language, nor does it make any assumptions about a particular PML, making it generically applicable. (We will return to the issue of language- vs. system-based approach in Section 5.2.6.)

2.2.3 The Treaty

In the following discussion, the following notation is used:

- E_i denotes an instantiated environment.
- A_i is used to denote a set of process steps that form a common sub-process. Note that in terms of the definition of an environment, A_i is a subset of P (process), i.e., it does not necessarily contain a subset of T (tools), D (data), U (users), 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. A pre-requisite to this operation is that $A_1(E_1)$ was previously exported in E_1. The successful outcome of this operation generates $A_1(E_2)$, a local replicated 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 (Section 3.2.3 shows a concrete implementation of *import*). Note that the name of A_1 must be distinct from any other pre-existing or new activity in E_2 so that it can be uniquely identified at runtime.

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 permanently attached to the *export* and *import* operations in some fashion, e.g., to associate a request to execute on remote data with the *export* operation. However, early experiments with our implementation revealed that these are indeed orthogonal concerns that should be distinguished. Thus, we separate the issue of how to provide common multi-site activities from the concern of how to restrict or control their application.

We define the following two execution privileges directives, each of which could be used in conjunction with either of the above operations:

1. *request*(A_1, E_1, E_2) — E_1 specifies an intent 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) — E_2 allows A_1 to be used by E_1 on data from E_2. Once again, A_1 could be originally defined at E_1 (or at some third site from which E_1 imported it), in which case it was later imported by E_2, or it could be exported by E_2 and imported by E_1.

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:

$$T_{A_1}(E_1, E_2) = \mathit{export\_request}(A_1(E_1), E_2); \mathit{import\_accept}(A_1(E_1), E_2) \quad (1)$$

$$T_{A_1}(E_1, E_2) = \mathit{export\_accept}(A_1(E_2), E_1); \mathit{import\_request}(A_1(E_2), E_1) \quad (2)$$

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 expression (1) A_1 is initially defined in E_1 and is exported to E_2, which imports it; whereas in expression (2) 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 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 to the relevant SubEnv by another SubEnv.

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 earlier regarding execution privileges. 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:

$$T_{A_1}(E_1, (E_2 \ldots E_n)) = \bigcup_{i=2}^{n} T_{A_1}(E_1, E_i) \quad (3)$$

In words, it is the union of all pairwise (simple) Treaties with E_1 as the source SubEnv. 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:

$$FT_{A_1}(E_1, E_2) = T_{A_1}(E_1, E_2); T_{A_1}(E_2, E_1) \tag{4}$$

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

$$FT_{A_1}(E_1, E_2 \ldots E_n) = \bigcup_{i<j} FT_{A_1}(E_i, E_j) \tag{5}$$

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 as in simple Treaties.

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:

$$T_A(E_1, (E_2, E_3)) \tag{6}$$
$$T_A(E_2, (E_1, E_3)) \tag{7}$$

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.

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 withdraw 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 further use of 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 mechanism for establishing shared common sub-processes, 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.

2.2.4 Local Evolution and Dynamic Treaty Verification

In Section 2.2.1 we identified the need to be able to perform process evolutions and Treaty-leaving operations locally with minimum interaction with remote SubEnvs, while still being able to dynamically check the validity of Treaties. We begin with a definition of a valid (or consistent) Treaty, analyze all possible ways in which it can be invalidated, and discuss our dynamic Treaty-verification algorithm.

A (simple) Treaty from E_1 to E_2 on A_1 is said to be valid if and only if all three conditions below hold:

1. Either:
   (a) A_1 is marked at E_1 as exported to E_2, and is marked at E_2 as imported from E_1.
   (b) A_1 is marked at E_1 as imported from E_2, and is marked at E_2 as exported to E_1.
2. A_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. A_1 is identically defined in both SubEnvs. Since there is no shared space in which Treaties are stored, there must be a way to guarantee that original Treaties have not been altered by the time they are invoked on remote data. We refer to this condition as the "common sub-process invariant".

The first condition is invalidated whenever *unexport* at the exporting site, or *unimport* at the importing site, is issued. When an activity is issued, *unexport* can be easily detected locally at the invoking site — the invocation is rejected if the issued task is not (anymore) exported. *unimport* is also easily detectable since when the task is requested on the remote site, if it is part of a sub-process which has been unimported (and thus removed from the process' set of tasks) the requested activity will simply not be found.

As for the second condition, both *request* and *accept* privileges have to be checked for their validity. E_1 can lose its *request* privileges on A_1 if the equivalent of $cancel(A_1, E_1, E_2)$ was issued. This can occur in one of two ways, depending on the method by which the *request* privileges were originally obtained: (1) If through *export-request*, then an *unexport-request* on A_1 from E_1 to E_2 revokes *request* privileges. This can be validated at E_1 locally when the multi-site activity is invoked, at the same time that the *export* privileges are checked. (2) If through *import-request*, then an *unimport* on A_1 at E_1 invalidates condition 2. Thus, validity checking is similar to that for condition 1.

E_2 can revoke *accept* privileges from E_1 on S_1 whenever the equivalent of $deny(A_1, E_1, E_2)$ occurs at E_2. This can also occur 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 verify this case, E_2 must explicitly check for proper *accept* privileges every time an activity in A_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 activity will simply not be found.

The third condition, requiring identical copies of the Treaty sub-processes at the participant SubEnvs, can become unsatisfied as a result of various (local) process evolutions, and is more complicated to check for. The main problem occurs when a Treaty sub-process is modified at the exporting ("source") SubEnv. Regardless of the process privileges attached to the exported task, such evolution violates the common sub-process invariant.

One method to address this (which was implemented in Oz) is based on *evolution timestamps*. The idea is for the local SubEnv to assign a "timestamp" each time a process is compiled and loaded locally. When a sub-process is imported, its timestamp is also shipped and stored at the importing SubEnv. At run-time, whenever a multi-site activity 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) sub-process, or by reloading the process (perhaps automatically), which also fetches the up-to-date versions of all imported strategies from the exporting SubEnv(s).

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 transfers the responsibility of upgrading the imported rules to each individual SubEnv when it actually needs to use them. This "lazy update" approach fits well with the general decentralized philosophy. Figure 3 summarizes this section by presenting the dynamic Treaty verification algorithm that is executed in the acceptor SubEnv prior to invocation of each multi-site activity.

2.2.5 Treaty Summary

Treaties are the abstraction mechanism used for the *definition* of 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

```
verify-treaty (TaskId, SrcSubEnv, DstSubEnv):
                                                        /* Executes at DstSubEnv */
                                                                 /* condition 1 */
    if ( find task with the given TaskId )
    then
                                                                 /* condition 2 */
        if ( DstSubEnv accepts TaskId from DstSubEnv)
        then
                                                                 /* condition 3 */
            if (Tasks's remote timestamp = Tasks's local timestamp)
            then
                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
            end if
        else
            There is no Treaty on that Task, reject execution
            Reason: an equivalent of cancel occurred
            Reactivation: DstSubEnv needs to accept the Task
        end if
    else
        Requested task does not exist in local SubEnv, cannot execute
        (Re)activation: DstSubEnv needs to (re)import the task
    end if
```

Figure 3: Dynamic Treaty Verification Algorithm

(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 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, although Treaties are defined pairwise, multi-site Treaties involving an arbitrary number of sites can be formed. It might appear that our approach suffers from being too low-level in that it makes it somewhat complicated to define multi-site Treaties by requiring to form pairwise Treaties. However, this formalism ensures maximum process autonomy. Further, 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 may decide to bundle some of the operations into a single built-in command. For example, it could set defaults for combining *export* and *import* with *request* and *accept* but allow the expert process administrator to modify them. Finally, the PSEE 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 Section 3).

2.3 Multi-Process Execution: the Summit

The Treaty mechanism establishes common sub-processes between sites, and defines execution privileges over the common multi-site activities. However, it does not impose a particular approach on how to *execute* these shared processes. This is the role of Summits.

2.3.1 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. Multi-site activities that are defined in a common-sub-process are executed at the coordinating SubEnv by fetching to it all remote data, while local activities emanating in each local process from the common-sub-process are executed locally at each site with local data.

2.3.2 The Summit

Following the "international alliance" metaphor mentioned earlier, our decentralized execution 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). It is important to note that each of the two metaphors, namely Treaty and Summit, are independent from each other in our model. That is, whereas in the international community Summits (may) lead to Treaties, in our model Treaties actually enable Summits.

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 remote 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 by the underlying single-process PSEE.) We call the process from which the multi-site activity is invoked the *coordinating process*. The Summit protocol consists of the following phases:

1. *Summit Initialization and Treaty Verification* — 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 executing the Treaty-verification algorithm shown in figure 3.

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 and tools, in the local SubEnv.* Pre-Summit actions include: (1) Verification that prerequisites imposed by the process step enclosing the activity are satisfied locally; this may be regarded as "internal" constraints. (2) Verification that the activity can be executed with respect to the overall task workflow; this may be regarded as "external" constraints (see [31] for more on this distinction). (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. Note that Pre-Summit requires that all involved SubEnvs identify the <u>same</u> requested activity, in order to know what to verify/satisfy. This is guaranteed through the *import* mechanism of the Treaty.

   One optimization that can be made in some cases (depending on the PML as well on the specific activity) is for the coordinating process to determine locally whether or not launching a 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 execution, 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. The activity is executed synchronously, and it may or may not execute at one location depending on the kind of tools associated with the activity. For example, it may launch a cooperating set of tools, on one or more sites, involving one or several users.

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 and tools, in the local SubEnv.* Post-Summit actions include: (1) Assertions on the process and product data that reflect the fact that the various activities were executed (depending on the PSEE, 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). If any Summit activity 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.

Thus, both pre- and post-Summit phases occur asynchronously in each SubEnv *only* according to its local process, while execution of the Summit phase occurs synchronously and 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.

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

2.3.3 Example

The following example illustrates the execution of Summits. Assume there are three development teams working in separate sub-environments **SE1**, **SE2**, and **SE3**, who are responsible for three disjoint components of a system **S**, labeled **S1**, **S2**, and **S3**. The teams operate at different sites and reside in different geographical areas. They each work on their own artifacts (e.g., files, documentation) using their private tool set and their own processes. 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 should be taken: (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. 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.

Figure 4 illustrates the enactment of this example as a (composite) Summit. 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

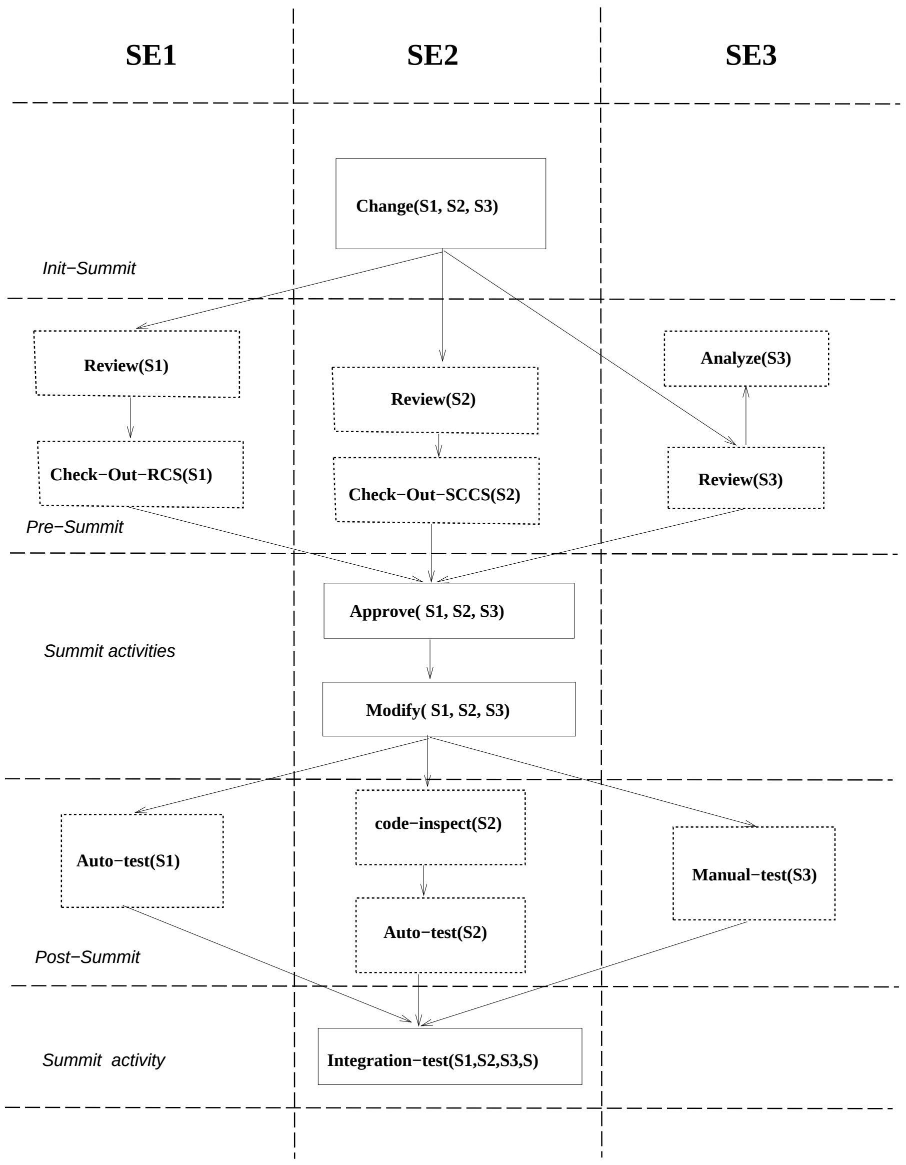


Figure 4: An Example Summit

the other SubEnvs perform automatic 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 4 depicts one particular execution trace of the process, not the whole process. For example, a different execution would occur if the `review` activity failed at **SE1** (i.e., the reviewer did not accept the proposed change). In this case, a revision phase would be followed, after which a second review would be scheduled, and so forth, until the review succeeds.

3 Realization of the Interoperability Model in Oz

The generic model, as a high-level abstraction, leaves many aspects undefined and unresolved, both technical and conceptual. We address some of these issues here by discussing the realization of Treaties and Summits. The architectural aspects of Oz, including site interconnectivity, configuration, transactions, database and cache management, are beyond the scope of this paper and can be found elsewhere [7, 8].

3.1 Oz Overview

Oz is a multi-process PSEE (as defined in Section 2.1.3), and it supports definition and execution of autonomous multiple SubEnvs following the Treaty and Summit models. When not interoperating with other SubEnvs, the functionality of a local SubEnv resembles that of Marvel [11], the predecessor to Oz. 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. The process modeling language of Oz is based on the Marvel Strategy Language (MSL) [36]. 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 rules which correspond to the notion of process-steps in the generic context hierarchy. A rule consists of: a signature, i.e., names and types of formal parameters; a binding section, where additional objects (termed derived parameters) are bound to the rule as a result of querying the database; a pre-condition consisting of a (composite) predicate over the arguments; an activity which interfaces to external tools and data; and a set of mutually-exclusive effects. A rule can be fired either directly by a user (via a client, see below), or indirectly, as a result of rule chaining. When a rule is fired, its condition is evaluated. If the evaluation fails (i.e., the predicate evaluates to false), the rule processor attempts to automatically satisfy the rule by backward chaining to other rules whom effect may satisfy the failed condition, recursively. If the condition is (or has become) satisfied, the activity of the rule is spawned and executed on behalf of the user who invoked the rule (or, in case of a chained rule, the user who issued the rule that chained to this rule). Upon completion, the activity returns a return code that determines which effect of the rule to assert. The rule processor then attempts to forward chain to rules whom condition have become satisfied as a result of the assertion, recursively. Thus, process steps are implicitly interrelated by logical matchings between effects and conditions of rules. In order to enable finer control over the degree of chaining, several chaining directives can be applied on rule predicates. For example, a `no_forward` directive on a rule's effect disables any forward chaining from that rule.

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 multi-server "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.

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 the local tools (encapsulated in rule activities), on local data objects, under the local process. 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. 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 manipulation of remote data can be done only by binding remote objects as parameters to Treaty rules.

For example, figure 5 shows how the client for user `israel`[2] is connected to the local server of SubEnv `NY`, with a (default) view of the local objectbase[3] (parent-child relationships are depicted with straight lines and links by curved lines). Figure 6 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, and he supplies arguments to the rules by clicking on objects from the objectbase. In particular, if a remote objectbase is open, `israel` 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. 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

[2]The user's name is shown in the upper left corner of the interface window.

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

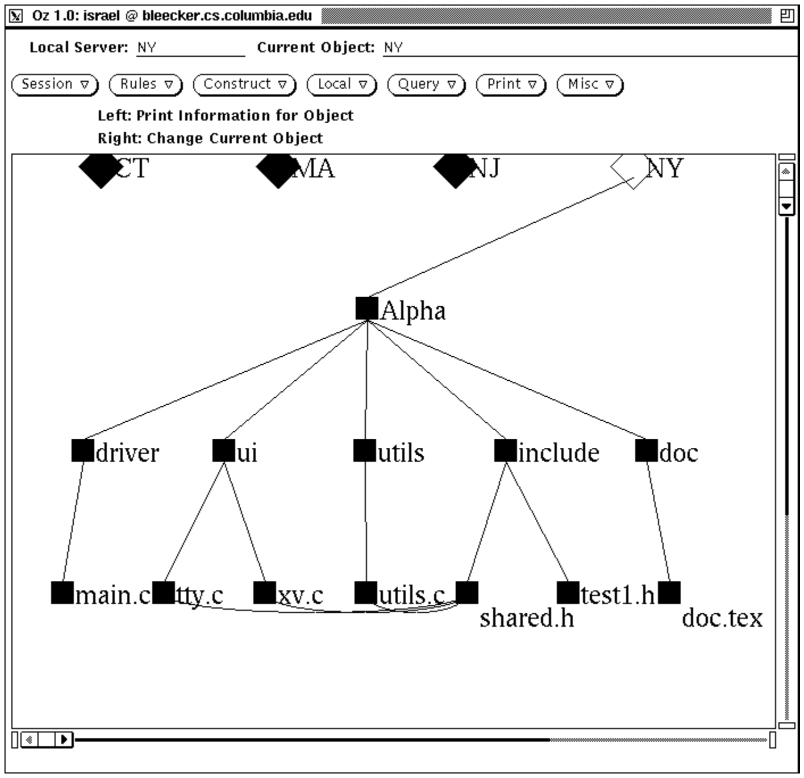


Figure 5: An Oz Environment

no explicit access to remote processes, it cannot invoke "remote Summits", thus all Summits are initiated by local clients.

3.2 Treaty in Oz

Treaties in Oz follow the formal Treaty model. The basic unit of commonality in Oz is the *rule*. However, as a "syntactic sugar", the unit that is exported and imported is the *strategy*, a bundling construct for rules, somewhat analogous to a module consisting of functions in modular programming languages.

Oz provides five built-in commands for establishing Treaties: *export*, *import*, *unexport*, *unimport*, and the "non-standard" *treaty* operation. 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 in Section 2.2.

3.2.1 export

The *export* operation is defined as:

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

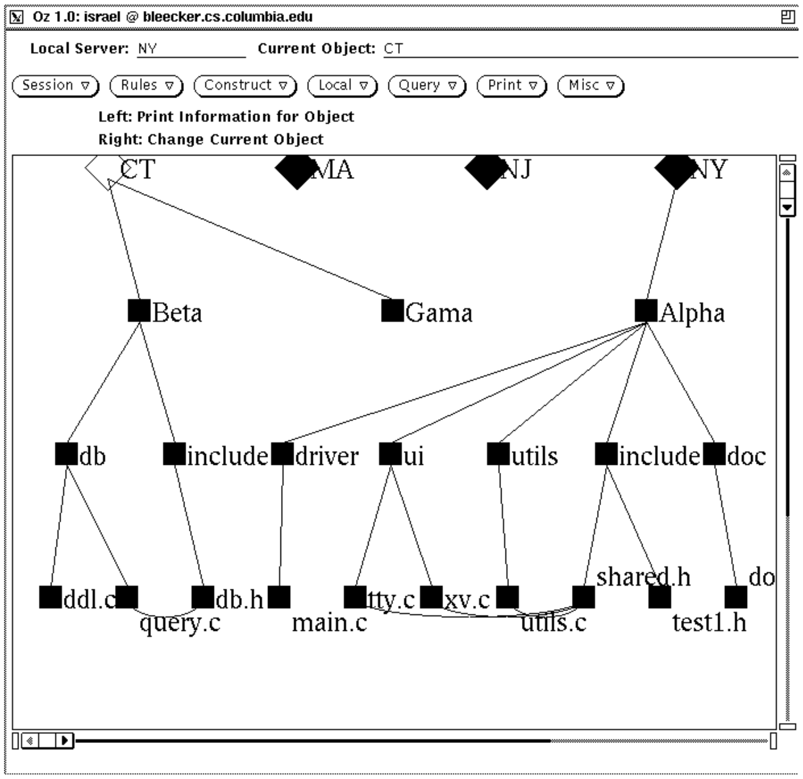


Figure 6: Oz Environment with one open remote site

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 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.

3.2.2 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.

3.2.3 import

The import operation is defined as:

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

import is the main operation in Treaties. We assume the existence of the necessary underlying infrastructure to communicate with the remote SubEnv (This topic is beyond the scope of this paper, see [7]). In particular, there must be a connection from *DstSubEnv* to *SrcSubEnv*, since the operation is initiated at *DstSubEnv* but it involves both SubEnvs. The realization of *import* consists of four distinct phases:

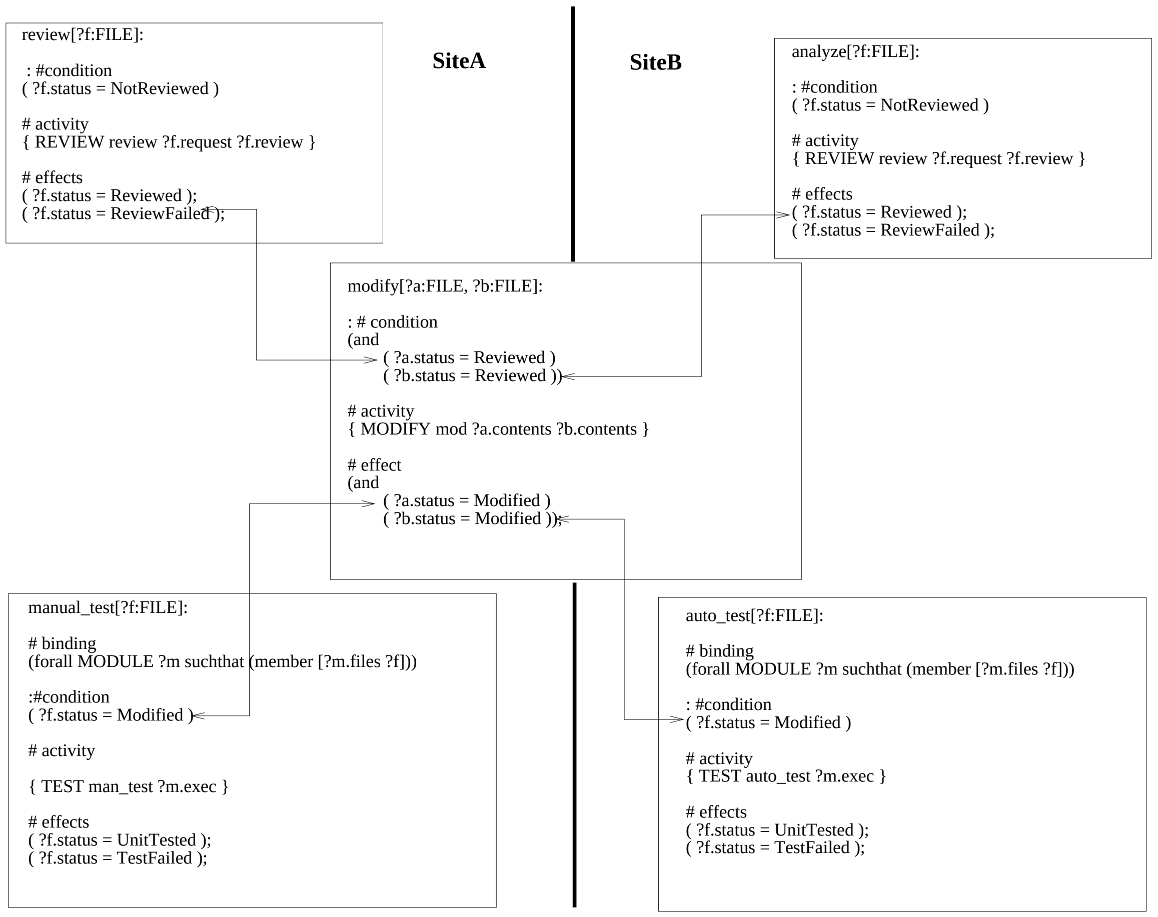
1. *Select* — Since remote strategies are not normally visible to SubEnvs, the *import* interface must supply the administrator at *DstSubEnv* with a list of the available strategies at *SrcSubEnv* that were explicitly exported from it to *DstSubEnv*. Further, this information must be generated dynamically, since the list of exported strategies at *SrcSubEnv* can change at any time as a result of issuing local *export* or *unexport* operations.

2. *Copy* — Once the importer at *DstSubEnv* selects the strategy to import, the strategy is copied from *SrcSubEnv* along with additional information needed for runtime validation (e.g., timestamp). The source-code of the strategy is used only during the integration phase, however, and cannot be manipulated by administrators at *DstSubEnv*, to ease dynamic verification of Treaties.

Note that *import* fetches only the rules, without the tools and their envelopes (i.e., the wrapping mechanism used to integrate tools into Oz, see [20]). While this is not a problem with the default *import-accept* option (where 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 either the activity and its associated tools already exist at the importing SubEnv, or they can be copied explicitly. If this is not the case (e.g., a tool is bound to a specific machine and cannot be copied), then this combination should not be used.

3. *Integrate* — This is the main step. First, the imported strategy is parsed and checked to be schema-compatible with the local process. Next, the rules in the parsed strategy are integrated with the rule network, 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. 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 Summit rules would be automatically enacted.

Figure 7 illustrates the integration phase. Suppose the `modify` rule is imported by two different sites, `SiteA` and `SiteB`. In `siteA`, `modify` is backward connected to rule `review` through



```
review[?f:FILE]:

 : #condition
( ?f.status = NotReviewed )

# activity
{ REVIEW review ?f.request ?f.review }

# effects
( ?f.status = Reviewed );
( ?f.status = ReviewFailed );
```

```
analyze[?f:FILE]:

: #condition
( ?f.status = NotReviewed )

# activity
{ REVIEW review ?f.request ?f.review }

# effects
( ?f.status = Reviewed );
( ?f.status = ReviewFailed );
```

```
modify[?a:FILE, ?b:FILE]:

: # condition
(and
     ( ?a.status = Reviewed )
     ( ?b.status = Reviewed ))

# activity
{ MODIFY mod ?a.contents ?b.contents }

# effect
(and
     ( ?a.status = Modified )
     ( ?b.status = Modified ));
```

```
manual_test[?f:FILE]:

# binding
(forall MODULE ?m suchthat (member [?m.files ?f]))

:#condition
( ?f.status = Modified )

# activity

{ TEST man_test ?m.exec }

# effects
( ?f.status = UnitTested );
( ?f.status = TestFailed );
```

```
auto_test[?f:FILE]:

# binding
(forall MODULE ?m suchthat (member [?m.files ?f]))

: #condition
( ?f.status = Modified )

# activity
{ TEST auto_test ?m.exec }

# effects
( ?f.status = UnitTested );
( ?f.status = TestFailed );
```

Figure 7: Integration of Imported Rules

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. 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 execution. Notice that in general an imported rule may connect to zero, one or more local or other imported rules.

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 context-less rules, as well as the fine granularity of rules as process building blocks, also pay off handsomely.

Due to the coupling of *import*/*export* with *accept*/*request* in Oz, it is necessary to make *import* idempotent with respect to the compilation mentioned above, and to allow a SubEnv to export an imported strategy. This is particularly important for multi-site Treaties. 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 execution-privileges are 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 refer to such operation as a "faked" *import*.

4. *Acknowledge* — An acknowledgment is sent to *SrcSubEnv*. This acknowledgment is not critical, however, since Treaties are verified at runtime. Its sole purpose is to notify users at *SrcSubEnv* of the new Treaties that are available to them.

There are two more properties that the *import* operation must possess. 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, since the acknowledgment phase is optional, there is no need to guarantee cross-site atomicity for *import*. The atomicity of the operation has to be preserved only in the importing server. This fits well with the general decentralized requirements.

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

3.2.4 unimport

The *unimport* operation is defined as:

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

Unlike its *import* counterpart, *unimport* is a local operation. 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 site 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 an *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 main point is that the equivalent semantics of the formal model are fully obtainable in Oz.

3.2.5 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 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 to 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 as part of a multi-user global environment, in which case a global administrator (and a corresponding global Treaty operation) was essential. Therefore, Oz supports the explicit *Treaty* operation, which bundles *export* and *import*, as explained below.

In order to be eligible for executing a 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:

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

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.

3.3 Summit in Oz

Summits are the main means by which multiple SubEnvs actually interoperate, and as such, they encompass all the support that is required to enable execution of multi-process "Treatified" 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.

3.3.1 Summit Initialization and Treaty Verification

A Summit task is initiated 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. The first action taken by the coordinating server is to fetch copies of the remote objects from their origin SubEnvs, and bind them to the parameters of the rule (in addition to the obvious binding of local objects, but as we focus here on inter-site issues we will ignore from now purely local aspects).

The second step involves Treaty verification. 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 from figure 3.

The reader might wonder 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. 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 [26]. Thus, 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.

3.3.2 Pre-Summit

The coordinating server evaluates the rule's condition. If the condition is not satisfied, the server fans out to the participating sites and triggers 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.

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, since the chained-to rules are regular rules that may contain arbitrary activities. 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 (see [50]). 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 topic is beyond the scope of this paper, see [7].

3.3.3 Summit Activity

If the condition of the rule is satisfied, the multi-site activity is fired at the coordinating site. Since typical Oz activities involve (possibly large) files — as opposed to pre- and post-phases which access "light" process state information — multi-site activities require sites which are physically remote to transfer the remote files to the coordinating server.

Another issue regarding multi-site activities is the association with users. In case of a single-user activity, Oz associates the activity with the user whose on behalf the Summit rule was invoked, or to a delegated user if it was specified in the rule. In case of a multi-user groupware activity (e.g., virtual whiteboard), Oz provides mechanisms to define the participants and bind the activity to them at run time, see [10].

3.3.4 Post-Summit

The first step in Post-Summit asserts the appropriate (local and remote) effects of the Summit rule, depending on the output from the activity. Since the executed rule is identical at all participating sites (because of the common sub-process 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 objects (which are remote to the coordinating server and local to each remote server), or the coordinating server itself asserts the effects on the replicated objects and sends the updates to the remote servers. 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 for object cache management. Therefore, Oz employs the latter approach. In order to enable forward local chaining, the coordinating server sends, in addition to the object updates, a pointer to the asserted effect, and each of the remote servers acts as if it had asserted the effects locally to explore forward chaining possibilities.

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.

3.3.5 Inference of Summit Rules

There are several approaches to modeling and enacting multi-step Summit rules. Technically, the coordinating server must distinguish chains which are part of the local fan-out from those which are "global" Summit rules. One alternative is to add "Summit" directives, similar to 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 mechanism for dynamic binding of parameters [26] to handle binding of both local and remote objects to chained rules, the basic rule-inference mechanism can infer Summit rules — these are simply the rules that happen to have been instantiated with (some) remote objects as parameters. Thus, the inference of Summit rules has been extended to operate in the same manner as local inference is done. However, unlike normal rules, when Summit rules are inferred they are enqueued in a separate Summit queue and are scheduled for execution only after local forward chaining has completed in all sites that are part of that Summit (see below).

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.

3.3.6 Summit Completion

Once local forward chaining completes in all involved SubEnvs, they notify the coordinating server, which in turn checks if there are any rules in the Summit queue. If there are none, it completes the task and releases resources that were allocated for the Summit (e.g., transaction locks, which are beyond the scope of this paper, see [7]). If there are pending Summit rules, the coordinating SubEnv reiterates to the Summit initialization phase, except it bypasses the manual parameter binding phase which is (automatically) performed by the extended parameter binding mechanism. 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.

4 Application of the Model to Other PMLs

We now outline how the interoperability model may be applied to two other families of PSEEs categorized by the paradigm underlying their PMLs, namely Petri nets and Grammars (application to imperative process programming such as APPL/A can be found in [7]). These families, together with rules, cover most kinds of PSEE [33]. 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.

4.1 Petri Nets

The Petri net [45] is a 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 PSEEs is influenced primarily by SLANG [2] and FUNSOFT [21], and their corresponding PSEEs, 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 use for the most part general Petri net terminology.

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, and *Tokens* represent the current state of the process under enactment and the product data used in the activities (i.e., the actual parameters).

A *predicate* 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. A transition is said to be enabled when its input places contain the sufficient quota of tokens and the predicate(s) on the transition is satisfied.

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 in general altering the input or output places of a transition requires to modify the transition itself (analogous to changing the number or types of the formal parameters to a function in a conventional programming language). Also, the predicate is a local constraint on the transition and therefore conceptually part of it.

Integration of a process step into an existing net as part of the *import* operation 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. This in turn might imply further modifications in the neighboring transitions to accommodate the changes in input-output places. 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 there are no pre- and post-Summit actions during its enactment.

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 PSEE):

Summit initialization — The coordinating SubEnv binds the data arguments to its input places, and 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 be restricted 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.

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 execution 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 required for executing the Treaty subnet.

4.1.1 An Example

The following example, depicted in Figure 8, 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 [3] describing SLANG.

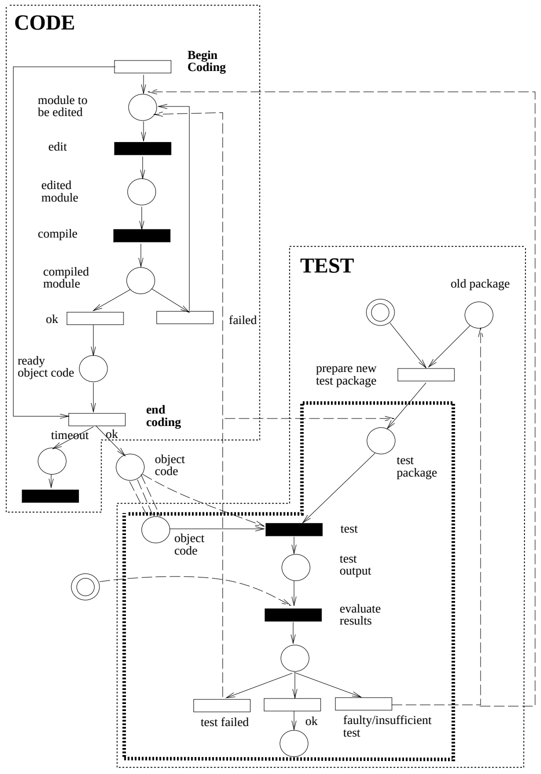


Figure 8: Example Multi-Process Petri-net

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 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 and 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 into 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 (i.e., the Summit) 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.

4.2 Grammar-Based PMLs

The grammar hierarchy [13] and the corresponding automata provide another powerful set of formalisms 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 modeling, analogous to sentence generation at one end (what Heimbigner calls a prescriptive process [22]) to sentence recognition (parsing) at the other (proscriptive) [29]. The PDL project employed the former for context-free grammars [28], while the implementation of the Activity Structures Language on top of Marvel follows the latter approach [31]. One group experimented with both in the context of attribute grammars for HFSP [32] and Objective Attribute Grammars [47], 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 simply rule conditions, and in HFSP they are decomposition conditions. Symbols are associated with formal and actual parameters in some fashion specific to the PML and PSEE. 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 possibly be generated during 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, (which is the one actually taken in Oz to address naming of rules) 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 (skip the first and last phases):

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 to enacting that symbol are checked in each of the participating SubEnvs, each according to their own local process. In principle, a recognition-oriented PSEE 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 PSEEs. This could be regarded as a form of sentence generation.

Summit — Assuming all SubEnvs ultimately agree, the symbol is enacted in the coordinating SubEnv. If, however, this is a non-terminal symbol representing a 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.

Post-Summit — All the participating SubEnvs are notified by the coordinator to complete the symbol. For example, in the case of a generation-oriented PSEE, 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.

5 Experience and Evaluation

We now discuss how the interoperability model fulfills the requirements set forth in Section 1.1. We base our evaluation mainly on our experience in using Emerald City, an Oz environment that has been used to develop the Amber [1] rule processor, Pern [25] transaction manager, and the Darkover [35] object management system. Most importantly, Emerald City was used for the re-engineering of the Oz kernel itself to be constructed by these components[4].

[4]The latest version of Oz , 1.1.1, already uses Darkover and Pern as its components, and the integration of the Amber process-server is in progress.

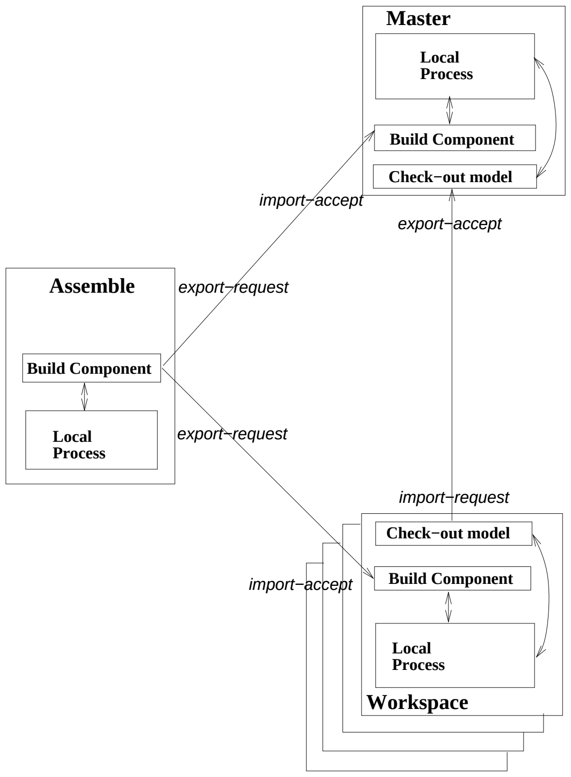


Figure 9: The Emerald City Environment

5.1 The Emerald City Environment

Emerald City consists of three types of processes: a "Master" process that is used to maintain stable versions of components as well as additional "glue" modules that together comprise Oz; an intermediate "Assembly" process used for system (re)-engineering from components; and a "Workspace" process for individual intra-component development. Although processes can be in general instantiated for a variety of projects, Emerald City was tailored specifically to support the complex development and re-engineering tasks of Oz, so we will not distinguish from now between the processes and the actual environment. Note, however, that many local (i.e., non-Treaty) rules have been reused from earlier Oz and Marvel environments, particularly from OzMarvel, the Marvel environment which was used for the production of the earlier Oz 1.0.

Figure 9 shows the site interconnections in Emerald City. It comprises of a single Master SubEnv, a single Assembly SubEnv and multiple Workspace SubEnvs. The Workspace SubEnvs are mostly similar but not identical to each other, and unlike the multi-user Master and Assembly SubEnvs, they are mostly single-user although nothing prevents them from being used by multiple users (as some have). The Master SubEnv interoperates with Workspace SubEnvs via the `check-out-model` strategy that contains various cross-site rules for reserving and depositing artifacts across the sites, and for updating local information as a result of changes in other SubEnvs. A sample Treaty rule for updating function interfaces is listed in Appendix A. The rules in `check-out-model` are executable from Workspace, and have originated at Master, as indicated in the figure by the *request* and *export* labels, respectively.

Assembly maintains a three-way Treaty on the `build-component` strategy with the Master SubEnv and with each Workspace SubEnv that is involved in the re-engineering effort. This is a *simple* Treaty from Assembly to Master and Workspace, i.e., only the Assembly SubEnv can execute rules from that Treaty on data from Master and Assembly. A representative rule from `build-component` is listed in Figure 10.

This rule takes 3 arguments, one from Master, one form Assembly, and one from a Workspace SubEnv (line 1). It then binds the proper subsystem object from Assembly (line 5), the executable from the local workspace SubEnv (line 8), the local repository of header files (line 10), the main function from the local project (if exists) or from the master project (lines 12 - 15), and source files from the local and the master projects (lines 17 - 18). The condition (lines 20-24) states that all source files have been compiled and do not require recompilation (e.g., due to changes made to external function prototypes). Notice that this rule may backward-chain to a local `compile` rule vie the predicate in line 22. The activity (lines 25-28) invokes a builder tool with objects from all three sites. Finally, the effects (lines 29-31) indicate whether the build activity was successful (first effect) or not (second effect).

The Workspace SubEnv is where most of the development is done, where each developer tailors his/her own rules and tools to suit his/her needs. Thus, Emerald City is prescriptive and allows freedom in the creative aspects of programming (carried out in Workspaces) while providing automatic utilities and proscription for the complex and mechanical aspects of connecting the individual pieces together.

Emerald City has been in use since April 1995 and is constantly evolving. In its present configuration it consists of 16 SubEnvs: 1 Master, 1 Assembly, and 14 Workspace SubEnvs. The Master process consists of 34 local rules (in addition to the 15 standard rules used for site configuration [8] and for built-in operations, e.g., `copy object`) and 21 Treaty rules. Figure 11 shows a snapshot of user `kaiser` working in the Master SubEnv and listing the rules. The number of rules in the menu is smaller than the total number of rules because some of the rules are "hidden", i.e., they are intended to be fired only through chaining and not explicitly invoked by users, and other rules are overloaded, e.g., there are 6 different `reserve` rules for the various types of objects reserved, and for different types of destination SubEnvs.

The Assembly process has 27 local process-specific rules and 2 Treaty rules. A typical Workspace process contains 22 new rules, in addition to the imported Treaty strategies from Assembly and Master. Thus, 19% of the total distinct rules in Emerald City are Treaty rules. This figure, which can be used as a (static) measure of the level of site-interoperability, seems to be typical for Oz environments; in another experimental environment that implemented the ISPW9 "benchmark scenario" [44], this interoperability measure was 15% (see [7]). Another (dynamic) measure of site-interoperability is the percentage of actual invocations of Treaty rules in Summits from the total invocations. Table 5.1 summarizes the runtime statistics made for 11 active project members (taken from execution log files generated by Oz). The `built-in` column includes operations such as printing an object and browsing

```
# ----------------------------------------------------------------
# Build a system using a local main function
#       ?lp is the local (workspace) project object
#       ?ap is the assembly project object
#       ?mp is the master project object, where code needed by a
#       variety of workspace SubEnvs is stored.
# ----------------------------------------------------------------
build-system [?lp:LOCAL_PROJECT, ?ap:PROJECT, ?mp:PROJECT]:
   # RULE BINDINGS:
  (and
    # find the proper subsystem in re-project to link to.
    (exists SUBSYSTEM ?s suchthat (and (ancestor [?ap ?s])
                                       (?s.Name = ?lp.subsystem)))
    # find the local binary
   (exists BIN   ?lb    suchthat (member  [?lp.bin  ?lb]))
   # find the repository of local header files
   (forall INC   ?i     suchthat (member  [?lp.inc  ?i]))
   # Find Main file, if it exists, in the local project or Master
   (exists CFILE ?main suchthat (or (and (linkto   [?lp.main  ?main])
                                         (ancestor [?lp ?main]))
                                    (and (linkto   [?mp.main  ?main])
                                         (ancestor [?mp ?main]))))
   # bind all source files in the local project
   (forall COMPILABLE ?c suchthat (or (member   [?lp.files ?c])
                                      (member   [?mp.files ?c]))))
  :
  # RULE CONDITION: all source files are compiled and are not marked
  #                 for recompilation
  (and no_forward (?c.compile_status = Compiled)
       no_chain   (?i.recompile_mod = false)
       no_forward (?main.compile_status = Compiled))

  # RULE ACTIVITY: build an executable with objects from all sites
  { COMBINE_TOOLS build_local_main ?lb.executable ?lp.build_log
                 ?s.libraries ?s.build_order ?s.med_libraries
                 ?main.object_code ?c.object_code }

  # RULE EFFECTS:
  (?lb.build_status = Built);      # build succeeded
  (?lb.build_status = NotBuilt);   # build failed
```

Figure 10: Three-site Build

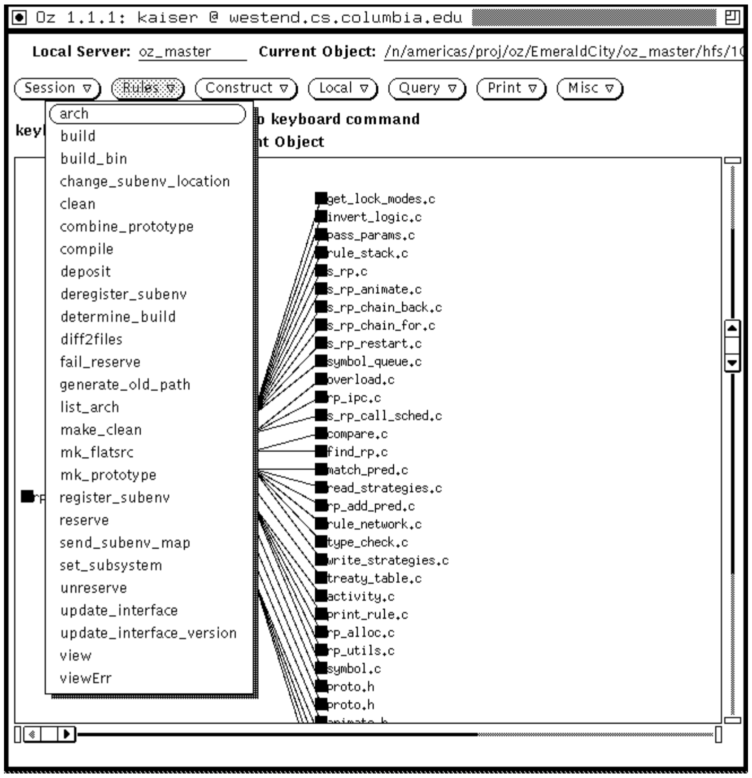


Figure 11: A Snapshot from the Master SubEnv

the hierarchy, and are in general not part of a specific process. The `treaty` column lists the "meta" administrator commands to establish/update/remove treaties. They were mostly issued by three administrators of Master and Assembly (other users have issued such commands sparsely, to connect their Workspace to other SubEnvs). The `summit` and `local` columns list invocations of Summit and local rules, respectively, and the `int. measure` lists the dynamic interoperability measure, i.e., the percentage of Summit rules from the sum of Summit and local invocations. We can see that the overall dynamic interoperability measure is 22%, meaning that approximately 80% of the development efforts were local, an overall positive result.

5.2 Evaluation

5.2.1 Autonomy

Throughout the paper 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. 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

users	built-in	treaty	summit	local	Total	int. measure
1 admin	17141	661	2096	4908	24806	.30
2	2116	0	450	1145	3711	.28
3 admin	7707	134	589	1591	10021	.27
4	2812	29	735	1942	5518	.27
5	4661	23	330	1271	6285	.21
6	1250	0	110	446	1806	.20
7	631	13	133	519	1296	.20
8 admin	7049	198	678	3528	11453	.16
9	3968	1	294	1592	5855	.15
10	360	0	12	120	492	.09
11	11427	4	107	1782	13320	.06
Total:	59122	1063	5534	18844	84563	.22

Table 1: Summary of usage in Emerald City

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 generic 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 design of Oz (as opposed to the generic model). The most important one concerns the global configuration mechanism and the global objectbase browsing facility, both of which cannot be "turned-off" and thus they violate autonomy. This was evidenced in Emerald City, where individuals working in their workspaces sites did not want to provide *any* access to other workspaces. To overcome this problem, the configuration mechanism has been modified to allow for partial visibility of remote sites which is determined autonomously, but a more general solution is needed.

5.2.2 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 work in an equivalent single instance running under the Marvel single-site PSEE is negligible, because the infrastructure overhead impacts only interoperability.

5.2.3 Interoperability

Given that autonomy was a crucial requirement, this "competing" requirement seems to also have been adequately addressed. The Treaty abstraction appears to support particularly well interoperability modeling of process and data. Two areas that still need improvements are in modeling interoperability at the user and the tool levels, which are related to groupware technology. Preliminary work has been done in [10].

Work in Emerald City revealed another area that requires improvements in Oz, namely better support for multi-site operations between trusted sites, particularly for interoperability modeling. The Treaty operation as a single command (with the issuer being administrator in both sites) was a step in that direction. Other improvements include commands for defining multi-site Treaties, more selective Treaty invalidation procedures that do not invalidate Treaties unnecessarily, and automatic updates of strategies without requiring to re-establish Treaties. Finally, work in Emerald City showed that establishing cross-site links at the data level is important for facilitating multi-site activities, although it may violate autonomy.

5.2.4 Support for Pre-existing and Heterogeneous Processes

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. This is particularly true for the declarative rule-based PML. For other PMLs, however, the addition of a new-subprocess may require more work and special tools. Another problematic issue with supporting pre-existing processes is with their *schemas*, particularly in 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 [15]). Alternatively, PMLs might need to sacrifice some of their typing restrictions, at least for Summit activities, to accommodate heterogeneous schemas, and to be able to check for schema (sub)compatibility.

5.2.5 Scaleability

The Treaty/Summit model scales up mainly because it does not assume any global authority or centralized control. However, it does not provide means to form hierarchies over a set of interoperating sites, and they are all treated flatly as peers. This might have a negative impact on scaleability, particularly for top-down oriented environments. For example, in Emerald City it may have been advantageous to define a hierarchy of workspace SubEnvs with individual student's workspaces below "component" workspaces.

5.2.6 Language vs. System approaches to Treaty Definition

We already discussed in Section 2.2.2 some advantages of using the system-based approach. One disadvantage of this approach is that it is impossible to define a "Treaty" program with site classes as formal parameters. In Emerald City, for example, this capability would have allowed to automatically form a Treaty upon instantiation of a new workspace site in Emerald City. Instead, it was necessary to form a Treaty manually between each new workspace SubEnv and the other sites. Another disadvantage of the system-based approach was that it was necessary to create a set of built-in system calls not only for creating Treaties but also for removing, listing, and updating them. This suggests language constructs for Treaties in conjunction with system calls that are called from them, as an improved approach.

6 Related Work

ISTAR [14], one of the earliest software engineering environments (or "Integrated Project Support Environments"), provided comprehensive support to the software development life-cycle, including both management and software engineering. The main idea in ISTAR was the *contractual* approach, in which a "contractor" (e.g., a group of programmers) provides services to a client (e.g., a 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 PSEE (it had a somewhat hard-coded process), its architecture was an important step towards decentralization.

Shy, Taylor, and Osterweil were among the first to explicitly identify decentralization as a key environment technology [48]. Their theoretical work draws an analogy between software development and the business corporation, and they advocate a "federated decentralization" model for PSEEs 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 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; and (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 [23] 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 [24] is an attempt to address heterogeneity at the language level. The main idea in the ProcessWall 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 centralized.

ProcessWEAVER 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" [16]. 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 can 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 [17] 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 [40], process execution is performed by HP's SynerVision, and Matisse [18] (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 [6] is another PSEE 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 sub-processes 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 2 [5], an active configuration 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.

7 Conclusions and Future Work

Two key concerns guided this research: (1) maximizing local autonomy, both physically and logically, so as not to force a priori any global or inter-site constraints 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.

The high-level 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*. In essence, a Treaty is an abstraction that specifies 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: 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 each Treaty is formed over a single and a small sub-process unit. Still, complex Treaties can be formed (and subsequently executed) between any number of sites and involve arbitrarily large sub-processes, by successive invocations of simple Treaties (which could be optimized from the user interface perspective). 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 (combined with dynamic Treaty verification), on demand.

Inter-process execution was achieved by the complementary *Summit* model. Summits are the execution abstraction for Treaty-defined sub-processes. They support multi-site enactment 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, tools, and/or users from multiple sites, and the latter is used for the autonomous execution of any private subtasks emanating from prerequisites and consequences of the shared activities.

7.1 Future Work

The first issue to further explore is extensions of the basic Treaty/Summit model with more abstractions that support alternative modes of interoperability, both in modeling and in execution. For example, enabling to model and enact local activities that execute simultaneously. Another extension concerns enhanced groupware modeling facilities for tools and users. Finally, support for site hierarchy should be explored.

Addressing heterogeneity and interoperability at the PSEE and PML levels in conjunction with the process-interoperability model described in this paper, are other important avenues to explore.

Finally, 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 software 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.

Acknowledgments

We would like to thank the members of the Programming Systems Laboratory at Columbia university and particularly George Heineman and Peter Skopp who contributed significantly to this research and to the development of Oz. We also thank Lee Osterweil and K. Narayanaswamy for insightful comments on earlier versions of the interoperability model. Finally, we thank Karen Huff for discovering a bug in an earlier version of the Petri net example.

References

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

[2] 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.

[3] 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.

[4] Naser S. Barghouti. Supporting cooperation in the MARVEL process-centered SDE. In Herbert Weber, editor, *5th ACM SIGSOFT Symposium on Software Development Environments*, pages

21–31, Tyson's Corner VA, December 1992. Special issue of *Software Engineering Notes*, 17(5), December 1992.

[5] 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 Press.

[6] 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.

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

[8] 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.

[9] 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.

[10] Israel Z. Ben-Shaul and Gail E. Kaiser. Process support for synchronous groupware activities. Technical Report CUCS-002-95, Columbia University Department of Computer Science, January 1995.

[11] 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.

[12] 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.

[13] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2), 1959.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] Dimitrios Georgakopoulos, Mark Hornick, and Amit Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3:119–153, 1995.

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

[21] 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.

[22] 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.

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

[24] 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.

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

[26] 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.

[27] SynerVision for SoftBench: A Process Engine for Teams, 1992. Marketing literature.

[28] 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.

[29] Gail E. Kaiser, Israel Z. Ben-Shaul, and Steven S. Popovich. Implementing activity structures process modeling on top of the MARVEL environment kernel. Technical Report CUCS-027-91, Columbia University, September 1991.

[30] 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.

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

[32] 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.

[33] 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.

[34] 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.

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

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

[37] Ted G. Lewis. Where is client/server software headed? *Computer*, 28(4):49–55, April 1995.

[38] Jeff Magee, Naranker Dulay, and Jeff Kramer. Structuring parallel and distributed progrmas. *Software Engineering Journal*, 8(2):73–82, March 1993.

[39] Jeff Magee, Jeff Kramer, and Morris Sloman. Constructing distributed systems in Conic. *IEEE Transactions on Software Engineering*, 15(6):663–675, June 1989.

[40] 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.

[41] Peiwei Mi and Walt Scacchi. Process integration in CASE environments. *IEEE Software*, 9(2):45–53, March 1992.

[42] Naftaly H. Minsky. Law-governed systems. *Software Engineering Journal*, 6(5):285–302, September 1991.

[43] John R. Nicol, C. Thomas Wilkes, and Frank A. Manola. Object orientation in heterogeneous distributed computing systems. *Computer*, 26(6):57–67, June 1993.

[44] Maria H. Penedo. Life-cycle (sub) process scenario. In Carlo Ghezzi, editor, *9th International Software Process Workshop*, pages 141–143, Airlie VA, October 1994. IEEE Computer Society Press.

[45] James L. Peterson. *Petri Net Theory and The Modeling of Systems.* Prentice-Hall, Englewood Cliffs NJ, 1981.

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

[47] 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.

[48] 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.

[49] 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.

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

[51] 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: A Sample Treaty Rule from Emerald City

```
# ---------------------------------------------------------------
# A Treaty rule for Updating interfaces for Workspace Project
# (LOCAL_PROJECT) based on changes in the Master Project (PROJECT)
# ---------------------------------------------------------------

update_interface[?lp:LOCAL_PROJECT, ?p:PROJECT]:
  (and (exists SUBSYSTEM ?s  suchthat (and (ancestor [?p ?s])
                                          (?s.Name = ?lp.subsystem)))

       # Find local interface
       # --------------------
       (forall PROTOTYPE ?LPT suchthat (member   [?lp.proto    ?LPT]))
       (exists INC       ?i   suchthat (member   [?lp.inc      ?i]))
       (forall HFILE     ?h   suchthat (member   [?i.hfiles    ?h]))
       # Find master interfaces
       # ----------------------
       (forall PROTOTYPE ?PT suchthat (member   [?p.proto ?PT]))
       (exists INC       ?ii suchthat (member   [?lp.interface ?ii]))
       (forall COMPONENT ?CD suchthat (ancestor [?s           ?CD]))
       (forall LIB       ?l  suchthat (linkto   [?CD.lib      ?l]))
```

```
    (forall MODULE    ?m  suchthat (linkto   [?m.library    ?l]))
    (forall SRC       ?sr suchthat (member   [?sr.libs      ?l]))
    (forall INC       ?ri suchthat
                              (or (member [?sr.incs        ?ri])
                              (linkto [?m.related_incs ?ri])
                              (member [?p.common_incs  ?ri]))))
:

{ TREATY_TOOLS install_interface ?i ?h.contents ?ii ?ri.directory
               ?sr.sys_includes ?p.tags ?lp.tags
               ?PT.contents ?LPT.contents
               return ?i_path ?ii_path ?combine }

(and no_chain (?lp.interface_version = 0)
              (?lp.interface_version = ?p.interface_version)

    no_chain (?lp.sys_includes = ?combine)
    no_chain (?i.directory = ?i_path)
    no_chain (?i.recompile_mod = false)

    no_chain (?ii.directory = ?ii_path)
    no_chain (?ii.recompile_mod = false));
```