Logically Central, Physically Distributed Control in a Process Runtime Environment

Aaron G. Cass **Barbara Staudt Lerner**[*] **Eric K. McCall** **Leon J. Osterweil**
Alexander Wise

Department of Computer Science
University of Massachusetts
Amherst, MA 01003-4610
+1 413 545 2013
{acass, blerner, mccall, ljo, wise}@cs.umass.edu

ABSTRACT
An effective process definition language must be powerful, yet clear. It must also have well defined semantics to support powerful and definitive analysis. End users require that a runtime interpreter for the language faithfully implement the semantics used in analysis of process definitions, and that the interpreter be efficient and scalable. In addition to all of the above, the language, and its interpreter must also be readily evolvable. In this paper, we describe the architecture of Juliette, a process execution environment designed to address all of these requirements. We outline the tensions posed by these strong objectives and describe Juliette's modular approach and its novel distribution strategy, indicating how they address the tensions set by our objectives. While we explain the Juliette architectural approach in the context of the interpretation of Little-JIL, the approach applies to the interpretation of a broad class of process definition languages.

Keywords
Process, Workflow, Distribution, Static Analysis, Dynamic Execution

1 INTRODUCTION

Process technology can address important problems in such diverse areas as software engineering, workflow, and electronic commerce. For process technology to become widely accepted and applied, however, we believe that there are several important goals that must be reached. First, process descriptions must be precise, clear, and accurate descriptions of the work to be accomplished. Second, it must be possible to reason formally about the processes to ensure that they satisfy the needs of the process users. Third, the runtime execution of a process must faithfully implement the semantics defined in the process description. Fourth, the processes must effectively and efficiently coordinate the activities of humans and machines at runtime. We have addressed the first two of these goals in the design of Little-JIL. In this paper, we describe Juliette, a process execution environment for this language. Juliette meets the latter two goals, namely faithful semantic implementation and efficiency. Juliette does this by providing a logically centralized, physically distributed control component.

Our approach is based upon the premise that processes are profitably viewed as engineerable artifacts, and that their development, evaluation, and evolution can benefit substantially from borrowing approaches used in software engineering. Earlier explorations of this premise have focused on modeling, coding, executing and improving processes. With process improvement, the emphasis has been on dynamic analysis approaches, such as measurement and the gathering of data about the progress of process executions. We note, however, that in software engineering, quality assessment and improvement are more effectively pursued by a combination of dynamic and static analysis. Dynamic testing is a sampling technique, and while it can be quite useful in discovering defects and studying actual runtime behavior in specific contexts, it is useless in demonstrating the absence of defects. Static analysis complements the dynamic approach by offering the possibility of demonstrating the absence of certain classes of errors for all possible executions, and supporting reasoning about diverse sorts of properties. We believe that applying these complementary analysis approaches is even more important for processes than for standard software, as it is usually not feasible to perform myriad dynamic test runs using different test inputs and different interleavings of the concurrent activities. Furthermore, the

[*] Currently at Department of Computer Science, Bronfman Science Center, Williams College, Williamstown, MA 01267, +1 413 597 4215, lerner@cs.williams.edu

concurrent interleavings are likely to be even more varied in the process domain because the interleaving is determined by the behaviors of agents, which are likely to vary from one execution to another. It is therefore both less feasible to do extensive testing and more difficult to make general claims based on limited testing.

To be sure that static analysis assurances hold during executions of processes, however, it is necessary to be sure that the semantic models used in the static analysis of process definitions are also the semantic models used to support the dynamic execution of these processes. There are important and difficult tensions entailed in being able to achieve both goals for comprehensive languages. Certainly more comprehensive languages are more difficult to analyze. But also, semantics that are easy to reason about can sometimes pose complications for execution engines. Moreover, languages that are comprehensive and mechanically analyzable can often be incomprehensible to humans.

In other work [19] we have described the powerful semantics that Little-JIL includes to support the definition of processes and experiences that have indicated that Little-JIL is effective as a vehicle for specification of important process details. In still other work [3], we have succeeded in applying static dataflow analysis technology to prove properties such as dangerous race conditions and improper event sequencing. While this work demonstrates the possibility of applying powerful reasoning to process definitions, the results are only binding on actual executions if we are confident that Juliette correctly enforces Little-JIL's semantics.

In addition to ensuring faithful execution of the language semantics, we want Juliette to be acceptable to real users as a platform for supporting actual process execution. Thus, we have designed Juliette to support distributed process execution to support scalability. This further complicates reasoning about Juliette's accurate implementation of Little-JIL semantics. While the Little-JIL language architecture has clearly shaped Juliette, it is also true that the modular architecture of the system has had interesting impacts upon the semantic model of the Little-JIL language. In particular, Little-JIL adopts a modular approach to process definition, causing it to be amenable to important types of language evolution. But it also strongly suggests a model of distributed concurrent process execution by a heterogeneous collection of human and computer tool agents. Thus, the decomposition of Juliette into components, and its distribution and intercommunication architecture became important issues. This paper addresses particularly the decomposition structure, distribution strategy, and agent coordination approaches that we developed, and the interplay between the characteristics of Juliette and the semantic model of Little-JIL.

2 RELATED WORK

As noted, a central challenge in work such as ours is to balance the stated desire for a powerful process definition language that has precisely defined executable semantics with the substantial demands this inevitably places on an execution platform that must be faithful to the defined language semantics, while also being efficient and scalable.

Some researchers have opted to implement languages with relatively weak semantics in order to simplify the task of providing such an execution platform. For example IDEF0 [13] supports the development of process definitions that are essentially dataflow diagrams. The execution of such diagrams entails assuring the delivery of input and output artifacts to process steps and the proper sequencing of those steps. One major drawback here is that more complex execution semantics (for example, concurrency details and exception handling) cannot be represented by IDEF0.

Similarly, STATEMATE [8] supports the development of an integrated set of diagrams, which can be used to represent different views of a process (e.g., dataflow, control flow, and finite state machine). STATEMATE uses simulation to reason about runtime behaviors and static analyzers to detect dataflow inconsistencies, and certain types of inconsistencies among the different diagrams. Here too, however, the semantics of the STATEMATE diagram set are insufficient to represent clearly and precisely such important process features as exception management, resource contention, and concurrency control.

There has been considerable interest in devising process and workflow languages that are capable of representing broader and more detailed execution semantics than those in systems such as IDEF0 and STATEMATE. Typically these languages are decomposed into components each of which encapsulates different semantic features of the language. This can be seen, for example, in the Workflow Management Coalition's Workflow Reference Model [10], in which a workflow engine can incorporate the notion of a Role Model so that workflow process definitions need not specify details of how roles are to be performed. This frees the execution engine from having to implement the semantics required to define those details.

Some recent process programming languages have carried this idea still further, by demonstrating modular approaches to incorporating broader semantics while reducing execution platform complications [4, 19, 17]. In programs written in a modular language, different types of language semantics are addressed by different language modules, with the runtime behavior of the process realized through the coordination of separate components. For example, a process language might modularize the description of resource allocation policies and data management approaches in components that are separate from each other and from details of execution sequencing and concurrency. It has been argued that this approach benefits users of the language because they can comprehend their processes more readily by being able to focus their attention separately on different narrower aspects [17].

This approach has the additional advantage of allowing the different semantic modules of the language to be defined in different languages, each of which can presumably be selected to be more appropriate for the particular semantic feature.

This modularization of the process description language can be used to determine an appropriate modularization of the process execution environment. Thus, for example, recent versions of the APEL execution engine are implemented as separate components, where each component is responsible for a different aspect of process execution [5]. APEL uses a ProcessWall [9], a blackboard-like system that captures all runtime state about the process execution, for needed communication among components. When one component puts an item on the ProcessWall, the others can be notified through an event mechanism. Components register interest in the parts of the ProcessWall in which they are interested. This approach provides for easy component composability and directly supports the modularization of the process definition language. The way in which the ProcessWall is used for communication among components, however, has the unfortunate effect of reducing the power of the reasoning that can be applied to process definitions. In APEL, any component can view and respond to any part of the ProcessWall in whatever way it chooses. Thus, the range of possible sequences of executions of steps in an APEL process is extremely broad. This implies that execution models of APEL processes must encompass very wide classes of possible executions, which tends to reduce the possibility of obtaining conclusive analytic results and thus reduces the set of properties for which assurances can safely be assumed to hold at runtime.

APEL allows an optional control component to help address this problem [5]. APEL's control component can inhibit messages from the ProcessWall so that other components cannot act on them. This capability could be used to more directly route particular events to certain components of the system. However, since any component can generate events, this control component would have to know about many events and would therefore be difficult to maintain and evolve.

Endeavors [2] uses a similar approach to providing an open infrastructure for the development of process execution systems. The Endeavors architecture is based on a layered object model with well defined APIs and an event mechanism to notify the other layers of state changes within a layer. This layered architecture supports the separation and distribution of different process language modules. For example, it is possible to separate the process and artifact state servers [11]. Like APEL, Endeavors uses a blackboard-like structure (like ProcessWall), called the Foundation layer. Semantic services such as the interpreter, located in the System layer, register to receive state change messages from the objects in the Foundation. As with APEL, this approach provides for easy composability but distributes responsibility for the semantic correctness of a process execution among the services in the system layer. As with APEL this allows broad diversity in the possible execution orderings, and therefore reduces the possibility of obtaining conclusive results from static analyses performed on process definitions. An approach that combines the composability of APEL and Endeavors with the ability to also specify important details of control flow would offer the advantages of flexibility as well as the benefits of analyzability. Juliette offers this combination by providing a level of composability that is comparable to that offered by APEL and Endeavors, but Juliette also provides a logically centralized facility for interpreting Little-JIL's diverse and powerful control flow semantics to facilitate reasoning.

However, one problem with centralized control flow interpretation is that the interpreter might then become an execution bottleneck. Such a bottleneck would prevent the runtime system from being suitably efficient, and from scaling up to large, industrial-sized processes. A finer-grained distribution that allows the control component itself to be physically distributed would promise to be more scalable. But this needs to be done very carefully to maintain correctness. One finer-grained approach, with good prospects for efficiency and scalability, is to allow each task in the process definition to be represented by a different object at runtime, and to allow those objects to be distributed across different execution platforms, with the distribution determined at design or installation time.

The RainMan [15] system takes this approach by defining a process to be a system of tasks, where each task is defined by an object that implements a `PerformerAgent` interface, and this object can perform the task in whatever way, including calling other `PerformerAgents`. Because the different `PerformerAgents` are invoked by using a Remote Method Invocation (RMI) protocol, the `PerformerAgents` can reside on different execution platforms. Clearly, responsibility for controlling flow of execution is distributed in this case. Unfortunately, here again we note that the possible sequences of task execution are quite broad, as much of the responsibility for determining them is left to the separate tasks. Thus, the possibilities for conclusive static analysis of such process descriptions is limited. This approach is also taken by the ORBWork [16] system, in which each task is handled by a potentially distributed Task Scheduler that is implemented as a CORBA [14] object.

This approach takes advantage of the process language's facilities for specifying a process in terms of well defined units of work, thus defining a natural level of distribution granularity at runtime. However, with fine-grained distribution, it becomes particularly important to determine an appropriate distribution topology. Since the execution agents in the process represent an existing topology, it seems natural to use this topology to determine the distribution of the execution components. However, in situations where the agent locations are not known until runtime, or where the agents move, it seems unlikely that a statically-defined distribution topology of the execution objects will match the distribution topology of the agents.

In the case of such mismatches, the agents and execution objects are not co-located and therefore agents that need to interact with these objects will have to rely on potentially slow network communications. These mismatches also make it difficult to deal with agent mobility and detachment. Even when an agent is temporarily detached from the network, either by actively disconnecting or by establishing a low reliability connection like a wireless link, it is clearly desirable that the agent still be able to get work done. If the execution object with which the agent must communicate is co-located with the agent, its execution can continue without the need to reconnect to the rest of the process execution engine. Note also that co-locating a step execution agent with its interpreter seems to be inherently more equitable in that the interpreter for the step is using that step's computing resources, and does not need to borrow resources from elsewhere.

The RainMan and ORBWork approach makes it difficult to ensure that agents are co-located with their execution objects, thus losing the performance, mobility, and equitability benefits mentioned above. This is particularly true in the absence of an *a priori* specification of the agent distribution topology. In contrast, Juliette accomplishes fine-grained distribution and provides a mechanism for keeping the process execution topology and the agent topology congruent, while also supporting the language modularization that was previously described.

3 OUR APPROACH

While the Juliette runtime environment was designed for the execution of Little-JIL, we believe that the approach can be used for a variety of languages. It will simplify later discussions to give a quick introduction to Little-JIL, and thus also describe the characteristics of languages for which this approach seems appropriate.

A Quick Introduction to Little-JIL and Its Execution

Little-JIL is a process definition language intended to be used to describe the coordination between human, software, and hardware participants to accomplish a task. The various participants in a Little-JIL process are called agents, and they are assumed to be autonomous.

Little-JIL process descriptions are steps, representing tasks to be completed, hierarchically decomposed into substeps with data and control passing from step to step as the process executes. In order to more fully describe the coordination of agents to accomplish a task, a process description specifies the data and control dependencies between steps, but also the resource requirements of the steps that make up a process and the types of data that flow between steps.

In order to support language evolution and ease the development of a language execution engine, Little-JIL is defined as a modular language in that the semantics of a process definition is composed of different sets of semantics defined in different language modules. Therefore, in addition to the control and data flow expressed directly in the Little-JIL coordination model, a process definition must also incorporate the following definition components:

- A resource model describing the types and instances of resources the process uses, including the agents that will carry out the work,
- An artifact model providing type definitions for the data flowing through the process, and
- A set of agents to assume responsibility for the execution of the steps.

In Juliette, the Little-JIL coordination model is implemented directly by a logically centralized interpreter. Juliette also requires the use of a resource manager that coordinates the allocation of resources to the steps in the process. Juliette is implemented in Java and thus the Java type model and runtime system serve as Juliette's artifact model. . Juliette provides a mechanism for communicating with the agents that are otherwise external to Juliette and run as separate operating system processes, presumably on separate platforms.

The Little-JIL coordination model provides the glue that binds these components together by implementing the semantics of Little-JIL step execution, which includes specific details of interaction with other components. The Little-JIL step execution lifecycle is defined formally as a finite state machine. A simplified, informal description of the key states and transitions of the finite state machine, and how they are executed by an interpreter, follow[1]:

- *Elaboration Phase*
  - The interpreter copies parameter values in from the step's parent based upon parameter bindings declared in the Little-JIL process definition.
  - The resource manager acquires an agent that matches the specification present in the Little-JIL process definition.
  - The interpreter assigns the step to the agent.
- *Starting Phase*
  - The agent chooses to start the step. When this is done is entirely up to the agent.
  - The resource manager acquires (any additional, non-agent) resources that match the specifications present in the Little-JIL process definition.
- *Execution Phase*
  - If the step is a leaf step, the agent to which the step is assigned will perform the task for the step and indicate when it is finished. Performance of the task is entirely up to the agent.

[1]For simplicity, a number of Little-JIL language features have been left out of this discussion. See [19, 18] for details.

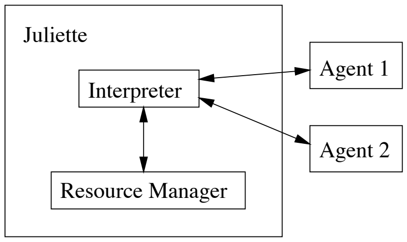

Figure 1: Logical Architecture of Juliette

  - If it is a non-leaf step, the interpreter drives the execution of each of the substeps using this same step lifecycle.
- *Finishing Phase*
  - The interpreter copies the values of the out parameters from the step to its parent as specified in the Little-JIL process definition.
  - The resource manager releases resources no longer needed.

Note that in this lifecycle, we have defined specific points at which the interpreter must interact with other components. It is important to note that these interactions are through well-defined APIs that restrict the types of information communicated. For example, the API for the resource manager involves only the identification, acquisition, and release of resources. The resource manager cannot affect the data or control flow in the process because the interpreter only interacts with the resource manager through an API that does not expose these aspects of the language semantics. Similarly, the API used by the interpreter to communicate with agents is only concerned with the assignment of steps and notifications of when the steps are started and completed. Also, because the interaction between the interpreter and the agents is asynchronous, the agents cannot "steal control" of the process. So, even though the autonomous agent decides when to start and stop the step, the interpreter handles all flow of data and control into and out of the step. In this way, the interpreter assures that the control and data flow semantics of Little-JIL are enforced. In addition to enforcing the language semantics, this frees the interpreter from having to take responsibility for any semantics other than those specified as part of Little-JIL itself (i.e., agent coordination and data flow semantics). For example, the interpreter need not know the resource management policies or artifact type structure in order to perform its duties.

As shown in Figure 1, the interpreter is central to the Juliette architecture. It reads in a Little-JIL process definition and executes the coordination and data flow semantics by assigning work, and effecting data flows, to the other components at appropriate points in the step lifecycle.

While Juliette has been implemented to execute Little-JIL process definitions, the high-level architecture of Juliette seems to us to be quite appropriate for a variety of modular, step-hierarchic process description languages with well-defined step lifecycle semantics.

A Modular Execution Environment

As mentioned earlier, modularization of the language can be used to determine an appropriate modularization of the execution environment. This is one way of controlling the complexity of the execution environment, thus enabling complex language semantics without adversely affecting our ability to execute programs written in the language.

For example, in Juliette the responsibility for resource handling semantics is assigned to a resource manager (see Figure 1). The interpreter communicates with the resource manager at the appropriate points in the execution of a step. This is similar to APEL [5] in that different parts of the common model are in different modules. However, because the step lifecycle executed by the interpreter defines when the different execution components are used, the interpreter has control over the interactions between the components. Thus the control flow semantics of the language are executed by the interpreter. This assures that Juliette's execution of a Little-JIL process implements Little-JIL's semantics, thus making it easier to ensure that properties verified by static analysis must always hold at runtime.

Clearly other language modules can also be implemented as different components in an execution architecture and the interpreter can interact with them also at the appropriate points in the lifecycle. The power of this approach is that the interpreter, while concerned with control and data flow semantics, need not be concerned with the semantics of the other language modules. Furthermore, this approach is not limited to Little-JIL, but should be effective as the architecture of an execution environment for any language with separate modules and a well-defined control and data flow module.

The Communication Mechanism

Recall that our goals include allowing for scalability and efficiently handling both expressed and implied parallelism in process definitions. This suggests that it should be possible for the resource manager and the various agents to each reside in a different address space and communicate with the interpreter via some inter-process communication mechanism. However, this mechanism should also support ease of composability and enforce the APIs on which the interpreter depends. To satisfy these varying requirements, Juliette uses an Agenda Management System (AMS) [12].

The AMS is implemented on top of a Distributed Object Substrate (DOS) that provides on-demand caching of objects and a cache coherence protocol to keep the objects consistent. The DOS also supports object migration: when a method is called on a distributed object, the object gets cached wherever the method was called and a thread of control on that node executes the method.

The AMS manages a collection of distributed *agendas* (lists of tasks to be performed), one of which is allocated to each agent and component in Juliette. The AMS provides a vehicle for the passing of messages between the agents and components that relieves senders of having to know about the physical locations of recipients. Thus for example, during the elaboration phase of a step's lifecycle, the interpreter uses the AMS to assign the step to an agent. Figure 2 shows the interpreter's `postItem` method used to assign an *agenda item*, representing the step to execute, to an agent by posting the item on the agent's agenda. First, the interpreter determines which agent is responsible for the step. Next, it asks the DOS for the agent's agenda. Only the DOS knows where this distributed object actually resides. It then sets the status of the item to indicate that the item is newly posted and adds the item to the agent's agenda.

The AMS also provides an event-based notification mechanism implemented with the observer pattern [6, page 293]. Communicating agents and components can register interest in agendas or agenda items by calling the `addPropertyChangeListener` method on the item or agenda, and when these objects change, all registered agents and components, called listeners, will be notified by calling their respective `propertyChange` methods[2].

Figure 2 also shows an agent's code for responding to an item being placed on its agenda[3]. Since the agent has registered interest in its own agenda, its `propertyChange` method is called when the item is added to the agenda. In this method, it first checks to see if the reason it was called was because a new item was added to its agenda. If so, it sets the item's status to `Starting` to notify the interpreter that it wants to start the step. Before doing so, it adds itself as a listener[4]so that it is informed when the interpreter gives it the go-ahead to continue. It is only necessary to do this for leaf steps because only leaf steps are actually executed by agents (agents merely supervise the coordination of substeps in non-leaf steps) .

When the status of the item is set, this causes a property change notification to be fired, thus calling the interpreter's `propertyChange` method. As shown in Figure 2, this method first checks to see if an agent is attempting to start a step. If so, it acquires the resources for the step. The code for acquiring resources is not shown. As shown in Figure 3, the resource manager also has an agenda. The code for acquiring resources communicates with the resource manager by placing requests on the resource manager's agenda and listening for an event indicating that the resource manager completed that request. Also note that we have omitted the

[2]This is the same API as is used by JavaBeans [7], but the implementation is customized to provide distributed event notification.

[3]For human agents, an *agenda viewer* allows the human user to interact with the system by modifying its agenda items.

[4]The agent uses addTransientChangeListener which is implemented so that the agent is not migrated when it receives update notifications.

Methods from the interpreter class:

```
postItem (Item theItem)
{
  String agentName = theItem.getAgentName();
  Agenda theAgenda = Directory.getAgenda (agentName);

  theItem.addPropertyChangeListener(this);
  theItem.setStatus ("Posted");
  theAgenda.addItem (theItem);
}

propertyChange (PropertyChangeEvent e)
{
  if (e.getSource() instanceof Item
      && e.getPropertyName.equals("setStatus")
      && e.getNewValue().equals("Starting")) {
    // An agent wants to start a step

    Item theItem = (Item) e.getSource();

    // ... code to acquire resources

    // Tell the agent that it is ok to continue.
    theItem.setStatus ("Started");
  }
}
```

An agent's `propertyChange` method:

```
propertyChange (PropertyChangeEvent e)
{
  if (e.getSource() instanceof Agenda
      && e.getPropertyName.equals("addItem")) {
      // a new item was added to my agenda

      Item theItem = (Item) e.getNewValue();
      if (theItem instanceof LeafItem) {
        // Listen to the new agenda item,
        // need to react when the interpreter
        // says ok to execute step.
        theItem.addTransientChangeListener (this);
      }

    // Tell the interpreter I want to start the step
    theAgendaItem.setStatus ("Starting");
  }
  else if (e.getSource() instance of LeafItem
           && e.getPropertyName.equals("setStatus")
           && e.getNewValue().equals("Started") {
    // Interpreter says OK to execute the step

    LeafItem theItem = (Item) e.getSource();

    // Do the step-specific activity
    executeStep (theItem);

    // Tell Interpreter I'm done with the step
    theItem.setStatus ("Completing");
  }
}
```

Figure 2: Assigning work to an agent

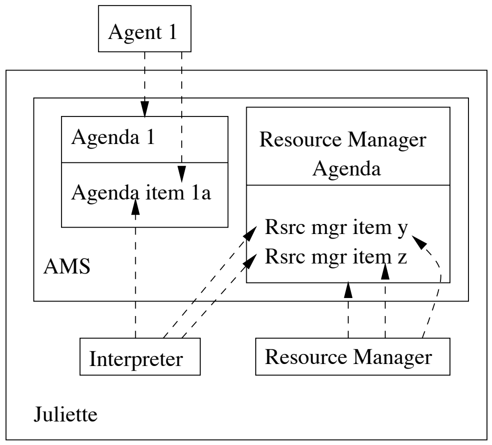

Figure 3: Agenda management view of Juliette

error handling that would occur if the resource request could not be satisfied. Once resources are acquired, the interpreter sets the status of the item to `Started` to indicate that the step can proceed. In the case of a leaf step, the agent will be notified, thus having its `propertyChange` method called again. Returning to the agent's `propertyChange` method shown in Figure 2, we see that the agent reacts by performing the step. Exactly what is involved here depends on the specific step, of course. Finally, the agent communicates back to the interpreter that it is done by setting the item's status to `Completing`.[5]

By registering interest in, and updating, distributed agendas and agenda items, we can easily run the agents and resource manager in processes separate from the interpreter. This also gives us great flexibility in the distribution of these processes across a network of machines. The interpreter remains in charge of orderly execution of steps since it is an intermediary in all communication and carefully steps through the state machine defined by the semantics of Little-JIL.

Distribution of the Interpreter

Even though the AMS allows the resource manager and the agents to each reside on different platforms, the interpreter component still represents a potential bottleneck. All work requests originate from the interpreter and many of the events generated by the AMS are handled by the interpreter. Because processes are inherently distributed and parallel, this bottleneck can quickly become unacceptable.

Our approach is to decompose the interpreter into components that can be distributed around the network. As we've seen, RainMan [15] and ORBWork [16] establish a fine-grained granularity by assigning a distributable runtime object to each task in the workflow, thus enabling a scalable architecture. This approach is possible because the language semantics define a task as a well-defined unit of work. Our similar approach is to use the language semantics to help determine the distribution granularity.

The step semantics of Little-JIL represent a good unit of distribution granularity for at least two reasons:

- A step represents a well-defined unit of work. The step concept bundles together resource acquisitions, parameter assignments, and a task description so that each step can be executed independent of the step's parents or siblings.
- A step's completion is determined only by the completion of its substeps, if there are any, or by agent actions if there are no substeps. This minimizes the amount of information a step needs about the global state of the process.

As mentioned above, agents in a process get work requests in the form of steps that appear as items on their agendas. When an agent completes a step, it must notify the interpreter. At the same time, other steps may be executing and need to communicate with the interpreter. In order to take advantage of this inherent parallelism in process execution, our approach has been to associate a simple *step interpreter* with each step instead of using a complex central interpreter for the whole process definition.

In Juliette, during the elaboration phase of a step's lifecycle, the parent step's interpreter creates a step interpreter for the step[6]. This step interpreter immediately registers interest in receiving update notifications from the agenda item representing the step. Therefore, when the agent starts the step, this step interpreter is already listening for updates and can respond appropriately.

We also take advantage of the hierarchical structure of Little-JIL process definitions to keep each step interpreter simple. Because the completion of a step in Little-JIL is determined only by the completion of that step's substeps, if it has any, we've designed the step interpreter such that it responds to events only of its own step and substeps. This minimizes the amount of global knowledge that each step interpreter requires.

Even though each step's agent needs to communicate with "the interpreter," it does this by communicating with its own step interpreter and, eventually, with the step interpreter of its parent. Since there is exactly one step interpreter for each step, we can unambiguously control how the process proceeds.

Effecting the Desired Distribution Topology

Now that we have an appropriate unit of distribution gran-

[5]From this example, it might appear that there is a great deal of code that one must write to turn a piece of software into a software agent suitable for integration into a Little-JIL process. In fact, the code shown above is defined in an abstract `Agent` class and can be inherited by real agents. Only the `executeStep` method must really be defined by the individual agents.

[6]The root step is a special case because it has no parent. A special `ProcessStarter` object creates the root's step interpreter.

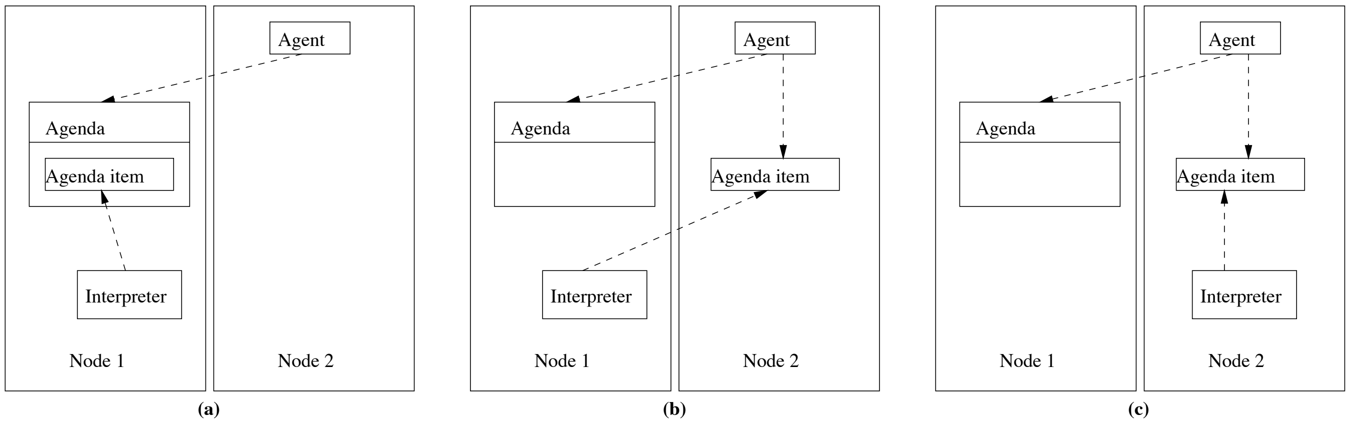

Figure 4: Migration of the interpreter

ularity, we must address the question of how those units should be distributed. In other words, we need to determine an appropriate distribution topology for the interpreter. We already have one topology inherent in the problem: the topology of the agents. Recall that agents communicate with step interpreters by setting the status of the agenda items representing steps. To reduce communication overhead, it therefore seems sensible to co-locate the step interpreter with the agenda item it is observing and therefore also with the agent executing that agenda item. However, a process designer should not have to know where the agents will be at runtime and furthermore, the agents might move during the process execution. Since agents may be mobile, we therefore want to allow agenda items and step interpreters to be mobile as well. In Juliette, agents, their agenda items, and their step interpreters are co-located without an *a priori* specification of the agent locations. This is accomplished by taking further advantage of the Distributed Object Substrate on which the AMS is built.

The communication between agenda items and step interpreters is facilitated by setting up appropriate observer or listener relationships and by allowing the agenda items and step interpreters to migrate from node to node as needed. Thus, when methods are called on the agenda items or step interpreters, they migrate to the node from which the method call originated.

Figure 4 shows this migration in action. First, an interpreter creates an agenda item to correspond to a step being elaborated and also creates a step interpreter for that agenda item. The new step interpreter is a listener to the new agenda item. The agenda item is placed on the appropriate agent's agenda by calling `addItem` on the agenda. This causes the agenda to migrate to the node where the new agenda item was created. Figure 4a shows the state at this point. The agenda and the new agenda item and step interpreter are all on one node. The agent on the other node is a listener of its own agenda, which is now on the first node.

Because the agent is observing the agenda it is notified that a new item exists. The agent can then choose to start the step represented by the item by changing the item's status. This is a call to the `setStatus` method on the agenda item, which causes the agenda item to first migrate, as seen in Figure 4b, before `setStatus` is called locally. Note that the agenda does not migrate to Node 2 since the agenda is not modified by the agent's actions. Instead, the agenda on node 1 now references a cached copy of the new agenda item. Node 2 has a cached copy of the agenda.

After the call to `setStatus`, the status change results in an update notification to the step interpreter. Since the update notification is accomplished via a call to the `propertyChange` method, the interpreter migrates to the agent's node (see Figure 4c). Now, future communication between the step interpreter and the agenda item is local. When the step interpreter creates substeps, they will originate on this node. If these substeps are assigned to the same agent, as we expect to happen frequently, the interaction with the agent for those substeps will also be local. Thus, migration only occurs when a step and one of its substeps uses a different agent running on a different node.

Using this approach, we have been able to achieve fine granularity distribution and at the same time keep the step interpreters co-located with the agents with which they must communicate.

4 EVALUATION

The implementation of Juliette had two major goals: First, it needed to support the faithful execution of the language semantics used as the basis for static analysis, and second it needed to be scalable and efficient.

Faithful Execution of Semantics

As discussed earlier, Juliette adopts a modular approach in which the resource and artifact models are implemented as separate modules and are separated from the interpreter, which is responsible for implementing the coordination model semantics. The semantics of the coordination model are expressed in terms of a finite state machine, whose behaviors are rather straightforwardly implemented. This approach has proven effective in simplifying the execution of the semantics of the coordination model.

It has had another very desirable effect in that it has also facilitated evolution. Because we have implemented the separate Little-JIL semantic features, such as artifact and resource management, as separate components, not only can we change their implementations, but we can also change their syntax or semantics by providing a new manager that implements the new syntax or semantics.

In fact, we have made numerous changes to the semantics of the coordination model, usually primarily by modifying the finite state machine model. In addition, we are currently in the process of implementing a new resource model and resource manager that is expected to seamlessly integrate into the existing Juliette architecture.

Efficient Interpretion
Because we have distributed the interpreter, and co-located the step interpreters with the agent with which they interact, the interpreter does not represent the bottleneck about which we had been most concerned. We can still view the interpreter as logically centralized even though it is not physically centralized. Thus, an agent that is communicating with the interpreter does not have to contend with other agents also needing to communicate with the interpreter.

Also, as noted before, an approach that co-locates the interpreter and the agent gives a measure of interpretation equitability. When an agent is performing part of a Little-JIL process in the Juliette runtime environment, the interpreter is using the agent's computing resources, not those of another agent.

5 FUTURE WORK

In addition to supporting efficient interpretation, we expect our distribution approach to give additional advantages. We plan in the future to demonstrate the effectiveness of distribution toward other goals, including robustness, mobilility, and scalability.

Robustness
If the process execution engine had a physically centralized interpreter, then the entire system would have a key single point of failure. However, in a decentralized approach such as Juliette, there is the hope that the parts of the process that are not being interpreted on that platform can continue. Our approach is designed to allow this by distributing the step interpreters around the network and thus eliminating the single point of failure. However, the current cache coherence protocol used by the DOS does not deal with failure of network nodes or data transport. We are currently experimenting with checkpointing of process executions as an approach to reduce the impact of a failure, but future work is needed to improve the caching protocols.

Mobile computing
In a centralized interpreter model, if one of the agents in a process has disconnected from the network, or is on a lower reliability connection like a wireless link, then there will be times when the mobile agent is unable to contact the interpreter's node. However, having the interpreters with which the agent must interact co-located with the agent, as in the Juliette approach, makes it possible to allow temporary disconnection. However, advanced planning is required, for example to make sure that resources will be available to substeps if they are needed while the agent is disconnected. We have developed a framework of mobility issues [1], and plan to continue research in that direction. Also note that a solution to the mobile computing problem requires first a solution to the robustness problems mentioned above.

Scalability
We have briefly experimented with comparing a centralized approach to the distributed approach we present here. With all objects, including agenda items, agendas, and step interpreters, created on one node and not migrated during execution, the system is noticeably slower to respond to agent requests. We have also begun to experimentally compare our approach to one in which the agendas and their items migrate but the step interpreters are centralized.

While our approach is promising and our implementation is efficient, there is still more work to be done to achieve high performance and scalability. As previously mentioned, objects implemented using the DOS migrate for every call to a method that might change the state of the object. Some method calls, however, need not cause a migration. For example, in the current implementation, when a parent step's interpreter is notified that a substep has changed, the parent step's interpreter migrates because its `propertyChange` method is called. In the case of steps with many substeps that execute in parallel, this parent migration can cause performance degradation. We are beginning to look at changing the migration policy.

6 CONCLUSIONS

In this paper, we have described an architecture that we have implemented for a process runtime environment. There were two main goals of this work: to faithfully implement the process language semantics and to efficiently execute those semantics.

As processes become more and more distributed, involve more and more parallelism, and involve more and more autonomous and heterogeneous agents, the complexity of process understanding will surely rise. In order to control this complexity and ensure that processes behave is ways we wish them to, some sort of analysis is required. In the process domain, it is usually not feasible to perform myriad dynamic test runs, and it will become less and less feasible for more complex processes. This suggests that static analysis is therefore quite important. And as static analysis results become available, we will need to ensure that those results are binding on all executions. By taking advantage of the modules of the process definition and giving the coordination model a logically central role in the architecture, the approach we describe allows us to ensure that static analysis results based on the control and data flow semantics of a process description are true at runtime.

Yet, even if the process provably behaves according to properties of interest, if the process execution engine is inefficient, it will not be adopted. This is particularly true because processes are inherently parallel and therefore, numerous autonomous agents will need execution services simultaneously. By taking advantage of the well-defined unit of work of the process definition language, our approach achieves a fine-grained parallelism for process execution. By using a novel communication mechanism, our approach also enables agents and the process execution fragments with which they must communicate to be co-located without an *a priori* specification of agent locations. In addition to efficiency, this approach was designed to support robustness, agent mobility, and scalability.

While this approach has been implemented to execute Little-JIL semantics, the high-level architecture of Juliette is appropriate for a variety of modular, step-hierarchic process description languages with well-defined step lifecycle semantics.

ACKNOWLEDGEMENTS
We would like to thank Rodion Podorozhny for his efforts in developing the resource management component of Juliette.

We also acknowledge the support of the Defense Advanced Research Projects Agency, and Rome Laboratories under grants F30602-94-C-0137 and F30602-97-2-0032.

REFERENCES

[1] S. Bhattacharyya and L. J. Osterweil. A framework for relocation in mobile process-centered software development environments. Technical report, Univ. of Massachusetts at Amherst, Aug. 1996.

[2] G. A. Bolcer and R. N. Taylor. Endeavors: A process system integration infrastructure. In *Proc. of the Fourth Intl. Conf. on Software Process*, Dec. 1996.

[3] J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil. Verifying properties of process definitions. Technical Report 99-63, Univ. of Massachusetts at Amherst, Nov. 1999. Submitted to ICSE 2000.

[4] S. Dami, J. Estublier, and M. Amiour. APEL: A graphical yet executable formalism for process modelling. *Automated Software Enginnering*, Mar. 1997.

[5] J. Estublier, M. Amiour, and S. Dami. Building a federation of process support systems. In *Proc. of Int'l Joint Conf. on Work Activities Coordination and Collaboration*, Feb. 1999. San Francisco, CA.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[7] G. Hamilton, editor. *JavaBeans API Specification Version 1.01*. Sun Microsystems, July 1997.

[8] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Trans. on Software Engineering*, 16(4):403 – 414, Apr. 1990.

[9] D. Heimbigner. The ProcessWall: A process state server approach to process programming. In *Proc. of the Fifth SIGSOFT Symp. on Software Development Environments*, pages 159–168, Dec. 1992.

[10] D. Hollingsworth. The Workflow Reference Model. Document Number TC00-1003, Issue 1.1, Workflow Management Coalition, Jan. 1995.

[11] P. J. Kammer, G. A. Bolcer, R. N. Taylor, and A. S. Hitomi. Supporting distributed workflow using HTTP. In *Proc. of the Fifth Intl. Conf. on Software Process*, June 1998.

[12] E. K. McCall, L. A. Clarke, and L. J. Osterweil. An Adaptable Generation Approach to Agenda Management. In *Proc. of the 20th Intl. Conference on Software Engineering*, pages 282–291, Apr. 1998.

[13] National Institute of Standards and Technology (NIST). *Integration Definition For Function Modeling (IDEF0)*, 1993. Federal Information Processing Standards (FIPS) 183.

[14] Object Management Group. *Common Object Request Broker Architecture*, July 1995.

[15] S. Paul, E. Park, and J. Chaar. RainMan: A workflow system for the Internet. In *Proc. of the Usenix Symp. on Internet Technologies and Systems*, 1997.

[16] A. Sheth and K. J. Kochut. Workflow applications to research agenda: Scalable and dynamic work coordination and collaboration systems. In *NATO ASI on Workflow Management Systems and Interoperability*, Aug. 1997. Istanbul, Turkey.

[17] S. M. Sutton, Jr. and L. J. Osterweil. The design of a next-generation process language. In *Proc. of the Joint 6th European Software Engineering Conf. and the 5th ACM SIGSOFT Symp. on the Foundations of Software Engineering*, pages 142–158. Springer-Verlag, 1997. Zurich, Switzerland.

[18] A. Wise. Little-JIL 1.0 Language Report. Technical Report 98-24, Univ. of Massachusetts at Amherst, Apr. 1998.

[19] A. Wise, B. S. Lerner, E. K. McCall, L. J. Osterweil, and S. M. Sutton Jr. Specifying coordination in processes using Little-JIL. Technical report, Univ. of Massachusetts at Amherst, Nov. 1999. Submitted to ICSE 2000.