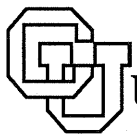
The ProcessEngine:

A Process State Server Approach

to Process Programming

Dennis Heimbigner

CU-CS-544-91 Revised, April 1992



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

The ProcessEngine: A Process State Server Approach to Process Programming

Dennis Heimbigner

CU-CS-544-92 Revised 20 April 1992

University of Colorado at Boulder

Technical Report CU-CS-544-92
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

The ProcessEngine: A Process State Server Approach to Process Programming

Dennis Heimbigner

Revised 20 April 1992

Abstract

The ProcessEngine is a process state server providing storage for process states plus operations for defining and manipulating the structure of those states. It separates the state of a software process from any program for constructing that state. Instead, client programs implement the processes for operating on the process state. This approach has a number of potential benefits such as support for process formalism interoperability, support for multiple process languages, and low-cost retro-fitting of process into existing environments. The process server interface provides descriptive mechanisms for representing process state as well as product structure. A classification of client programs is provided to show how the state server can be used in a variety of ways.

To appear in the Proceedings of the Fifth ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments

This material is based upon work sponsored by the Defense Advanced Research Projects Agency under Grant Number MDA972-91-J-1012. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Author's E-Mail address: dennis@cs.colorado.edu

1 Introduction

Much of the research into *process programming* [14] has been concerned with the formalisms needed to model and support processes. These formalisms are typically made explicit through *process programming languages* (PPL's) [12], whose purpose is to support the definition of specific processes. These formalisms, and hence the associated PPL's may be divided into two classes: modeling and execution (or enaction). As a rule, modelling formalisms emphasize concise descriptions of the normal operation of a process. Models intentionally ignore many details of a process in order to achieve a concise and clear description of process. This is consistent with the idea that process formalisms for modeling are used primarily for explanation, education, and analysis.

Process formalisms for execution are designed to drive so-called *process–centered* (or *process–driven*) environments. A process–centered environment is one in which the programmer is guided in the task of producing software according to some methodology. Such an environment extends the more traditional tool–oriented environment by adding the capability to specify the process by which software is to be constructed. This is in contrast to a typical tool based environment in which the programmer is presented only with a collection of tools and is given no help in deciding how to apply those tools to produce a software product.

To date, most of the work in process programming has been concerned with the definition of appropriate process languages [12] and with the construction of example process programs [13, 16] to test out the utility of those languages. What has been missing from this work is a consideration of how, concretely, such languages can be used to *drive* an environment. There is agreement that the environment must have some form of process "component", but the exact nature and role of that component has not been decided. Additionally, there is some dispute [6] about the correct style of programming to be used in executable process programs; roughly, the debate may be characterized as rule–based versus procedural. Each style has its merits and demerits, yet no one has proposed a satisfactory method by which multiple styles and multiple process languages can usefully co-exist in an environment. Other issues such as incremental process evolution also have yet to be completely addressed.

This paper explores the use of a process state server approach as a concrete mechanism for solving (and in some cases bypassing) many of these problems. The *ProcessEngine*[1] is the name of the prototype state server under construction at the University of Colorado. It provides storage for process states plus operations for defining and manipulating the structure of those states. A state server allows for the separation of the state of a software process from any program for constructing that state. Instead, separate client programs implement the processes for operating on the process state. Rather than focusing on the

[1] With apologies to Gibson and Sterling [4].

process program (written using some specific process language), the state server stores the state of a process in execution. It says as little as possible about how a process state is constructed and instead focuses on the structure of the process state and the legal modifications that can be applied to that state.

In effect, the ProcessEngine shifts attention from formalisms for describing process code to formalisms for describing process states. This separation allows for a degree of interoperability between different code formalisms (i.e, mixing different process languages) as long as they adhere to a common state structure.

While the state server approach is influenced by a number of previous efforts, a primary source comes from observations on the ISPW6 Software Process Example and its solutions [13]. That example required the solutions to be able to dynamically create and abort various engineering tasks as part of the process. A common approach used in the solutions was to define and manage a "reified" (explicit) representation of the engineering tasks as part of the process program. In the APPL/A [17] solution, for example, each task in the problem was represented by a corresponding Ada task in the program, thus providing a direct representation of the process elements in code. Additionally, the program included several APPL/A relations that stored information about the attributes and structure of these tasks. In effect, the APPL/A program contained two distinct representations of the process.

- The APPL/A tasks provided the directly executable code for the process.
- The APPL/A relations provided an explicitly manipulable representation of those tasks.

Consistency between those two representations was maintained by embedding maintenance operations in the APPL/A tasks and by using triggers to propagate information between relations and between relations and the APPL/A tasks. A number of other languages such as AP5 [2] and Marvel [10], produced similar solutions to the ISPW6 problem.

The state server idea elaborates on this use of an explicit process state. The novel feature is in recognizing that this explicit state can be usefully described and maintained independently of the languages which manipulate it.

This paper is organized as follows. First, there is a discussion of the merits of the state server approach. Second, the architecture of the ProcessEngine is sketched. Third, the state and product formalisms used in the ProcessEngine are described. Fourth, the possible client-server interactions are discussed. Fifth, some miscellaneous features (meta-data and transactions) are discussed. Finally, some related work and project status is presented.

2 Merits of the State Server Approach

Separating policy from mechanism is the primary benefit in applying a state server approach. In this case the policy is the process programming language and all of the "baggage" that comes along with it. This baggage includes a bias toward some form of programming style (e.g., rule-based versus procedural). It includes the burden of complex features such as reflection, reification, and exception handling. The state server pushes all of that complexity out to other components of the system. This is not to say that the state server has *no* biases; it will be clear (see section 4) that the choice of state model and product model introduces bias. But it still seems less restrictive than a complete process language.

The style problem (rule-based versus procedural) [6] represents another kind of policy. The procedural style is good for describing the normal execution path for processes. Unfortunately, it is not psychologically plausible since it does not allow for programmer flexibility or unanticipated actions. The rule-based style, by contrast, is quite flexible and can handle unanticipated actions. But it is difficult to understand the process flow. The REBUS program [16], for example, uses this style and is not as easy to understand as one could wish.

Style choices are policy[2] in that they say something specific about how to construct a process state. The state server side-steps the whole problem by only providing a mechanism for constructing states and saying nothing about how it must be done. Section 8 expands on this issue.

The state server can provide a migration path for retro-fitting existing environments with process support. One can introduce a state server into an environment along with a minimal set of tools to define state models, browse states, and modify states mostly manually. Even this minimal set of tools can be useful in allowing programmers and managers to record their activities and then experiment with process elements.

Once a state server has been introduced into an environment, it is possible to begin the automation process by "wrapping" tools so that they begin to automatically record their activities in the state server. There is a strong analog to such systems as the HP encapsulator [8] which incrementally add control integration to environments by wrapping tools to signal and receive events. Here, instead of wrapping for events, one is wrapping for process.

Returning to the process language issue, the state server has the advantage that no special language is required to write process control. Any language that can be made to access the server can be used to define some piece of process code. Programmers and managers can use a variety of languages to write process. This can significantly lower the barriers to experimentation with process in an environment.

[2]Assuming that the process language can support both a rule-based style and a procedural style.

Once the state has been separated out, it should be easier to view it. Little has been written about the mechanisms for providing visualizations of the state of a process when that state is inextricably tied to some process language. Is it expected that the visualization system will be written in the process language? Or will it need special interfaces written specifically for visualization? Or will the visualization have to be part of the run-time system for that language? Any of these choices will complicate either the process language or the run-time system, or both. Using a state server, writing many kinds of visualization tools is straightforward. The tool is written in whatever language is appropriate using calls into the server to obtain the state data needed for the visualization.

It has been hypothesized that process languages need mechanisms for managing on-the-fly changes to the process. This is viewed (correctly) as a difficult task because it is assumed that this requires the language to be reflective; that is, the language should be able to materialize, examine, and modify its own state. Separating state from the language avoids the need for reflection in the process language. The state will have already been materialized (in the server) and so it can be manipulated by normal programs. This is not to say that changes to process will be easy. There are still many problems in determining the consequences of an on-the-fly change.

Handling unanticipated actions (sometimes termed "process exceptions") should also be easier. An unanticipated action is a situation in which some exceptional condition occurs that requires a modification in the normal process flow, but does not require a permanent change to the way the process is to be done in the future. For example, it might be the case that the normal process requires that a design must be approved by some manager before it is accepted. In any given "execution" of this process, there may be circumstances (illness, deadlines, or whatever) that dictate that some designs are accepted without review. A state server can handle such exceptions by modifying the state to reflect the exception and possibly marking some tasks as completed even though the normal process has not been followed.

3 Process State Server Architecture

The state server architecture is currently being implemented as a straightforward client-server architecture. Client processes (in the operating system sense) communicate with a server process using remote-procedure calls (RPC). The server is actually composed of a number of modules:

Server Interface: The *server interface* handles the details of receiving requests from clients, invoking the appropriate local procedure to field the request, and returning any result back to the client.

Catalog: As described in section 6, this module maintains a queryable meta-database of information about the structure of the process state (process goal types and product types).

Event Dispatcher: The event dispatcher provides functionality similar to the broadcast message server of Field [15]. It maintains a database of clients registered to receive events along with the event patterns defining the events of interest to each client.

Process States: These are the actual state graphs and product objects and tuples. Its general structure must be in conformance with the schema elements defined in the catalog.

Persistent Storage: It is desirable for the state of the server to be persistent for very long periods of time (including over machine failures). This module provides for persistent storage of states. In practice, it is often also responsible for providing concurrency control (see section 7).

4 A Formalism for Describing Process State

A state server such as the ProcessEngine, via its external interface, implicitly defines a formalism for defining and manipulating process states. This formalism is defined by the interface calls to the server and the associated constraints on the legal order in which those calls may be invoked.

This formalism consists of two parts. As one part, there must be some representation of the tasks (or goals or procedures) that have been completed or that should be completed. As part of this task representation, there should also be some equivalent of variables to define the flow of data between various tasks.

As the other part of the formalism, there must be some representation of the product (broadly construed) that is being produced by the process. The product can be expected to include more than just the final code. It will consist of a constellation of data objects (e.g., requirements, design, configurations) that are produced during the execution of the process.

In the following subsections, these various elements of the formalism are discussed in more detail.

4.1 Task Representation

In the ProcessEngine, a process state is represented as a directed acyclic graph (DAG) of task nodes, which are instances of some collection of task types. Within a graph, the node instances are connected by two kinds of edges. One edge type in this graph has the

semantics of "has-subtask", or inversely "is-subtask-of." The other class of edge in the ProcessEngine formalism has the semantics of "precedes."

Figure 1(b) shows a simple task graph that will be used as a short example in this discussion. It is in fact a simple Makefile [3], and corresponds to a particular process for constructing an object "x" from objects "y.o" and "z.o." For simplicity, the task types are not indicated. Figure 1(a) shows the corresponding Makefile. This example is admittedly simplistic, but should suffice for purposes of demonstration.

The vertical edges in the figure represent the "has-subtask" relationships. Note that they partition the graph nodes into layers. The horizontal edges in the figure represent "precedes" edges. These edges are constrained to only connect nodes in the same layer that share a common parent in the immediately higher layer. These edges induce a partial order over all subtasks of a task, and transitively, all descendants of those subtasks. For example, task "ld([y.o,z.o],x)" must be preceded by task "make-y.o" and thus must also be preceded by task "cc(y.c,y.o)."

A portion of the server interface provides operations for constructing and manipulating task graphs. For our example, we will use the following operations:

Initiate: Initiate a new process graph and instantiating its root task.

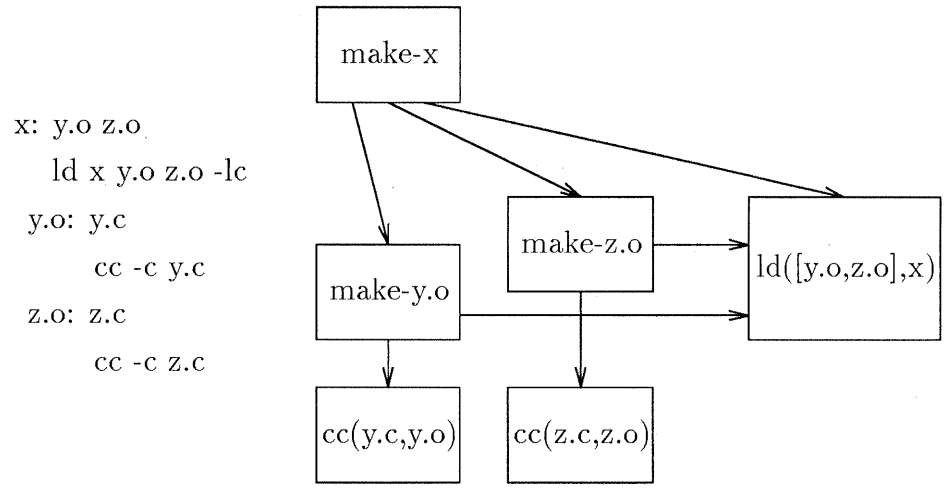
Create Task: Create a task object (as an instance of some task type).

Precede: Adds a predence edge between two tasks.

SubTask: Adds a subtask edge between two tasks.

Using these server operations, the fragment of Figure 1(b) might have been constructed using the following sequence of operations:

1. Invoke *Initiate* to create a process to make object "x" with initial task "make-x."
2. Invoke *Create-task* for tasks "make-y.o," "make-z.o," and "ld([y.o,z.o],x)." The [] notation is intended to bracket lists of elements.
3. Invoke *Sub-task* to create sub-task edges from the task "make-x" to the sub-tasks "make-y.o," "make-z.o," and "ld([y.o,z.o],x)."
4. Invoke *Precedes* to create precedence edges from the tasks "make-y.o" and "make-z.o" to the task "ld([y.o,z.o],x)."
5. Invoke *Create-task* and *Sub-task* again to create the task "cc(y.c,y.o)" as a sub-task of the task "make-y.o."
6. Invoke *Create-task* and *Sub-task* again to create the task "cc(z.c,z.o)" as a sub-task of the task "make-z.o."



(a). Makefile Contents. (b). State Graph.

Figure 1: Example Task Tree.

As a task is created, it is reasonable to assume that some external activity must be initiated as a consequence of establishing the task. When (and if) this activity is completed, it is desirable to somehow note this fact. This notation will be termed "task satisfaction." In its most primitive form, task satisfaction is marked by invoking the *Satisfied* operation in the server interface to declare that a task is considered satisfied. Note that this operation says nothing about the semantics of task satisfaction, only that some external client deemed the task to have been satisfied. In some cases, this marking may be carried out automatically. It is possible to attach (via an event) a *tool* to a task and to specify that satisfaction of that task can be obtained by successfully executing the attached tool.

Returning to our example graph, the following sequence might be used to satisfy all the tasks in the graph.

1. *Satisfy* task "cc(y.c,y.o)" by executing the C compiler as an attached tool.
2. *Satisfy* task "make-y.o"
3. *Satisfy* task "cc(z.c,z.o)" by executing the C compiler as an attached tool.
4. *Satisfy* task "make-z.o"
5. *Satisfy* task "ld([y.o,z.o],x)" by executing the loader an attached tool.

6. *Satisfy* task "make-x"

Note that this sequence is not unique. Other orders of satisfaction could have been applied to achieve the same effect. Additionally, the impression may be left that task expansion must also precede all task satisfaction. This is not the case; the two processes may be intermixed subject to the constraints implied by the edges in the graph as embodied in two obvious rules.

1. No task may be marked as satisfied until all of its subtasks are marked as satisfied.

2. No task may be marked as satisfied until all of its preceding tasks are marked as satisfied.

4.2 Task Parameters

In addition to simple task objects, the task representation includes *task parameters*, which are a simple form of variable.

Task parameters server two purposes. First, the parameters serve to define the flow of data between task nodes. Second, many processes can be more concisely specified if task types can be parameterized and reused with different parameters at various points in the process. In the Makefile example, it is inconvenient to have a different task "cc(X.c,X.o)" for every possible X. Rather, it is preferable to have a parameterized task type of the form "cc(C,O)" which is satisfied when C is bound to a source file and O is bound to the corresponding object file.

Each task type has an associated set of task parameters divided into input parameters or output parameters. In line with this, task parameters are marked as either *in* (input) or *out* (output) arguments[3]. When a task instance is created, a set of actual arguments (instances of the parameters) is created and associated with the task instance. Initially, all of these arguments are unbound in value. A simple form of unification, as in logic languages, is the model for assigning values to arguments in this state model. Operations are available to bind arguments with either product values (see section 4.3) or with other task arguments. In effect, this last case binds two variables together so that if one becomes bound to a value, then the other is bound to the same value.

Data flow between tasks can be represented by defining a variable as the output of one task and the input of a succeeding task. The semantics of variable binding ensure that when the variable is bound to a value by the first task then that same value will appear as an input to the later task.

Since one rarely would want a tool to be invoked until all the inputs to a task are bound, a distinction is made between the actual creation of a task, and when its last

[3]The equivalent of *in-out* is deliberately not provided.

input argument is bound to a real value. This latter situation is termed *task ready.* When a task is marked as satisfied, that too is an event of interest. But because there is more control over when this occurs, the binding state of output arguments is handled differently for task satisfaction. A task can be marked as satisfied even if some of its output arguments are not bound to real values. The choice to leave arguments unbound is deliberately left to the client that marks a task as satisfied.

4.3 Product Representation

In general, processes are intricately interwined with the structure of the products to be produced. In light of this, it is reasonable to include some form of product type model as part of the server.

Debates about appropriate type models are endless (and probably fruitless). The ProcessEngine uses a form of entity-relationship model. This model for product structure is based on the notion of object identity plus a relation model. Experiences with the APPL/A process language, and examination of other potential process languages such as AP5 indicate that this is a plausible choice, but other choices are certainly defensible.

This type model consists of the following elements:

Scalar Types: A selected set of primitive scalar types are part of the model. These types currently are *integer*, *string*, and *atom.*

Abstract Object Domains: It is possible to define abstract domains of *objects* (e.g., *text-file*). These domains consist of only object identifiers: i.e., values unique for every object across all domains. It is possible to define subtype domains (e.g. *source-code* **isa** *text-file*). Union types can be defined also. A distinguished type named *object* is defined as the parent of all other domain types. It is important to note that objects (as identifiers) do not necessarily have any state associated with them.

Relation: One can define n-ary relations with named attributes (fields) over either object domain types or scalar types. Relations are bags of tuples and not sets (a deliberate choice). Relations are the state-carrying elements in this model. If an abstract object needs state, then relations must be defined with one attribute coming from the domain of that abstract object and one or more other attributes defining the associated state. Relations also inherit down subtypes. This means that if relation R is defined over Type1 $\times$ Type2 and Type3 **isa** Type1, then R may contain fields whose first value is an instance of Type3.

Given a product schema defined using the above elements, the server interface provides operations for defining that schema to the process server using operations for creating objects and relations and tuples.

As a special help, a client can define a domain type and define specific byte strings as elements in that domain. This allows the clients to use externally generated handles in the product structure. Note that ambiguity is possible here if two semantically different external handles have the same byte string encoding.

Finally, it is assumed that the product type model can also be used to represent the task representation graphs, including task types and task parameters. This provides a capability for clients to attach client-defined annotations (such as "task-stopped") to, for example, task nodes in a state graph. In fact this is the underlying mechanism used to implement operations such as *Satisfied.*

5 Event Management

Event dispatching systems such as Field [15] and HP-SoftBench [8] have proven to be remarkably useful for integrating the control of tools in an environment. One tool can signal an event such as editing a source code file. Another tool, *Make* for example, can register with the dispatcher to receive such events and cause the source code file to be automatically re-compiled.

Changes in the state of the process state server also represent events of which clients should be notified. For example, an event can provide one means for connecting a task and a tool. When a specific task is added to the process state, this can be defined to signal an event with a specific structure. This event can be fielded by the tool responsible for satisfying tasks of that type.

To facilitate client notification, the state.server provides an event definition, registration, and notification facility as part of its interface[4][5]. Essentially all changes to the process and product state can generate an event notice. But the changes of primary interest are task creation, task satisfaction, task ready (i.e., last input bound), object creation, and tuple creation and deletion.

For compatibility with other event systems, events generated by the state server are strings which encode information about actions. For example, when a task is instantiated, an event is generated with a string encoding such things as the state server, the task object and the operation. This event might be fielded by a tool designed to satisfy tasks of this type (though in practice, most tools will wait for a *task ready* event to guarantee that all the inputs are bound to values).

Events are generated in two ways. First, process state and product state changes generate events. Second, the dispatcher interface is exposed to clients, so they may

[4]Event signalling takes the state server out of a role of passive component and makes it an active component.

[5]Note that if the environment already has an event server, then the state server can use it instead of its internal one.

generate arbitrary events as well. Clients must register with the dispatcher in order to receive signalled events. Note that there is no need to have the dispatcher accept events from outside. The equivalent effect can be had by defining a client to capture those external events and perform whatever state actions are required.

6 State Model Catalog

The basic structure of a state model is simple: task nodes connected by edges. But for the server to function it must have more detail about the model being supported. In effect, the server needs a specification of the allowable task types, along with their input/output structure, and the types and relations of the product model. In most systems, there would be a defined language for these specifications and a "program" in that language (a specification) would be input at either the time of server construction or at the time of server invocation.

But it seems desirable to allow more flexibility with the specification and so we take a meta-data catalog approach as opposed to a language approach[6]. In effect, the catalog stores a data structure (the meta-data) that contains the same information as would be defined using the specification language. The primary advantage of a catalog structure vis-a-via a language is that it allows for browse and update of the schema elements. It is possible to query the catalog to see what tasks, types, relations, and events are defined and to dynamically modify those elements of the model. This is useful in defining new tasks representing the availability of a new tool, for example. Dynamic addition of event receivers is also important.

It is assumed that the elements in the catalog have names and so the catalog implicitly defines a name space consisting of those elements. The initial implementation assumes a flat name space[7] for top level objects of tasks, types, relations, and explicitly defined events. This means that each such top level element must have a name that is unique. Secondary elements such as task inputs and outputs have names, but they are conceptually "contained" within other elements (tasks types in this case). Those secondary names need only be unique within their containing element. Of course, this means that technically, the name space is not flat, but it is still less structured than, for example, the Unix file system.

Part of the server interface is devoted to operations for manipulating and accessing the catalog structure. This interface makes significant use of *handles*, which are references to objects in the server. The client can only get handles from the server, copy them

[6]The two should not be considered mutually exclusive. There is no reason why the catalog cannot be initialized by means of some specification written in some language.

[7]A flat space is acceptable for an experimental prototype but a more sophisticated and structured name space will be needed beyond that.

around, and send them as arguments back to the server. The client has no knowledge of the internal structure (if any) of the handles.

The catalog interface consists of a number of operations for each kind of element. One operation is to *lookup* an element by name and return a handle to it,. There also operations to create and destroy instances of an element. Other operations are provided to browse the catalog by enumerating the elements in it. By appropriately invoking these operations, a client can specify an arbitrary collection of elements.

7 Transactions

It is to be expected that the state server will be accessed simultaneously by multiple clients. Additionally, it is also desirable to support atomicity for clients so that a client failure will not compromise the integrity of the server. This combination implies a need for some form of general database transaction mechanism.

As an initial solution, it is assumed that the server interface provides the following standard interface operations for supporting transactions.

Begin Transaction: Initiate a transaction between a client and the server.

End Transaction: Terminate (successfully) a transaction between a client and the server and commit any modifications performed during the transaction.

Abort Transaction: Terminate (unsuccessfully) a transaction between a client and the server and ensure that the effects of all operations performed during the transaction are erased.

Explicit locking operations are not required since the kind of lock (read or write) can be deduced from the particular operation and its operands.

8 Client Paradigms

Given a state server with the architecture as described previously, one is left with the question: how does one actually use it? This reduces to the problem of constructing clients to define a process state and to manipulate it through the server interface. As has been indicated, the state server approach deliberately emphasizes mechanism and leaves policy for the clients. Decisions such as task expansion, task satisfaction, and constraint maintenance are pushed out of the server and into the clients.

Proper construction of clients is, frankly, still the subject of experimentation. Because they may be arbitrary programs, it is difficult to crisply characterize all possible clients. Nevertheless, it is possible to discern three rough classes of clients: *tools*,

process-constructors, and *process-constrainers*. Tools are what you might expect; they are monolithic independent programs for performing some action. Typically a tool is associated with a leaf task in the process state. When the task is instantiated, the tool is invoked. When the tool completes successfully, the task is marked as satisfied. Tools may be interactive, which means that from the point of view of the process, the associated task is non-deterministic.

There is a wide variety of clients which may be characterized as *process-constructors*. This variety directly reflects the range of possible styles for constructing processes. Broadly, it is possible to distinguish three sub-classes of constructors:

1. *Procedural* constructors operate "top-down." These clients looks for a task of a given structure and expands it by adding a fixed set of subtasks. This is more-or-less "backward-chaining" or "procedure-call."

2. *Rule-based* constructors operate "bottom-up." These clients looks for a collection of tasks, creates a new task, and converts the collection of tasks into subtasks of the new task. This may be viewed as a form of forward chaining. Sometimes, the supertask may already exist in the process state, and this kind of client can act to merge previously independent process fragments.

3. *Planning* constructors are actually a generalization of procedural constructor. A good example of a process formalism using planning can be found in [9]. A planning system would look at the task to be expanded, and at a range of tasks actions and try to create a specific set of subtasks to satisfy the parent task. A procedural constructor can be seen as a rather simplistic planner in that it uses the same plan (sequence of subtasks) for every parent task. A true planner might produce different subtasks depending on additional information such as the input values associated with the parent or knowledge about the product state.

It is intriguing to note that all three types of constructor clients might simultaneously be acting on a single state server. This provides one approach to interoperability between various process program styles and formalisms.

The third identifiable class of client is the *process constrainer*. This is a client that is responsible for checking and enforcing any constraints on the legal structure of the process and product state. Constraint in this context should not be thought of as a predicate (as in APPL/A or AP5), but rather as an arbitrary piece of code which examines the state of the process server and decides if it is correct or not. The code of the client might have been generated automatically from such a predicate, but it could be constructed in some other fashion as well.

In a typical situation, a constructor client might add tasks to the state. These tasks would generate events that would cause the activation of some set of constrainers. The

constrainers would examine the state and if any constraint were violated, then the constrainer would initiate some form of repair. Repair might entail modification of the state, or even rollback by detaching the new tasks from the task graph and possibly even destroying tasks.

It is worth repeating that this classification of clients is purely ad-hoc. It reflects the typical styles enforced by the various formalisms defined in the process literature. But the process state server does not force any client to fit into any of these categories. If a client chooses to mix styles, or use some other style altogether, or be simultaneously a constructor, tool and constrainer, then that is perfectly acceptable (though perhaps not desirable).

In fairness, it should be noted that this extreme flexibility in clients has an important drawback; it may be difficult to comprehend the process program that is generating some particular state in the state server because the "program" is spread out over a number of client programs possibly written in a variety of programming languages. This may imply the need for some separate specification of the process that can be understood, and from which the various clients can be derived (manually or automatically). But this is a topic for future consideration.

9 Related Work

As previously mentioned, the primary motivation for a process state server came from an examination of the solutions for the ISPW-6 Software Process Example. It was clear from those solutions that process state was important separate from any formalism for constructing it.

The style problem [6] was also a factor. the problem there was to somehow reconcile the need for both prescriptive and proscriptive process languages. The key insight was to recognize that both styles could share a common state, even if they were constructed in different ways (by rules or procedurally).

But others are certainly recognizing the importance of process state. For example, a number of the papers in [18] relate to the issue:

- The Process Virtual Machine (PVM) of Balzer is substantially more ambitious than a process state server. But, it does assume that part of the PVM will be a grammar for a process state. If that grammar happens to match the state structure used by a process state server, then that server probably could be used as a component of that PVM.

- Kaiser is proposing a rule-based process server based on Marvel [11]. Unlike the state server, this server contains a complete process programming language (Marvel). This style of server seems to have more policy in that it uses the Marvel

rule-based language to manipulate any state in the server. Many of the tasks of the Rule-Based server seem similar to those of the Process Virtual Machine. As an aside, one may hypothesize that if the underlying state of a Marvel system were made explicit, then it would look very much like a process state server as defined in this paper.

- The Articulation system of Mi and Scacchi appears to be moving to separate how a process is constructed from what is constructed. Its *development instance* concept could be viewed as an approximation of process state.

- Penedo addresses the issue of separating modelling from implementation. As with the PVM, implementation includes a state component and so there is a place to insert the process state server technology.

The ECMA/NIST Reference Model [5] explicitly includes a notion of *Process State Services*. Again, the notion of state in that model is more general than that of the process state server defined in this paper, but it is clear that the capabilities of the ProcessEngine overlap with the functionality of the ECMA-NIST Process State Service.

The key difference between the ProcessEngine and these other proposals is that the process state server is intentionally "minimal." To reiterate, it emphasizes mechanism over policy. Other proposals include a specific language or language style as part of the server. Additionally, constraints on the form of the process state are included as part of the server. The process state server defined here pushes such constraints out of the server and requires clients to enforce them. One consequence is that the ProcessEngine has the potential to address process-formalism interoperability, which is has not been addressed by these other systems.

It may be possible to reconcile all of the proposals by defining additional servers to act as clients to the ProcessEngine. A client could implement, for example, the rule part of the Rule-Based Server and leave the state definition to the ProcessEngine. This would have the advantage that other language servers could share the same state server and provide a certain degree of interoperability.

10 Status

A version of the ProcessEngine is currently under construction. This initial version is actually being implemented using modules from the existing Triton system [7]. Triton may be briefly characterized as a serverized object repository providing persistent storage for typed objects, plus functions for manipulating those objects. Triton uses an existing object manager, Exodus [1], to provide much of its functionality.

The process state server has many similarities to an object manager and so it should not be surprising that much of Triton can be used for the ProcessEngine. Triton has

modules for persistent storage (Exodus), for server interface management, for a catalog, and for a form of event management: all of which can be used in the ProcessEngine.

11 Conclusion

The state server approach introduces a powerful separation of mechanism from policy into process programming by separating the state of a process from the languages that might be used to construct that state. This has a number of benefits such as support for multiple languages and styles, simplification of process languages, support for retro-fitting, and visualization. This indicates that this approach has significant value as an approach to process programming.

References

[1] Michael Carey, Dave Dewitt, Goetz Graefe, Doug Haight, Joel Richardson, David Schuh, E. Shekita, and S. Vandenberg. The EXODUS Extensible DBMS Project: an Overview. In Stan Zdonik and David Maier, editors, *Readings in Object-Oriented Databases*. Morgan Kaufmann, San Mateo, CA, 1990.

[2] Don Cohen. *AP5 Manual*. Univ. of Southern California, Information Sciences Institute, March 1988.

[3] Stuart I. Feldman. Make – a program for maintaining computer programs. *Software – Practice and Experience*, 9:255 – 265, 1979.

[4] William Gibson and Bruce Sterling. *The Difference Engine*. Bantam Books, 1991.

[5] NIST ISEE Working Group and the ECMA TC33 Task Group on the Reference Model. Reference Model for Frameworks of Software Engineering Environments. Technical Report NIST Special Publication 500-201, Technical Report ECMA TR/55, 2nd Edition, ECMA (European Computer Manufacturers Association) & NIST (National Institute of Standards and Technology, United States Department of Commerce), December 1991. Revision of Technical Report ECMA TR/55 (Dec. 1990).

[6] Dennis Heimbigner. Proscription Versus Prescription in Process–Centered Environments. In *Proceedings of the 6th International Software Process Workshop*, Hokkaido, Japan, October 1990.

[7] Dennis Heimbigner. Experiences with an Object-Manager for A Process-Centered Environment. In *Proceedings of the Eighteenth International Conf. on Very Large Data Bases*, Vancouver, B.C., 24-27 August 1992.

[8] Hewlett-Packard. *HP Encapsulator: Integrating Applications into the HP SoftBench Platform*, 1989. HP Part No. B1626-90000.

[9] Karen E. Huff. Plan-based intelligent assistance: An approach to supporting the software development process. Technical Report COINS Technical Report 89-97, University of Massachusetts, Amherst, September 1989.

[10] Gail E. Kaiser. Rule-Based Modeling of the Software Development Process. In *Proc. 4th International Software Process Workshop*, October 1988. Published in ACM SIGSOFT Software Engineering Notes, v. 14, n. 4, June, 1989.

[11] Gail E. Kaiser, Naser S. Barghouti, and Michael H. Sokolsky. Preliminary Experience with Process Modeling in the Marvel Software Development Environment Kernel. In Bruce D. Shriver, editor, *23rd Annual Hawaii International Conference on System Sciences*, volume II, pages 131–140, 1990. Kona, Hawaii, January, 1990.

[12] T. Katyama, editor. *Proceedings of the 6th International Software Process Workshop: Support for the Software Process.* IEEE Computer Society Press, Hakodate, Hokkaido, Japan, 28–31 October 1990.

[13] Marc I. Kellner, Peter H. Feiler, Anthony Finkelstein, Takuya Katayama, Leon J. Osterweil, Maria Penedo, and Dieter Rombach. Software Process Modeling Example Problem. In Takuya Katayama, editor, *Proceedings of the 6th International Software Process Workshop: Support for the Software Process.* IEEE Computer Society Press, 1991. Hakodate, Hokkaido, Japan, October, 1990.

[14] Leon J. Osterweil. Software Processes are Software Too. In *Proc. Ninth International Conference on Software Engineering*, 1987. Monterey, CA, March 30 – April 2, 1987.

[15] Steven P. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, pages 57–67, July 1990.

[16] S. M. Sutton Jr., H. Ziv, D. Heimbigner, M. Maybee, L. J. Osterweil, X. Song, and H. E. Yessayan. Programming a Software Requirements Specification Process. In *Proceedings of the First International Conference on the Software Process*, Redondo Beach, CA, October 1991.

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

[18] Ian Thomas, editor. *Preprints for the 7th International Software Process Workshop.* Rocky Mountain Institute of Software Engineering, Younteville, Ca., 15–18 OCtober 1991.