Programming Process Coordination in Little-JIL

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

Computer Science Department
University of Massachusetts
Amherst, Massachusetts 01003
{lerner, ljo, sutton, sandy}@cs.umass.edu
http://laser.cs.mass.edu **

Abstract. Process programming languages have not been readily adopted by practitioners. We are addressing this problem through the development of Little-JIL, a language that focuses on the coordination aspects of processes and provides a visual representation, yet one that is rigorous enough for execution and formal reasoning. We have used Little-JIL to program several software engineering processes, knowledge discovery processes, and are working on processes to coordinate robot teams. We believe the simplicity gained by focusing on coordination and visualization should make Little-JIL both readily adoptable and widely useful.

1 Introduction

Process programming remains a challenging problem for software process technology support. The complexity of process languages and the effort of process programming using these languages has hindered both the development and adoption of process programming technology. We are addressing these problems of process language design and development through a three-part strategy:

1. Focusing on a subset of process requirements related to *coordination*, in particular, to the coordination of activities and agents
2. Defining *appropriate abstractions* and language constructs to capture important coordination aspects of software processes
3. Devising *visual* yet *rigorous* representations to aid adoption of the language

By focusing on coordination, we address a necessary aspect of all software processes. Moreover, our experience suggests that the programming of activity

** This research was supported by the Air Force Research Laboratory/IFTD and the Defense Advanced Research Projects Agency under Contract F30602-97-2-0032. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Air Force Research Laboratory/IFTD or the U.S. Government.

and agent coordination is largely sufficient for software processes in many contexts. Additionally, we can reduce the complexity of the language, simplifying its definition and development, and facilitating its use and evaluation.

We have identified the following language features as especially important for programming coordination: 1) a multi-paradigm control model, 2) a few basic but general constructs for proactive control, 3) preconditions and postconditions on process steps, 4) resource specification, modeling, and management, 5) flexible exception handling, 6) explicit, controlled data flow, 7) agent specification, binding, and delegation.

The main features missing from the above list are the usual features of a general programming language, such as imperative commands and data type definitions; also missing are constraint (or rule) specifications, common in process languages. These omissions simplify language definition, implementation, and use. Moreover, we have found that many processes decompose into high-level coordination and low-level computations. Coordination operators are found in some process languages (e.g., [1, 2]) and blackboard languages (e.g., [3]). We propose using a coordination language to program the high-level coordination aspects, while using a computational process language (such as JIL [1]) or a general programming language to implement the computational elements.

Finally, to facilitate use of the language, we have adopted a visual syntax. An important characteristic of this syntax is that it has rigorous semantics that allow it to be executed. While there are many graphical process definition languages, most lack adequate semantic depth or rigor to support execution. Exceptions include Petri-net-based systems and a few others (e.g., [4–7]). A distinguishing characteristic of our approach is the style of program representation, which emphasizes the hierarchical composition of individually simple steps rather than the typical approach of arbitrarily complex, nested flow graphs. In our experience this has helped to promote the clarity of our process coordination programs without sacrificing expressiveness.

2 Example

Little-JIL is a process language we have developed according to the three-part strategy outlined above. To demonstrate the coordination abstractions of Little-JIL, we use a high-level description of Booch's object oriented design (BOOD). In BOOD, designers first identify classes and objects, then identify their semantics and the relationships between them, and then design the internal implementation of the classes. This process repeats until all of the classes are elaborated in sufficient detail.

Figure 1 shows a partial implementation of BOOD. We highlight how Little-JIL supports coordination by discussing the example, leaving many details unexplained. Also, it is important to realize that this is not a complete definition of BOOD. The Little-JIL editor allows the user to selectively hide information to avoid a cluttered presentation and allow a concise high-level picture of a process. A detailed understanding is gained by selectively displaying detail.

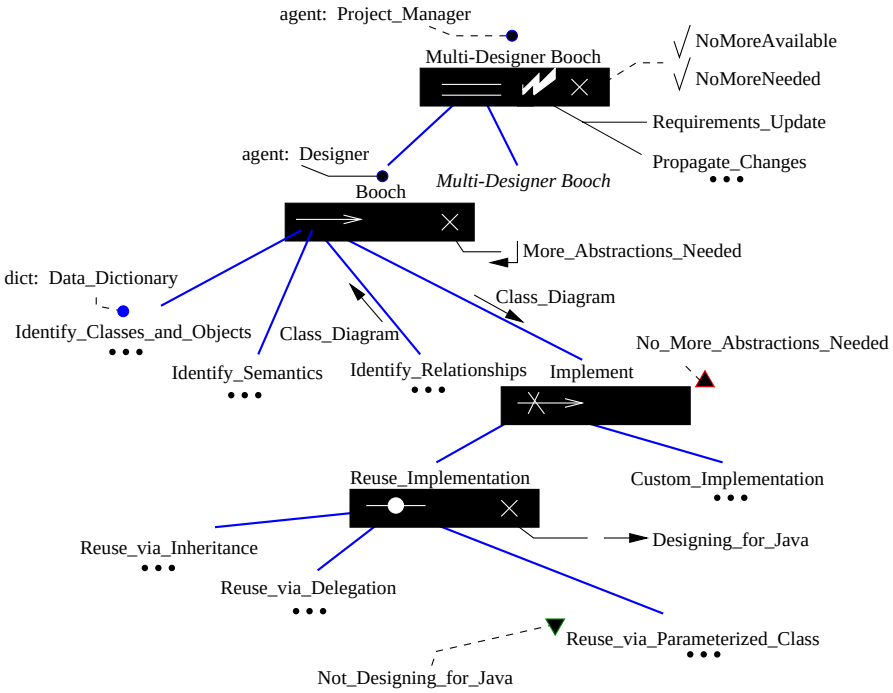

Fig. 1. A Multi-User Booch Process (BOOD) in Little-JIL

Each step in a process is visually represented by a step name surrounded by several graphical badges which represent aspects of the step. The black bar below the step represents the control of a step's substeps. The left-most badge in the step bar defines the *proactive control flow*. For example, step `Multi-Designer Booch` has a `parallel` badge which allows, but does not require, the substeps to be performed in parallel. Step `Booch` contains a *sequential* badge indicating that its substeps should be executed from left to right sequentially. Step `Reuse_Implementation` is a *choice* indicating that the agent assigned to the step should choose one of the substeps as a means of completing the task. `Implement` is a *try* step which requires alternatives to be tried sequentially until one succeeds. The arrows next to the arcs represent *data flow* between steps and substeps.

Resources are part of a step interface, identified by a circle above a step name. (Step interfaces also include declarations of formal parameters, local objects, and exceptions thrown, which are omitted from the figure.) By default, resources are acquired in mutual exclusion, but we are investigating a variety of forms of sharing and delegation to allow more concurrency. For example, the `Data_Dictionary` is updated when identifying classes and objects. It is declared as a resource for that step, ensuring coordinated use of the data dictionary.

Each Little-JIL step has an *execution agent* associated with it. An agent may be human or automated. Agents are treated as special types of resources. If a

step does not declare its own agent, it uses the same agent as its parent. In the figure, we see two types of agents: a project manager and a designer.

Resource-bounded parallelism is a Little-JIL idiom. In this example, a project manager assigns tasks to individual designers. Each designer can work in parallel with the manager making assignments. At any point the manager can stop making assignments or the project could run out of designers to whom assignments can be made, limiting the parallelism by the number of designers available.

Little-JIL steps may have *preconditions* and *postconditions*. Preconditions are represented by downward pointing triangles to the left of the name of a step and postconditions by an upward pointing triangle to the right of the step name. Conditions are checked at the beginning and end of the execution of a step to ensure that the step can be, or was, executed correctly. If a condition fails, it throws an exception either preventing the step from executing, or preventing it from completing successfully. *Exception handlers* are identified by an "X" at the right end of the step bar. An exception handler may specify a step to be executed when the exception is raised and how the step should continue after handling the exception. The `Reuse_Implementation` step reacts to the `Designing_for_Java` exception, thrown by the `Not_Designing_for_Java` precondition, by requiring the designer to try another alternative (indicated by the arrow) to get reuse.

The "lightning bolt" in the step bar represents *reactive control*, that is asynchronous reactions to global events. For example, when the requirements change, a step is started that propagates the changes to the affected designers.

The syntax of Little-JIL is unique and therefore requires some learning. We have found, however, that it provides a very concise, high-level picture of a process while allowing details to be expanded selectively. Furthermore, the precisely defined semantics of the constructs, although not fully described here, allow meaningful discussion and rigorous analysis of process definitions.

3 Discussion

The Little-JIL strategy focuses on coordination while deemphasizing computation. This is most appropriate for processes that rely on elements that are external to the process program. This includes processes that are largely manual or otherwise agent based, where the agents can benefit from coordination or where the process can benefit from agent control. It also includes processes that require coordinating use of independent tools, artifacts, etc. The approach is generally applicable when coordinative and computational aspects can be separated, or to processes that are simply non-computational.

We have found coordination processes to be common in many domains. We have written Little-JIL programs for several software processes, including object-oriented design, formal verification, and static analysis, as well as for the various domains noted above. Coordination process programming naturally lends itself to workflow processes, which frequently have human agents and use existing external tools. We have also been exploring the applicability of Little-JIL to processes of knowledge discovery in databases (KDD) [8]. At a low level, KDD

processes can be computationally intensive. At higher levels, though, we have found that KDD processes involve combinations of human and automated agents, and management of resource usage that make them suitable for coordination process programming. We are also beginning to explore the use of Little-JIL in coordinating teams of reconfigurable autonomous robots.

4 Status and Directions

Preliminary versions of the Little-JIL language and editor have been developed and demonstrated; work is being completed on the first standard version (1.0). A supporting agenda manager [9] and resource manager are being developed (prototypes are operational), and work on integrating Little-JIL with AI-based scheduling [10] is in progress. Work on support for visualization, analysis, measurement, and evaluation is beginning. We plan to continue developing coordination processes in a variety of domains.

References

1. Sutton, Jr., S.M., Osterweil, L.J., The Design of a Next-Generation Process Language. In: Proc. of the Joint 6th European Software Engg. Conf. and the 5th ACM SIGSOFT Symposium on the Foundations of Software Engg. Springer-Verlag (1997) 142–158.
2. Canals, G., Boudjlida, N., Derniame, J.-C., Godart, C., Lonchamp, J.: ALF: A Framework for Building Process-Centred Software Engineering Environments. In: Finkelstein, A., Kramer, J., Nuseibeh, B., (eds): Software Process Modelling and Technology. John Wiley & Sons Inc. (1994) 153–185.
3. Montangero, C., Ambriola, V.: OIKOS: Constructing Process-Centered SDEs. In: Finkelstein, A., Kramer, J., Nuseibeh, B., (eds): Software Process Modelling and Technology. John Wiley & Sons Inc. (1994) 33–70.
4. Bandinelli, S., Fuggetta, A., Ghezzi, C., Lavazza, L.: SPADE: An Environment for Software Process Analysis, Design, and Enactment. In: Finkelstein, A., Kramer, J., Nuseibeh, B., (eds): Software Process Modelling and Technology. John Wiley & Sons Inc. (1994) 223–248.
5. Gruhn, V., Jegelka, R.: An Evaluation of FUNSOFT Nets. In: Proc. of the Second Eurpoean Workshop on Software Process Technology. Trondheim, Norway (1992).
6. Dami, S., Estublier, J., Amiour, M.: APEL: A Graphical yet Executable Formalism for Process Modelling. Automated Software Engg., **5** (1998) 61–96.
7. Young, P.S., Taylor, R.N.: Human-Executed Operations in the Teamware Process Programming System. In: Proc. of the Ninth Intl. Software Process Workshop. (1994).
8. Jensen, D., Dong, Y., Lerner, B.S., Osterweil, L.J., Sutton Jr., S.M., Wise, A.: Represening and Reasoning about Knowledge Discovery Processes. In: Seventh Intl. Conf. on Information and Knowledge Management. (1998). Submitted.
9. McCall, E.K., Clarke, L.A., Osterweil, L.J.: An Adaptable Generation Approach to Agenda Management. In: Proc. of the 20th Intl. Conf. on Software Engg. (1998).
10. Garvey, A., Decker, K., Lesser, V.: A Negotiation-Based Interface between a Real-Time Scheduler and a Decision-Maker. In: Proc. of the Workshop on Models of Conflict Management in Cooperative Problem Solving. AAAI (1994).