Specifying Coordination in Processes Using Little-JIL

Alexander Wise, Barbara Staudt Lerner
Eric K. McCall, Leon J. Osterweil and
Stanley M. Sutton, Jr

CMPSCI Technical Report 98-38
August 1998

Laboratory for Advanced Software Engineering Research
Computer Science Department
University of Massachusetts

Effort sponsored by the Defense Advanced Research Projects Agency and the Air Force Research Laboratory, USAF, under agreement number F30602-97-2-0032. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon."

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, the Air Force Research Laboratory, USAF, or the U.S. Government.

Specifying Coordination in Processes Using Little-JIL

Alexander Wise, Barbara Staudt Lerner, Eric K. McCall,
Leon J. Osterweil, and Stanley M. Sutton Jr.
Computer Science Department
Lederle Graduate Research Center
University of Massachusetts
Amherst, MA 01003-4610 USA
+1 413 545 2013
{wise, lerner, mccall, ljo, sutton}@cs.umass.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 focus 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 an orthogonal issue. Little-JIL provides a rich set of control structures while relying on separate systems for support in areas such as resource, object, and agenda management. The second hypothesis is that processes are executed by agents who know how to perform their tasks but who can 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. Two general issues are prominent in process and workflow design. The recognition in many languages that processes can and should be described in terms of a wide variety of semantics: organizations, activities, artifacts, resources, events, agents, exceptions, and so on has lead to a number of languages that are powerful and semantically rich but also complex and challenging to use, especially for non-programmers. Counter to this, language designers have adopted a number of strategies toward simplification; these include narrowing of semantic focus or depth and the use of graphical representations. While such strategies may indeed foster linguistic simplicity, the processes that they must attempt to capture retain their semantic breadth and complexity. Consequently, the practical utility of simplified languages has been limited. In this paper, we present Little-JIL, a language that attempts to resolve these two apparently conflicting objectives, semantic richness and ease of use.

Little-JIL is strongly rooted in our past research on process programming languages [19, 20], 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 and potentially independent treatment of various semantic elements of a process. We believe that coordination is a logically central aspect of process semantics and is an especially important focus for a factored process language. Coordination structures serve as a natural focus to which other factors (such as objects, resources, or agents) can be related and through which the use or involvement of other factors in the process can be orchestrated. Little-JIL is designed to interoperate with additional specification languages and supporting services to allow for the expression of process factors that are not addressed in Little-JIL.

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 still supports process semantics in a rigorous way more typically associated with textual languages. This is facilitated in part because the focus of the language is narrowed to coordination-related elements.

We believe that the approach of minimizing the process language and factoring out related components 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

The work described here continues our ongoing efforts to demonstrate how the precision, rigor, and operational semantics of programming languages can be used to provide the benefits of understandability, analyzability, expressivity, and executability to software development and workflow processes.

In previous work, we have investigated extending a conventional programming language with process-motivated extensions (APPL/A [19]). 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 [20]. 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 and extends the lessons of JIL, but with a more precise focus. 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 orthogonal elements called factors. Little-JIL attempts to meet this objective in three ways. First, we adopted semantics that closely match those in the process domain. Second, we separated those semantics into linguistic elements that could be specified and managed more-or-less independently. Third, we focused Little-JIL on factors related to coordination.

Note, though, that while a language may select specific semantic factors to address, additional factors are still generally required for complete and effective process representation and support. 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 functionality that is not deemed central to the expression of coordination. These systems provide, for example, resource management, data management, and agenda management. This factored approach allows the core coordination language to be simpler and easier to understand, develop, and use. Additionally, by factoring certain functions, they can be developed and evolved in independent ways, as appropriate to the environments and organizations in which they will be used.

Further following on JIL, Little-JIL retains the step as the central abstraction and scoping mechanism but refines the features in terms of which a step is defined. Within this framework, 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, expressivity, or simplification of programs. Furthermore, by using a factored approach and concentrating on the key factor, coordination support, we were able to simplify language design relative to that of general-purpose programming language. To help make the language accessible to both developers and readers, we adopted a primarily visual syntax.

Expressivity: 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 interpretation and execution of process programs. Second, precision supports the *analyzability* of process programs, as well as fostering clarity of expression and surety of communication. 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 not unrelated, however, so the design of Little-JIL has also involved balancing tradeoffs. For example, adding a control construct may increase expressivity, 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 the step is either successfully *completed* or *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 [21].

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

- 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 are composed of one or more substeps whose execution sequence is determined by the step kind. A sequential step's substeps are posted in left to right order until either all complete or one terminates. A parallel step's substeps are simultaneously posted and it completes when all its substeps complete. A try step's substeps are posted in sequence until one is completed. A choice step's substeps are posted simultaneously until one is started (at which time, the others are retracted). With just these four step kinds, the language remains simple to use yet our experience to date indicates that they are sufficiently expressive to capture a wide range of step orderings.
- *Requisites* are a generalization of pre- and post-conditions. 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 with great care 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 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 [21].
- *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 program 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.
- *Parameters* passed between steps allow communication of information necessary for the completion 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 that 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 details are modeled outside of the process program. As with parameters, the language attempts to minimize the requirements placed on the resource model: Little-JIL requires that the model 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 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.

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 an-

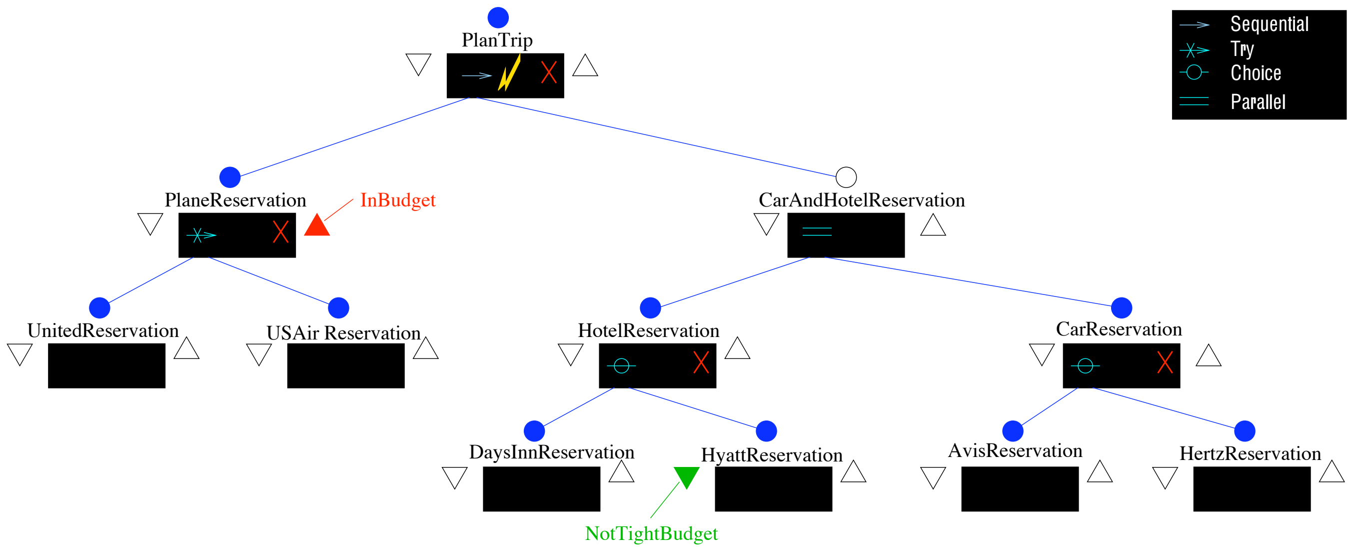

Figure 2: Reservation process showing proactive control: step kinds, requisites.

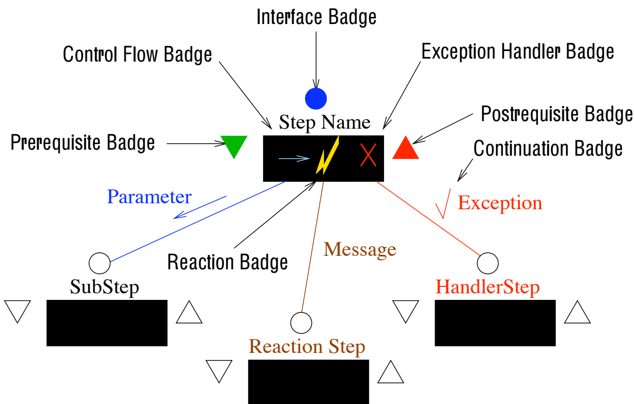

Figure 1: Legend

other 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 only the continuation badge 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 emulate 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 to make plane reservations before car and hotel reservations, a try step to try United first, then USAir, a parallel step to allow the two secretaries to make car and hotel reservations simultaneously, and choice steps to allow a secretary to choose which hotel chain or car company to try first.

Note that the process program is relatively resilient to common 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.

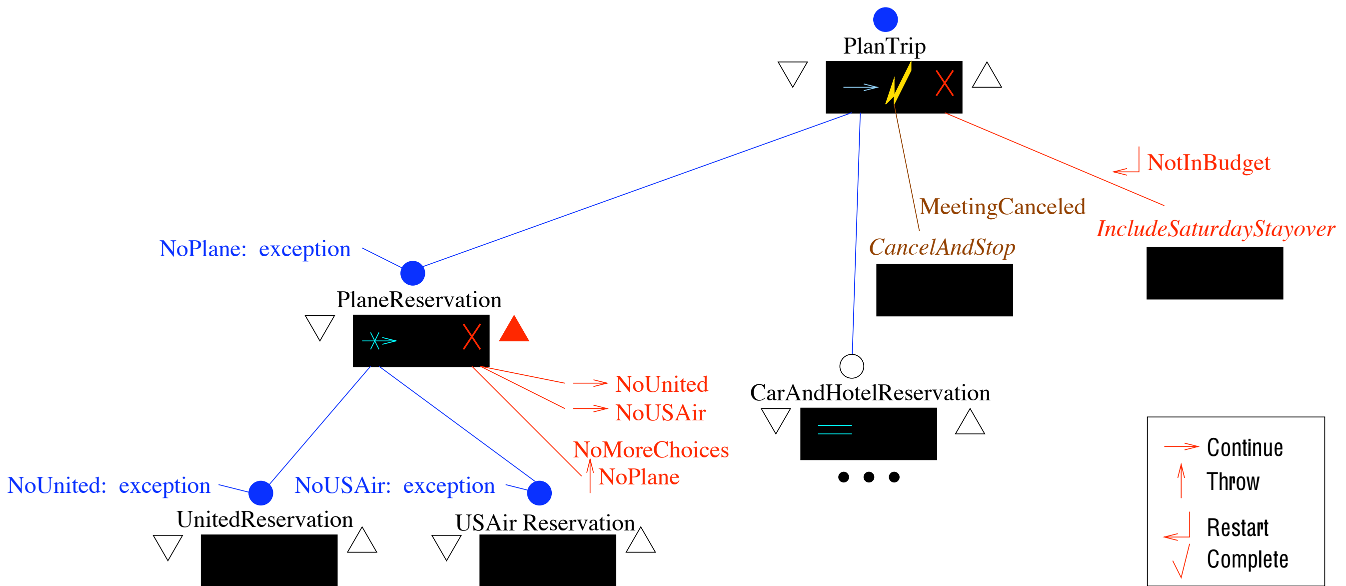

Figure 3: Reservation process showing reactive control: exceptions, messages.

Requisites
There are two cases in the example (figure 2) where requisite steps have been used (though many more opportunities exist). 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 agent 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 can allow an agent to delegate tasks to subordinates then check that the tasks have been satisfactorily completed. This is shown in the PlaneReservation step. If, for example, the travel budget were sensitive information, 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.

Similar to the way in which 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 placed on the traveler's agenda. 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).

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

[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.

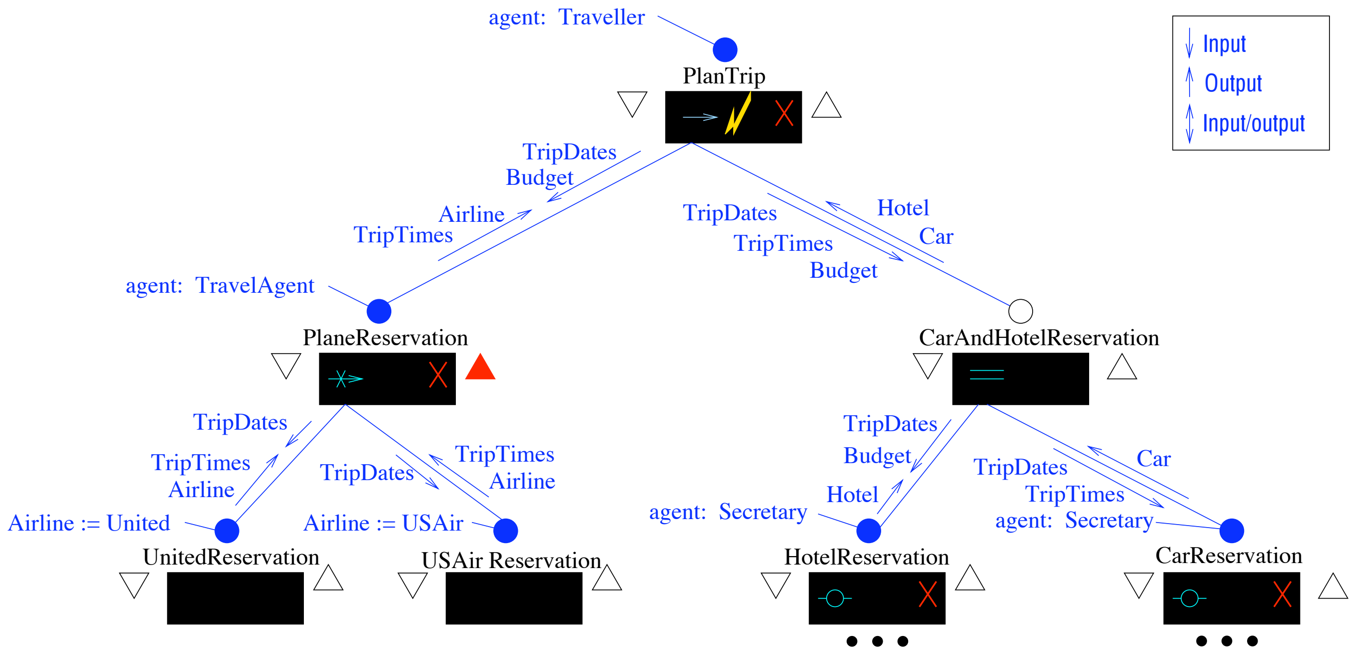

Figure 4: Reservation process showing data flow.

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
Annotations on a step interface denote resource requirements for the step. 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 annotations 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.

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 [21]) 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 [16]. These processes have focussed on programming coordination among designers, 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 [8]. 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 [15].

We have explored the application of process programming to data mining as well. In [12] 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 meet the need for vehicles to integrate 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 the workflow domain, such as the example used in this paper.

Several idioms have emerged that simplify the design and understanding of processes. *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.

Runtime environment

In previous experiences we have been able to learn much about the expressivity, precision, breadth, and clarity of Little-JIL. In order to gain more understanding of the effectiveness of Little-JIL in guiding actual process execution, however, it seems necessary to actually execute Little-JIL process programs. In this area our experience has been more limited.

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 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).
- Object manager: the object 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, has a mixture of human and tools as its execution agents, a highly distributed interpreter, a resource manager, the Java 1.1 runtime system and a filesystem as its object manager, and an agenda management system [17] for communication. An early prototype of the system has been used to interpret part of the BOOD process previously mentioned.

5 Evaluation

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 has 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 and write.

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.

Expressivity: Factoring issues unrelated to coordination, such as the specification of types, resource modeling, and the communication mechanism, has 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 expressivity. For example, while resources were factored from the language so that a resource model defini-

tion is independent from Little-JIL, resource specifications appear in the interface badge, and a Little-JIL interpreter must be prepared 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 expressivity. 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.

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 our analysis has been limited to manual evaluation of processes, but we believe 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.

Our evaluation of Little-JIL is continuing through the definition of processes from a variety of domains, implementation of an interpreter and supporting components, 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.

6 Related Work

In keeping with the multiple semantic aspects of software and other processes, Little-JIL is a semantically broad language, despite deferring some semantics to factors that are separate from the language (e.g., agents, resources, data). The general categories of feature in Little-JIL are adopted from JIL. Several other process languages are also semantically rich, with various combinations of features. For modeling process tasks EPOS [7] has instance-level attributes, procedures, and triggers, and type-level attributes and procedures. The type-level attributes include pre/postconditions, parameters, tools, substeps, and "role" (i.e., agent kind). 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). ALF lacks explicit exception handlers and assumes that agents are specified and assigned separately. PEACE [1] has input/output, pre/postconditions, in/out events, and "intrinsic role" (a human agent).

To help manage linguistic and process complexity, Little-JIL is a factored language. Factoring has not been an explicit theme in the design of many other process languages. JIL [20] is a factored language, but it retains all of its factors, since it is intended to be a full-featured process language. Several other languages are similarly intended to be full-featured, or at least nearly so (e.g., ALF, EPOS, Merlin [13]). Many other languages achieve an effect like factoring in that they either depend on, or are intended to capture, aspects of a process related to externally defined elements, such as agents, tools, or artifacts. Some examples include ALF (where agents are defined wholly outside the MASPs, and operators and objects are bound to external tools and artifacts), and ProcessWeaver [10] (in which external agents, tools, and artifacts are coordinated).

In keeping with the need for high-level semantics for process support in general and for language factoring in particular, Little-JIL uses the process step is the central abstraction. A number of process languages based on general-purpose programming languages or Petri-Nets, such as APPL/A [19], AP5 [6], and SLANG [2], lack such high-level, process-oriented abstractions. Other languages have also focused on process steps or tasks, including HFSP [14], EPOS, ProcessWeaver, Teamware [22], JIL, and APEL [9]. Still other languages, such as ALF, Merlin, and Adele-Tempo [3], focus on notions related to "work contexts" (which may be correlated with steps). Oikos [18] uses several high-level abstractions.

Little-JIL is based on the premise that coordination of execution agents is a central, key factor in process specification and support. Many process languages provide no first-class representation of execution agent. However, external execution agents are also associated with processes in languages including JIL, Merlin, PEACE, EPOS, Teamware, and ALF. In most of these, some form of agent specification is given as

part of the process (ALF being an exception where the agent specification is factored out). Most of these languages are specifically concerned with *human* agents; automated entities are addressed by mechanisms that incorporate "tools." In Little-JIL (and JIL) the notion of "agent" subsumes both human and automated entities both, where the latter may include, for example, tools and robots.

Many process languages are entirely or significantly textual. Little-JIL is primarily a visual language. Its graphical model is distinctive in that it emphasizes the hierarchical breakdown of a process while keeping the within-step flow simple. There are a number of other graphical process languages. Many use net-based models, including SLANG, Melmac, ProcessWeaver, and Teamware. These generally emphasize the "horizontal" flow within a step (although typically still allowing hierarchical decomposition). Statemate [11] provides three coordinated graphical views, based on a state model, that incorporate hierarchy with a nested representation. Oikos uses diagrams to represent structural relationships among process entities. APEL provides control and data flow diagrams between activities and state diagrams within activities. Little-JIL is also distinctive in graphically capturing requisites, proactive and reactive control and exception handlers in its process structure. No other graphical languages represent this variety of control modes, although some include reactions to events.

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, 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.

7 Summary

Little-JIL is a new process programming language based on a factored design. Little-JIL focuses on agent coordination as a key process factor. The premise of this focus is that processes are conducted by agents who understand their tasks but who can benefit from coordination with other agents.

In Little-JIL, processes are modeled as compositions of steps. Steps are defined with respect to their interfaces, prerequisites and postrequisites, and substeps. Interfaces include resources, parameters, messages propagated, and exceptions thrown. Every step has an execution agent as a distinguished resource. Substeps are organized into proactive substeps (for which there are four basic control kinds), reactions, and exception handlers. Reactions occur to globally broadcast messages. Exceptions are scoped and propagated up the step hierarchy; a varied continuation mechanism allows for flexible responses.

This variety of features in Little-JIL is necessary to capture the variety of semantics that may affect process coordination. However, in keeping with a factored approach to language design, Little-JIL relies on separate specification languages and mechanisms to address aspects of process semantics that can be treated independently of coordination structures. These currently include object management, resource modeling, and agenda management. This approach to process language design helps to keep the coordination language relatively simple while, through a combination of factors, providing the comprehensive support that processes require.

The design of Little-JIL was based on three main principles: simplicity, expressivity, and precision. Simplicity is required for ease of development and use. Expressivity is needed to capture the widely varied aspects of process applications. Precision is essential in supporting executability and analyzability. All language design decisions have been considered carefully in light of these goals. To further promote ease of use and comprehensibility, Little-JIL has a graphical syntax. This syntax is relatively unusual, though, in that it emphasizes the hierarchical break-down of process steps while keeping the within-step flow simpler than in most graphical process languages.

We believe that the combination of a factored design, close adherence to the principles of simplicity, expressivity, and precision, and the use of an innovative graphical syntax makes Little-JIL an especially effective and appealing language for expressing a wide variety of software, workflow, and other processes.

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

REFERENCES

[1] Selma Arbaoui and Flavio Oquendo. PEACE: Goal-oriented logic-based formalism for process modeling. In Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh, editors, *Software Process Modelling and Technology*, pages 249–292. John Wiley & Sons Inc., 1994.

[2] Sergio Bandinelli, Alfonso Fuggetta, and Sandro 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] Noureddine Belkhatir, Jacky Estublier, and Melo L. Walcelio. ADELE-TEMPO: An environment to support process modeling and enaction. In Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh, editors,

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

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

[5] Gérome Canals, Nacer Boudjlida, Jean-Claude Derniame, Cladue Godart, and Jaques Lonchamp. ALF: A framework for building process-centred software engineering environments. In Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh, editors, *Software Process Modelling and Technology*, pages 153 – 185. John Wiley & Sons Inc., 1994.

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

[7] 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 Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh, editors, *Software Process Modelling and Technology*, pages 33 – 70. John Wiley & Sons Inc., 1994.

[8] Matthew B. Dwyer and Lori 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.

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

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

[11] 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, April 1990.

[12] David Jensen, Yulin Dong, Barbara Staudt Lerner, Eric K. McCall, Leon J. Osterweil, Stanley M. Sutton Jr., and Alexander Wise. Coordinating agent activities in knowledge discovery processes. In *Int'l Joint Conf. on Work Activities Coordination and Collaboration*, July 1998. submitted.

[13] 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.

[14] Takuya 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.

[15] Marc I. Kellner, Peter Feiler, Anthony Finkelstein, Takuya Katayama, Leon J. Osterweil, and Maria H. Penedo. ISPW-6 software process example. In *Proc. of the First Int'l Conf. on the Software Process*, pages 176 – 186, 1991.

[16] Barbara Staudt Lerner, Stanley M. Sutton, Jr., and Leon 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, April 1998.

[17] Eric K. McCall, Lori A. Clarke, and Leon 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.

[18] Carlo Montangero and Vincenzo Ambriola. OIKOS: Constructing Process-Centered SDEs. In Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh, editors, *Software Process Modelling and Technology*, pages 33 – 70. John Wiley & Sons Inc., 1994.

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

[20] Stanley M. Sutton, Jr. and Leon 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.

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

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