Proceedings of the 22nd International Conference on Software Engineering. 4-11 June 2000. Limerick, Ireland. Pages 754 – 757.

Little-JIL/Juliette: A Process Definition Language and Interpreter

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

Laboratory for Advanced Software Engineering Research
Department of Computer Science
University of Massachusetts
Amherst, MA 01003-4610 USA
+1 413 545 2013
{acass, lerner, mccall, ljo, sutton, wise}@cs.umass.edu

ABSTRACT
Little-JIL, a language for programming coordination in processes is an executable, high-level language with a formal (yet graphical) syntax and rigorously defined operational semantics. The central abstraction in Little-JIL is the "step," which is the focal point for coordination, providing a scoping mechanism for control, data, and exception flow and for agent and resource assignment. Steps are organized into a static hierarchy, but can have a highly dynamic execution structure including the possibility of recursion and concurrency.

Little-JIL is based on two main hypotheses. The first is that coordination structure is separable from other process language issues. Little-JIL provides rich control structures while relying on separate systems for resource, artifact, and agenda management. The second hypothesis is that processes are executed by agents that know how to perform their tasks but benefit from coordination support. Accordingly, each Little-JIL step has an execution agent (human or automated) that is responsible for performing the work of the step.

This approach has proven effective in supporting the clear and concise expression of agent coordination for 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. We present Little-JIL, a process language that attempts to resolve the apparently conflicting objectives of providing constructs to support a wide variety of process abstractions such as organizations, activities, artifacts, resources, events, agents, and exceptions 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 [5, 6], but it makes some important breaks with this earlier work. Little-JIL differs from our prior work in that it is primarily a graphical language. This helps to promote understandability, adoption, and ease of use. However, Little-JIL language constructs are still defined using the sort of precise semantics that is more typically associated with textual languages. This is facilitated in part because the focus of the language is narrowed to coordination-related elements. This is facilitated by another break from our earlier work, namely the *factoring* of the language into modular language components. Little-JIL focuses on the coordination component

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"[1]. From this perspective, it can be seen that coordination is a logically central aspect of process semantics. As with Linda [1], in Little-JIL we have separated coordination from such other language semantic elements as artifact management, resource management, communication, and real time specification. In addition, because we are concerned about coordination *processes*, Little-JIL incorporates control constructs into the coordination language. This allows processes to constrain coordination when necessary while leaving control decisions in the hands of agents when those constraints are unnecessary.

[*] Currently at Department of Computer Science, Williams College, Williamstown, MA, +1 413 597 4215, lerner@cs.williams.edu
[**] Currently at HP Laboratories, Palo Alto, CA, +1 650 236 2882, emccall@hpl.hp.com
[***] Currently at IBM TJ Watson Research Center, Hawthorne, NY, +1 914 784 7128, suttonsm@us.ibm.com

Because minimizing the process language and factoring out related components permit language complexity to be added incrementally, we believe that this approach can lead to benefits in many areas, including process analysis, understanding, adaptation, and execution.

2 APPROACH

In Little-JIL we identify what we believe to be a viable factoring for a process programming language, and have designed what we believe to be a viable set of linguistic elements that initially focuses on the coordination factor. We have defined this factor so that flexibility can be offered to the agents in the process at appropriate points, and we rely on separate systems for the definition of resource requirements, artifact specification, and agenda management. This factored approach allows the core coordination language to be simpler and easier to understand, develop, and use. Additionally, by factoring out certain aspects of process definition, these aspects can be developed and evolved in independent ways, as appropriate to the environments and organizations in which they will be used. To demonstrate this approach we have also developed examples systems to support these other factors.

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

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

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

Precision: The language semantics are precisely defined. This precision contributes to several important goals. First, it enables automatic execution of process programs (See [2] for a description of Juliette, our runtime environment that executes process programs based on this precise semantics). Second, precision supports the *analyzability* of process programs (see [3] for a discussion of work aimed at static analysis of processes based on this precise semantics). 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.

Flexibility: In some cases, a process programmer needs to control the order in which steps are executed because only one step sequencing may be acceptable. Traditional workflow languages have allowed the specification of this type of sequential control. However, in cases where more than one sequencing is acceptable, or when a different set of steps may be acceptable, a fixed step sequencing imposed by the process program would not be appropriate. Little-JIL is designed to give process programmers the power to choose what level of flexibility to give to the process's agents.

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 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 step types, each of which can be multiply instantiated at runtime. The leaves represent the smallest specified units of work and the tree's structure represents the way in which this work will be coordinated.

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

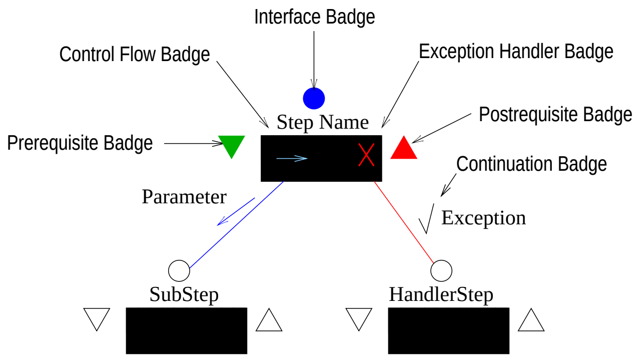

Figure 1: Legend

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 of the step 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 postreq-

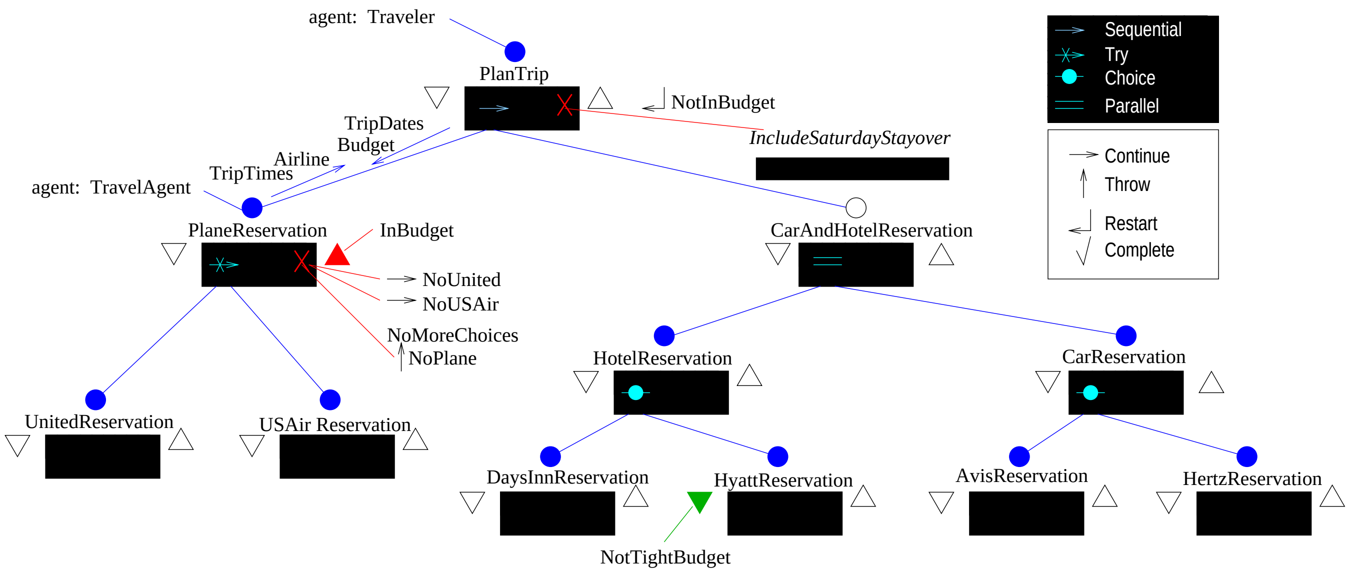

Figure 2: Reservation process showing proactive control: step kinds, requisites.

uisite badge to which a postrequisite step may be attached. Within the box below the step name are two more badges. From left to right, they are the control flow badge, which specifies the order in which substeps are to be executed, and to which the substeps are attached, and the exception handler badge, to which exception handlers are attached. These badges are omitted if there are no child steps or handlers, respectively. The edges that come from these badges can be annotated with parameters (passed to and from substeps) 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.

There are five 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 [7].

In explaining the language features we refer to an example process program for planning a trip (Figure 2). The process describes the activities of first getting a plane reservation, then getting hotel and car reservations for a trip, while remaining within budget. The main features of the language and their *raisons d'être* follow:

Four non-leaf step kinds provide control flow. These four kinds, "sequential," "parallel," "try," and "choice," are the bare minimum for which a need has been clearly established to date. Non-leaf steps consist of one or more substeps whose execution sequence is determined by the step kind. Substeps of sequential steps (such as PlanTrip in Figure 2) are all executed in left to right order. Substeps of parallel steps (such as CarAndHotelReservation in Figure 2) can be executed in any order (including in parallel). Substeps of try steps (such as PlaneReservation in Figure 2) are executed in left to right order stopping when one completes successfully. Exactly one of the substeps of choice steps (such as HotelReservation or CarReservation in Figure 2) is executed with the decision of which to execute being made dynamically by the agent. It is important to note how the parallel and choice step kinds accord to human users the power to exercise their judgment and to make choices about the order in which the subtasks of an item should be performed or how a particular item of work is to be done. While the language can be used to constrain the alternatives, the human agent is left free to make the choices.

Requisites are a mechanism to add checks before and after a step is executed to ensure that all of the conditions needed to begin a step are satisfied and that the step has been executed "correctly" when it is completed. A prerequisite is a step that must be completed before the step to which it is attached. Similarly, a postrequisite must be completed after the step to which it is attached. For example, the postrequisite of PlaneReservation in Figure 2 is a separate step called InBudget. When PlaneReservation is finishing, the InBudget step is posted and executed like any other step. If it throws an exception, this is an indication that the plane reservation has over-extended the budget. 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 to be common in process programs and requisite step semantics seem different enough from other kinds of sequential steps that a special notation was introduced. These constructs also seem ideal for supporting the monitoring of process execution.

Exceptions and handlers augment the control flow constructs of the step kinds. Exceptions and handlers are used to indicate and fix up exceptional conditions or errors during program execution and provide a degree of reactive control that we believe allows a process programmer to simply and accurately codify common processes. The exception mechanism in Little-JIL has been designed to be simple yet remain expressive. It is based on the use of steps to define the scope of exceptions and handlers. Exceptions are passed up the step decomposition tree (call stack) until a matching handler is found. Our experience has indicated that it is necessary to allow different exception handlers to work in a variety of ways. After handling an exception, a continuation badge determines whether the step will continue execution, successfully complete, restart execution at the beginning, or rethrow the exception. For example, in the process in Figure 2, the handler attached to PlanTrip for the NotInBudget exception has a restart continuation badge, indicating that if we are not InBudget, we should first IncludeSaturdayStayover and then restart the trip planning process. Detailed semantics are provided in [7].

Parameters passed between steps allow communication of information necessary for the execution of a step and for the return of step execution results. The type model for parameters has been factored out of Little-JIL, thus removing issues such as type definition and equality, which are unrelated to coordination. For lack of space, we don't show all the parameters and their bindings in Figure 2. The left side of the diagram shows some of the parameter passing on the edges (for example, the PlanTrip step passes the TripDates and Budget to the PlaneReservation step, and gets back TripTimes and Airline).

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. In Figure 2, PlanTrip has an agent specified as Traveler and PlaneReservation has an agent specified as TravelAgent. Resource management is not done in Little-JIL, but is carried out by a resource specification and management factor. As such, the resource specifications in Figure 2 are interpreted only by the resource management factor. As with parameters, Little-JIL attempts to minimize the requirements placed on the resource specification factor: Little-JIL requires little more than that the factor support the identification of resources that match a specification, and that it support resource acquisition and release to avoid usage conflicts.

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

We have developed Juliette, a runtime environment based on Little-JIL that provides implementations of these other language factors so that process programs can be executed [2]. Resources are managed by a prototype resource manager, the Java type system is used as the type model for parameters, and an Agenda Management System [4] is used to assign work to possibly mobile agents. The Little-JIL language interpreter of Juliette is built on top of a Distributed Object Substrate that allows the pieces of the interpreter to be distributed close to the agents with which they must interact.

4 CONCLUSION

To date we have used Little-JIL to define a wide range of processes from domains as diverse as software engineering, robot control, and electronic commerce. We have successfully supported the execution of many of these processes and have begun to apply powerful static analyzers to the process definitions to prove critical properties. This experience suggests that our approach to process definition has very broad applicability.

Reports on the details of Little-JIL can be found at `http://laser.cs.umass.edu`.

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, Timothy Sliski, 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 and by the U.S. Department of Defense/Army and the Defense Advance Research Projects Agency under Contract DAAH01-00-C-R231. 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/IFTD, the U.S. Dept. of Defense, the U. S. Army, or the U.S. Government.

REFERENCES

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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