Specifying Coordination in Processes Using Little-JIL

Alexander Wise[⋆] **Barbara Staudt Lerner**[⋆⋆] **Eric K. McCall**[⋆] **Leon J. Osterweil**[⋆]
Stanley M. Sutton Jr.[⋆]

[⋆]Department of Computer Science
University of Massachusetts
Amherst, MA 01003-4610 USA
+1 413 545 2013
`{wise, mccall, ljo, sutton}@cs.umass.edu`

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

ABSTRACT

Little-JIL, a new language for programming coordination in processes is an executable, high-level process language with a formal (yet graphical) syntax and rigorously defined operational semantics. The central abstraction in Little-JIL is the "step." Little-JIL steps serve as foci for coordination and provide a scoping mechanism for control, data, and exception flow and for agent and resource assignment. Steps are composed hierarchically, but Little-JIL processes can have highly dynamic structures and can include recursion and concurrency.

Little-JIL is based on two main hypotheses. The first is that the specification of coordination control structures is separable from other process programming language issues. Little-JIL provides a rich set of control structures while relying on separate systems for support in areas such as resource, artifact, and agenda management. The second is that processes can be executed by agents who know how to perform their tasks but will benefit from coordination support. Accordingly, each step in Little-JIL is assigned to an execution agent (human or automated); agents are responsible for initiating steps and performing the work associated with them.

This approach has so far proven effective in allowing us to clearly and concisely express the agent coordination aspects of a wide variety of software, workflow, and other processes.

Keywords
Process, process programming, Little-JIL, workflow, coordination

1 INTRODUCTION

There is a growing need for process and workflow specification in many contexts. This is evidenced by both a growing marketplace as well as a thriving research community. In this paper we present Little-JIL, a process language that attempts to resolve the apparently conflicting objectives of providing constructs to support a wide variety of process abstractions such as organizations, activities, artifacts, resources, events, agents, and exceptions, which can easily make a language large and complex, and creating a language that is easy to use and understandable by non-programmers.

Little-JIL is strongly rooted in our past research on process programming languages [25, 26], but it makes some important breaks with this earlier work. Of primary importance for this paper is the focus on the *coordination* of activities and agents and the premise of process language *factoring*. Process language factoring is the separation of various semantic elements of a process so that they can be treated independently.

Coordination, as defined by Carriero and Gelernter is "the process of building programs by gluing together active pieces" and is a vehicle for building programs that "can include both human and software processes"[6]. From this perspective, it can be seen that coordination is a logically central aspect of process semantics and is an especially important focus for a factored process language.

As with Linda [6], in Little-JIL we have separated coordination from other language elements. Unlike Linda, which is made up of a set of common primitives for the construction of multiple coordination paradigms and removes all computational elements, in Little-JIL we have selected a single higher-level coordination paradigm that we believe fits naturally with the domain of process and workflow specification and included a small set of computational constructs to allow the programmer to further refine the ways in which the major computational elements interact. Furthermore, the paradigm we have selected serves as a natural focus to which other factors (such as artifacts, resources, and agents) can be related and through which the use of these factors can be orches-

Draft of paper submitted to ICSE 2000.

trated.

Little-JIL also differs from our prior work in that it is primarily a graphical language. This helps to promote understandability, adoption, and ease of use. However, Little-JIL language constructs are still defined using the sort of precise semantics that is more typically associated with textual languages. This is facilitated in part because the focus of the language is narrowed to coordination-related elements.

Because minimizing the process language and factoring out related components permit language complexity to be added incrementally, we believe that this approach can lead to benefits in many areas, including process analysis, understanding, adaptation, and execution. In this paper, we present the design of Little-JIL and evaluate our experience with it.

2 APPROACH

In previous work, we have investigated extending a conventional programming language with process-motivated constructs (APPL/A [25]). This work suggested that it would be preferable to develop a new special purpose, high-level language designed specifically for process programming. This new language, JIL, has been described elsewhere [26]. Preliminary evaluation of JIL has suggested: 1) the value of high-level, process-oriented semantics, 2) the appropriateness of the "step" as a central abstraction, 3) the use of the step construct as a scoping construct for other features, and 4) the possibility of a factored language design. Both APPL/A and JIL aimed to be comprehensive in their features and were concerned with supporting full process implementations, including necessary computational and data-modeling functionality. However, this work also underscored difficulties both in developing and in using large and complex languages.

The language described here is called Little-JIL. Little-JIL draws on the lessons of JIL by retaining the step as the central abstraction and scoping mechanism but refines the features in terms of which a step is defined. A main objective of the design of Little-JIL is to pursue a factored approach, by which we mean separating different aspects of process definition into elements called factors. In Little-JIL we identify what we believe to be a viable factoring for a process programming language, and have designed what we believe to be a viable set of linguistic elements that initially focuses on the coordination factor.

Note, though, that while a language may select specific semantic factors to address, additional factors are still generally required for an effective process representation. A well-factored language is thus part of a larger process environment along with additional notations and systems. Little-JIL relies on separate systems for the definition of language factors other than coordination. These systems provide, for example, for the definition of resource requirements, artifact specification, and agenda management. This factored approach allows the core coordination language to be simpler and easier to understand, develop, and use. Additionally, by factoring out certain aspects of process definition, these aspects can be developed and evolved in independent ways, as appropriate to the environments and organizations in which they will be used.

The design of Little-JIL features was guided by three primary principles:

Simplicity: To foster clarity, ease-of-use, and understandability, we made a concerted effort to keep the language simple. We added features only when there was a demonstrated need in terms of function, expressiveness, or simplification of programs. Furthermore, by using a factored approach and concentrating on coordination, we were able to simplify the language relative to that of a general-purpose programming language. To help make the language accessible to both developers and readers, we adopted a primarily visual syntax.

Expressiveness: Subject to (and supportive of) the goal of simplicity, we made the language highly expressive. Software and workflow processes are semantically rich domains, and a process language, even one tightly focused on coordination, must reflect a corresponding variety of semantics. We wanted the language to allow users to speak to the range of concerns relevant to a process and be able to express their intentions in a clear and natural way.

Precision: The language semantics are precisely defined. This precision contributes to several important goals. First, it enables automatic execution of process programs. Second, precision supports the *analyzability* of process programs. Analysis is key to assuring that process programs indeed have properties that are desirable for process safety, correctness, reliability, and predictability (or, conversely, for showing that those properties cannot be guaranteed). Analysis also contributes to process understanding and validation.

We also followed many other software and language design criteria, such as hierarchic decomposition, scoping, and so on, but the three principles described were the primary concerns for Little-JIL. These concerns are related, however, so the design of Little-JIL has also involved balancing tradeoffs. For example, adding a control construct may increase expressiveness, but it may also increase complexity in terms of the number of language features. Some additional complexity may be warranted if new features will be widely used or they result in a simplification of programs, but such considerations may be difficult to weigh. Fortunately, our design principles can also be complementary: separating out components of the language has increased its simplicity. In the next section we describe the features of Little-JIL. We show how Little-JIL can be used to clearly and effectively express the coordination aspects of agent-based processes using the familiar problem of trip planning.

3 LANGUAGE AND EXAMPLES

Capturing the coordination in a process as a hierarchy of steps is the central focus of programming in Little-JIL. A Little-JIL program is a tree of steps whose leaves represent the smallest specified units of work and whose structure represents the way in which this work will be coordinated.

As processes execute, steps go through several states. Typically, a step is *posted* when assigned to an execution agent, then *started* by the agent. Eventually either the step is successfully *completed* or it is *terminated* with an exception. Many other states exist, but a full description of all states is beyond the scope of this paper.

There are six main features of the Little-JIL language that allow a process programmer to specify the coordination of steps in a process. Due to space constraints, we can only give an overview of the language. Detailed language semantics are provided by the Little-JIL language report [27].

The main features of the language and their *raisons d'être* are the following:

- Four non-leaf *step kinds* provide control flow. These four kinds, "sequential," "parallel," "try," and "choice," are the bare minimum for which a need has been clearly established to date. Non-leaf steps consist of one or more substeps whose execution sequence is determined by the step kind. A sequential step's substeps are all executed in left to right order. A parallel step's substeps can be simultaneously executed. A try step's substeps are executed in left to right order stopping when one completes successfully. Exactly one of a choice step's substeps is executed with the decision of which to execute being made dynamically. It is important to note how the parallel and choice step kinds accord to human users the power to exercise their judgment and to make choices about in what order the subtasks of an item should be performed or how a particular item of work is to be done. While the language can be used to constrain the alternatives, the human agent is left free to make the choices.
- *Requisites* are a mechanism to add checks before and after a step is executed to ensure that all of the conditions needed to begin a step are satisfied and that the step has been executed "correctly" when it is completed. A prerequisite is a step that must be completed before the step to which it is attached. Similarly, a postrequisite must be completed after the step to which it is attached. While requisites decrease the simplicity of the language, we have found them necessary to allow process programmers to naturally describe common step contingencies. The need for pre- and post-requisites appears common enough in process programs and requisite step semantics seem different enough from other kinds of sequential steps that a special notation was introduced.
- *Exceptions and handlers* augment the control flow constructs of the step kinds. Exceptions and handlers are used to indicate and fix up exceptional conditions or errors during program execution and provide a degree of reactive control that we believe allows a process programmer to simply and accurately codify common processes.
  The exception mechanism in Little-JIL has been designed to be simple yet remain expressive. It is based on the use of steps to define the scope of exceptions and handlers. Exceptions are passed up the step decomposition tree (call stack) until a matching handler is found.
  Our experience has indicated that it is necessary to allow different exception handlers to work in a variety of ways. After handling an exception, a continuation badge determines whether the step will continue execution, successfully complete, restart execution at the beginning, or rethrow the exception. Detailed semantics are provided in [27].
- *Messages and reactions* are another form of reactive control and greatly increase the expressive power of Little-JIL. The greatest difference between exceptions and messages is that messages do not propagate up the step decomposition tree, being global in scope instead – any executing step can react to a message. Thus, messages provide a way for one part of a process program to react to events without being constrained by the step hierarchy. Because messages are broadcast, there may be multiple reactions to a single message.
  The semantics of messages are not fully worked out and thus they are not implemented yet, but experience so far has convinced us that the coordination factor of a process language must be both able to drive execution forward through proactive mechanisms, and be able to react to events from the environment.
- *Parameters* passed between steps allow communication of information necessary for the execution of a step and for the return of step execution results. The type model for parameters has been factored out of Little-JIL, thus removing issues such as type definition and equality, which are unrelated to coordination.
- *Resources* are representations of entities that are required during step execution. Resources may include the step's execution agent, permissions to use tools, and various physical artifacts; resource specification is not done in Little-JIL, but is carried out by a resource specification factor. As with parameters, Little-JIL attempts to minimize the requirements placed on the resource specification factor: Little-JIL requires little more than that the factor support the identification of resources that match a specification, and that it support resource acquisition and release to avoid usage conflicts.

What's missing from the above feature list is also important to note. As noted above, Little-JIL does not specify a data type model for parameters and resources. It also omits expressions and most imperative commands. Little-JIL relies on agents to know how the tasks represented by leaf steps are performed: Little-JIL is used to specify step coordination, not execution. These typical language features have been factored out, thus simplifying Little-JIL.

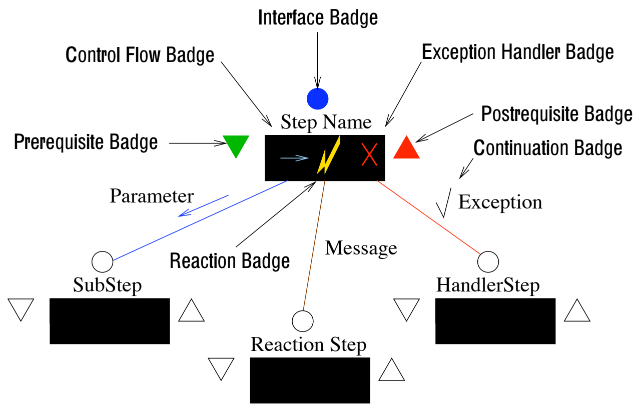

Figure 1: Legend

The graphical representation of a Little-JIL step is shown in Figure 1. This figure shows the various badges that make up a step, as well a step's possible connections to other steps. The interface badge at the top is a circle to which an edge from the parent may be attached. The circle is filled if there are local declarations associated with the step, such as parameters and resources, and is empty otherwise. Below the circle is the step name, and to the left is a triangle called the prerequisite badge. The badge appears filled if the step has a prerequisite step, and an edge may be shown that connects this step to its prerequisite (not shown). On the right is another similarly filled triangle called the postrequisite badge to which a postrequisite step may be attached. Within the box (below the step name) are three more badges. From left to right, they are the control flow badge, which tells what kind of step this is and to which child steps are attached, the reaction badge, to which reaction steps are attached, and the exception handler badge, to which exception handlers are attached. These badges are omitted if there are no child steps, reactions, or handlers, respectively. The edges that come from these badges can be annotated with parameters (passed to and from substeps), messages (to which reactions occur), and exceptions (that a handler should handle). It is possible for an exception to have a null handler, in which case the continuation badge alone determines how execution proceeds.

To better motivate each of these language features and to illustrate their use, we present in Figures 2, 3, and 4 a trip planning process, coded in Little-JIL. The process is based on one presented in [4]. Our version involves four people: the traveler, a travel agent, and two secretaries. The basic idea is to make an airline reservation, trying United first, then USAir. If (after making the plane reservation) the traveler has gone over budget, and a Saturday stayover was not included, the dates should be changed to include a Saturday stayover and another attempt should be made. After the airline reservation is made and travel dates and times are set, car and hotel reservations should be made. The hotel reservations may be made at either a Days Inn or, if the budget is not tight, a Hyatt, and the car reservations may be made with either Avis or Hertz.

The separation of the semantic issues into separate graphical components, as described above, allows an editor tool to selectively display information relevant to a particular factor of a Little-JIL program. Indeed, we illustrate this approach to visualization in the subsequent figures to highlight various language features.

STEP KINDS

Figure 2 depicts the overall structure of the Little-JIL trip planning process program. Each of the four step kinds are used where appropriate:

- A sequential step is used to make plane reservations before car and hotel reservations,
- A try step is used to try United first, then USAir,
- A parallel step is used to allow the two secretaries to make car and hotel reservations simultaneously, and
- Choice steps are used to allow a secretary to choose which hotel chain or car company to try first.

Note that the process program is relatively resilient to many expectable sorts of changes. For example, changing the process program to express a preference in hotel or car rental companies or deciding to attempt all reservations in parallel, i.e., changing the way in which these activities are coordinated, can be accomplished with a straightforward change of step kind.

Requisites

There are two cases in the example (Figure 2) where requisite steps have been used. A postrequisite has been attached to the PlaneReservation step to check that the airfare hasn't exceeded the budget. This means that after the travel agent has successfully made an airline reservation, the traveler should complete the InBudget step. A prerequisite for the HyattReservation step is also shown. This prerequisite could be considered an optimization that is based on the assumption that staying at a Hyatt depletes one's travel budget more than staying at a Days Inn. If a secretary chooses to reserve a room at the Hyatt and the budget is too tight, that step aborts immediately because it will definitely cause costs to exceed the budget.

While the English description of the process does not specify who should check the budget, the Little-JIL program specifies that the traveler is responsible for this task. Postrequisite steps help clarify how the delegation of work can be done. For example, a subordinate can be specified as the agent for a step, but the subordinate's supervisor could then be specified as the agent responsible for executing a postrequisite step that checks the acceptability of the work done by the subordinate. This is shown in the PlaneReservation step. If, for example, the travel budget were sensitive information,

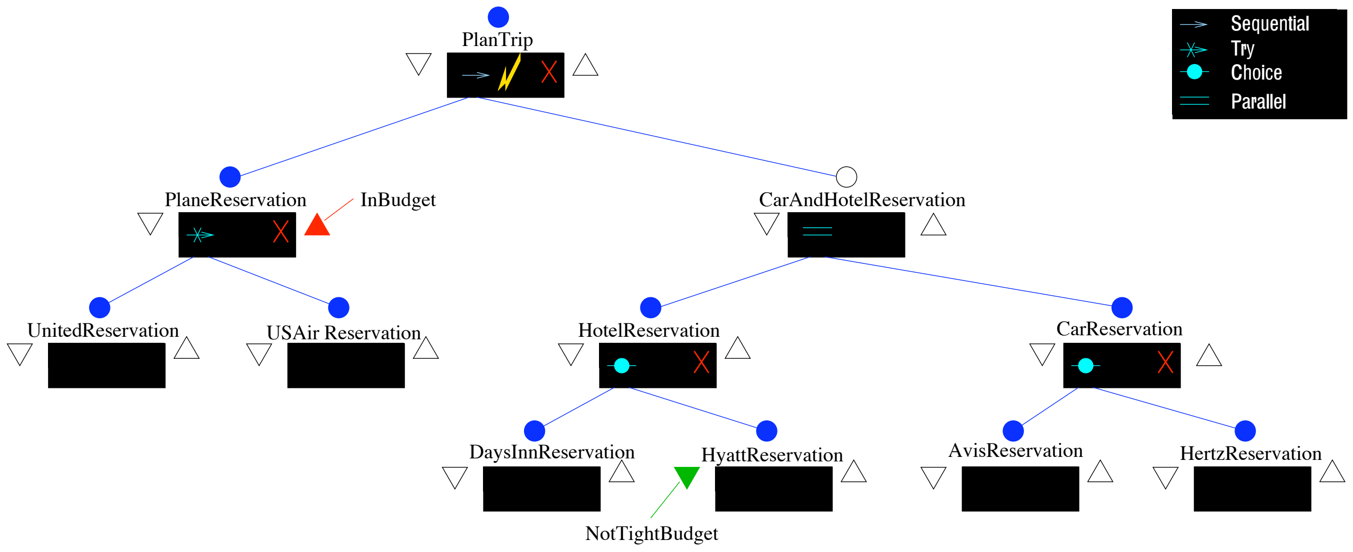

Figure 2: Reservation process showing proactive control: step kinds, requisites.

the execution agent for PlaneReservation could assign the UnitedReservation and USAirReservation steps to other agents without divulging the budget.

Exceptions and handlers
If the agent cannot complete the InBudget prerequisite step previously mentioned (because it determines that the budget has been exceeded), an exception, NotInBudget (not shown), is thrown to the parent. The parent step's handler, IncludeSaturdayStayover (in Figure 3[1]), would check to see that a Saturday stayover was not already included, and if not, it would change the travel dates and restart the PlanTrip step with the new travel dates. If there was already a Saturday stayover, the handler could throw another exception (not shown) that would be propagated higher in the tree or would terminate the program.

Just as different step executions result from the different step kinds, different executions result from different continuation badges. If, for example, IncludeSaturdayStayover were rewritten to make alternative plans, the continuation badge would be changed to "complete," indicating that the exception step had provided an alternative implementation of PlanTrip.

Messages and reactions
An example of a reaction, the "handler" for a message, appears in Figure 3. Here, when the MeetingCancelled message is generated, the CancelAndStop substep of PlanTrip is assigned to the traveler. In this case, there may be very little information associated with that step; it is assumed that the agent will take appropriate action (e.g., phoning the travel agent and secretaries and asking them to abort).

[1]In the figures, ellipses indicate when substeps have been omitted for clarity. In practice, we expect a visual editor to elide information at the user's request.

Parameters
In the example, it is clear that information must be passed from step to step. For example, the PlaneReservation step must pass the trip dates and times to the other reservation steps so that a hotel room and car are reserved for the correct times. Information is passed between steps via parameters. Parameter passing is indicated by annotations made on the step connections, shown in Figure 4. Three parameter passing modes are defined in Little-JIL. Arrows attached to the parameters indicate whether a parameter is copied into the substep's scope from the parent, copied out, or both.

The treatment of the budget says a lot about the approach we have taken with Little-JIL. The language only specifies that a parent step's appropriate parameter values are copied to and from its child steps as specified in the program. Thus, it is assumed that the agents executing steps that need to consult the budget know how to do so; "budget" is not explicitly modeled in the Little-JIL program. Thus, the Little-JIL program provides guidance about when to check the budget, but doesn't dictate any particular way of doing so.

Resources
Resource requirements for a step are indicated by annotations on the step's interface specification. Resources play a central role in the execution of Little-JIL programs, however resource management has been factored out of Little-JIL. By identifying and acquiring resources at run time, a resource management component enables a Little-JIL program to adapt to different environments, allowing more dynamism during process execution. Because resource management has been factored out of the language, the details of a resource model do not have to be represented in each process program.

In Figure 4 execution agent resources are specified as anno-

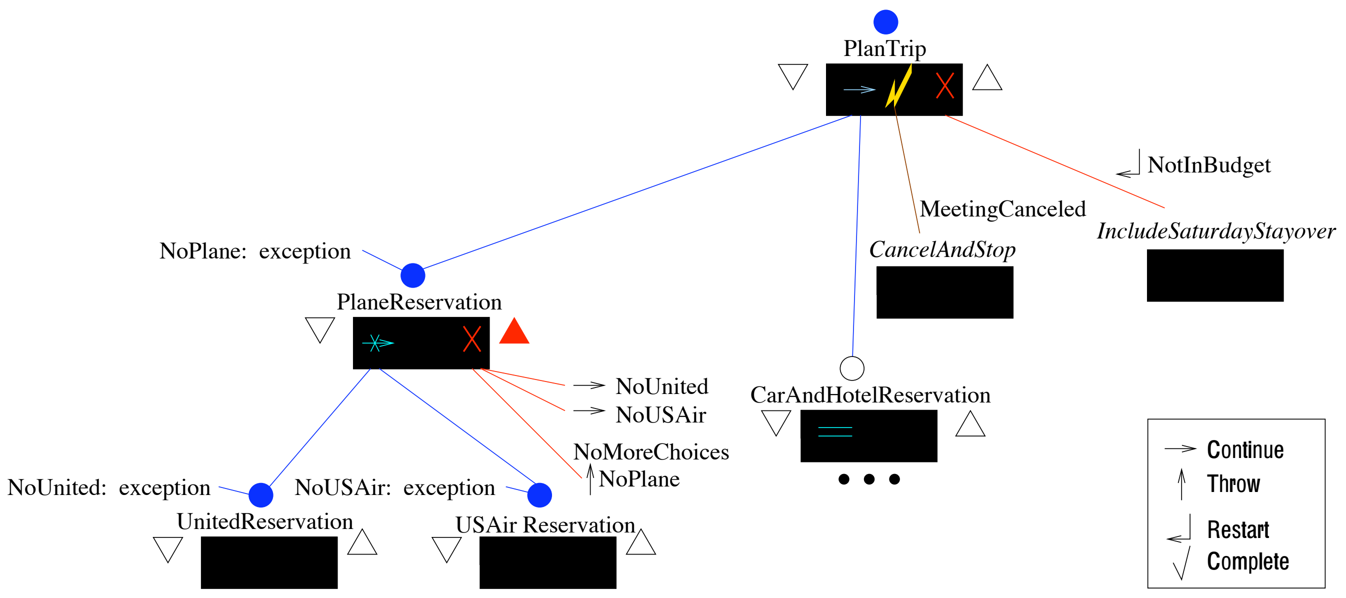

Figure 3: Reservation process showing reactive control: exceptions, messages.

tations on the interface badge. The steps for HotelReservation and CarReservation specify a secretary as the agent responsible for the task. We expect that these tasks would be done in parallel by two different secretaries – but in an environment with only a single secretary, both of these tasks would automatically be assigned to the same secretary who might interleave the activities or perform them sequentially.

In the example, only the agents are being managed as resources, however, resources can be any artifact for which the resource manager's ability to identify artifacts and avoid usage conflicts would be an asset.

4 EXPERIENCE

Process programs

The development of Little-JIL began in 1997, and has proceeded as a series of iterative cycles of design and evaluation. The current version of the language (version 1.0 [27]) is the product of at least three such iterations, each of which entailed the writing of process programs from a variety of application areas. With each iteration, existing features have been honed, and new features have been added only when a clear need has been demonstrated.

In the software engineering domain, we have written process programs for coordinating the actions of multiple designers doing Booch Object Oriented Design [22] and the assignment and tracking of bug reports from submission through regression testing. These processes have focussed on programming coordination among programmers, and also on how to assure that the processes provide support to humans, while not appearing to be too prescriptive or authoritarian. We have also written process programs for guiding the use of the FLAVERS dataflow analysis toolset [13]. In this work we have been particularly interested in using Little-JIL to support both novice and expert users in being more effective in using several tools in this complex toolset. We have also written process programs for guiding the application of formal verification methods and tools, but here our experience has been rather limited. Finally, we have also used Little-JIL to program the ISPW 6 software development process [21].

We have explored the application of process programming to data mining as well. In [17] we describe the applicability of process programming to this domain, and present some example Little-JIL data mining process code. The focus of this work has been to explore how well Little-JIL seems to coordinate diverse tools in this area and program important interactions among tools that focus on distant phases of overall data mining processes.

We are also exploring the use of Little-JIL in programming high-level strategies for coordinating teams of robots. In this work we have been particularly interested in coordinating the activities of humans with those of robots, and in evaluating the effectiveness of our approach to resource specification.

We have also demonstrated the applicability of Little-JIL in programming processes taken from electronic commerce and the workflow domain, such as auctions, the monitoring of assigned tasks, and the example used in this paper.

Several idioms have emerged that simplify the design and understanding of processes. For example, *resource-bounded recursion* allows a step to be repeated multiple times executing with a different resource on each iteration and ceasing when there are no more resources (by completing on a ResourceNotAvailable exception). *Resource-bounded parallelism* is similar to resource-bounded recursion except that in this case the iterations are allowed to happen in parallel. The

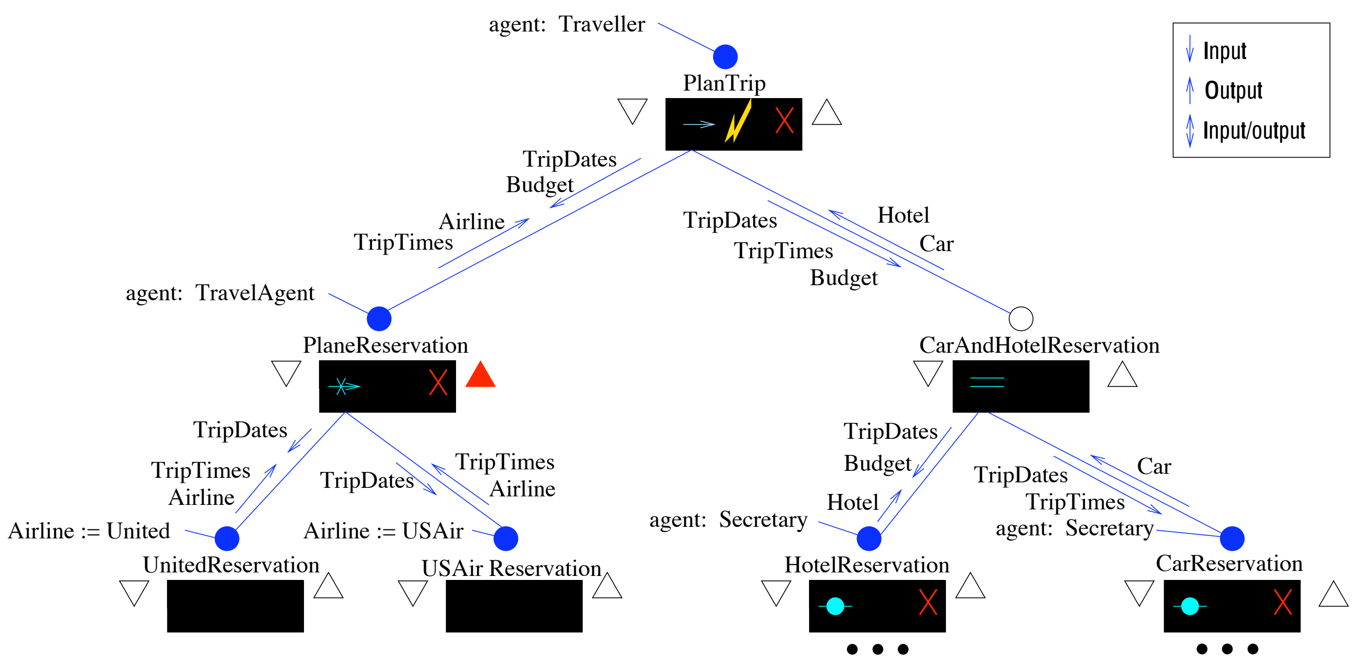

Figure 4: Reservation process showing data flow.

prevalence of situations in which these idioms seem effective in being simultaneously terse, yet clear and precise, suggests that they represent high level language abstractions that may well deserve to be formalized in process programming languages.

Runtime environment
Over the evolution of Little-JIL, the specification of the various processes we have examined has taught us a great deal about the strengths and weaknesses of the language. However, to gain more understanding of the effectiveness of Little-JIL in coordinating agents, we must execute Little-JIL process programs. Unfortunately, we have less experience here due to the fact that the other factors as well as the Little-JIL interpreter were developed after the coordination language.

As has been emphasized earlier, the Little-JIL language has been designed to allow clean separation of process environment components that are not integral parts of the process language. In order to execute Little-JIL process programs, these separated components must be provided. A Little-JIL execution environment consists of the following components:

- Execution agents: these components are required to accomplish the tasks codified in the process program. They do the real work in the process, and make decisions such as when a step should be started or which exception a step throws.
- Little-JIL interpreter: this component interprets the process program by interacting with the other components of the environment as dictated by the semantics of the Little-JIL program being interpreted. It keeps track of and responds to the state changes of steps that occur during execution.
- Resource manager: the resource manager is responsible for managing the resources required by a process program. Its tasks include processing resource management requests generated by the interpreter (including requests for execution agents for a step) and handling model change requests generated by execution agents (upon, for example, the production of a resource needed by other steps in the process).
- Artifact manager: the artifact manager is responsible for managing artifacts produced and needed by the process. Among other things, it provides the type model used by the system for parameter type checking and passing.
- Agenda manager: this component handles the communication of the agents (and interpreter) during process execution. It is responsible, for example, for notifying an execution agent when the interpreter assigns it a step for execution.

A variety of software systems could be used to serve as each of these components. Our prototype Little-JIL process execution environment, called Juliette [7], has a mixture of human and tools as its execution agents, a highly distributed interpreter, a locally developed resource manager, the Java 1.1 runtime system and a file system as its artifact manager and an agenda management system [23] for communication.

5 RELATED WORK
In our research, we have constructed a richly-featured pro-

cess language with a factored design. Little-JIL has a variety of language features: both proactive and reactive control constructs, data flow specification, pre- and post-requisites, etc. At the same time, Little-JIL focuses only on allowing specification of agent coordination and defers specification of the resource management, object (artifact) type modeling and management, and agenda management factors to external languages. The prototype execution environment is composed of several connected modules, thus mirroring this factored approach.

Many other process language and workflow systems have a focus similar to that of Little-JIL, however most lack some of the language features we have identified in Little-JIL, such as exception handling or scoped parameter passing. Many more do not follow a factored approach, instead combining into a single specification language the specification of agent coordination and artifact or resource specifications. APEL [14] is a notable exception. APEL is similar to our work in that it defines separate models for different process factors, but unlike Little-JIL does not seek to make the modeling languages separately evolvable.

InConcert [24] is billed as an "object-oriented client-server workflow management system." A process is specified by a solution developer as tasks (i.e., steps in Little-JIL), roles (execution agents), and references (to documents or objects). InConcert's workflow language is not factored, however. For example, workflow modeling is wound up with resource modeling, and, as with many workflow systems, document management is integrated with workflow execution. We believe the factored approach taken in the design of the Little-JIL language will allow us to run the same process program with a variety of resource models (managed by a variety of resource management components); such flexibility may be difficult to achieve with InConcert.

The approach taken in Little-JIL also contrasts with that taken in workflow management systems such as Action Technology's Metro [1]. In Metro, process enactment is handled by Metro Server, which is also used to manage the workbox (the mechanism for assigning tasks to agents). In Little-JIL, these are separate factors in both the language and implementation. Thus, in the execution environment the interpreter component controls when steps are assigned to agents, but the determination of exactly which agent is assigned the step and how the step is assigned are handled by other components, namely the resource manager and agenda manager.

A number of process languages based on general-purpose programming languages or Petri-Nets, such as APPL/A [25], AP5 [9], and SLANG [2], lack high-level, process-oriented abstractions and a focus on the "step" as the unit of work.

Languages that have focused on process steps or tasks include HFSP [20], EPOS [10], Teamware [28], and APEL [12]. While none of the features in Little-JIL are unique, the way in which they are combined to form a step-oriented process language is. For example, ALF [5] "MASPs" include an object model (parameters), tools (with pre/postconditions), ordering constraints on operators (path expressions), rules (reactions) and "characteristics" (postconditions on the MASP as a whole). However, ALF lacks explicit exception handlers and treats human agents and tools separately.

ProcessWeaver [15], Merlin [18], and Adele-Tempo [3], focus on notions related to "work contexts" (which may be correlated with steps). In most process languages, some form of agent specification is given as part of the process, often giving human and software agents different treatment; frequently work contexts are assigned only to humans. Resource specification is frequently included as an integral part of process modeling, and many languages lack effective visual representations, sufficient reactive control constructs, or both.

A particularly distinctive feature of Little-JIL is its explicit, scoped exception handling. Support for exception handling in other process languages, if it exists, usually takes one of two forms. Some languages provide consistency rules for violation of consistency conditions (one kind of exception), for example, Merlin, Marvel [19], and AP5. Other languages provide general reactive mechanisms that might be used to handle exceptional events, although these may not be differentiated from normal events. Some examples include ALF, Adele-Tempo, and Statemate [16].

Some process languages have achieved an effect similar to the factored approach taken with Little-JIL by defining interfaces to external aspects of process *execution*. Some examples include ALF (where agents are defined wholly outside the MASPs, and operators and objects are bound to external tools and artifacts), ProcessWeaver (in which external agents, tools, and artifacts are coordinated), and OPSS [11] (in which a separate State Server is integrated with human, tool, and software agents via a software bus). Little-JIL, in contrast, explicitly factors those elements external to the process *specification*, not just the execution. As as result, resource management, agent coordination, and artifact management are specified outside of the process language itself. ProcessWeaver's process modeling language, for example, has notions of concurrency control with semaphores built in; and though the agent coordination mechanism seems factored in OPSS, the state server still models artifacts, agents and resources in addition to the state of process enactment.

6 CONCLUSIONS

Our experience with Little-JIL thus far has been encouraging. In general, we have found that the focus on coordination has made it relatively easy to express the process semantics that we desire. More importantly, separating out factors did not hinder such expression, simplified language development, and made it easier to adopt and use. In this section, we revisit our main design principles to identify where the factored approach taken in the design of Little-JIL has succeeded and where work remains.

Simplicity: By separating out many process-related factors not directly relevant to coordination, Little-JIL remains fairly small and easy to understand. This has been evidenced by our interactions with researchers from other domains, specifically from data mining, static analysis, and robotics, who have found Little-JIL to be easy to read.

The factored approach has allowed the creation of a graphical notation centered around the step, which we have identified as the focus of coordination. While the notation is centered on the step, other factors are still well represented.

Expressiveness: Factoring issues unrelated to coordination, such as the specification of types, resource modeling, and the communication mechanism, have not prevented us from expressing a wide variety of processes in Little-JIL. As components were factored from the language, features that represented the interfaces to the factored components were added to maintain expressiveness. For example, while resources were factored from the language so that a resource model definition is independent from Little-JIL, resource specifications appear in the interface badge, and a Little-JIL interpreter must be prepared to communicate with a resource manager to identify, acquire, and release resources as it executes process programs.

To maintain its simplicity, we have resisted impulses to add features to the language, but our experience indicates that it may yet be necessary to add some traditional language features to improve expressiveness. In particular, Little-JIL processes often use exceptions for non-exceptional conditions, such as terminating resource-bounded recursion and parallelism, which would be more naturally terminated by testing whether resources exist rather than failing when resources are depleted. We are currently considering adding looping and conditional constructs as well as a simple expression language to reduce the inappropriate use of exceptions. We have also begun to explore how to add a scheduling factor to Juliette and the impact of timing on coordination.

Thus far in our experience, reactions have been used less than the other control mechanisms. We believe that this is attributable more to the fact that they have been added to the language relatively recently than to their inherent utility. As we get more experience with them, we expect their semantics to shift somewhat.

Precision: We require precision in our language for two reasons: executability and analyzability.

As as result of the factored approach, components such as an agenda manager, resource manager, and execution agents must be provided. We have developed these components as well as an interpreter for a subset of Little-JIL. We have executed processes written in that subset and are confident that all of Little-JIL is executable.

Complex processes typically involve a great deal of concurrent activity being performed by multiple agents. We want to reason about common concurrency problems, such as ordering of activities, possibilities for deadlock or starvation, and so on. Much of the detailed behavior of a process is imprecise. Rather it is left to the agents. Since we believe micromanagement of an agent's process is inappropriate – it has been factored out. Because this and many other factors are not completely represented in Little-JIL, it will be interesting to discover what the practical limits of analysis are. It will likely be necessary to perform analysis across the representational boundaries imposed by this factoring.

Thus far most of our static analysis has been limited to manual evaluation of processes, but Little-JIL is precise enough to allow application of static analysis technology, especially to the analysis of issues directly related to the coordination of step execution [8]. In recent work we have begun to demonstrate success in applying the FLAVERS static dataflow analyzer to Little-JIL process programs. This work has been very revealing. We have succeeded in demonstrating the presence of specific bugs in some Little-JIL process programs, and the absence of bugs in others. But have also discovered that apparently simple and intuitive Little-JIL constructs such as the parallel and choice steps (especially when used in conjunction with recursion) often conceal considerably semantic complexity. This buttresses our contention that these language constructs are important additions to process programming languages, as they are intuitively clear, yet represent substantial semantic content.

Our evaluation of Little-JIL is continuing through the definition of processes from a variety of domains, and use and analysis of the resulting processes. We expect to learn a great deal from these experiments and expect to continue to refine Little-JIL as experience directs us.

ACKNOWLEDGMENTS

The authors would like to thank Rodion Podorozhny for his early contributions to Little-JIL and the resource management factor, Aaron Cass for his substantial work on Juliette, and Yulin Dong, Hyungwon Lee, and Marcia Zangrilli for programming in and providing feedback about many versions of the language.

REFERENCES

[1] Action Technologies, Inc. Metro 3.0 product overview. http://www.actiontech.com/metro/overview/.

[2] S. Bandinelli, A. Fuggetta, and S. Grigolli. Process modeling in-the-large with SLANG. In *Proc. of the Second Int'l Conf. on the Software Process*, pages 75–83. IEEE Computer Society Press, 1993.

[3] N. Belkhatir, J. Estublier, and M. L. Walcelio. ADELE-TEMPO: An environment to support process modeling and enaction. In A. Finkelstein, J. Kramer, and B. Nuseibeh, edi-

tors, *Software Process Modelling and Technology*, pages 187 – 222. John Wiley & Sons Inc., 1994.

[4] E. Bertino, S. Jajodia, L. Mancini, and I. Ray. Multiform transaction model for workflow management. In *Proc. of the NSF Workshop on Workflow and Process Automation in Information Systems*, May 1996.

[5] G. Canals, N. Boudjlida, J.-C. Derniame, C. Godart, and J. Lonchamp. ALF: A framework for building process-centred software engineering environments. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modelling and Technology*, pages 153 – 185. John Wiley & Sons Inc., 1994.

[6] N. Carriero and D. Gelernter. *How to Write Parallel Programs A First Course*. MIT Press, 1990.

[7] A. G. Cass, B. S. Lerner, E. K. McCall, L. J. Osterweil, and A. Wise. Logically central, physically distributed control in a process runtime environment. Technical report, University of Massachusetts at Amherst, Nov. 1999. Submitted to Internation Conference of Software Engineering 2000.

[8] J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil. Verifying properties of process definitions. Technical Report 99-63, University of Massachusetts at Amherst, Nov. 1999. Submitted to Internation Conference of Software Engineering 2000.

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

[10] R. Conradi, M. Hagaseth, J.-O. Larsen, M. N. Nguyên, B. P. Munch, P. H. Westby, W. Zhu, M. L. Jaccheri, and C. Liu. EPOS: Object-oriented cooperative process modelling. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modelling and Technology*, pages 33 – 70. John Wiley & Sons Inc., 1994.

[11] C. Cugola, E. D. Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *Proc. of the 20th Int'l Conference on Software Engineering*, pages 261–270, Apr. 1998.

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

[13] M. B. Dwyer and L. A. Clarke. Data Flow Analysis for Verifying Properties of Concurrent Programs. In *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering, New Orleans*, pages 62–75. ACM Press, December 1994.

[14] 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.

[15] C. Fernström. PROCESS WEAVER: Adding process support to UNIX. In *Proc. of the Second Int'l Conf. on the Software Process*, pages 12 – 26, 1993.

[16] 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.

[17] D. Jensen, Y. Dong, B. S. Lerner, E. K. McCall, L. J. Osterweil, S. M. Sutton Jr., and A. Wise. Coordinating agent activities in knowledge discovery processes. In *Int'l Joint Conf. on Work Activities Coordination and Collaboration*, July 1998. submitted.

[18] G. Junkermann, B. Peuschel, W. Schäfer, and S. Wolf. MERLIN: Supporting cooperation in software development through a knowledge-based environment. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modelling and Technology*, pages 103 – 129. John Wiley & Sons Inc., 1994.

[19] G. E. Kaiser, N. S. Barghouti, and M. H. Sokolsky. Experience with process modeling in the MARVEL software development environment kernel. In B. Shriver, editor, *23rd Annual Hawaii International Conference on System Sciences*, volume II, pages 131–140, Kona HI, Jan. 1990.

[20] T. Katayama. A hierarchical and functional software process description and its enaction. In *Proc. of the 11th Int'l Conf. on Software Engineering*, pages 343 – 353. IEEE Computer Society Press, 1989.

[21] M. I. Kellner, P. Feiler, A. Finkelstein, T. Katayama, L. J. Osterweil, and M. H. Penedo. ISPW-6 software process example. In *Proc. of the First Int'l Conf. on the Software Process*, pages 176 – 186, 1991.

[22] B. S. Lerner, S. M. Sutton, Jr., and L. J. Osterweil. Enhancing design methods to support real design processes. In *9th IEEE Int'l Workshop on Software Specification and Design*, pages 159–161. IEEE Computer Society Press, Apr. 1998.

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

[24] S. K. Sarin. White Paper: Object-Oriented Workflow Technology in InConcert. http://www.inconcertsw.com/solution/sunil.htm.

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

[26] 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.

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

[28] P. S. Young and R. N. Taylor. Human-executed operations in the teamware process programming system. In *Proc. of the Ninth Int'l Software Process Workshop*, 1994.