A Metalinguistic Approach to Process Enactment Extensibility

Gail E. Kaiser Israel Z. Ben-Shaul* Steven S. Popovich
Stephen E. Dossick

Columbia University
Department of Computer Science, MC 0401
New York, NY 10027, UNITED STATES
kaiser@cs.columbia.edu

Abstract

We present a model for developing rule-based process servers with extensible syntax and semantics. New process enactment directives can be added to the syntax of the process modeling language, in which the process designer may specify specialized behavior for particular tasks or task segments. The process engine is peppered with callbacks to instance-specific code in order to implement any new directives and to modify the default enactment behavior and the kind of assistance that the process-centered environment provides to process participants. We realized our model in the Amber process server, and describe how we exploited Amber's extensibility to replace Oz's native process engine with Amber and to integrate the result with a mockup of TeamWare.

1. Introduction

The essential concept underlying Process-Centered Environments (PCEs) is language-based extensibility. That is, each PCE provides a language in which users specify the desired tailoring of the system's behavior to their particular needs and requirements. In other words, they represent the process in the process modeling formalism provided by the PCE, and this process model is then interpreted or executed by the PCE's process enactment engine. Such "first-order" extensibility has been widely investigated for a decade or so, and several major paradigms for process modeling formalisms have been investigated, including rules, Petri nets, grammars, task graphs, and imperative code [20].

A related "second-order" kind of extensibility involves the ability to modify or re-engineer the process, perhaps dynamically while a given process instance is in progress. Termed process evolution, this concept has also been investigated in recent years in the software process community [22]. Reflection has been one influential language-based approach to evolution [2].

Such first- and second-order, fixed-language and fixed-interpreter, extensibility is inherently limited, however, in two major respects:

- *Language* — The expressive power of the particular process modeling language and its computation model determines the scope of extensibility, even when *in vivo* evolution is supported. That is, the assistance the PCE can give the user(s) in carrying out a given process is a priori restricted when the process model is written, or modified, to those processes that can be defined in the language and how that language is enacted by the process engine. For example, process-wide constraint enforcement is readily supported in most rule-based process formalisms via overloaded (data type-specific) pre-conditions on primitive operations (like OzWeb's `read` and `write` [11]), whereas constraints usually must be specified on a per task basis in Petri nets (e.g., predicates in FUNSOFT nets [12]). On the other hand, modular process hierarchy, which is built-into many (extended) Petri net formalisms, may not be directly supported in rule-based PCEs.

  While some extensions to the basic language paradigm may be done at the time the language is designed (such as extending Marvel rules with control annotations that indicate partial rule chains should be enacted as all-or-nothing transactions [4]), clearly not all desirable functionality may be envisioned *a priori*, when the language and its engine are designed and implemented. The inherent challenge of extensibility lies in the fact that the added-on feature(s) had not been thought of at the time of the initial design, or else they would already

*Dr. Ben-Shaul is now at Technion-Israel Institute of Technology, Department of Electrical Engineering, Technion City, Haifa 32000, ISRAEL

be in the language/system in the first place.

- *System* — Process modeling may be viewed as supplying a language-based application programming interface (API) to the services provided by the PCE. Again, the single and fixed API, and single and fixed interpreter or execution engine, effectively limits the capabilities of the system with regards to its behavior in supporting human process participants. For example, adding process monitoring, e.g., for measurement purposes, may require addition of interception techniques and logging facilities to track and record process activities. Adding guidance support, e.g., to notify users when tasks become enabled and allow them to select among currently enabled tasks, may demand addition of an agenda mechanism in which to store some representation of the enabled tasks [29].

  Such extensions might be written directly in an imperative process programming language like APPL/A [28], but for most declarative process modeling formalisms adding on such functionality necessarily involves modifying the underlying process enactment engine and/or integration with other (sub)systems, independent of extensions to the language per se. In other words, the process modeling API generally does not provide the primitives necessary to substantially change the mechanisms by which the PCE supports process enactment.

There are two main alternatives for advancing to "third-order" beyond the fixed single-language/single-interpreter extensibility: One is to eliminate the "singleness", through a *multi-lingual* approach, and the other is to eliminate the "fixedness", through a *meta-lingual* (henceforth metalinguistic) approach. These approaches are not mutually exclusive, they could in principle be combined, but we do not address this here.

Multi-lingual extensibility can itself be sub-divided into two different approaches: The first enables multiple languages to co-exist as peers, their independent process engines interfacing to a common process state (or enabled process task) representation, in the style of ProcessWall [13], perhaps but not necessarily with an external task scheduling mechanism [24]. The second approach is based on layers, with translators from higher to lower levels and a "process assembly language" at the bottom. We have begun investigation of the former approach, see [7], and studied the latter extensively as described in [26, 14, 19].

While enhancing the level of abstraction and potentially realizing the various enactment models intended for the different languages, the multi-lingual approach is ultimately still dependent on the capabilities of the underlying interpreter (and/or the assembly language) in the layered case, or on the process state/task server, in the peer case.

In our *metalinguistic* approach to "third-order" extensibility, the basic process modeling language can be extended with new syntax to include user-invented process enactment directives, and the interpreter can be augmented with new semantics to implement the new directives, to interpret the original syntax in a multiple different ways, and to add new process assistance services. This approach is analogous to metalinguistic abstraction in Scheme [10]. Then tailored instances of the PCE can be employed in a great variety of applications, including those not envisioned at the time the system was originally developed. It is important to note that here we generically extend the process *system* itself (i.e., the PCE), as opposed to refining, or evolving, a specific process *definition* (e.g., as in [1]).

Metalinguistic extensibility introduces difficult technical problems. In particular, the process interpreter must be designed to allow "deep" insertion of, and a structure for invoking, independently-written code that changes its internal behavior — without affecting or conflicting with built-in capabilities or other external extensions.

In the rest of this paper, we focus on a particular process paradigm, i.e., rule-based, to which we have applied our metalinguistic approach. While we believe that metalinguistic extensibility is, in principle, generically applicable to other popular process formalisms, we have no supporting evidence yet. We demonstrate here only that our metalinguistic approach is very effective for extending the process assistance afforded by rule-based PCEs.

Section 2 describes the application of our metalinguistic approach to rule-based PCEs in general, and the range of potential extensibility adaptations. Section 3 discusses the mechanisms required to realize the metalinguistic approach, and presents an extensible rule-based process server, Amber, in which such mechanisms were implemented. Section 4 illustrates Amber's extensibility with respect to two different kinds of process enactment functionality: one that added multi-server connectivity and process interoperability in the style of Oz [6, 5], and another that added more intuitive process modeling (than plausible with rules) as well as online process visualization by integrating Amber/Oz with a mockup of the TeamWare research prototype from the University of California at Irvine [9]. Section 5 summarizes the contributions of this research.

2. Spectrum of (Rule-based) Metalinguistic Extensibility

Rules come in many forms. Some kinds have two parts (condition and action, or logical precedent and consequence), while others have three parts (condition, operator and effect, or event, action, postcondition). Some systems support only forward chaining, some only backward chaining (or inferencing), some both; AI planning systems ef-

fectively simulate chaining to draw up a specified plan. To generalize the discussion on rules (while focusing on PCE rules), however, we assume that each rule-based process modeling language (PML) has the following constructs:

- A *condition*, which specifies a predicate to be checked before the rule is executed. The condition may be formed over local or global process state variables as well as system variables.
- An *activity* that encapsulates the actual process step modeled by this rule. An activity typically involves invocation of an external "tool" application, on data which may be modeled in the PCE (in PCEs that support data integration) or also external, and by (possibly a set of) users which may or may not be designated in the body of the activity (e.g., by roles).
- A set of *effects*, which specify the assertions to be made on the process state as a result of the activity invocation. Although some rule formalisms combine activities and effects as "actions", it is usually more convenient to separate these sections in PCEs due to the fact that activities are typically external to the PCE engine.
- *Chaining* policies, which determine the control-flow of the process, i.e., when and how one rule invokes automatically other related rules as a result of a logical *matching* between them, as well as means to manually restrict automatic invocation. >From PCE perspective, chaining is effectively the process enactment mechanism. Matching may be intentionally formed by the process modeler who wishes to bind several activities, or automatically *inferred* by the rule interpreter, although they are in general indistinguishable from the process engine perspective.

Due to the high-level nature of rules, there is a wide range of possible adaptations that can be made with respect to process enactment. (Recall that we discuss here extensions to the language and/or interpreter which affect *all* process instances, as opposed to tailoring a specific instance.)

The first extensibility aspect concerns types of rules and their special properties. Rule type extensibility is open-ended. For example, to add guidance support, a special kind of "guidance" rules may be introduced, which entail special runtime support that may be supplied via callback functions (see Section 3. Another kind of rules may weaken the constraining notion of condition and continue execution even if the condition is not satisfied (e.g., to permit and monitor process exceptions). In order to support such extensibility, the original language must have a basic rule categorization mechanism into which new "basic" types can be added.

The second aspect is the chaining directions and modes. As mentioned above, the two basic directions are *backward* chaining, i.e., firing rules which may satisfy an unsatisfied condition, and *forward* chaining, i.e., firing rules whose condition became satisfied (or enabled) as a result of asserting the effects of another rule. One extension in systems with one direction might be to add the other direction. However, such modification would be in general very hard to attain as an afterthought, as it requires to essentially reimplement the core of the interpreter. On the other hand, bi-directional chaining allows for two other hybrid extensions, both of which can be extended: forward chaining during backward-chaining, and backward chaining during forward chaining. The former may be attempted when the effect of a rule satisfies the conditions of other rules. Similarly, the latter may be used when a condition is partially but not completely satisfied by the previous rule's effect. Note that the extension may or may not involve syntax changes, depending on whether the chaining modes are selectively or unconditionally applied, respectively.

A third aspect combines the two above aspects, namely, static (matching) and dynamic (chaining) relationships between rules. An obvious static extension is to adapt the valid matchings between (pre-existing or extended) types of rules. For example, an interpreter extended with guidance rules may only allow guidance rules to be associated with other guidance rules or with mandatory system-exception rules but not with other ordinary rules. Similar extensions may include syntax to turn-on and turn-off chaining, wholesale or on a per-rule basis.

Another aspect of inter-rule relationship concerns the dynamic property of their execution. For example, the capability to *atomically* execute a set of linked rules may be added. This may involve syntax notations to define and or derive the atomicity boundaries, but ultimately requires interaction with and support of a transaction manager. Besides the fact that such transaction manager component must exist (or be implemented) in order to support atomicity, the interesting point from extensibility perspective is that the interpreter must enable interface to other sub-systems in the PCE, such as the process data- and transaction- managers, as well as enable integration with external sub-systems.

The final extensible property is the order of rule evaluation. Some rule systems employ breadth-first invocation, i.e., firing all enabled rules at one iteration, followed by all subsequently enabled rules in the following iteration, and so forth; other rule systems operate in a depth-first mode, firing an enabled rule followed by all rules which have become enabled as a result of its effects, and so forth. As with chaining direction, adding from scratch a basic evaluation-order would be hard, but extending the basic algorithm, e.g., to include both modes and supply syntax to determine which mode to apply, is a feasible extension. Another ordering policy may prioritize rule execution based on (extended) rules types.

3. Mechanisms for Metalinguistic Extensibility in Amber

Several key design issues enable Amber to support metalinguistic extensibility, i.e., to extend the syntax and semantics of its base language. To focus our discussion, we ignore many other aspects of the language/interpreter which are not closely related to supporting extensibility (for a complete account of Amber, see [26]).

Syntactic extensions can be made (only) by means of *rule annotations*. Annotations are strings that can be attached to the different sections of the rules and affect the (default) behavior before, during, or after the execution of the rule-section as well as the default chaining behavior into or from the rule. Once they are added to the language, they become "new" keywords. For example, a `weak-enforce` annotation in the condition of the rule may only raise a warning message when a condition is not satisfied, but otherwise continue the execution of the rule. Note that by introducing such annotation we give the *option* for process modelers to use weak-enforcement, but don't change the default behavior. One can also change the default of its Amber instance to be weak-enforcing, but such a change does not require any syntax changes, only semantic ones. The restriction to use annotations as means of syntactic extensibility provides a clean and relatively simple extension-interface with the interpreter, because the basic rule structure, and thus the skeletal logic of the rule-processor, are preserved. As a result, the number of entry points for code extensions is reasonably small, simplifying the extension task while still enabling to insert arbitrarily complex functionality in each of these entry points, as will be seen in Section 4.

Thus, all *semantic* extensions, including but not limited to those which correspond to syntax extensions, are also constrained to be made at well-defined specific entry points in the interpreter, essentially to ease the extensibility task and avoid complex control changes which might require knowledge of the interpreter-internals. Note that if such intimate familiarity is required, such "extensibility" is mostly worthless because it is limited only to the implementors of the interpreter. Recall that our goal is to enable process *administrators* to tailor the language/interpreter to fit their special needs.

To facilitate such modular extensibility, the interpreter is *iterative*, rather than a recursive one. (This is also the reason why Amber's rule interpreter was largely rewritten although the basic language is similar to the Marvel and Oz rule languages [17]). While the latter may be viewed as more natural for implementation of rule-processing due to the recursive nature of chaining, it is far less extensible due to the deeply intertwined and inter-dependent rule phases. Instead, the rule execution algorithm iterates over a sequence of a fixed rule-phase dispatches, discussed below.

Process engine extenders can insert/revise/replace their own functionality *between* phases, using a table-driven *callback* mechanism. The callbacks are made to a *mediator* that tailors the process engine's semantics and interfaces to other environment components, in a similar fashion to [30, 23]. The callbacks can interface with other sub-systems, Callbacks can break the sequential execution, and access, in a controlled manner, internal state of the interpreter. The proper callback function to invoke is determined by a combination of the current rule-phase, the (static) rule annotation (if any), and the (dynamic) state of the rule (such as whether the rule is in backward or forward chaining). This means that callbacks reside in a multi-dimensional function array. An interesting aspect of the callback array is that even the number of dimensions may be extended to allow for multiple state attributes.

Finally, since Amber supports context-switching among multiple tasks operating concurrently or sequentially on behalf of one or more clients, a mechanism is required to be able to extend the rule execution ordering policy. This is made possible by a parameterizable *multi-priority* queue, where a rule's priority is determined by its type, as defined by the (extended) rule annotation or by the default setting. Thus, by adding a rule type annotation and assigning to it a priority, the execution ordering can be extended.

We proceed with an overview of the language followed by the full rule execution algorithm.

3.1. Language Overview

Amber's PML is based in part on the Marvle Strategy Language (MSL) first developed for the Marvel process-centered environment [18] and later extended in Oz (with mostly semantic changes); most of the syntax, except for the extensible annotations, was previously elaborated in [3]. The PML incorporates an object-oriented data definition sublanguage for defining classes whose members represent process state and product artifacts (design documents, source code, test cases, etc.), and a rule sublanguage that specifies the actions that can be taken by a user or by the environment.

The fixed portion of the rule syntax is similar to the generic structure mentioned earlier, consisting of a rule signature (name and typed formal parameters); a binding section for retrieving object that are related to the parameters (derived parameters); a condition, specified as a compound first-order predicate logic over (derived and regular) parameters; an activity that indicates a shell script *envelope* which interfaces the PCE to external tools and executes with specified (possibly transformed) arguments; and a set of mutually exclusive effects, each of which matches exactly one of the possible return values from the tool envelope.

An Amber process server loads a collection of rules when

it starts up; different Amber instances may thus interpret different rule sets, representing different processes. The Loader utility parses a rule set and translates it into an efficient internal form, basically a rule network whose nodes are rules and whose edges represent matches between the condition (or bindings) of one rule and an effect of another; the network thus reflects all possible chaining.

```
setup_review[?p:PROJECT,
     ?design_doc:DESIGN_DOCUMENT]:

 # this rule prepares for a review of
 # one of the project's C files
 # against its design document.  It is
 # triggered first when a defect
 # is found in the C file, and then
 # afterward whenever the C file
 # is revised.

(and (bind MODULE ?m suchthat
       (member [?p.srcs ?m]))
     (bind CFILE ?c suchthat
       (and (member [?m.cfiles ?c])
            (?c.bug_status = Defected)))
     (bind DOCFILE ?d suchthat
       (member [?design_doc.docfiles ?d])))
    :
(forall ?c
    (or (?c.review_status = Revised)
            automation forward
        (?c.bug_status = Defected)
            automation forward))

{ REVIEW_TOOLS init_review
    ?c.change_request
    ?c.bug_report ?d.change_request
    ?d.bug_report }

(and
    (?d.review_status = ReviewRequested)
        automation forward
    (?c.review_status = ReviewRequested)
        automation forward);
```

Figure 1. Example Amber Rule

The rules of a process form the command set or interface of the Amber instance. The user client or an encompassing program issues requests to Amber that result in one or more rules being instantiated and evaluated. Amber supports by default both backward and forward chaining, as well as backward during forward and forward during chaining modes, although either of these modes can be altered or removed, either optionally (by introducing rule annotation) or globally (by changing the default behavior).

Figure 1 shows an example rule that triggers code inspection of a C file with respect to its design document. This rule is adapted from an Oz demo environment for the ISPW9 example process [25]. The rule specifies that the review is performed whenever either of the conditions (a buggy C file has been revised or a bug has been found in a C file) are satisfied. The assertions in the effects trigger further chaining that results in a groupware document inspection application being run, assisting the designer and coder to together inspect the code.

Notice the `automation forward` annotation which is attached to each predicate in the condition and each assertion in the effect. In Amber, each chaining type (in this case `automation`) is followed by `forward` and/or `backward` to indicate whether the chaining type applies during forward vs. backward chaining, respectively (i.e., `forward` and `backward` are built-in, but `automation` is not). Multiple annotations may be attached to the same predicate or assertion, e.g., both forward chaining into and backward chaining out of the same predicate may be supported, and/or multiple chaining types may be indicated. If no annotation is given, then normally there cannot be chaining to/from that predicate or assertion. This can be changed in a given instance, to default to "on" rather than "off" for any or all of the instance-specific chaining types (in which case `no_chain` annotations would need to be introduced). Particular chaining types may require various arguments. For example, we have devised a new chaining type we call `guidance`, which works like `automation` except that after instantiating a rule and satisfying its condition, the rule instance is placed into a persistent agenda ("to do" list), so one argument is *whose* agenda (user or group) [29].

3.2. Amber Rule Execution Algorithm

There are two auxiliary data structures, in addition to basic rules, that are used to manage rule execution: tasks and bulges. A *task* is a set of rules, together with all of their forward and backward chaining implications. It represents the context for all the rules invoked as the result of a top-level rule selection from a client (a human user or program). A *bulge* is also a set of rules, which represent a callback-specifiable dependency among a collection of rules, where either they must be executed in sequence, or a single rule must be chosen from the bulge to run while the rest are discarded.

Amber's rule execution algorithm consists of two intertwined parts: rule selection (or scheduling) and rule execution. During the execution of a rule, it may have to be suspended when waiting. In this case, its state is preserved,

it is context-switched, and a new rule, if any, is selected using the selection algorithm.

Rule Selection — New tasks (which correspond to newly issued commands) are always placed in the top-level (i.e. highest-priority) execution queue, to optimize interactive (user-issued) tasks. This policy is in fact hard-wired (perhaps unnecessarily). When a task has been instantiated, the rule interpreter runs the task until such time as it has no more rules ready to execute. Individual rules are marked as either "runnable" or "waiting". A task may not have any runnable rules because its has finished, or all of its rules are waiting, generally for either an activity or another rule to complete. Concurrent tasks are interleaved at the natural breaks afforded by activity invocations and backward or forward chaining.

If a runnable bulge is found, one of its runnable rules is selected for execution, and no more bulges are considered in this cycle. Each bulge is also checked to see if it contains a rule that is waiting for the completion of an activity. If no runnable bulge is found, but there is a waiting bulge, execution of the entire task is suspended to wait for the activity. Any rule that may remain to be executed is in a lower-priority queue than the waiting rule.

Rule Execution — As outlined earlier, rule execution consists of fixed phases, separated by callback entries.

1. *Enque* — A rule is instantiated with parameters and placed in a bulge in the appropriate queue, according to its priority.

2. *Begin* — An instantiated rule is selected from the current bulge.

3. *Bindings* — Local (to the rule) variable bindings are established via queries to the environment's data repository.

4. *Condition evaluation* — The rule's condition is evaluated. Backward chains may be instantiated and enqueued during this phase. A callback function may filter certain entries from being enqueued, or modify the entries to be enqueued.

5. *Waiting for backward chaining* — If required by the failure of the condition in this particular Amber instance, the rule interpreter attempts to perform all possible backward chains from the rule's unsatisfied predicates, until the rule's condition is satisfied or all possibilities have been exhausted. (This is the essence of `automation backward` chaining, which is built into Amber as the default chaining type for when chaining is permitted at all; a given Amber instance may completely disallow any form of chaining.) Clauses in the rule's condition are considered in the order determined by a chaining callback function, and are never reconsidered after satisfaction or exhaustion of backward chaining possibilities, to prevent infinite cycling (this decision could easily be changed, but is not currently parameterized by a callback).

6. *Activity initiation* — Once the rule's condition has been satisfied (or other implementor-defined requirements for "success" have been met), the rule's activity (a tool envelope) is sent to the originating client with the arguments derived for it in the rule.

7. *Activity completion* — The client returns a status code and other return values at an arbitrary later time when the tool envelope terminates.

8. *Effect assertion* — When the activity finishes, one effect is asserted according to the status code. etc.). Forward chains may be instantiated. (Automatic enactment of those chains is the essence of `automation forward` chaining, which is built into Amber as the default chaining type for when chaining is permitted; a given instance of Amber may support `automation forward` chaining, `automation backward` chaining, both or neither.) The ordering of forward chains emanating from a given effect is determined by a chaining callback.

9. *Waiting for forward chaining* — After an effect has been asserted, the rule is retained during any forward chains emanating from it, from its children, etc. Amber does not use the ancestors for anything, but another component of the environment might, e.g., in our Oz integration the transaction manager treats `atomicity` implications of a rule as nested subtransactions [15].

10. *End* — Clean up after chaining, e.g., free memory allocated to the rule and its chains. This is done *after* the rule *and* its forward and/or backward chaining "children" have finished, so the data structures are available for reference throughout the full chain.

Each rule instance contains a `state` field indicating one of these phases. Whenever a rule is invoked, the Amber rule interpreter runs through these states, performing each of the phases in turn, incrementing the state after each step, and invoking the phase-specific callback. Note that the waiting for backward chaining and condition evaluation phases may cycle, as each clause in the condition is considered. The callback functions may also request repetition of a phase or skipping ahead (e.g., to `end`, to terminate a rule early). Each phase callback is passed a pointer to the current rule instance, and can modify its `state` field as well as access the tree of parent and child rule instances through Amber's application programming interface (API) [21]; the rule also contains a "work area" where the callbacks can store a block of their own data for use by other callbacks.

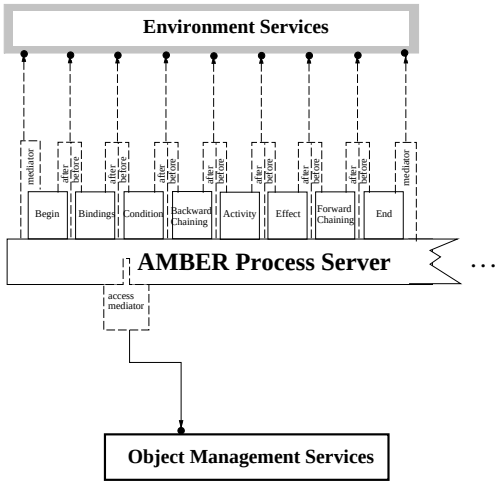

Figure 2. Amber Internal Architecture

3.3. Amber Callback Interface

Amber's rule execution phases are illustrated in Figure 2 Each phase has two callback functions: The *before* function is invoked before that phase is initiated; its return value determines whether or not the phase is actually performed. The *after* callback is executed after the phase completes, and is passed the return value from the rule interpreter's phase execution. Callback functions may skip or repeat a phase, augment a phase with additional wrap-around code, replace the phase completely with instance-specific functionality, or jump to any other phase. The reason for separating an after callback of one phase from the before callback of the next phase (as opposed to combining them) is due to the fact that callbacks can break the sequential execution and therefore a before callback may be invoked even though the previous phase (in the sequential order), and thus its after callback, have not been invoked.

We have also found it important to designate different callbacks for forward vs. backward chaining and for different chaining types, e.g., an instance implementor might not want to allow `automation forward` chaining during `automation backward` chaining, whereas he/she would almost certainly want to allow `atomicity forward` chaining during `automation backward` chaining (otherwise all atomicity guarantees would be thwarted). Thus there are separate callback functions for both forward and backward chaining with respect to each chaining type. Not all of the possible callback functions will be used in any particular Amber instance; some of them may never be used, but all options are provided in the interest of flexibility. The set of callback functions are specified in Amber's `action` array of function pointers. Currently this mediator code is linked into the Amber instance at compile-time; an improved version would support dynamic linking. Additional details can be found in [26].

4. Sample Extensions

4.1. Multi-Process Collaboration

Oz is a multi-site PCE that enables collaboration between heterogeneous and autonomous process instances running on homogeneous process engines. Each Oz environment has its own process model, data schema, objectbase and tools. User clients are always connected to their one "local" server and may also open and close connections on demand to "remote" servers. Servers communicate among themselves to establish *Treaties* — agreed-upon shared subprocesses automatically added on to each affected local process, and to coordinate *Summits* — enactment of Treaty-defined process segments that involve data and/or local clients from multiple sites. We stretch the International Alliance metaphor, since Treaties among sites precede and specify Summits rather than *vice versa*.

A Treaty consists of a set of shared rules exported by one site and imported by one or more other sites. A Summit occurs when a Treaty rule is enacted with actual parameters from two or more sites, as follows:

1. A user client selects the rule and provides arguments from its local objectbase and any of its open remote objectbases. The client's local server is said to be the *coordinating* site.

2. The rule parameters and variables are bound by the process engine at the coordinating site, with remote data automatically transferred to and cached at this site as needed.

3. The rule's condition is evaluated and any consequent chaining is performed by each allied environment with respect to its own data and according to its own local process. In particular, if a condition predicate is not currently satisfied on its argument(s), i.e., parameters and/or variables, the server where the offending object or objects reside performs backward chaining according to its own process rules to attempt to satisfy the predicate.

4. Once the condition is satisfied, the rule activity is run by the initiating user client.

5. When the activity completes, one of the rule's effects is asserted. Any implied chaining is performed at each site, again with respect to its own data and according to its own process. In particular, conditions of rules in the environment's own process model may be satisfied

by the assertions, so these rules are triggered, perhaps satisfying the conditions of other local rules, and so on.

6. After all such chaining has completed and the sites have synchronized, the original Summit rule may itself forward chain to one or more Summit rules. Then the cycle repeats, enacting forward chains among Summit rules in depth-first order.

The interleaving of local and Summit forward chaining is actually more complicated than presented above. First all local `atomicity` chaining is completed, then all Summit `atomicity` chaining, followed by local `automation` chaining and finally Summit `automation` chaining. The motivation for always completing local chaining of a given chaining type spawned from a Summit rule prior to any Summit chains due to that same rule, of the same chaining type, is to guarantee local consistency prior to initiating global operations. This is analogous to two-phase commit in distributed transactions, but also applies to non-`atomicity` chaining.

Although Oz had its own process engine, we decided to replace it with Amber to enable site administrators to extend their process engines. The interesting aspect of Amber/Oz from the extensibility perspective is that Treaties and Summits were added using the extension protocol rather than modifying Amber itself. We focus here on Summits.

4.1.1 Amber/Oz Implementation Details

Neither `atomicity` nor `automation` chaining, nor the concept of Summits, are built into Amber in any way. `atomicity` vs. `automation` is specified in the extended syntax, i.e., predicate annotations, and the semantics are implemented in the mediator callback code. Altogether, eight chaining types were defined in Amber's `action` array, from highest to lowest priority: Local `atomicity forward` chaining; Summit `atomicity forward` chaining; Local `automation backward` chaining; Summit `automation backward` chaining; Local `automation forward` chaining; Summit `automation forward` chaining; User-invoked local rules; User-invoked Summit rules.

Several callback functions work together to realize Summits. `after_eval` is invoked when a rule condition evaluates to false. It checks whether the rule is local or Summit, i.e., the rule is defined in an active Treaty and the rule's arguments include at least one remote object. `after_eval` does nothing for local rules. For Summits, the coordinating site requests remote backward chaining at each affected site, whose own object(s) failed a condition predicate. `after_eval` set the rule frame to a waiting state until all the remote backward chains complete. It flushes remote objects used by the rule from the objectbase cache before starting up remote chaining, since the remote chaining may modify these objects. `after_eval` is one of several callback functions where the Amber/Oz mediator interfaces to the environment framework's database.

`after_bc` is called at the end of a backward chaining cycle, to attempt to satisfy a condition predicate. If there is a failure of local backward chaining, for any reason (e.g., there might be a concurrency control conflict regarding necessary arguments for one of the chained-to rules), that failure is reported to the user client. In the case of Summit rules, however, remote backward chaining is still performed regardless of the success or failure of local backward chaining at the coordinating site. An alternative model would attempt any local backward chaining needed at the coordinating site first, and only if it succeeded in satisfying the relevant predicates, would any remote backward chaining (to fulfill the remaining unsatisfied predicates) be attempted. But that would reduce parallelism, and not necessarily reduce the apparently "unnecessary" work incurred via the remote backward chaining in such cases. We presume that in most cases the user really does intend to perform the requested Summit rule eventually, and will make arrangements for eventually fulfilling all the prerequisites and try again.

`set_chaining_types` is called when instantiating forward and backward chaining, with the list of new rule frames generated by chaining from a particular rule. When that chaining was ultimately triggered by a Summit rule (that is, a rule running as part of a Summit on a non-coordinating site), it sets the chaining type of the new rule frames to the Summit chaining type corresponding to the type that has been placed in the rule frames by the Amber process engine. For example, if a Summit `automation` rule frame generates `automation` and `atomicity` children, those children will have their types set to Summit `automation` and Summit `atomicity`. If it were not for this rather subtle callback function, remote rule chains triggered by a Summit at a non-coordinating site would not be recognized as part of a Summit by their local process engine.

The rest of the callback functions are concerned with transaction management. When `atomicity forward` chaining is instantiated between the currently running rule and the other rules whose conditions it satisfies, `enqueue_atomicity` is called for each chained-to rule to initialize a transaction nested within the parent rule's transaction, with the standard commit and abort dependencies between the transactions. In contrast, when `automation forward` chaining, `automation backward` chaining, or a top-level rule is instantiated, `enqueue_automation` is called. In the case where there is no parent rule frame (i.e., a top-level rule), it starts up a new top-level transaction. Otherwise, it starts up a transaction nested within the parent rule's transaction, but with no dependencies between the transactions. That is, each rule chained to during

`automation` is treated as an independent transaction, that can commit or abort separately from the rest of the enclosing rule chain, rather than as a nested subtransaction where all or none of them commit.

When a rule is selected for enactment, `summit_tx` is called for Summit rules and `tx_init` for all other rules. The process engine *is* able to distinguish Summit vs. local here, due to the "Summit" keyword in the chaining annotation. `tx_init` links the rule frame and the current transaction (recall that callback functions are passed a rule frame pointer). `summit_tx` prepares for a Summit by retrieving copies of any needed remote objects not already cached, and links the Summit rule and the current transaction. (`summit_tx` is called as a subroutine by `tx_init` for a top-level Summit rule, because then the process engine cannot distinguish local vs. Summit cases.)

`rp_acquire_locks` is called after the binding phase for both local and Summit rules. It traverses the lists of bound objects in the rule's parameters and local variables, acquiring locks on all of the objects that the rule might access. `finish_rule` is called after effect assertion. In the case of a local rule, it checks whether any "child" rule frames were generated by forward `atomicity` chaining. If there are no such children, it commits the transaction associated with the rule. But if there are any atomicity children, the transaction is left open until the `end` phase callback (`commit_rule` below). In the case of a Summit rule, the subroutine starts up remote forward chaining at all relevant sites, and then proceeds as for a local rule. It sets a site counter to the number of remote sites involved in the Summit, for later reference by `commit_rule`.

`commit_rule` is called at the end of rule execution. It commits any remaining transactions associated with the rule (if it had `atomicity` chaining children when `finish_rule` was called). `commit_rule` checks if the just-completed rule was the last remaining `atomicity` child of a parent rule and, if so, it commits the parent's transaction. For rules created by remote chaining during a Summit, it notifies the coordinating site that chaining is complete. For Summit rules (i.e., at the coordinating site), it decrements the site counter used for termination detection.

Oz's process engine was directly cognizant of locks and transactions. Although Amber/Oz uses the same transaction management component as the original Oz, described in [16], Amber itself knows nothing at all about locks or transactions; all code concerned with concurrency control and failure recovery of rules and rule chains is located in callback functions (or mediator utilities called by those functions).

4.2. Integration with micro-TeamWare

TeamWare [31, 32] represents the process model as a task graph, where nodes in the graph define process steps or tasks. After the process engine is informed that a task has been completed, a user selects among the connected nodes by traversing an outgoing edge (there may be several alternatives, iterations, etc.). Each node is associated with a script that may launch some external tool. The work of multiple users may be guided by the same task graph. TeamWare thus provides nice process visualization features. It is well-suited to modeling process topology, the "big picture" of concern to managers and other non-technical users, as well as clearly representing the workflow path and the choices for what to do next.

In contrast, Amber/Oz's process animation (inherited from Oz) graphically shows chaining from rule to rule as the chain unfolds, and the full rule network showing all possible chaining options can be displayed. But there is no "roadmap", the user has to *know* what to do next. (Only simple single-user processes can easily be written as one long forward chain, where the user is *told* what to do next, although we describe two different experimental extensions that would support this for multiple users in [19, 8].) Amber/Oz, and the original Oz, provide powerful execution facilities (e.g., both goal-driven and event-driven automation), and sophisticated multi-user/context-switching support (with collaborative concurrency control policies achieved through the mediator interface to an external transaction manager [15, 16]). But the user interface limitation has been a severe block to serious use of Oz (and Marvel before it) by anyone other than the development group.

Thus it seems valuable to *integrate* TeamWare and Oz, to exploit the advantages of both. We did not integrate Amber/Oz with the real TeamWare, which was temporarily out of order, but instead with our toy version called micro-Teamware (μTeamWare). We plan to later integrate Amber/Oz with Endeavors [9], the successor to TeamWare, following the same integration architecture.

4.2.1 micro-TeamWare Integration Approach

The gist of the integration, from the process perspective, is to use μTeamWare task graphs to model the process topology, the sequencing of tasks, that Amber/Oz users otherwise have to *know* off the top of their head. The users can then look at the status of the task graph to determine what to tell Amber/Oz to do next. μTeamWare activities correspond to entry points into Amber multi-rule chains, from which backward and/or forward chaining could occur, and perhaps also any follow-on rules directly selected by the user to complete the required work. Thus μTeamWare activities

would be relatively coarse-grained compared to Amber's individual rules.

We took advantage of Amber's extensibility to add *agendas* ("to do" lists), such that an enabled task intended to be performed by a particular user automatically appears in that user's agenda. The user can select an agenda entry for enactment at his/her convenience. Agendas are persistent, so that work could easily be assigned to a user who is not currently available. Finally, group (or role) agendas allow for any member of the group to select the entry, so it is not necessary to pick a particular user *a priori* when any user in the group (or who fulfills the role) will do. We had previously developed a similar notion of agendas [29], but it had never been integrated in the mainstream version of Oz — although we were able to port some of the old code to Amber. The agenda implementation, e.g., persistent maintenance of agendas, is implemented entirely in "mediator" code, not hard-wired into Amber.

The μTeamWare to Amber interface was simple: Amber's TCP/IP message interface, in the Oz mediator code, was extended so (1) a client can request that a particular instantiated rule (with parameters) be added to a specified agenda, and (2) a client can request the display of a particular agenda. Then the μTeamWare scripts, triggered when a task node is enacted, simply send messages to add rules to agendas. Regarding Amber's rule phases, the callback after the *Enque* phase jumps to the *End* phase. That is, the rule is not executed until later selected by a user from an agenda, when it goes through the phases normally.

An Oz graphical user interface client (based on Motif) was modified to display agendas in the form of menus, allowing the user to select instantiated rules from an agenda for enactment at his/her convenience. The user can request his/her own agenda, any other user's agenda, and/or one or more group agendas. The user interface also permits manual construction of agenda entries. There is currently no access control to limit agenda display or even such assignments to, say, the requisite user and his/her manager, but on the other hand there is nothing forcing a user to ever invoke any of his/her assigned tasks or preventing him/her from manually deleting them.

The Amber to μTeamWare interface was a bit more complicated: Amber needed to have some way to indicate to μTeamWare when a task node was completed and thus an outgoing edge could be followed. The obvious way to achieve this would seem to be through mediator callbacks. One model would be for the mediator to maintain state information regarding whether or not a given rule chain was triggered by selection of an agenda item that had been originally placed in the agenda by μTeamWare (as opposed to by a user manually). Then the *End* callback for the top-level rule, invoked after all chaining emanating from that rule has completed, would send a message back to μTeamWare . But this was problematic: Amber "knows" when a multi-rule chain completes, but does not "know" when a set of related chains is done. It is desirable to permit a μTeamWare task node to model larger process segments than might be appropriately implemented by a single rule chain, e.g., when edit-compile-debug iteration is needed, or for upstream activities like design, which may not map nicely to a sequence of tool invocations.

A better model, also using mediator callbacks, would be to invent a new annotation, that when included in the asserted effect, would be interpreted by the *Effect* callback function as signaling the end of the μTeamWare task. The rule containing this annotation might be automatically chained to, in cases where it is appropriate to implicitly end the task, or might be explicitly selected by a user when the task should only be ended at a human's discretion. However, this had the disadvantage of including specific knowledge about μTeamWare in the mediator code, whereas it would be preferable to keep agendas completely general — there are certainly other useful purposes for them unrelated to the μTeamWare integration.

So, we decided on a variant of the second model: instead of placing an annotation in the effect of a rule whose enactment is intended to signal the end of the encompassing μTeamWare task, that rule's activity indicates an envelope whose invocation would send the proper message to μTeamWare (such messages can also be tacked onto the end of regular tool invocation envelopes). The approach restricted all knowledge by Amber of μTeamWare and *vice versa* to their envelopes and scripts.

5. Contributions

We sketched an approach to extensible process enactment systems to enable addition/modification of the assistance a process-centered environment provides its users. We concentrated on the rule-based paradigm both because our experience lies there and because we had already shown that other higher-level process formalisms could easily be translated into rules for enactment. However, we plan future work to investigate extensibility for other process paradigms, notably task graphs.

We presented our realization of an extensible rule-based process modeling language and process interpreter in Amber. Amber's syntax follows closely the notation we had already developed for Marvel and Oz, with moderate changes. However, the rule interpreter was largely rewritten in a completely different style, basically iterative over a sequence of rule-phase dispatches, rather than recursive with the rule phases deeply intertwined. It was simple to insert the table-driven callbacks, and it should be relatively easy to make other parts of the interpreter parameterizable.

We had planned all along to replace Oz's process engine

with Amber, and then to add various new process assistance functionality to Amber/Oz such as guidance chaining and parallelized automation chaining. But the TeamWare integration idea came much later, and demonstrates the versatility of our extensibility approach to change the process enactment model in a way we had not originally envisioned: integration with a second process engine. We cannot present it here due to space constraints, but Amber's extensibility was also a significant factor in our experimental replacement of ProcessWEAVER's native Petri net-based process engine. The power and flexibility of the callbacks mediating between the two systems was particularly critical since ProcessWEAVER is a commercial product and we had no access to its source code or internal documentation; see [27].

Acknowledgments

Andrew Zhongwei Tong introduced our concept of agendas, which was reimplemented for Amber by Wenyu Jiang. Jack Yang assisted in integrating the Amber process server into Oz and making it practical for our daily use. George Heineman helped with Amber's mediator interface to the Pern transaction manager, and Wenke Lee wrote part of the Amber API manual [21]. Dick Taylor and Greg Bolcer provided useful suggestions and feedback regarding our development and integration of μTeamWare, based on their TeamWare version 1.0, and its integration with Amber/Oz.

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

References

[1] R. Balzer and K. Narayanaswamy. Mechanisms for generic process support. In D. Notkin, editor, *1st ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 21–32, Los Angeles CA, December 1993. Special issue of *Software Engineering Notes*, 18(5), December 1993.

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

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

[4] N. S. Barghouti. Supporting cooperation in the MARVEL process-centered SDE. In H. Weber, editor, *5th ACM SIGSOFT Symposium on Software Development Environments*, pages 21–31, Tyson's Corner VA, December 1992. Special issue of *Software Engineering Notes*, 17(5), December 1992.

[5] I. Ben-Shaul and G. E. Kaiser. *A Paradigm for Decentralized Process Modeling*. Kluwer Academic Publishers, Boston, 1995.

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

[7] I. Z. Ben-Shaul and G. E. Kaiser. An architecture for federation of process-centered environments. Technical Report CUCS-015-96, Columbia University Department of Computer Science, May 1996.

[8] I. Z. Ben-Shaul and G. E. Kaiser. Integrating groupware activities into workflow management systems. In *7th Israeli Conference on Computer Systems and Software Engineering*, pages 140–149, Herzliya, Israel, June 1996. IEEE Computer Society Press.

[9] G. A. Bolcer and R. N. Taylor. Endeavors: A process system integration infrastructure. In W. Schäfer, editor, *4th International Conference on the Software Process: Software Process – Improvement and Practice*, December 1996. In press.

[10] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. MIT Press, Cambridge MA, 1992.

[11] W. J. Gail E. Kaiser, Stephen E. Dossick and J. J. Yang. An architecture for www-based hypercode environments. Technical Report CUCS-037-96, Columbia University Department of Computer Science, August 1996.

[12] V. Gruhn and R. Jegelka. An evaluation of FUNSOFT nets. In J. Derniame, editor, *Software Process Technology Second European Workshop*, number 635 in Lecture Notes in Computer Science, pages 196–214. Springer-Verlag, Trondheim, Norway, September 1992.

[13] D. Heimbigner. The ProcessWall: A process state server approach to process programming. In H. Weber, editor, *5th ACM SIGSOFT Symposium on Software Development Environments*, pages 159–168, Tyson's Corner VA, December 1992. Special issue of *Software Engineering Notes*, 17(5), December 1992.

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

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

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

[17] G. T. Heineman, G. E. Kaiser, N. S. Barghouti, and I. Z. Ben-Shaul. Rule chaining in MARVEL: Dynamic binding of parameters. *IEEE Expert*, 7(6):26–32, December 1992.

[18] G. E. Kaiser and P. H. Feiler. An architecture for intelligent assistance in software development. In *9th International Conference on Software Engineering*, pages 180–188, Monterey CA, March 1987. IEEE Computer Society Press.

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

[20] M. I. Kellner and H. D. Rombach. Session summary: Comparisons of software process descriptions. In T. Katayama, editor, *6th International Software Process Workshop: Support for the Software Process*, pages 7–18, Hakodate, Japan, October 1990. IEEE Computer Society Press.

[21] P. S. Lab. *Amber Manual*, July 1996. ftp://ftp.psl.cs.columbia.edu/pub/psl/oz.1.2.manuals/V.Amber/.

[22] N. H. Madhavji and M. H. Penedo, editors. *Special Section on the Evolution of Software Processes*, volume 19:12 of *IEEE Transactions on Software Engineering*. December, 1993.

[23] D. Notkin and W. G. Griswold. Extension and software development. In *10th International Conference on Software Engineering*, pages 274–283, Raffles City, Singapore, April 1988. IEEE.

[24] L. J. Osterweil. Presentation at Software Process Architectures Workshop, March 1995.

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

[26] S. S. Popovich. *An Architecture for Extensible Workflow Process Servers*. PhD thesis, Columbia University Department of Computer Science, July 1996. CUCS-014-96.

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

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

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

[30] G. Wiederhold. Mediators in the architecture of future information systems. *Computer*, 25(3):38–49, March 1992.

[31] P. S. Young and R. N. Taylor. Human-executed operations in the teamware process programming system. In C. Ghezzi, editor, *9th International Software Process Workshop: The Role of Humans in the Process*, pages 78–81, Airlie VA, October 1994. IEEE Computer Society Press. Position paper.

[32] P. S. C. Young. *Customizable Process Specification and Enactment for Technical and Non-Technical Users*. PhD thesis, University of California Irvine, March 1994.