Specifying Process Coordination 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
This paper presents Little-JIL, a new language for programming agent coordination. Little-JIL 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 other coordination- supporting features and provide a scoping mechanism for control, data, and exception flow and for agent and resource assignment. Steps can be composed hierarchically, but Little-JIL processes can have highly dynamic structures and include recursion and concurrency.

Little-JIL is based on two main hypotheses. The first 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. The second hypothesis is that the specification of coordination control structures is a separable issue. In this regard Little-JIL provides a rich set of control features; however, it relies on separate systems for support in areas such as resource, object, and agenda management.

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 and workflow processes.

Keywords
Process programming, Little-JIL

1 Introduction

There is a growing need for process and workflow specification in many contexts. This is evidenced by both a growing marketplace as a well as a thriving research community. Most process and workflow languages tend to fall in one of two camps: rigorous programmatic descriptions or easily understood, graphical languages. There are advantages to each. Rigorous specifications support analysis and execution, while graphical specifications support high-level description and understandability. There are also disadvantages to each. Programmatic specifications can be extremely difficult to understand, particularly by non-programmers, while with graphical specifications, it is often difficult to capture many important details, such as exception handling. In this paper, we present Little-JIL, a rigorous, graphical language for specifying processes and workflow.

Little-JIL is strongly rooted in our past research on process programming languages[**?**, **?**], but makes an important break in two respects. First, it is a graphical language. Second, it abstracts certain process elements into separate specification languages that are integrated with Little-JIL, rather than creating a single monolithic language. In this way, the graphical language remains simple and can present a concise high-level view of a process. Integration with other components maintains the ability to express detailed processes where necessary.

The focus of Little-JIL processes is coordination: coordination of autonomous agents, coordination of shared resources, coordination of information flow. Coordination is explicit in the graphical notation. Agents are autononoums entities, human or software. Assignment of tasks to agents is explicit in Little-JIL processes, but the details of how these agents perform their tasks is specified externally to the Little-JIL process. Similarly, resources are external entities whose use must be monitored and controlled. The resource needs of a process are specified in a Little-JIL process, while the detailed workings of the resource model are specified externally. Finally, information/data must be propagated between agents in order for them to accomplish their work. This flow is expressed in the Little-JIL process, while the detailed definition of the data types and how they may be manipulated is again specified externally to the process.

We believe that this 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 our experiences with it is a graphical, coordination process language.

2 Approach

In previous work, we have investigated extending a conventional programming language with process-motivated extensions (APPL/A [**?**]). 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 [**?**]. Preliminary evaluation of JIL has suggested: 1) the value of high-level, process-oriented semantics, 2) the utility of the "step" as a central abstraction, 3) the use of the step construct as a scoping construct for other features, and 4) the validity of a factored language design, which allows, insofar as possible, the various aspects of a process to be described independently and as needed. 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 the difficulties, both in developing and in using, such large and complex languages.

The work described here draws on and extends the lessons of JIL, but pursues a more focused and visual approach. It retains the step as the central abstraction but refines the features in terms of which a step is defined and emphasizes the role of the step as a scoping mechanism. It also uses a visual representation to reinforce these notions, and to enhance usability and comprehensibility. This work also pursues the idea of separating different aspects of step definition into independent factors, and advances the hypothesis that a particularly useful collection of factors centers on agent coordination.

The language described here, which is based on a subset of JIL, is called Little-JIL. The design of the Little-JIL has four primary themes.

Simplicity: To foster clarity, ease-of-use, and understandability, a concerted effort has been made to keep the language simple. Features have been added to the language only when there has been a demonstrated need in terms of function, expressivity, or the simplification of programs.

⇒ Furthermore, by focusing on coordination [REMARK: *rather, by using a factored approach in general*] , we have been able to simplify language design relative to that of a computationally complete process language. To help make the language accessible to both developers and readers, we have adopted a primarily visual syntax.

⇒ **Expressiveness**: [REMARK: *look up best word*] Subject to (and supportive of) the goal of simplicity, Little-JIL should be an expressive language. Software and workflow processes are semantically rich domains, and a process language, even one focused on coordination, must reflect a corresponding variety of semantics. The language should allow the user to speak to the range of concerns relevant to a process, and to express their intentions in a clear and natural way.

⇒ **Separability**: In keeping with the principles of simplification and factoring [REMARK: *circular definition?*] , Little-JIL relies on separate systems for certain functionality that is not deemed central to the expression of coordination structures. These include, for example, resource management, data management, and agenda management. This allows the core coordination language to be simpler and easier to understand, develop, and use. Additionally, by factoring out certain functions, we hope to allow them to be developed and evolved in independent ways, as appropriate to the environments and organizations in which they will be used.

Precision: Little-JIL is an executable language. The goal of executability has two main benefits in process language design. First, it helps to foster on semantic rigor: the language must be substantially complete [REMARK: *what does that mean?*] , consistent, and unambiguous. Second, it supports the execution of processes in accordance with their specification in the program. Through execution, what is known and validated about a process specification can be transferred to the process behavior itself. [REMARK: *we now must rewrite to say that the language is hopefully analyzable*] ⇐

There are many other software and language design criteria we followed, such as hierarchic decomposition, scoping, orthogonality, and so on, but the four goals described were the primary concerns for Little-JIL. These concerns are not orthogonal, 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 factors may be difficult to weigh. Fortunately, our design themes can also be complementary: separating out components of a language may increase 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 a running example based on a standard workflow-process problem.

3 Language and examples

As described previously, capturing a process as a hierarchy of steps is the central focus of programming in Little-JIL. Little-JIL is a graphical language, with a program looking like a tree of steps with the smallest units of work as leaves. All of the "real" work of a process is done by the agents assigned to these leaf steps.

There are six main features of the Little-JIL language we highlight in this paper. Due to space constraints, we can only give a feeling for what programming in Little-JIL is like; specific language semantics are provided by the Little-JIL language report [**?**].

The features of the language and *raison d'etre* are:

- Four *step kinds* provide control flow. These four kinds,

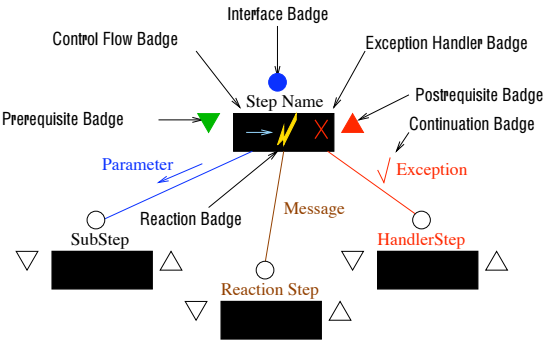

Figure 1: Legend

"sequential", "parallel", "choice" and "try", are the bare minimum we discovered we needed. With just these four, the language remains simple to use yet expressive enough to capture interesting processes.

- *Requisites* allow increased agent autonomy. While requisites decrease the simplicity of the language, we felt they were necessary to allow process programmers to write code that accurately reflects their intent. Requisites are roughly equivalent to sequential steps with three children (a prerequisite step, the "real" step, and a postrequisite step). We believe the need for pre- and post-requisites is common enough in process programs and have different enough meaning from other sequential steps that a special notation was introduced.
- *Exceptions and handlers* augment the control flow constructs of the step kinds and provide a degree of reactive control we believe is necessary. Exceptions allow 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 and executable.
- *Messages and reactions* allow reactive control, and greatly increase the expressive power of Little-JIL. They also provide independence from a program's hi-
⇒ erarchic structure... [REMARK: *add something to distinguish this from exceptions*]
- *Parameters* passed between steps provide data flow. The type model has been separated out – Little-JIL merely agrees to give (a named copy of) the parent step's appropriate parameter values to the child steps.
- *Resources* allow for more dynamic process execution, and allow "execution agents" (and a resource manager component) to be separated out.

What's "missing" from the above feature list is also important to note. Little-JIL does not specify a type model, conditionals, or any loop step kinds.

⇒ [REMARK: *Introduce basic states for steps here; posted, started, completed, terminated only.*]

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 by which this step is connected to its parent. The circle is filled if there are local definitions associated with this step, and is empty otherwise. Below the circle is the step name, and to the left is a triangle called the pre-requisite badge. The pre-requisite is a step that must be successfully completed for this step to begin execution. The badge appears filled if the step has a pre-requisite step, and an edge may be shown that connects this step to its pre-requisite (not shown). On the right is another similarly filled triangle called the post-requisite badge. The postrequisite step is begins execution immediately after the step completes execution, but must also successfully complete for the parent to be notified of the step's completion. 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 hidden if there are no child steps, reactions, or handlers. The edges that come from these badges are annotated with parameters (passed to and from substeps), messages (to which reactions occur), and exceptions (that a handler should handle) respectively.

To better motivate these features and to illustrate their use, we present a trip planning process, codified in Little-JIL. The process is based on one presented in []. 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. 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 Days Inn or, if the budget is not tight, the Hyatt, and the car reservations may be made with either Avis or Hertz.[1] If (after making the reservations) 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 reservation attempt should be made.

Step kinds

In figure 2 the framework of the Little-JIL program for the trip planning process is in place. 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

[1]The Little-JIL process programming team, LASER, and the University of Massachusetts in no way endorse any of these companies for travel planning, and are in no way liable for damages should travel mishap occur.

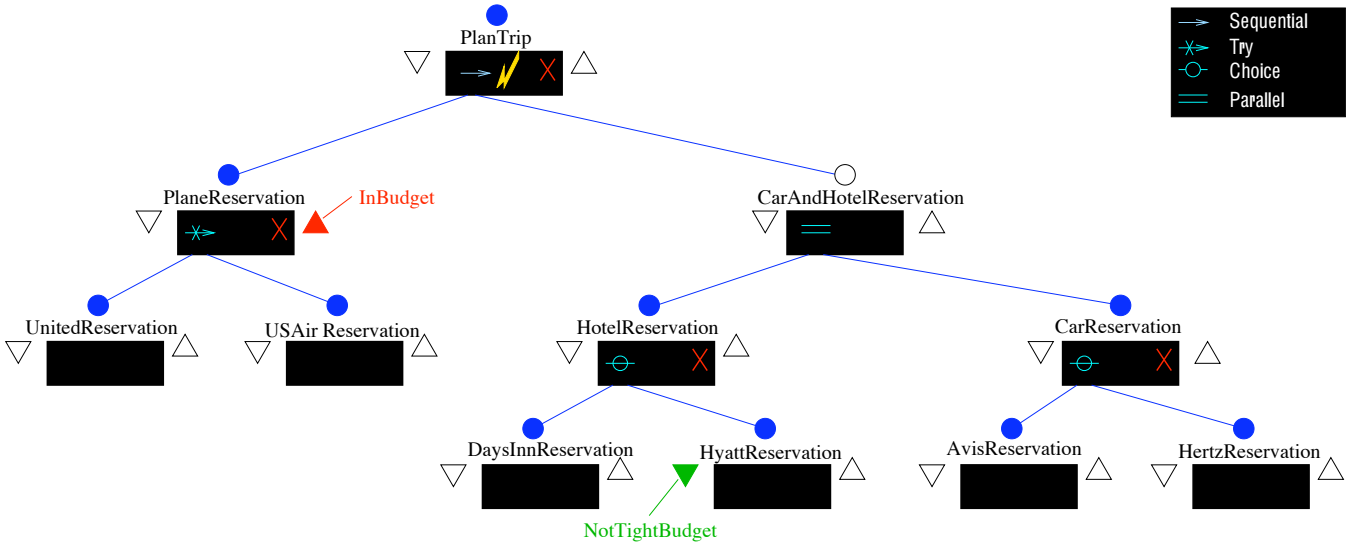

Figure 2: Reservation process: step kinds

deciding to attempt all reservations in parallel can be accomplished with a straightforward change of step kind.

⇒ [REMARK: *The treatment of the budget says a lot about the approach we have taken with Little-JIL. 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.*]

Requisites

The functionality of pre- and post-requisites has been previously described. 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 process program optimization that is based on the assumption that staying at a Hyatt depletes one's travel budget more than staying at a Days Inn. When a secretary chooses to reserve a room at the Hyatt, if the budget is too tight, the reservation step aborts immediately because we know that we will already go over budget.

⇒ [REMARK: *Comments about requisites vs. sequential steps go here?*]

Exceptions and handlers

Exceptions and handlers provide an extremely useful reactive control mechanism. Such mechanisms are common in both modern programming languages and in real word process execution. The exception mechanism in Little-JIL 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. There are four flavors of exception handlers: continue, complete, restart, and rethrow. The flavor of the handler helps to determine what happens to step execution if the exception is successfully handled by the step. Specific semantics are provided in [**?**].

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. [REMARK: *Should the rest (or all) of this discussion be moved to exception section? Chicken and egg problem, since these features interact.*] ⇐ The parent step's handler, IncludeSaturdayStayover, would check to see that a Saturday stayover was no already included, and if not, would change the travel dates and successfully complete. Because the exception has handled successfully, the step would restart with the new travel dates. If there was already a Saturday stayover, the handler could throw another exception that would be propagated higher in the tree (or would terminate the program).

Messages and reactions

Messages and reactions are intended to be another for reactive control in Little-JIL. While exceptions and handlers are used to indicate and fix up exceptional conditions or errors during program execution, messages and reactions are a more general mechanism. The greatest difference between these language features 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 communicate events in which another mostly unrelated part may be interested.

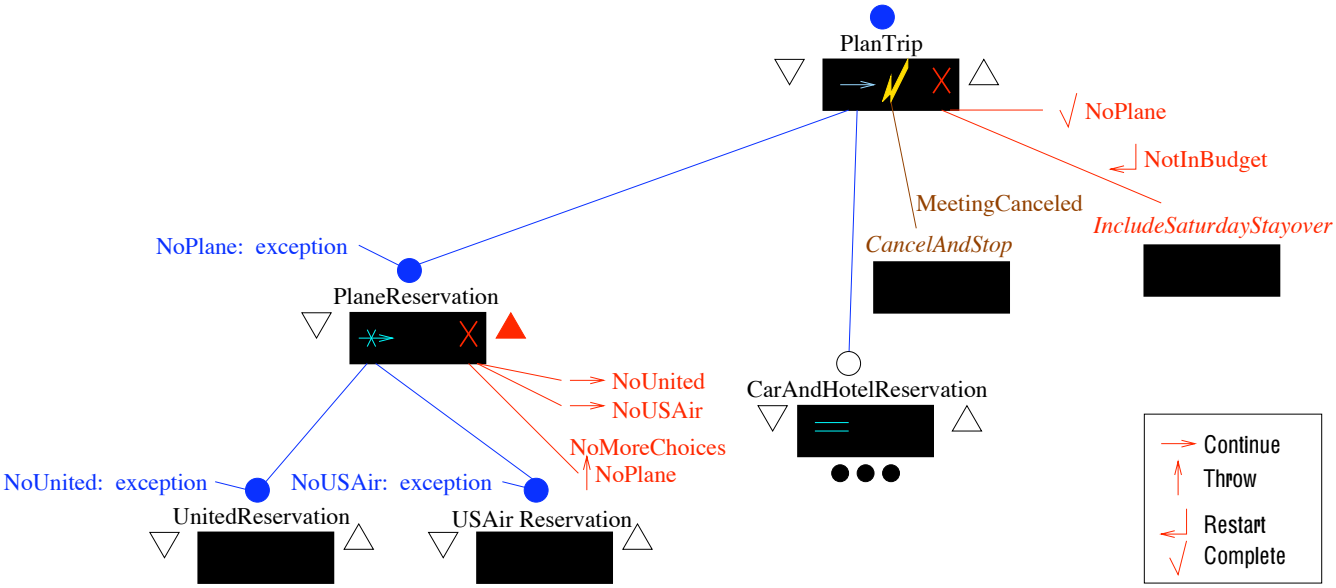

Figure 3: Reservation process: exceptions, requisites, messages

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).

Messages also provide a mechanism for agents to inject information into a running process program.

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. Parameters are indicated by annotations made on the step connections, shown in figure 4.[2] Arrows indicate whether the parameters are in, out, or inout parameters (as in languages such as Ada, [REMARK: *and...*]).

⇒ [REMARK: *Also describe variable scope, (explicit passing only, editor helps)*]

Resources

In figure 4, annotations on the step interfaces denote resource requirements for the step.

⇒ [REMARK: *There's definitely more we should say here.*]

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 [**?**]) 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 sharpened, and new features have been added with caution.

In the software engineering domain, we have written process programs for coordinating the actions of multiple designers doing Booch Object Oriented Design [**?**]. 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 [**?**]. 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 the several tools in this complex toolset. We have also written process programs for guiding 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 IPSW 6 process [**?**].

We have explored the application of process programming to Data Mining as well. In [**?**] 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 needs in this area for vehicles to integrate diverse tools, and program important interactions among tools that focus on distant aspects of overall data mining processes.

We are also exploring the use of Little-JIL to programming high-level strategies for coordinating teams of robots. In this

[2] 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.

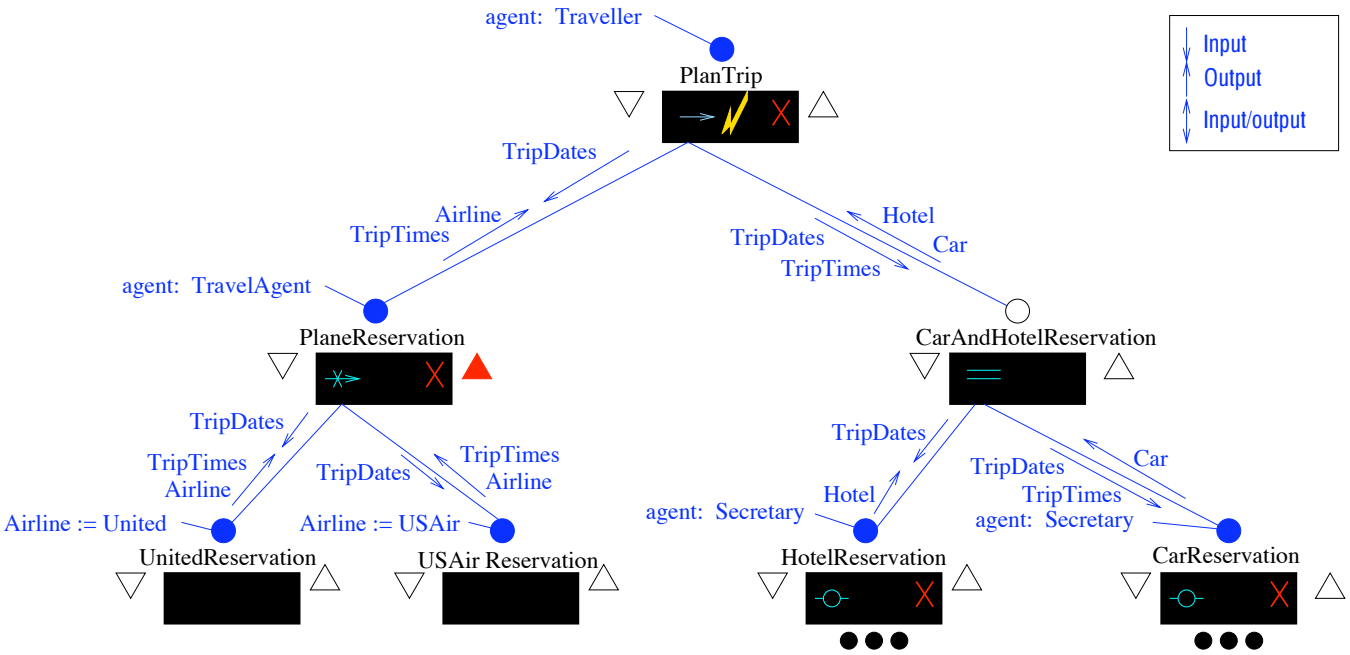

Figure 4: Reservation process: data flow

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. One example of this is a collection of processes aimed at coordinating the actions of both human and computer agents in planning a trip.

Process programs

Runtime environment

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 process program 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.
- Resource manager: the resource manager is responsible for managing the resources involved in 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.
- Coordination mechanism: this component handles the coordination 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 as its components a mixture of human and tool execution agents, a highly distributed interpreter, a resource manager, the Java 1.1 runtime system and a filesystem, and an agenda management system [**?**], respectively.

[REMARK: *Comments about pass-by-copy step parameter semantics supporting distributed execution go here. Talk a tiny bit about how this allows us to have a ditributed interpreter (one for each step).*] ⇐

5 Evaluation and future work

Our experience with Little-JIL thus far has been encouraging. In general, we have found it relatively easy to express the process semantics that we desire, and even more importantly, to use Little-JIL as a communication mechanism when interacting with domain experts. In this section, we re-

visit our four main design themes to identify where Little-JIL has succeeded and where work remains.

Simplicity: While the graphical notation of Little-JIL is idiosyncratic, we have found it to be easy to learn both to read and write the notation. This has been evidenced by our interactions with researchers from other domains, specifically from data mining, static analysis, and robotics.

Expressiveness: The syntax and the semantics of the language has been driven both by abstractly reasoning about processes, but also from experiecences directly using the language. In particular, the four step kinds and four exception continuation semantics were driven by needs found through experience. Since freezing the semantics with those constructs, we have found the language to be capable of expressing the control flow that we have needed in our processes. Through this experience, 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. For example, assign a design task to a designer. When the designer completes that task, assign another one. Stop making assignments when there are no more tasks left.
- *Resource-bounded parallelism* is similar to resource-bounded recursion except that in this case the iterations are allowed to happen in parallel. For example, assign a different design task to each software designer to work on in parallel. Stop making assignments when there are no more designers left. [3]

To maintain its simplicity, we have resisted impulses to add features to the language, but our experience indicates that it is probably necessary to add some traditional language features to improve expressiveness. In particular, processes often use exceptions for non-exceptional conditions, such as terminating resource-bounded recursion and parallelism. We are currently considering adding looping and conditional constructs as well as a simple expression language to reduce the inappropriate use of exceptions, being careful to balance the needs of simplicity and expressiveness.

Thus far in our experience, reactions have been used less than the other mechanisms. They appear to be quite important in the robotics domain. As we get more experience with them, we expect their semantics to shift somewhat.

Object management is another area where we recognize the need for future work. Of particular importance is the ability to express the sharing of information among steps and agents as well as asynchronouse information flow between steps that are executing concurrently.

[3] Note that resource-bounded recursion and parallelism could be intertwined so that tasks would be assigned in parallel as long as there were more designers and then sequentially to each designer as designs were completed.

Separability: The goal of separating out the concerns of agent management, resource management, and object management has been a tremendous benefit. This has resulted in a much simpler language, allowing more concise process representations. It is largely the separation of concerns that enables the graphical representation to be feasible, while still allowing the definition of detailed processes.

Precision: We require precision in our language for two reasons: executability and analyzability. We are in the process of developing an interpreter for Little-JIL. We have found the definition of Little-JIL to be precise enough for this purpose. To execute a process requires integration with external components, agents, an agent management system, a resource manager, and an object manager. Thus far we have written and executed processes integrating multiple agents (human and automated), an agent management system, and a resource management system. Object management has been done through the file system and resource manager. 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 to be inappropriate. [REMARK: *E*] volution is not discussed, but it should probably go here if we want to do it. ⇐

We also believe that analyzability is an important property for processes to have. 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. 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. It will be interesting to discover what the practical limits of analysis are, particularly as separability allows details to be missing in the Little-JIL representation. It will likely be necessary to perform analysis across the representational boundaries imposed by the Little-JIL architecture.

In conclusion, 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. [REMARK: *Here are notes we made about what to include in this section. Delete these as desired.*] ⇐

6 Related Work

In Little-JIL, 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 [**?**], AP5 [**?**], and SLANG [**?**], lack such high-level, process-oriented abstractions. [REMARK: *Check Marvel.*] ⇐ Other languages have also focused on process steps, including HFSP [**?**], ProcessWeaver [**?**], Teamware [**?**], and JIL [**?**]. [REMARK: *Check also EPOS and Melmac*] Still other lan- ⇐

guages, such as ALF [**?**], Merlin [**?**], and Adele-Tempo [**?**], focus on "work contexts" (which may be correlated with steps). [REMARK: *Check also APEL.*] Oikos [**?**] uses ⇒
a number of high-level abstractions (including processes, offices, and desks, among others).

Many process languages are entirely or significantly textual (at least in their process representation, if not their user interface); these include including APPL/A, HFSP, Merlin, Marvel, AP5, and ALF, among others. Little-JIL is primarily a visual process language. Its graphical model is distinctive in that it emphasizes the hierarchical breakdown of a process while keeping the within-step flow simple (i.e., based on the four control kinds). There are a number of other graphical process languages. Many use net-based models, including SLANG, Melmac, ProcessWeaver, and Teamware. These are variably high or low level in their semantics, but they generally emphasize the "horizontal" flow within a step (although typically still allowing hierarchical decomposition). Statemate [**?**] provides three coordinated graphical views that incorporate hierarchy albeit with a nested representation. 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. [REMARK: *Check APEL, Oikos.*] ⇒

Little-JIL is a semantically broad language, although some semantic details of Little-JIL programs must be defined within 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 combinations of features that are more or less comparable to those in Little-JIL. For modeling process tasks [REMARK: *check!*] , EPOS has instance-level attributes, procedures, and triggers, and type-level attributes and procedures. The type-level attibutes include pre/postconditions, parameters, tools, substeps, and "role" (i.e., agent kind). ALF work contexts ("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 [**?**] has input/output, pre/postconditions, in/out events, and "intrinsic role" (a human agent). [REMARK: *Check some others, e.g. Adele-Tempo.*] ⇒ ⇒

A particular feature that helps to distinguish Little-JIL is its explicit, scoped exception handling. Surprisingly (in view of the prevelance of exceptions in processes), few other language support this (exceptions being APPL/A and JIL). 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 [REMARK: *Others?*] . Other languages provide general reactive mechanisms that might be used to handle exceptional events, although these would not be differentiated from normal events. Some examples include ALF, Adele-Tempo, Statemate ⇐

As described in Section [**?**], Little-JIL, like JIL, is a factored language, but focusing on coordination-related factors, with other factors, such as resource management and data management, abstracted. JIL, although factored, is intended to be a full-featured (or nearly full-featured) process language, as are, evidently, some others (e.g., ALF, EPOS, Merlin). Some other languages are achieve an effect like factoring in that they either depend on or are intended capture aspects of a process related to externally defined elements, such as agents, tools, or artifacts. Some examples inlcude ALF (where agents are defined wholly outside the MASPs, and operators and objects are bound to external tools and artifacts), and ProcessWeaver (in which external agents, tools, and artifacts are coordinated). [REMARK: *Others reasonably in this category? APEL?*] ⇐

Little-JIL has as its primary function the coordination of execution agents. Many process languages provide no first-class representation of execution agent (e.g., APPL/A, HFSP, Marvel, SLANG, Melmac). However, external execution agents are also associated with processes in languages including JIL, Merlin, PEACE, EPOS, Teamware, and ALF. [REMARK: *Check handling in Teamware.*] 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. Little-JIL programs (more strictly, their interpreters) communicate with execution agents, human and automated, via an agenda management system. Other systems that provide agenda-like mechanisms, albeit for human agents, include Merlin, ALF [REMARK: *check*] , and ProcessWeaver. [REMARK: *cite SPADE-1 too – Eric*] ⇐ ⇐ ⇐

Agents in Little-JIL are a distinguished part of a general resource model. As for agents, many languages provide no specific features for resource modeling. The main exceptions support specificaion of (some subset of) agents, tools, and artifacts as specific kinds of entitites available to a step (e.g., PEACE, ALF, EPOS). More general resource modeling is supported in a few cases. [REMARK: *Say something about MVP-L.*] Merlin allows the general technical resources for a work context to be defined but provides no independent resource modeling capability. Oikos models customizable resource managers as services; these may include tool and product repositories, workspaces, actors, and process mod- ⇐

⇒ els. [REMARK: *Check Teamwear.*]

⇒ [REMARK: *The languages reviewed so far are incomplete, and some that are not mentioned in other remarks remain to be considered, although I don't expect any significant expansions of this text.*]

7 Conclusion