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

Israel Z. Ben-Shaul   Gail E. Kaiser
Department of Computer Science
Columbia University
New York, NY 10027

Abstract

We present a model for decentralized *Process Centered Environments (PCEs), which support concerted efforts among geographically-dispersed teams — each local team with its own autonomous process — and emphasize flexibility in the tradeoff between collaboration vs. autonomy. We consider both decentralized process* modeling *and decentralized process* enaction. *We describe a realization in the* Oz *decentralized PCE, which employs a rule-based formalism, and also investigate the application to PCEs based on Petri-nets.*

1 Introduction and Motivation

A *process* is a partially ordered set of steps for developing a software system. In addition to defining the steps and their interfaces to tools in some formal notation, a *process model* also specifies prerequisites to each step, the consequences of each step, and any synchronization among concurrent steps. Processes are "situated", in the sense that different processes are suitable for different personnel roles, lifecycle phases, projects, organizations, etc.

Process Centered Environments (PCEs) emerged in an attempt to provide flexible and extensible environments for developing software, by means of "process modeling" and "process enaction". PCEs (1) provide a formalism for defining project-specific software processes (modeling) and (2) take advantage of well-understood programming language implementation techniques to "execute" the software process (enaction). Process enaction assists in performing the software process as defined in the process modeling language (PML). Enaction can involve monitoring, guidance, automation, enforcement, constraining and controlling the workflow, and simulating parts or all of the software process. A specific instance of a PCE is constructed by tailoring a generic kernel that behaves as the enaction engine for the PML; this is typically carried out by a distinguished environment *administrator*, and is of no concern to the average environment *user* who participates in the process.

Why Decentralization?

Large-scale product development typically requires participation of multiple people, often divided into multiple groups, each concerned with a different facet of the product. For example, one software product may require teams for requirements elucidation, functional specification, design, coding, testing, documentation, and maintenance; another product may involve multiple teams, in this case with each responsible for full development of a distinct component of the system. Each team uses its own selection of tools, and possibly its own private data and management policies. At the same time, the teams need to cooperate in order to develop the product, and as studies in software engineering have shown, the interaction among team members accounts for a significant fraction of the total cost of the product being developed [12].

The degrees of *autonomy* and *collaboration* both depend on the nature of the product being developed and on organizational policies. Sometimes multiple organizations collaborate on a product, in which case autonomy (privacy or security) is a "hard" constraint that cannot be compromised. With the advent of high-speed networks and enhanced communication facilities, geographical dispersion and time shifting is an additional possibility, particularly among teams (as opposed to within a team).

This paper explores *modeling and enaction of inter-group collaboration among independent, autonomous and, possibly, pre-existing processes.* This is in con-

0270-5257/94 $3.00 © 1994 IEEE

trast to work on (1) supporting intra-group collaboration and synchronization, where multiple team-members (co)operate within the same process, possibly with different "views" [7, 3]; and (2) process modularization, where a single process is decomposed into sub-processes both for modeling and enaction purposes [21]. We further distinguish between distribution and decentralization. A *distributed* system is one that provides a single logical perspective, but is physically distributed into multiple computing units, usually across machines of a single site. (We use the term "site" to mean an administratively cohesive Internet domain sharing a network file system.) That is, a distributed system transparently shields the distribution from its applications. In contrast, a *decentralized* system is comprised of relatively *independent* subsystems with some degree of correlation between them, often (although not necessarily) spread among multiple sites. Here, transparency is intentionally not supported because it violates autonomy and because the range of access costs must be distinguished.

Environment distribution is a form of "vertical" scale-up, in that it allows for more users to work, but under the same process and within some bounded physical distance (typically a local-area network). This paper explores "horizontal" scale-up, where the number of users per group sharing the same process may not grow much, but the number of *groups* may be arbitrarily large, each group with its own private process and data but collaborating in a concerted effort with the other groups.

An "international alliance" metaphor for our model of DEcentralized PCEs (henceforth DEPCEs) is used throughout the paper, where the default is for independent countries to operate autonomously and co-operate only in accordance with treaties (in analogy to NATO). This is in contrast to the "corporation" model of decentralization suggested by Shy *et al.* [23], whereby subunits of a corporation do not have any existence independent of the corporation.

2 Requirements

The following is a list of the main requirements that a DEPCE should fulfill (for centralized multi-user PCE requirements, see [7]):

1. *Process Autonomy* — Each local subenvironment (henceforth *SubEnv*) should control autonomously its process and its data, while allowing access to them by remote SubEnvs under restrictions that are solely determined by the local SubEnv.

A related requirement is that a SubEnv should be *self-contained* and *operationally independent*. That is, it 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.

2. *Process Collaboration and Interoperability* — The main goal is to allow multiple groups of users, each group with its own process and schema, to execute operations that might affect the state and the product of remote processes. Interoperability is in general a hard problem, but is particularly difficult in this case, due to the heavy semantics associated with a "process".

We address interoperability in the sense of *multiple independent processes* but a *single PML* in which they are defined, as well as multiple schemas but a single data definition language. This is in contrast to ProcessWall [15], where *multiple formalisms* can be used to define a *single process*, or to the Activity Structures Language [19], where high and low level modeling formalisms are combined into one.

3. *Data Sharing and Presentation* — A DEPCE should provide non-transparent, but nevertheless efficient and highly available data access capabilities. In particular, SubEnvs should be able to access and browse through data residing at remote sites.

4. *Communication Modeling* — A DEPCE should support a range of geographical distances and different bandwidths between teams and within a team, ranging from local-area networks to long haul networks. This implies the necessity of flexible communication protocols between the various entities, according to a distance/bandwidth metric.

5. *Dynamic Reconfiguration* — A DEPCE should have the capability to dynamically add, delete, and move (among sites) inactive SubEnvs without disrupting the operation of the currently active SubEnvs.

A related issue is to enable a single-site environment with a pre-existing process to "join" a global environment with other pre-existing process(es), with minimal (re)configuration overhead. Similarly, a "split" of a SubEnv from its current global environment should be supported. This requirement is important when two or more organizations with established processes need to collaborate for a limited time.

6. *Decentralized Concurrency Control and Recovery* — Software development environments in general and PCEs in particular require sophisticated and flexible concurrency control (CC) mechanisms and policies [2]. DEPCEs add the dimension of remote vs.

local access. This complicates CC in that extended transaction models devised for centralized and even distributed systems are not adequate. For example, if semantics-based CC is employed, then different processes impact their local CC policy differently, requiring some sort of negotiation between local CC engines.

7. *Decentralized Query Processing* — A DEPCE must provide for and, preferably, optimize multi-site global queries on both process and product data.

This paper focuses on the first two requirements. Requirements 3 and 4 are mostly architectural, and are discussed in depth in [4]. Our approach to fulfilling requirement 5 is described separately in [6]. Requirements 6 and 7 are outside the scope of this work.

A Motivating Example

Assume there are three development teams using three SubEnvs **SE1**, **SE2**, and **SE3**, who are responsible for three disjoint modules of a system **S**, labeled **M1**, **M2**, and **M3**. The teams reside in different geographical areas, and they each use their own process and tools for development of their modules. Each module can be coded and unit-tested independently, but there is a library **L** shared by all groups. The following actions should be taken when a change request for **L** is made in **SE2**: (1) the change must be `pre-approved` (or reviewed) by all SubEnvs in advance; (2) the `change` is actually made, generating **L'**, and is propagated to all SubEnvs; (3) a static testing phase involving `code inspection` is performed; (4) a `unit-test` of each module linked with **L'** is performed; (5) an `integration-test` of all modules combined is performed. For simplicity, only the "successful" path is described.

Both `pre-approve` and `unit-test`, which are local operations, can differ at different sites. For example, one site might employ white box testing, while another site might use black box testing. Moreover, even an identical operations might imply execution of different related operations when issued in different sites.

A *wrong* and in some cases impossible solution would be to collect all the necessary data from the remote sites to the site that initiated the change, and then carry out all the process steps on all modules (and all their prerequisites and consequences) as defined in the local process. Besides the obvious performance limitations, this approach would be a clear violation of autonomy, since each site has its *own* process for unit-testing its module, which may not even be known to the initiating site. Another possible problem might be that some of the tools do not exist in all SubEnvs (e.g., due to licensing that binds a tool to a specific site or host), and other tools can be executed only in specific SubEnvs (e.g., special-purpose hardware).

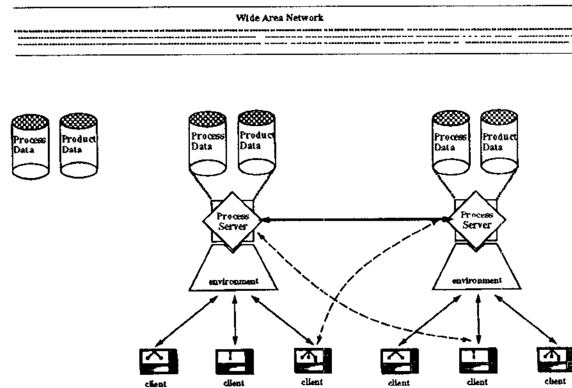

Figure 1: A Decentralized Environment

3 The Decentralized Model

The external view of a generic decentralized environment is depicted in figure 1. SubEnvs of the same or different global environment may reside arbitrarily on the same or different sites, but the discussion assumes exactly one SubEnv per site.

Each *active* SubEnv corresponds to a server that enacts its *local process*. The server also manages a private database for at least the process data for this process, and possibly the corresponding product data (the latter could alternatively be maintained in a separate database or in the native file system). The data for each SubEnv is said to be "owned" by its local process, and updates to both process and product data cannot be made from a remote process without prior approval of the owner process.

A *client* user interface connects each user to the local server, which by definition operates at the same site (but need not execute on the same host). A distributed communication layer enables connections among SubEnvs, both clients and servers.

We define a three-level hierarchy of nested contexts within a local process:

The *activity* level is where the PCE interfaces to actual tools. A *decentralized activity* is one that can operate on remote as well as local data. The activity itself is not decentralized in execution, that is, an individual activity executes on a single host. This definition excludes tools that are themselves multi-user (e.g., Flecse [10]). However, our decentralized model can be extended to include such tools, provided that

the PML and the corresponding PCE have mechanisms to describe the concurrent invocations by multiple participants in the process.

The *process-step* level encapsulates an activity with local prerequisites and immediate consequences (if any) of the tool invocation, as imposed by the process. 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.

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

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 the DEPCE provides global infrastructure support for interoperation among local processes, it explicitly avoids the need for a global "super" process — although such a process can be implicit.

3.1 Decentralized Process Enaction: The Summit Protocol

At first glance, there are two ways in which a decentralized task can be executed: (1) The task copies remote data into its own SubEnv 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 decentralized 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 the remote data may not be accessible to the local process, and even if it is, the prerequisites and consequences determined by the local process may 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.

We devised a hybrid that combines the two: At the activity level, fetch remote data and modify them locally; but at the process-step level, fork and execute any subtasks emanating from prerequisites and consequences in the remote SubEnvs. This permits activities with arguments from multiple SubEnvs, executes tools at the same site as their process, and maintains consistency in process and product data according to the processes owning the data.

Following the "international alliance" metaphor mentioned in the Introduction, our decentralized enaction model can be described as a "summit meeting": Before the meeting (one or more activities), each party (process) takes care of handling 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. Both the preparations to the summit meeting and results from that meeting can lead to further meetings (other sets of process steps in the same task), each involving a subset of the parties, possibly with different representatives (data arguments).

Inter-process collaboration takes place when an activity is invoked on both local data and data from one or more remote processes. We call the process from which the decentralized activity is invoked the *coordinating process.*

1. *Pre-Summit* — The involved processes (i.e., those that own some of the data requested by the activity) are notified, and all of them (including the coordinating process) perform simultaneously pre-Summit process actions, *each according to its local process,*

with its local data, in the local SubEnv site. Examples of pre-Summit actions include, not necessarily in this order: (1) Verification that prerequisites imposed by the process step enclosing the activity are satisfied; (2) Verification that the activity can be executed with respect to the task workflow; (3) Active invocation of related activities, e.g., to satisfy (1) and (2); and (4) Deriving and binding data arguments that are required by the activity but were not specified as parameters. The pre-Summit phase requires that all involved SubEnvs identify the <u>same</u> requested activity (clarified later).

2. *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. A Summit is not limited to a single step or activity, but may consist of several steps (and hence several activities), each of which might require more remote data to be fetched.

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

Thus, all participating SubEnvs act as if the activities in the Summit took place in their local process with local data only, but only the coordinating SubEnv really executes the activities. The interesting point here is that both pre- and post- Summit phases occur in each SubEnv *only* according to its local process, while execution of the Summit phase involves collaboration among the participating SubEnvs.

3.2 Solution to Motivating Example

Figure 2 illustrates the enaction of the motivating example described in section 2. Note that each box in the figure does not necessarily represent a single activity, but rather a subtask that might be broken down into a fine-grained set of process steps.

The change activity is initiated by the coordinating SubEnv **SE2**. Pre-Summit takes place in a decentralized manner, where each SubEnv performs the `pre-approve` activity locally according to its own process. For example, **SE3** requires an additional `analysis` step before performing pre-approval. Once done, the Summit begins, starting with the `change` activity, and continuing with the `code-inspection` activity. When complete, post-Summit begins, again in a decentralized manner. All SubEnvs perform a `unit-test` activity, 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 devising the input sequences for the test suites and actually performing the tests, whereas the other SubEnvs perform automatic testing. Once unit testing is complete (and assuming no errors are found), a second Summit takes place in the coordinating SubEnv, this time involving a single `integration-test` activity.

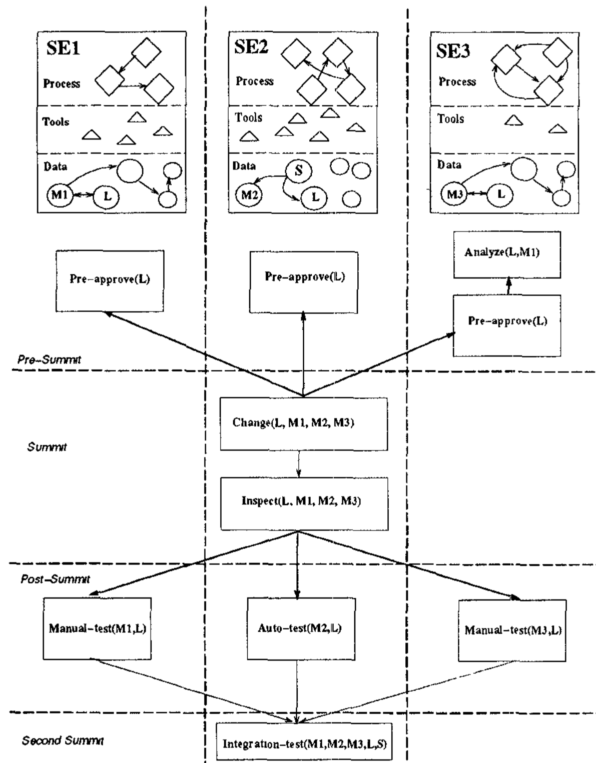

Figure 2: Solution to Motivating Example

3.3 Decentralized Process Definition: The Treaty Protocol

Returning to the "international alliance" metaphor, in order to enable multiple parties to exchange information and collaborate, they sign "treaties" determin-

ing *what* kinds of artifacts are allowed to be exchanged and *how* to perform the exchange/collaboration. Each party sends "diplomats" to each of the collaborating parties, both to enact the treaties and to ensure that they are not violated.

Similarly, in order to enable collaboration among processes, several requirements must be fulfilled:

First, there must be a way to define and agree on a *common sub-process* that is part of each local process intended to collaborate during that sub-process. At the very least, the decentralized activities must be commonly specified so that they can be identified during execution. But the "unit of commonality" might also be the process step, or even the task. In any case, this unit has to represent those process fragments that potentially involve multiple local processes. The decision as to what level (in the context hierarchy) to choose as the unit of commonality depends on the modeling primitives of the PML. For example, in a Petri-net formalism the transition (activity) seems a natural choice, whereas in rule-based PMLs the rule (process step) is more likely to be chosen.

Second, it must be possible for a common sub-process to be shared among only some of the local processes (SubEnvs) of a given global environment, not necessarily all of them. Further, the same local processes must be able to participate in multiple common sub-processes, together with the same or different collections of remote processes. There is usually some portion of each local process that is not shared with any other (a *private sub-process*).

Third, a *common sub-schema* for process and/or product data has to be defined (or identified) to enable operation of decentralized sub-processes.[1] Note that a shared sub-schema does not imply that any data instances are physically shared — only the "types" are shared. In addition, the collaboration might be restricted to data arguments from a subset of the data instantiating the common sub-schema.

Finally, the PML must allow for both *dynamic inclusion and exclusion* of common sub-processes, as well as *independent evolution* of private sub-processes. The former is particularly important when independent pre-existing processes decide to collaborate, perhaps only temporarily, while the latter is important for preserving the autonomy of local processes.

In order to form a common sub-process in a decentralized and incremental manner, the unit of commonality (treaty) defined in one SubEnv (the initiator of the treaty) has to be dynamically transferred to each of the other participating SubEnvs (parties to the treaty). And once transferred, the decentralized sub-process has to be integrated with each local process (ratification), which implies incremental "compilation" capabilities. (The meaning of "compilation" depends on the specific PCE, but most translate their processes into some internal format rather than repeatedly reparsing and reinterpreting the text.) In addition, each common unit has to be tagged in each SubEnv with some identification of the other participating SubEnvs (embassies) in order to enable verification of eligibility to operate the unit remotely (analogous to the role of diplomats). Verification can be done either when a common sub-process is included/excluded, at enaction time, or both.

4 Application of the Model

Our approach does not require the invention of a new PML intended for decentralization, nor does it make any assumptions about a particular PML. Instead, it suggests how to extend the available PCE's enaction engine to support our decentralized model. This makes it possible to upgrade a pre-existing centralized process to join a decentralized environment.

We describe now how the Summit protocol for decentralized enaction can be applied to two families of PCEs categorized by the paradigm underlying their PMLs: rules and Petri-nets. Application to PCEs with grammar-based PMLs — a third prominent paradigm — has been contemplated but is omitted due to lack of space (see [5]).

4.1 Rule-based PMLs

In general, a rule represents a process step in our context hierarchy, consisting of an optional *action* (activity) with its *pre-condition* (prerequisites) and *post-condition* (immediate consequences). The process step corresponding to a rule is enacted by first evaluating the pre-condition; the action is initiated only if the pre-condition is true. Completion of the action leads to asserting the post-condition.

Tasks are implicit in the possible rule chaining. *Backward chaining* involves matching the pre-condition of a rule with some rule whose post-condition might cause some subpart of the pre-condition to be satisfied. Then the firing of the second rule is considered recursively. *Forward chaining* arises when the action or post-condition of a rule fulfills the

[1] This is known in the database community as "schematic heterogeneity"; see [20]. The details of what constitutes a valid sub-schema is PML-specific and outside the scope of this paper.

pre-conditions of some rules, which are then fired recursively. Rule-based PMLs can be roughly divided into backward-chaining oriented such as Darwin [22], forward-chaining oriented such as AP5 [8], and those that incorporate both, like Merlin [25].

The rule is the minimal common sub-process that might be shared among local processes. When a decentralized rule is fired, either directly by a user or indirectly through automatic enaction (chaining), the following takes place:

Pre-Summit — The collaborating SubEnvs supply data to be bound to the symbols in the rule; the details of how this is accomplished depends on the PCE. Then the pre-condition of the rule is evaluated in the coordinating process. In backward-chaining PCEs, every participating SubEnv is notified if the pre-condition is not currently satisfied, or at least not already known to be satisfied. Each SubEnv may then activate other rules in its local process, in an attempt to satisfy (or verify) the failed pre-condition on its own data — possibly in a backtracking manner trying multiple alternatives. In any case, if the pre-condition cannot ultimately be satisfied, then the rule execution is halted in the coordinating process.

Summit — The action is executed in the coordinating SubEnv, involving both local and remote data. Then the local process engine must determine whether to continue enaction locally as part of a multi-rule Summit, or to complete the Summit and initiate the post-Summit phase. This can be specified by annotating the action or post-condition to indicate whether it might be part of a multi-rule Summit (if this cannot always be inferred).

Post-Summit — On completion of the Summit, the coordinating process fans-out with the relevant output to the remote SubEnvs. In forward-chaining systems, this leads to triggering of other rules in the local SubEnvs whose pre-conditions have become satisfied. In order to enable "follow-up" Summits (as with the `integration-test` in the example), chaining can also "fan-in" back to the coordinating site. Here again, some use of directives may be necessary to specify opportunities to fan-in.

4.2 Petri-net-based PMLs

The application of our decentralized model to Petri-net based PCEs is influenced primarily by MELMAC [13] and SPADE [1].

Transitions represent our notion of activities. *Places* represent the activity's data parameters. When places are typed, they can also be viewed as both prerequisites and immediate consequences on the transitions. 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. Thus the places and predicates together correspond to the process step. *Tokens* (or the marking of the net) represent the current state of the process under execution and the product data used in the activities. A transition is said to be *enabled* when its input places contain the sufficient quota of tokens. *Subnets* correspond to tasks. The unit of commonality is the transition, along with its predicates. (An alternative might be to also include the input and output places in the minimal common sub-process.) The Summit protocol starts when a common transition is attempted on data from local and remote SubEnvs:

Pre-Summit — The remote SubEnvs are notified, and each SubEnv checks if the transition is enabled locally. Note that an enabled state in the coordinating SubEnv is a necessary but not sufficient condition for firing a decentralized transition. Since Petri-net based PMLs are usually not extended to support the equivalent of backward chaining in rules, pre-Summit here is limited to verification and data binding.

Summit — If the common transition is enabled in all the collaborating SubEnvs (and the predicates are satisfied), the transition is fired. All associated SubEnvs fire the transition, but only the coordinating SubEnv executes the associated activity, with data supplied to it from the various SubEnvs. As in rules, a mechanism can be employed to determine when the Summit is complete. For example, subsequent transitions might continue to fire in the coordinating SubEnv as part of a multi-transition Summit.

Post-Summit — All associated SubEnvs transfer the appropriate tokens from their input to output places, each according to its own net (process). This can lead to further firing of transitions depending on the local nets, and may lead to subsequent related Summits.

5 Realization of the Model in Oz

We have implemented the Summit and Treaty Protocols in the Oz rule-based DEPCE, reusing as much code as possible from Marvel version 3.1 [7]. The main extensions were in:

1. Infrastructure for interaction between multiple SubEnvs (Marvel supports only a single process per environment instance, enacted by a centralized server). This includes server-to-server and client-to-remote-server communication, decentralized naming

schemes, and a new component for establishing remote connections [4].

2. Global browsing and querying capabilities. A user interface client can display and access remote data regardless of its physical location (a Marvel instance supports only a single local objectbase managed by the server). This enables the user to supply remote data arguments to a collaborative process. In addition, each client controls the "refresh" policy for its multi-objectbase display on a per objectbase basis.

3. Dynamic environment (re)configuration, to add and delete SubEnvs from a global environment. This is implemented as a process using the same notation and enaction engine as for general purpose process modeling [6].

4. A dynamic import-export mechanism for defining decentralized sub-processes, which takes advantage of the infrastructure's inter-process communication layer to implement Treaties.

5. Most significantly, substantial extensions to the process engine to support the Summit decentralized enaction protocol. However, the only change to the Marvel rule-based PML [17] was to add optional annotations to direct multi-rule Summits. Otherwise, rules maintain the identical syntax — and the semantics have changed only in those cases where remote data is accessed.

At the time of this writing, the first operational version of Oz has been completed.[2] Oz currently implements the Treaty and Summit protocols as described, except for the "fan-in" feature, which is still under development. Oz supports multiple *logical* sites, each with its own autonomous process, but not yet multiple *physical* sites: In particular, it assumes a network file system shared among the various SubEnvs and does not optimize its operation as a function of distance. Finally, while Oz provides cooperative transaction support for purely local operations (inherited from Marvel [16]), support for decentralized or distributed transactions is still minimal.

Oz currently consists of 170,000 lines of C, lex and yacc code and runs on SparcStations with SunOS 4.1.3, with XView and command-line client user interfaces; a Motif client is being developed.

6 Related Work

Shy *et al.* were among the first to identify decentralization as a key environment technology [23]. They draw an analogy between software development and the business corporation, and advocate a model for PCEs with global support for 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 corporation model differs from our "international alliance" model in that it still considers every sub-environment to be strongly affiliated with the corporation and necessarily abiding by some global rules. In contrast, our view of processes is that of independent units, which may or may not voluntarily decide to collaborate for a time.

Heimbigner argues that just like databases, "...environments will move to looser, federated, architectures ... address inter-operability between partial-environments of varying degrees of openness" [14]. He also notes that part of the reason for not adopting this approach until recently was due to the inadequacy of existing software process technology. However, his focus is on support for multiple formalisms, and in retro-fitting a process onto a process-ignorant environment. Heimbigner's *ProcessWall* [15] is an attempt to address one particular aspect of federation, namely process formalism interoperability. The main idea behind ProcessWall is the separation of process *state* from the *programs* to construct the state, so in theory, multiple process formalisms (e.g., procedural and rule-based) can co-exist and be used for writing different fragments of the same process. However, decentralization per se is not addressed, and the process state server is described as centralized.

Kernel/2r [18] supports a special case of process formalism inter-operation. The system identifies and divides the process into three distinguished kinds of process fragments, each with a separate process engine (and PML). The *interworking* process engine, MELMAC [9], supports cooperation between teams or within a team. An instance of the *interaction* process engine, WHOW, supports a single user working with a variety of tools to create, manipulate and delete development materials. The *interoperation* support, through the MUSE software bus, behaves like a process engine in that it controls partially ordered sequences of tool invocations where human intervention is not required. Although Kernel/2r does not

[2] We are using Marvel 3.1 to produce Oz, employing a process based on code re-engineering and componentization, but plan to soon start using Oz for its own further development.

directly support collaboration among multiple independent processes, MELMAC can, in principle, interface to teams who use another PCE (or who are not concerned with process at all).

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" [11]. This sounds remarkably similar to our approach. However, in his Process WEAVER system, "...processes are recursively structured into sub-processes of finer and finer granularity and detail." In other words, processes are defined top-down, whereas in our decentralized model, what is in effect the decentralized process of a global environment is defined bottom-up from the (collaborating) processes of the constituent SubEnvs.

Furthermore, Process WEAVER's distribution capability is essentially the reverse of what we have in mind: What we call pre-Summit and Summit may be distributed, but not decentralized. The process modeling formalism is based on Petri-nets, where transitions have pre-conditions and actions. After a transition has been enabled by tokens marking all its input places, each of its pre-conditions is evaluated in the global context until one of them is satisfied; there is no notion of breaking up a pre-condition into pieces evaluated according to distinct local processes. The input tokens are then removed and the action is executed. Finally, the output places are marked with tokens, and the cycle repeats. The opportunity for distribution arises in the pre-conditions, which can wait for events in the *work contexts* of a group, and in the actions, which can send these *work contexts* to a group; thus, the process can specify that multiple activities are undertaken at the same time. However, these work contexts are all part of the same global process, not defined autonomously.

7 Contributions and Future Work

The main contributions of this work are:

- Identifying the requirements for decentralization of Process Centered Environments, and distinguishing these from distribution;
- Designing a model for *process* decentralization, covering both process modeling and process enaction, which successfully trades off autonomy against collaboration, with flexible control by each participating local sub-environment;
- Sketching how to apply this model to two popular families of Process Modeling Languages, namely rules and Petri-nets;
- Developing a decentralized PCE architecture that handles all these issues; and
- Building an actual working system, Oz.

We plan to experiment with Oz and evaluate its utility in the near future.[3] Preliminary evaluation reveals that, as we have already discerned, there may be a tradeoff between ease of defining truly global processes, and maintaining autonomy among local processes collaborating only to the degree that they specify. Our approach is clearly geared towards supporting the latter, and so defining a truly global process, if desired, might be harder (although still possible!) in our international alliance model, compared to the corporation model mentioned above. On the other hand, defining a homogeneous distributed environment — whereby all sites employ identical process and only the data is distributed — can be easily defined, simply by including the entire process in a Treaty with all other processes. Finally, since pre- and post-Summit phases are effectively executed independently and in parallel, there is a substantial performance gain compared to centralized execution, as expected.

Open questions posed by this research include: (1) Support for multi-formalism inter-operability on top of our decentralized model; (2) Varying bandwidth has not yet been addressed (although see [24]); (3) Integration of "groupware" technology into the DE-PCE framework, including process support for multi-user tools, and incorporation of multi-media technology; (4) Decentralization of concurrency control and recovery policies suitable for collaborative work; and (5) Methodology for defining decentralized (sub-)processes.

Acknowledgements

We would like to thank George Heineman for his numerous contributions to the Marvel and Oz projects. Steve Popovich, Peter Skopp, Andrew Tong and Giuseppe Valetto are also working with us.

Ben-Shaul is supported in part by the New York State Center for Advanced Technology. Kaiser is supported by grants from NSF, Andersen Consulting, Bull, IBM Canada, and CAT

[3]We are involved in a multi-organization software development effort spread across Columbia University, the University of Illinois, and Bull's US Applied Research Lab.

References

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

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

[3] Naser S. Barghouti and Gail E. Kaiser. Scaling up rule-based development environments. *International Journal on Software Engineering & Knowledge Engineering*, 2(1):59–78, March 1992.

[4] Israel Z. Ben-Shaul. Oz: A decentralized process centered environment. CUCS-011-93, Columbia University Department of Computer Science, April 1993. PhD Thesis Proposal.

[5] Israel Z. Ben-Shaul and Gail E. Kaiser. A paradigm for decentralized process modeling and its realization in the OZ environment. CUCS-024-93, Columbia University Department of Computer Science, September 1993.

[6] 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*, March 1994. In press.

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

[8] Donald Cohen. Automatic compilation of logical specifications into efficient programs. In *5th National Conference on Artificial Intelligence*, volume Science, pages 20–25, August 1986.

[9] Wolfgang Deiters and Volker Gruhn. Managing software processes in the environment MELMAC. In *4th ACM SIGSOFT Symposium on Software Development Environments*, pages 193–205, December 1990.

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

[11] Christer Fernström. PROCESS WEAVER: Adding process support to UNIX. In *2nd International Conference on the Software Process*, pages 12–26, February 1993.

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

[13] Volker Gruhn and Rudiger Jegelka. An evaluation of FUNSOFT nets. In *2nd European Workshop on Software Process Technology*, September 1992.

[14] Dennis Heimbigner. A federated architecture for environments: Take II. Process Sensitive SEE Architectures Workshop, September 1992.

[15] Dennis Heimbigner. The ProcessWall: A process state server approach to process programming. In *5th ACM SIGSOFT Symposium on Software Development Environments*, pages 159–168, December 1992.

[16] George T. Heineman. A transaction manager component for cooperative transaction models. CUCS-017-93, Columbia University Department of Computer Science, July 1993. PhD Thesis Proposal.

[17] 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.

[18] Bernhard Holtkamp. Process engine interoperation in PSEEs. Process Sensitive SEE Architectures Workshop, September 1992.

[19] Gail E. Kaiser, Steven S. Popovich, and Israel Z. Ben-Shaul. A bi-level language for software process modeling. In *15th International Conference on Software Engineering*, pages 132–143, May 1993.

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

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

[22] Naftaly H. Minsky and David Rozenshtein. A software development environment for law-governed systems. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 65–75, November 1988.

[23] Izhar Shy, Richard Taylor, and Leon Osterweil. A metaphor and a conceptual architecture for software development environments. In *Software Engineering Environments International Workshop on Environments*, pages 77–97, September 1989.

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

[25] 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.