Using Little-JIL to Coordinate Agents in Software Engineering

Alexander Wise* Aaron G. Cass* 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
{wise, acass, ljo}@cs.umass.edu

**Department of Computer Science
Williams College
Williamstown, MA 01267 USA
lerner@cs.williams.edu

***HP Laboratories
Palo Alto, CA 94304 USA
emccall@hpl.hp.com

†IBM T. J. Watson Research Center
Hawthorne, NY 10532 USA
suttonsm@us.ibm.com

Abstract

Little-JIL, a new language for programming the coordination of agents is an executable, high-level process programming language with a formal (yet graphical) syntax and rigorously defined operational semantics. 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 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.

1. Introduction

Software engineering activities often involve many human agents and tools that must coordinate to produce a complex artifact such as the design of a large software system. The formalized specification and automated execution of these software engineering activities has been the goal of previous research on process programming. However, to support the complex coordination that must be achieved in software engineering processes, a coordination language has to provide 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. In this paper we present Little-JIL, a process language that attempts to resolve the apparently conflicting objectives of supporting this wide variety of abstractions 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 [27, 28], 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. 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"[7]. From this perspective, it can be seen that coordination is a logically central aspect of process semantics.

As with Linda [7], 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 elements of the process such as artifacts and resources can be related, and their use orchestrated.

Little-JIL is primarily a graphical language. This helps to promote understandability, adoption, and ease of use.

However, Little-JIL language constructs are still defined using precise operational semantics as textual languages typically have.

We believe that focusing on coordination, and allowing the process program to add additional layers of program complexity incrementally 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. We illustrate Little-JIL's features using an example process for solving the familiar problem of trip planning. While this process is not as complex as the software engineering processes for which Little-JIL is designed, it serves as an effective vehicle for demonstrating the language features.

2. Design Principles

Little-JIL draws on the lessons of our earlier work [28] by retaining the "step" as the central abstraction and scoping mechanism but refines the features in terms of which a step is defined. 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. 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. We believe that such decisions must be made through experimentation.

3. Coordination Paradigm

A coordinated activity consists of the following elements:

- A collection of agents each capable of carrying out one or more tasks in support of the activity,
- A communication mechanism enabling the agents to share information,
- A distribution mechanism enabling the agents to operate on separate machines,
- An assignment of tasks to agents, and
- A coordinating process that glues the agents and their tasks together in a manner conducive to accomplishing the coordinated activity.

The coordination paradigm of Little-JIL is one in which independent agents are coordinated in their ability to share information as well as being proactively assigned tasks. All communication between the process and the individual agents takes place via the agent's agenda, which can migrate from machine to machine. A new task is assigned to an agent by placing it on the agent's agenda along with data required to complete the task. The agent informs the Little-JIL interpreter when it has begun a task so that the interpreter can acquire resources on the agent's behalf to assist with the task. The agent also informs the interpreter when it has completed a task, reporting back information to the process such as updated data or exceptional situations that prevented satisfactory completion of the task.

The binding between agents and tasks is done dynamically. In particular, the process program contains declarations of the capabilities the agent must have. Just before assigning a task to an agent, the interpreter uses these declarations of required capabilities to select an agent who has those capabilities and is also available to do the task.

Agents may either be human or software. Little-JIL does not distinguish between them. In particular, both human and software agents have agendas. The distinction lies in how the agents connect to their agendas. Specifically, human agents use a GUI which interacts with the interpreter via a well-defined API, while software agents use this API directly.

Decisions on how a process should proceed may be based upon the following information:

- Whether an individual task was successfully completed or not,

- Which agents and resources are available to support future tasks,
- Decisions made dynamically by the intelligent agents participating in the process. In particular, agents are responsible for making context-specific decisions based upon the data within the process as well as criteria that is external to the process.

Little-JIL's coordination paradigm allows for a range of strictness or flexibility in the execution of the process. This is controlled by the process programmer's choice of constructs and the specificity in the agent capability requirements. These are explained more fully below.

4. Language and Illustrative Example

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. Steps provide a scoping mechanism for control, data, and exception flow and for agent and resource assignment.

As a process executes, 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 operational semantics are provided by the Little-JIL 1.0 Language Report [29].

The main features of the language and their justifications are the following:

- Four non-leaf *step kinds* provide control flow. These four kinds are "sequential," "parallel," "try," and "choice." 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; the agents dynamically decide which step to execute. It is important to note how the parallel and choice step kinds accord to agents the power to exercise their judgment and to make choices about the order in which the substeps of a step 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 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, respectively, 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 [29].
- *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 still undergoing evaluation and evolution, but experience so far has convinced us that 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 communica-

tion of information necessary for the execution of a step and for the return of step execution results.

- *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.

What's missing from the above feature list is also important to note. 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 excluded in order to focus the process program on coordination. We believe this makes the language more applicable to domains in which the agents are primarily autonomous.

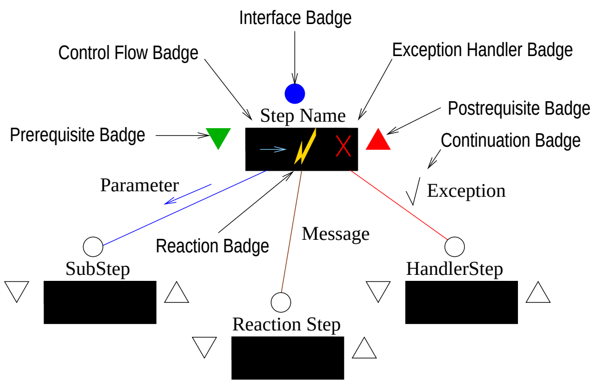

Figure 1. Legend

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 can be 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 [5]. 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 Visual-JIL (our editor for Little-JIL programs) to selectively display information relevant to a particular aspect of a Little-JIL program. Indeed, we illustrate this approach to visualization in the subsequent figures to highlight various language features.

4.1. 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 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.

4.2. Requisites

There are two cases in the example (Figure 2) where requisite steps have been used. A postrequisite has been

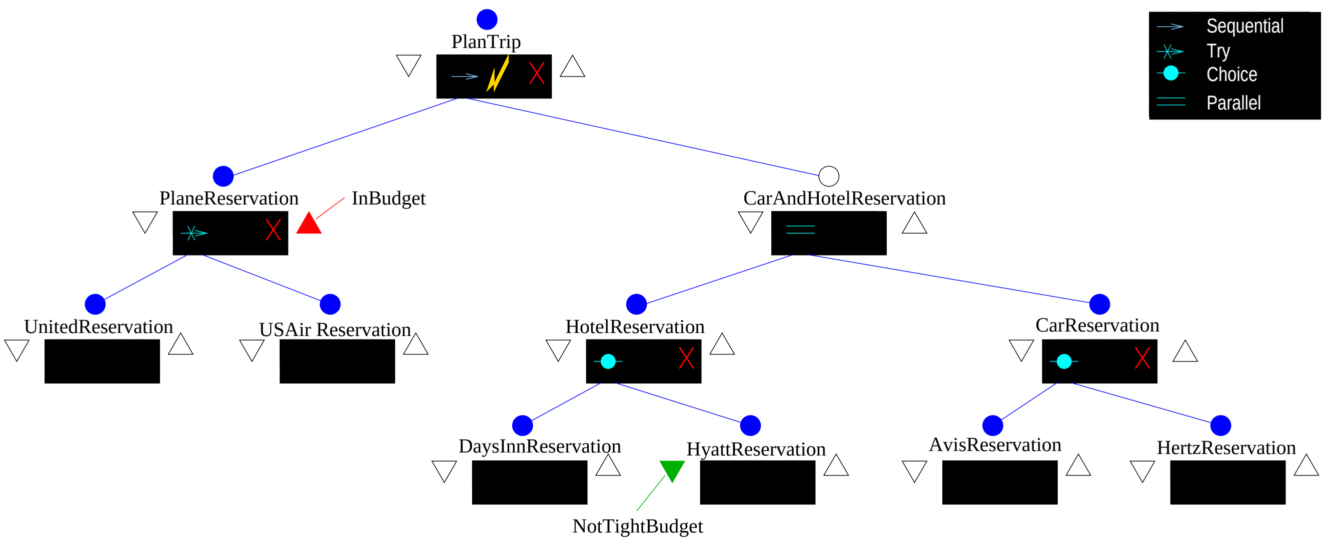

Figure 2. Reservation process showing proactive control: step kinds, requisites.

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 assigned to do the work associated with a step, but the subordinate's supervisor could be responsible for the postrequisite of the step to check 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, the execution agent for PlaneReservation could assign the UnitedReservation and USAirReservation steps to other agents without divulging the budget.

4.3. Exceptions and handlers

If the agent determines that the budget has been exceeded, the agent throws the NotInBudget exception to the parent. The parent step's handler, IncludeSaturdayStayover (in Figure 3[1]), would check to see if a Saturday stayover was 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 up the process 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.

4.4. 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).

4.5. 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

[1]In the figures, ellipses indicate where substeps have been omitted for clarity. Visual-JIL elides information at the user's request.

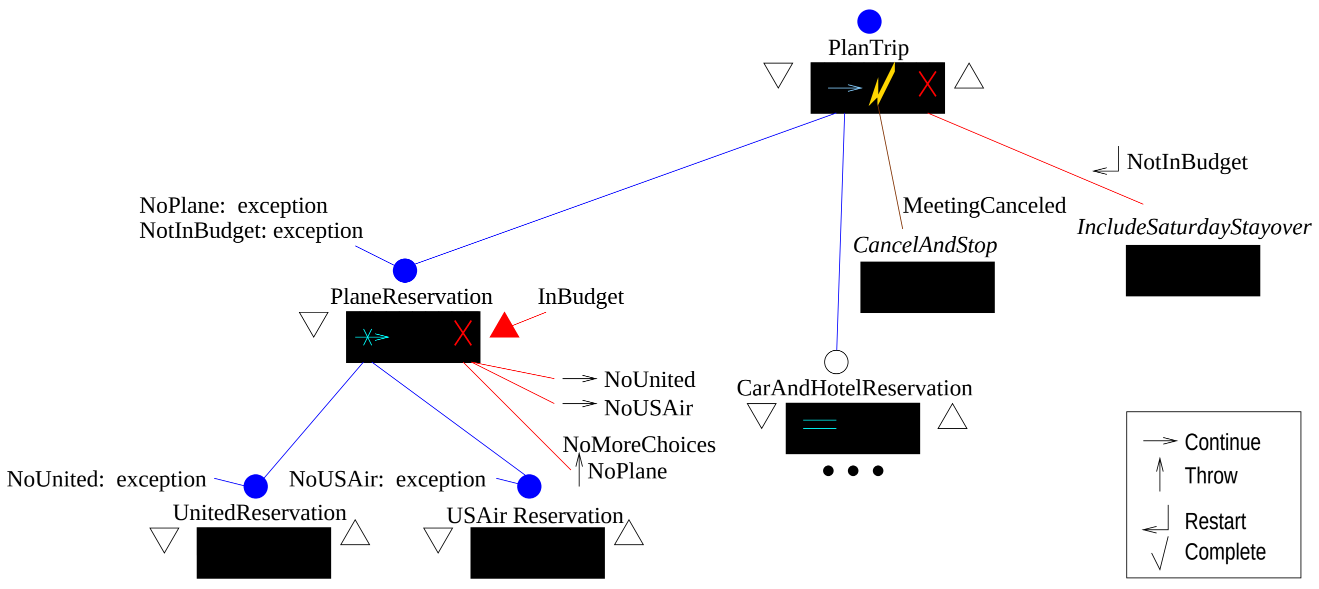

Figure 3. Reservation process showing reactive control: exceptions, messages.

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.

Because Little-JIL is focused on coordination, a process specifies at what points during execution parameter values should be copied to and from steps without specifying the computations to be performed on them. Thus, it is assumed that the agents executing those steps understand the meanings of the parameter values. For example, the use of InBudget as a postrequisite provides guidance about when to check the budget, but doesn't dictate any particular computation for doing so.

4.6. Resources

Resource requirements for a step are indicated by annotations on the step's interface specification and resources play a central role in the execution of Little-JIL programs. 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 the resource model is external to any one process and may be shared by multiple processes, the details of the resource model are not represented in the 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 allow for the possibility that these tasks could be done in parallel by two different secretaries–but in an environment with only a single secretary, we also allow for the dynamic assignment of both of these tasks 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.

5. Related Work

In our research, we have constructed a richly-featured process language including agent coordination, resource dependencies, proactive control constructs, both broadcast and scoped reactive control, data flow specification and pre- and post-requisites. These features are used to specify the set of tasks required, and the ways in which a collection of agents can cooperate to achieve a goal, but still offer the agents flexibility in the way the tasks are performed. This top-down approach contrasts with most work in coordination (e.g., [14, 3, 17, 22, 7]) in which coordination is specified from the perspective of the individual agents, and as such, our work is most directly comparable to workflow systems and other process programming languages. A notable exception is the 'set-plays' in [26]. Set-plays are multi-step multi-agent plans, and as such are similar to our approach of specifying the interactions between the agents from an integrated perspective.

While our use of high-level, process-oriented abstractions and a focus on the "step" as the unit of work separates

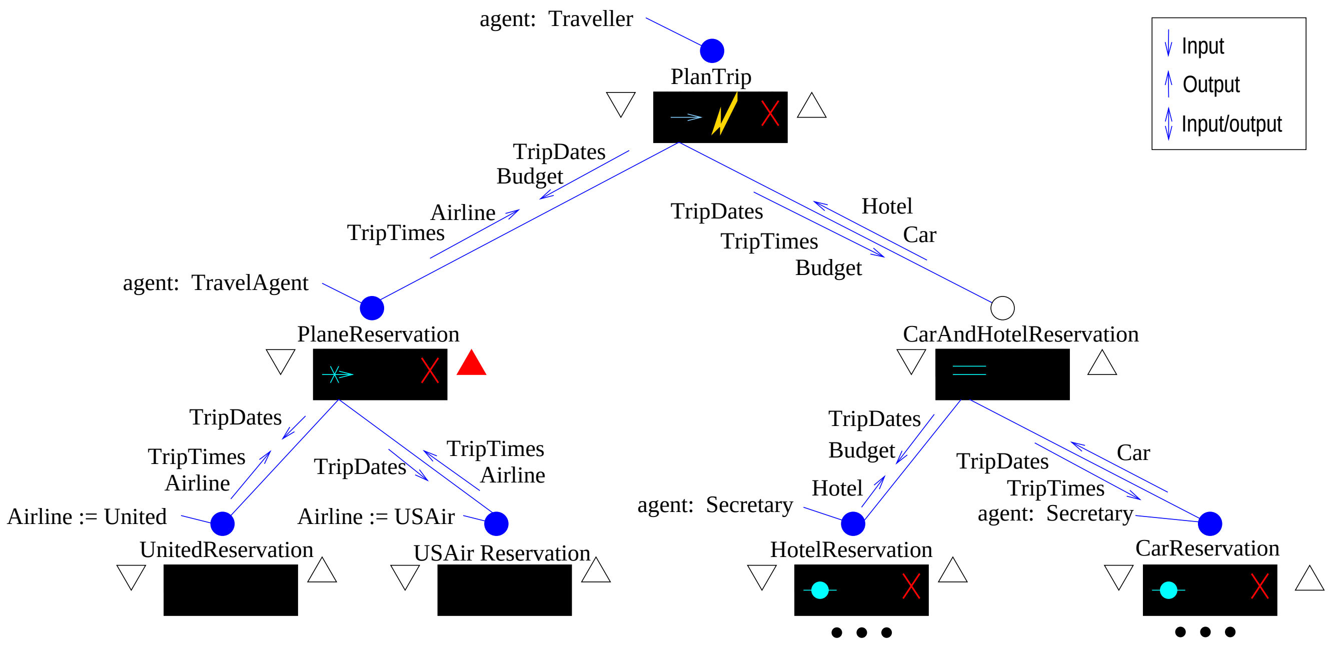

Figure 4. Reservation process showing data flow.

us from process languages based on general-purpose programming languages or Petri-Nets, such as APPL/A [27], AP5 [11], and SLANG [2], many process or workflow languages have focused on process steps (variously also called tasks or activities). For example HFSP [20], EPOS [12], Teamware [30], and APEL [13]. None of the features in Little-JIL are unique, but the way we have combined proactive and reactive control and resource and artifact management into a single consistent abstraction is. For example, while ALF [6] "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 treats human agents and tools separately. ProcessWeaver [16], Merlin [19], and Adele-Tempo [4], focus on notions related to "work contexts" (which may be correlated with steps). Work contexts are generally assigned only to humans and such languages treat tools and humans differently thereby requiring process programmers to determine agent assignments at design time.

APEL [13] is a process modeling formalism with many of our same goals, namely to provide a high-level, expressive, yet executable language suitable for many process domains. While the APEL project has defined many aspects of process modeling in great detail (such as the artifact model, which we leave to a separate model), we have chosen to concentrate on the coordination aspects. APEL's control flow mechanisms are similar to Little-JIL's in that both proactive and reactive control can be specified, and activities can be arranged hierarchically. However, the reactive control in APEL is limited to an un-scoped broadcast mechanism similar to Little-JIL's reaction mechanism. As such, Little-JIL's scoped exception handling has no direct analog in APEL. This scoped exception mechanism allows a flexible, yet careful way to deal with exceptional behaviors at run-time.

6. Experience and Conclusions

We have implemented several tools to support the definition and execution of Little-JIL programs. The tools are written in Java and have been used both on Linux and Windows platforms. The tools include a graphical editor, an interpreter [9], a distributed object substrate, an agenda management system [24], and a resource manager [25].

6.1. Process programs

We have applied Little-JIL to problems ranging from data mining [18], to electronic commerce [8], to the high-level coordination of teams of robots [1]. In the software engineering domain, we have written process programs for coordinating the actions of multiple designers doing Booch Object Oriented Design [23] 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 [15]. In this work we have been particularly interested in using Little-JIL to provide guidance 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].

6.2. Language extensions

To maintain its simplicity, we have resisted impulses to add features to the language, but our experience indicates that it may be necessary to add several new language features to improve expressiveness. For example, we have encountered several idioms that simplify the design and understanding of processes for which a formalism might be useful. The most common of these is *resource-bounded recursion* which allows a step to be repeated multiple times executing with a different resource on each iteration and ceasing when there are no more resources. *Resource-bounded parallelism* is similar to resource-bounded recursion except that in this case the iterations are allowed to happen in parallel. While we can express these idioms currently, introducing appropriate constructs would allow much more concise and understandable representation of the idioms. In particular, we have found that we currently need to use exceptions to terminate the resource-bounded recursion and parallelism. This is an inappropriate use of exceptions since those termination conditions are not exceptional but an essential piece of the idioms.

We have also begun to explore the role of timing in a process program and are developing constructs to support the definition of timing criteria. We intend to use that information to support deadlines for tasks and also to enable scheduling analyses of processes

6.3. Analysis

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 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. In recent work we have begun to demonstrate success in applying the FLAVERS static dataflow analyzer to Little-JIL process programs [10]. 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. We 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.

Much of the detailed behavior of a process is left unspecified in Little-JIL process programs. Rather it is left to the agents because we believe micromanagment of an agent's activities is inappropriate. Because this and many other aspects of the process 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 boundaries separating the process and the software agents to prove certain desirable characteristics of our processes.

Acknowledgments

The authors would like to thank Rodion Podorozhny for his early contributions to Little-JIL and the resource management system, and Yulin Dong, Hyungwon Lee, and Marcia Zangrilli for programming in and providing feedback about many versions of the language. This research was partially supported by the Air Force Research Laboratory/IFTD and the Defense Advanced Research Projects Agency under Contract F30602-97-2-0032. 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/IFTD, or the U.S. Government.

References

[1] E. Araujo, D. Karuppia, Y. Yang, R. Grupen, P. Deegan, B. Lerner, E. Riseman, and Z. Zhu. Software mode changes for continuous motion tracking. In *Int'l Workshop on Self Adaptive Software*, Apr 2000.

[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] M. Barbuceanu and M. S. Fox. COOL: A language for describing coordination in multi agent systems. In *Proc. of the First Int'l Conf. on Multi-Agent Systems*, 1995.

[4] 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,

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

[5] 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.

[6] 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.

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

[8] A. G. Cass, H. Lee, B. S. Lerner, and L. J. Osterweil. Formally defining coordination processes to support contract negotiations. Technical Report 99-39, University of Massachusetts at Amherst, Jun 1999.

[9] 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 99-65, University of Massachusetts at Amherst, Nov. 1999.

[10] 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.

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

[12] 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.

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

[14] K. S. Decker and V. R. Lesser. Designing a family of coordination mechanisms. In *Proc. of the First Int'l Conf. on Multi-Agent Systems*, 1995.

[15] 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.

[16] 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.

[17] T. Finin, Y. Labrou, and J. Mayfield. Kqml as an agent communication language. In *Software Agents*. MIT Press, 1997.

[18] 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.

[19] 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.

[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] K. Kuwabara, T. Ishida, and N. Osato. Agentalk: Coordination protocol description for multi-agent systems. In *Proc. of the First Int'l Conf. on Multi-Agent Systems*, 1995.

[23] 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.

[24] 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.

[25] R. M. Podorozhny, B. S. Lerner, and L. J. Osterweil. Modeling resources for activity coordination and scheduling. In *Proceedings of Coordination 1999*, pages 307–322. Springer-Verlag, Apr 1999. Amsterdam, The Netherlands.

[26] P. Stone and M. Veloso. Task decomposition, dynamic role assignment, and low-bandwidth communication for real-time strategic teamwork. *Artificial Intelligence*, 1999.

[27] 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.

[28] 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.

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

[30] 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.