HI-PLAN and Little-JIL: a Study of Contrast between Two Process Languages

Hyungwon Lee
Department of Computer Science
Kangnung National University
Kangnung, Kangwondo 210-702 KOREA
lhw@knusun.kangnung.ac.kr

Leon J. Osterweil
Department of Computer Science
University of Massachusetts
Amherst, MA 01003-4610 USA
ljo@cs.umass.edu

Abstract

A key issue for process languages is balancing the need for technical rigor with this need for ease of use. Little-JIL and HI-PLAN are new powerful, yet clear, process languages that attempt to resolve above two apparently conflicting objectives. This paper evaluates both the languages and provides valuable directions to the next-generation process language design. We describe their design goals and features, present solutions to a well-known benchmark process, ISPW-6 software process example, and analyze the relative strengths and weaknesses of each language through detailed comparisons on a wide variety of issues.

1. Introduction

Process language research was an early emphasis of software process studies, but it has still remained vital and challenging. There have been two colliding approaches to the process language design [6, 11, 13]. First, many process languages are based on the premise that software processes can and should be described in terms of a wide variety of semantics: organizations, activities, artifacts, resources, events, agents, exceptions, and so on. These languages are powerful and semantically rich, but likely to be complex and hard to use, especially for non-programmers. Contrarily, other language designers have adopted a number of strategies toward simplification; these include narrowing of semantic focus or depth and the use of graphical representations. While such strategies may indeed foster linguistic simplicity, the practical utility of simplified languages has been limited. Consequently, first-generation process languages have obvious limitations.

Little-JIL [6, 12, 13] and HI-PLAN [4, 5] are new process languages that attempt to resolve above two apparently conflicting objectives: semantic richness and ease of use. Little-JIL is an executable, high-level process language with a formal, yet graphical, syntax and rigorously defined operational semantics. It focuses on agent coordination as a key process factor. The premise of this focus is that processes are conducted by agents who understand their tasks but who can benefit from coordination with other agents [13]. HI-PLAN is a process language based on modeling formalism using extended data flow diagram. It focuses specification of process entities and their relationships. Both the languages have different design philosophies and valuable features that have not yet been incorporated into each other, which means both can benefit from each other.

In this paper, we introduce these two process languages and provide useful lessons for the next-generation process language design through extensive analyses on their relative strengths and weaknesses.

The applicability and efficiency of both the languages

have been explored and demonstrated by applying to a variety of processes. However, in order to compare and evaluate in the most objective way, both the languages are applied to the ISPW-6 software process example (ISPW-6 example) [3], a standard benchmark software process. ISPW-6 example, though may be simpler than some specific domain processes, enables to exercise a process language comprehensively, because it contains a large number of different types of process issues seen in real world and provides a firm and consistent basis for the solution.

The rest of this paper is organized as follows. Section 2 and 3 provides brief descriptions of Little-JIL and HI-PLAN, respectively. Section 4 discusses the relative weaknesses and strengths of two languages, and the conclusion is drawn in section 5.

2. ISPW-6 Example with Little-JIL

2.1 Overview of Little-JIL

Little-JIL is an *agent coordination* language; programs in Little-JIL describe the coordination and communication among agents that enables them to perform a process. A Little-JIL program is a tree of steps whose leaves represent the smallest specified unit of work and whose structure represents the way in which this work will be coordinated. As processes execute, steps go through several states. Normally, a step is *posted* when assigned to an execution agent, and then *started* by the agent. Eventually the step is either successfully *completed* or *terminated* with an exception. The followings are six main features of Little-JIL language that allow a process programmer to specify the coordination of steps in a process [13].

- Four *non-leaf step kinds* ("sequential", "parallel", "try", and "choice") provide control flow and are sufficiently expressive to capture a wide range of step orderings.
- *Requisites* are a generalization of pre- and post-

conditions. A prerequisite/postrequisite must be completed before/after the step to which it is attached.

- *Exceptions and handlers* are used to indicate and fix up exceptional conditions or errors during program execution and provide a degree of reactive control. Exceptions are passed up the tree until a matching handler is found. 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.
- *Messages and reactions* are another forms of reactive control. While exceptions propagate up the program tree, messages are global in scope so that any execution step can react to a message.
- *Parameters* passed between steps allow communication necessary for the completion of a step and for the return of step execution results. The type model for parameters has been factored out of Little-JIL.
- *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.

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. The interface includes the declaration of the agent who is to carry out the step, resource requirements of the step, variable declarations, exceptions that may be thrown, and messages that may be sent.

Below the circle is the step name. To the left is a triangle called the *prerequisite badge*. The prerequisite is a step that must be successfully completed for this
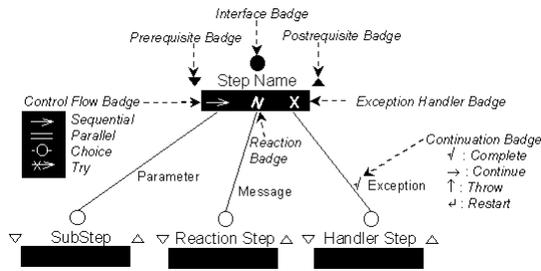
Figure 1 Little-JIL Legend

step to begin execution. If the prerequisite is not completed successfully, an exception is thrown and the step is not allowed to execute. 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*. The postrequisite step begins execution immediately after the step completes execution and must also successfully complete for the parent to be notified of the step's completion. If the postrequisite does not complete successfully, an exception is thrown.

Within the box below the step name are three badges. From left to right, they are the *control flow badge*, *reaction badge*, and *exception handler badge*. Substeps are connected to the control flow badge. The edge connecting a step to a substep is annotated with parameter passing. The control flow badge indicates the order in which the substeps may be executed. Little-JIL defines four control flow badges. A *sequential* step executes its substeps in order from left to right, beginning the next substep only after the preceding substep completes successfully. A *parallel* step allows the substeps to be executed concurrently. A sequential or parallel step requires all of its substeps to be performed. A *choice* step allows the agent performing the step to choose which single substep to execute. A *try* step identifies alternative ways of performing the step but hardwires the order in which the alternatives should be tried from left to right. A choice or try step requires exactly one of its substeps to be performed

successfully. A step with no control flow badge is a leaf step and is directly executed by an agent. A step whose name is in italics is a reference step and is defined elsewhere in the process.

The lightning bolt in the middle is the reaction badge to which reaction steps are attached. A reaction identifies a broadcast message that it responds to.

The X is the exception handler badge to which exception handlers are attached. An exception handler identifies the exception that it is handling, optionally a step to perform to handle the exception, and a *continuation badge* to indicate what to do after completion of the handler step. There are four continuation badges. The *continue badge* indicates that the execution of the step should continue. For sequential and parallel steps, this is as if the substep that threw the exception completed successfully. For choice and try steps, this allows an agent to perform a different alternative. The *restart badge* indicates that the entire step should be restarted from the beginning. The *complete badge* indicates that the entire step should be considered successful. The *rethrow badge* indicates that the entire step should be considered unsuccessful and the exception should be thrown again to the steps parent.

2.2 Little-JIL Solution to the ISPW-6 Example

In this section we illustrate the ISPW-6 example process in Little-JIL. Figure 2 shows the entire solution. The various steps of the figure could be elaborated to capture further details of this process, but space does not permit this here. The entire programs and descriptions are provided in [4].

Decomposition into the details

A Little-JIL program is a tree of steps with a root step that represents the entire process. Each step represents a unit of work in the process and may be decomposed into substeps. Ellipses indicate when substeps have been omitted for clarity.
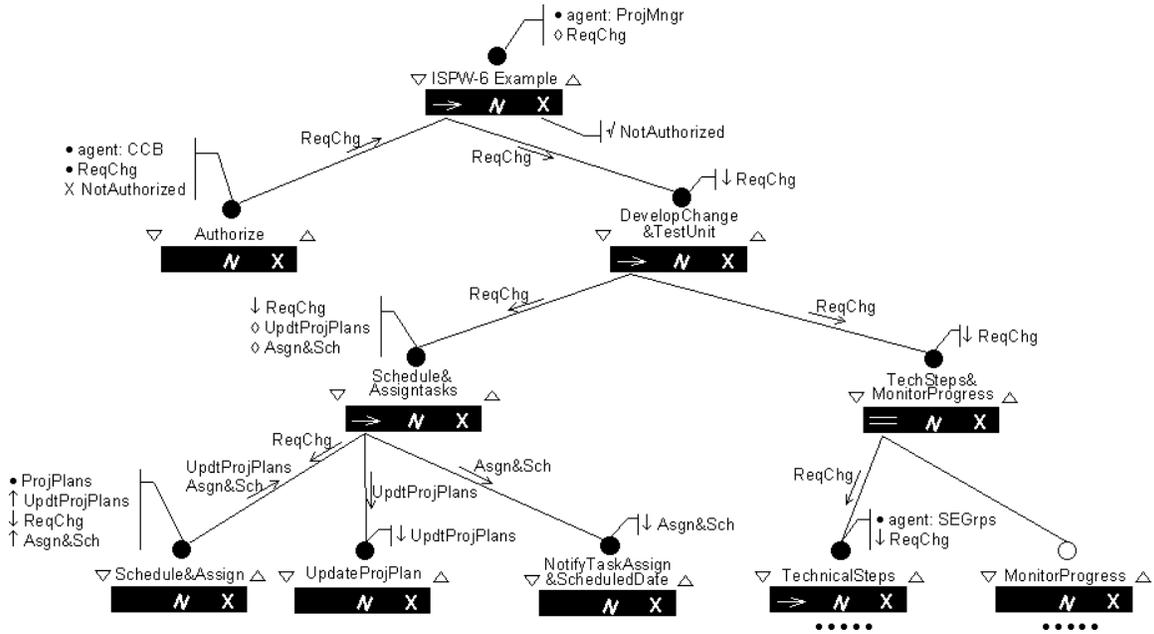
Figure 2 ISPW-6 in Little-JIL: Upper Steps

Step ordering

As described in section 2.1, the control flow badge of each step indicates how substeps should be executed. In figure 2, for example, the sequential step ISPW-6 Example is used to authorize the change before developing it. The TechnicalSteps and MonitorProgress steps can be executed concurrently as indicated by the control flow badge of their parent step TechSteps&MonitorProgress.

Input/source, output/destination, and agent

Local definitions associated with a step, such as resources and parameters are attached to the interface badge of the step. Resources of a step include an execution agent, physical inputs from file, and so on. The declaration of a resource is distinguished by a dot, such as ProjPlans in the Schedule&Assign step in figure 2. If no agent is specified for a step, the execution agent is inherited from the step's parent. Artifacts can be passed between steps via parameters. They have to be declared in the interface badge as one of

four parameter modes: *in* parameter (↓) whose value is copied into a step when the step is started, *out* parameter (↑) whose value is copied out when the step completes, *in/out* parameter (↓↑) whose value is copied into a step when the step is started and copied out when the step completes, and *local* parameter (◊) allowing the passing of information between its substeps. Arrows attached to the parameters indicate whether a parameter is copied into the substep's scope from the parent, copied out, or both. A sequence of parameter passing represents information flow. In figure 2, for example, the ReqChg is passed from the Authorize step to the DevelopChange&TestUnit step via the parent of both steps.

3. ISPW-6 Example with HI-PLAN

3.1 Overview of HI-PLAN

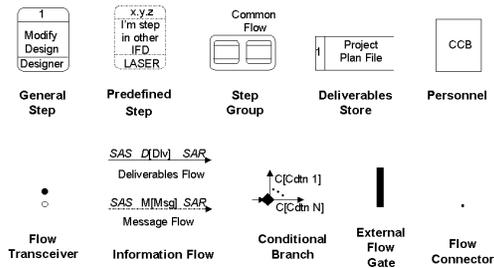HI-PLAN [5] is a structured process modeling language. The internal concepts and the external nota-

Figure 3 IFD Legend

Figure 4 State Transitions of Step in IFD

tions of HI-PLAN are based on the Ward and Mellor's extended structured analysis for the real time system. Process models of HI-PLAN are described with three modeling tools as followings.

- *Information Flow Diagram (IFD)*: A set of IFDs is to visualize the entities and their various relationships. A step in an IFD can be decomposed into the substeps that will compose a sub-IFD.
- *Information Definition Dictionary (IDD)*: An IDD is a form to define entities represented in IFDs. IDDs enable to describe artifacts, their physical storages, and agents.
- *Task Specification (TSPEC)*: A TSPEC is to describe a step in detail.

Due to space constraints, we can only give an overview of IFD. The legend of IFD is shown in figure 3. IFD provides three entity types: *step*, *personnel*, and *deliverables store*. A step of an IFD is considered as a black box whose agent is represented below step name. Steps can be grouped for sending or receiving common information. A personnel can be individual or decomposable organization. A deliverables store is the physical storage of related artifacts that can be used or produced by steps. These entities and the *information flow*s connecting them[1] allow representing both behavioral and functional perspectives together in a diagram and simulating or tracing step behaviors

---
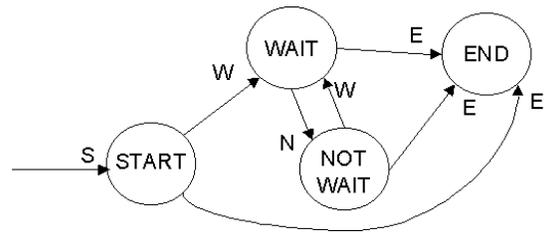
[1] At least, one of the entities should be a step.

easily. The basic idea is that the state of a step tends to be changed when the step produces some output, which influences the executions of other steps.

An information flow may be either *deliverables flow* or *message flow*. Whether the arc of an information flow is solid or dashed depends on its type. A deliverables flow can be used to transfer any artifact to an entity and a message flow can be used to transfer notification of message to an entity.

An information flow between entities can have two indicators at the ends of its arc: *SAS* (*State After Sending*) near the source entity and *SAR* (*State After Receiving*) indicator near the destination entity. The SAS indicator of an information flow represents the state of its source step after the step sends the designated information. The SAR indicator of an information flow represents the state of its destination step after the step receives the designated information. If any indicator of an information flow is not specified, the state of the step relevant to it is not changed after sending or receiving the designated information.

A step in IFD can be in one of four states *NEWS*: N (Not wait), E (End), W (Wait), and S (Start). Possible state transitions are shown in figure 4. Normally, a step starts when an artifact or message with S SAR indicator arrives to the step, then goes through suspension and resumption, and eventually ends.

In IFD, an information flow incident on a step should be connected to a *flow transceiver* attached to the step. Flow transceivers are used to reduce the ambiguity and increase the traceability of step behaviors when a number of information flows are connected between
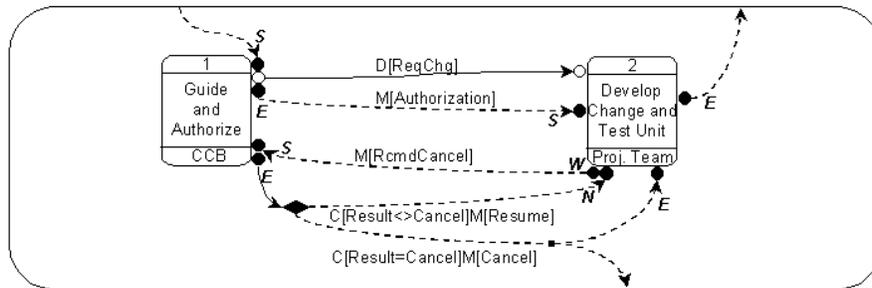
Figure 5 ISPW-6 in HI-PLAN: Highest Level IFD

entities. A flow transceiver is filled if the indicator relevant to it exists, and is empty otherwise. Flow transceivers can be utilized in several ways. If one transceiver is used for multiple input (output) information flows, the step state transition occurs only when all input (output) flows have been received (sent). A group of overlapping transceivers attached to a step represents a thread of step behavior, i.e. a sequence of step execution, when the step is included in multiple sequences of step execution.

Conditional branch is used for the representation of complex decision making to select or iterate more than one step attached to it. An information flow from a conditional branch can have a condition specification to decide which destination step(s) should be selected or iterated.

3.2 HI-PLAN Solution to the ISPW-6 Example

In this section we illustrate the ISPW-6 example process in HI-PLAN. The partial solution is composed of three IFDs.

Decomposition into the details

In HI-PLAN, any entity can be decomposed. Also, artifact can be decomposed. IFDs in figure 5, 6, and 7 show the decompositions of the Develop Change and Test Unit step and its substep Technical Steps in turn. Each component in an IDD can be decomposed further and defined in another IDD.

Step ordering

HI-PLAN has no explicit notation for the representation of step ordering. Instead, how steps should be executed is derived from the SAS/SAR indicators attached to the information flows and the conditional branches, which enables to derive various step orders sufficient to model the complex software processes. In figure 5, the Develop Change and Test Unit step is started after the Guide and Authorize step ends. In figure 7, the Test Unit step is started after both Modify code and Modify Unit Test Package steps end. In figure 6, the Technical Steps and Monitor Progress steps are started simultaneously after the Schedule and Assign Tasks step ends. In figure 7, after the Review Design step ends, the Modify Design step, the Modify Code step, or both the steps can be started according to the designated condition.

Input/source, output/destination, and agent

A step's agent represented in the step icon is defined in an IDD. Inputs and outputs for a step are represented by the information flows that flow into and out the step. Also, the source for each input or the destination for each output can be identified easily. A source or a destination can be a step, a file (i.e. deliverable store), or a personnel. An information flow to a step, a deliverables store, and personnel means transferring, storing, and notifying the designated information, respectively.
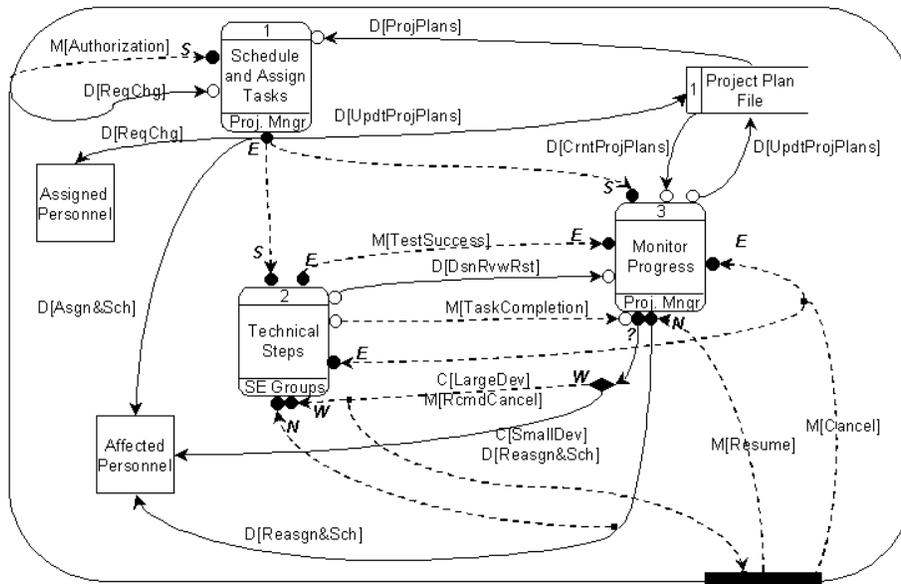
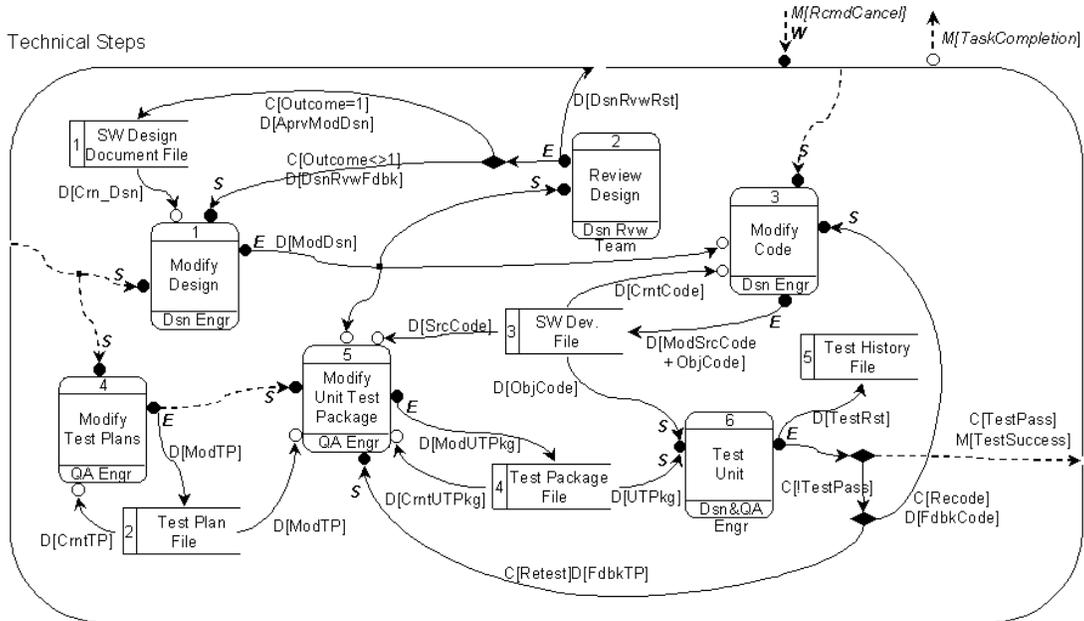**Figure 6 ISPW-6 in HI-PLAN: Upper Level IFD**

Figure 7 ISPW-6 in HI-PLAN: Technical Step IFD

For example, in figure 7, the Test Unit step gets the UTPkg artifact from the Test Package File and stores the TestRst artifact at the Test History File.

The sending or receiving order of the information flows for a step can be identified clearly within the sub IFD of the step.

Table 1 Little-JIL vs. HI-PLAN: Fundamentals

	Little-JIL	HI-PLAN
Main objective	Agent coordination	Process specification
Usage	Process execution	Process understanding & Communication
Structure	Hierarchical tree	Hierarchical network
Central abstraction	Step	Step, Storage, Personnel
Control structure	Control flow badge Pre/postrequisite badges Reaction badge Exception handler badge Continuation badge	Information flow Conditional branch

4. Comparisons

In this section, we evaluate Little-JIL and HI-PLAN on four aspects: fundamental issues, step decomposition issues, semantics issues, and qualitative issues.

4.1 Fundamental Comparison

Little-JIL and HI-PLAN are graphical process languages, but have different design principles, objectives, and features (Table 1).

First, Little-JIL is a language for programming agent coordination in processes. Because the focus is narrowed to coordination-related elements, Little-JIL provides a rich set of control structures for agent coordination. As mentioned earlier, Little-JIL is based on the hypothesis that processes are executed by agents who know how to perform their tasks but who can benefit from coordination support. Each step in Little-JIL is assigned to an execution agent (human or automated) responsible for initiating the step and performing the work associated with it [13]. On the other hand, HI-PLAN is a language for modeling the various aspects of a software process. The focus is the specification of process entities and their relationships. Accordingly, HI-PLAN concentrates on how easily and efficiently the entities and their relationships are

represented. Second, the major usage of a Little-JIL program is process execution, though agents are responsible for performing the work. The process specification written in HI-PLAN is for supporting process understanding and communication. Third, both the languages are different in their graphical models. Little-JIL uses hierarchical tree model that emphasizes the hierarchical breakdown of a process while keeping the within-step flow simple. HI-PLAN uses hierarchical network model that emphasizes the horizontal flow within a step while allowing hierarchical decomposition. Fourth, step is the only central abstraction in Little-JIL, but one of the central abstractions in HI-PLAN. Last, in order to describe detailed, precise, and adequate control, Little-JIL provides a variety of control structures for step execution. Contrarily, in order to maximize simplicity, HI-PLAN provides only two structures sufficient to represent step behavior.

4.2 Step Decomposition

Focus on step decomposition

In Little-JIL, step decomposition in Little-JIL is carried out based on both *process abstraction* and *control abstraction*, where process abstraction means that a step can be divided into smaller steps and control abstraction means that the control flow badge of a step determines the execution sequence of its substeps. Such step decomposition enables to represent a wide range of step orderings but causes two problems. First, Little-JIL programs are likely to have long depth due to the decomposition by control abstraction. Non-leaf steps are essential for the control of their substeps, but they may decrease understandability. Second, if both the abstractions come into conflict when a step is decomposed, control abstraction should prevail over process abstraction for execution efficiency and as the result, the steps constituting a step have to be scattered (See example 1 in the next.).

In HI-PLAN, step decomposition is based on only process abstraction. Control abstraction is handled in
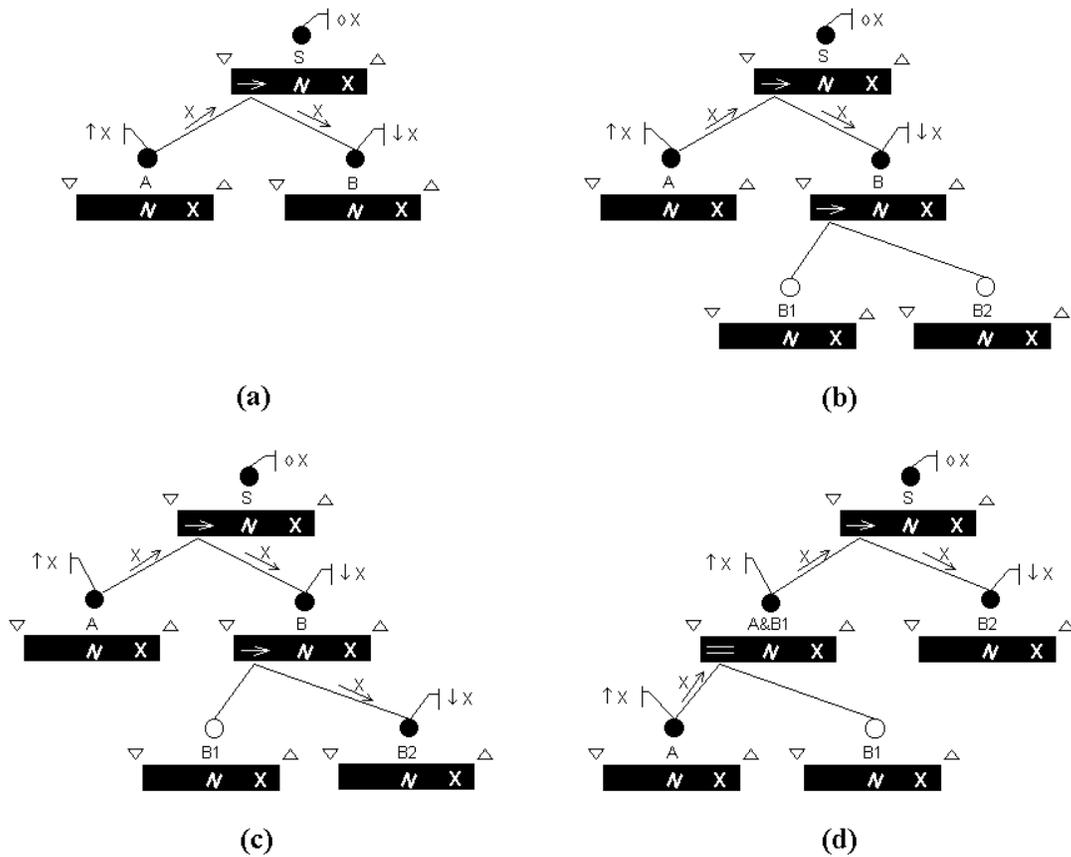
(a)

(b)

(c)

(d)

Figure 8 A Little-JIL Solution to Example 1

each IFD through information flows. Therefore, step decomposition is simpler and IFDs have shorter depth than Little-JIL program has. Decomposed steps, however, are modeled in separate diagrams and the overall process structure is not as clear at a glance as Little-JIL program.

Development approach

Little-JIL is not always suitable for top-down approach. Top-down development in Little-JIL programming may cause restructuring program structure. This is due to the facts that process abstraction can conflict with control abstraction while step decomposition and that inputs and outputs of a step are treated as parameters and results. HI-PLAN is adequate for top-down approach i.e. "Divide and Conquer". Let's

consider following example.

Example 1. Development approach
1) Step S is decomposed into substep A and B
2) Step A outputs x and step B uses x
3) Step B is decomposed into substep B1 and B2
4) Step B2 starts after completing B1
5) Only step B2 uses x (step B1 does not use x)

Figure 8 shows how Little-JIL solves example 1. From 1) and 2), step S seems to be a sequential step (figure 8(a)). From 3) and 4), step B must be decomposed into sequential substeps (figure 8(b)). Now, there are two possibilities to handle 5). One is to maintain process abstraction as in figure 8(c), where the starting step B1 will have to be delayed until step A is
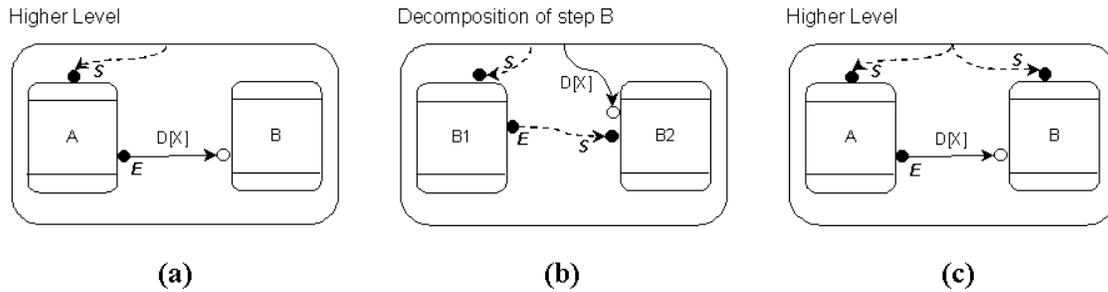
(a)   **(b)**   **(c)**

Figure 9 A HI-PLAN Solution to Example 1

terminated and it is impossible for step B2 to start as soon as step A is terminated. The other is to focus on control abstraction to maximize execution efficiency as in figure 8(d), where the program structure must be changed and the substeps of step B are scattered. If decomposition of these substeps is continued, re-structuring may be repeated.

Figure 9 shows a HI-PLAN solution. Figure 9(a) is the top-level IFD from 1) and 2). Figure 9(b) is the result of decomposition. After decomposition of step B, the top-level IFD does not have to be changed except adding the information flow to start step B (figure 9(c)).

4.3 Qualitative Issues

Understandability

Basically, both the languages are easy to understand due to their visual syntax. Especially, in order to maintain simplicity, Little-JIL separates out many process-related factors not directly relevant to coordination. Furthermore, its graphical notation centered on the step keeps well-organized program with few connections, and the various badges making up a step represents a variety of control structures in clear and easy manner. However, Little-JIL programs tend to have long depth, because step decomposition of Little-JIL concentrates on control abstraction as well as process abstraction.

The graphical notation of HI-PLAN is intuitive. IFD of HI-PLAN has shorter depth and fewer steps than Little-JIL program has, because step decomposition of IFD concentrates on process abstraction. But, IFD can become cluttered because it uses net-based model. Also, understandability may decrease according to the placements of entities.

Ease of use

In spite of rich and rigorous semantics Little-JIL provides, the separation of the semantic issues into separate graphical components makes programming in Little-JIL easier than that of general-purpose programming language. Little-JIL editor makes it easier to program and modify. A caution is that artifact dependencies among steps largely influence the program structure. So, careful analysis on those dependencies must precede actual programming.

HI-PLAN provides relatively small set of features familiar to most software engineers, which makes user comfortable. HI-PLAN is easy to use at the beginning of modeling, but it may become difficult to draw or modify due to the placements of entities even with editor, as diagram becomes complicated. So, careful placements and decomposition are required.

Ease of use is more important to Little-JIL because process is executed according to Little-JIL program; therefore, Little-JIL programmer is responsible for the results of process execution. Programmer describes not only every sequence of steps but also artifact dependencies between steps. Contrarily, HI-PLAN is a process modeling language; thus all sequences of processes are derived by (human) interpreter.

Flexibility

Little-JIL is designed to interoperate with additional specification languages and supporting services to allow for the expression of process factors that are not addressed in Little-JIL. Also, Little-JIL provides explicit, scoped exception handling to handle the abnormal situations. *Resource-bounded recursion* and *resource-bounded parallelism* [13] allows a step executed flexibly when resource constraints exist. The rich semantics of Little-JIL enable a control issue to be solved in several ways.

HI-PLAN provides flexible semantics on sending and receiving information. For example, it's possible for non-leaf steps to receive information before starting their executions. This enables top-down decomposition without having upper level IFDs affected. Also, any information transfer can cause state transition.

5. Conclusions and Future Work

Ease of use is an important requirement for process languages because the individuals and organizations responsible for defining processes are often not experienced at programming or modeling. The semantic richness of process languages means, however, that significant software engineering skills are required to program or model in them effectively. This is an impediment to the widespread adoption of process languages. A key issue for process languages is thus balancing the need for technical rigor with this need for ease of use [11]. In order to examine a process language whether it satisfies this or refine to do so, there should be some experiments to evaluate and analyze a process language by detailed and close comparison with other languages on a variety of issues concerning above two language design goals. This approach also enables to find what state-of-the-art process languages can provide and what next-generation process languages should provide.

In this paper, we have presented two powerful, yet clear, process languages, Little-JIL and HI-PLAN, and described their design philosophies and features. Also, we have identified their relative strengths and weaknesses through applying them to a well-known benchmark process, ISPW-6 software process example. We found that both the languages have their own merits, but also found that they should be refined to be better languages. We believe Little-JIL and HI-PLAN to benefit much from this experiment.

We will evaluate the languages continuously through applying to other process domains. Also, We are continuing to refine the languages so that our experience can be reflected in them.

REFERENCES

1. Aaron G. Cass, Hyungwon Lee, Barbara Staudt Lerner, and Leon J. Osterweil. Formally Defining Coordination Processes to Support Contract Negotiation. Technical Report 99-39, Department of Computer Science, University of Massachusetts at Amherst, June 1999.

2. David Jensen, Yulin Dong, Barbara Staudt Lerner, Eric K. McCall, Leon J. Osterweil, Stanley M. Sutton Jr., and Alexander wise. Coordinating Agent Activities in Knowledge Discovery Processes. In *Proceedings of International Joint Conference on Work Activities Coordination and Collaboration (WACC '99)*, February 1999, San Francisco, CA.

3. M. I. Kellner, P. H. Feiler, A. Finkelstein, T. Katayama, Leon J. Osterweil, M. H. Penedo, and H. D. Rombach. ISPW-6 Software Process Example.

In *Proceedings of the First International Conference on Software Process,* pp. 176-186, IEEE Computer Society Press 1991.

4. Hyungwon Lee. Evaluation of Little-JIL 1.0 with ISPW-6 Software Process Example. Technical Report 99-33, Department of Computer Science, University of Massachusetts at Amherst, March 1999.

5. Hyungwon Lee and Chisu Wu. HI-PLAN: A Structured Project Planning Method. In *Journal of Korea Information Science Society,* Vol. 23, No. 8, pp. 821-831, August 1996.

6. Barbara Staudt Lerner, Leon J. Osterweil, Stanley M. Sutton Jr., and A. Wise. Programming Process Coordination in Little-JIL - Towards the Harmonious Functioning of Parts for Effective Results. In *Proceedings of the 6th European Workshop on Software Process Technology (EWSPT '98),* number 1487 in Lecture Notes in Computer Science, pp. 127-131, Weybridge, UK, September 1998, Springer-Verlag.

7. Barbara Staudt Lerner, Stanley M. Sutton, Jr., and Leon J. Osterweil. Enhancing Design Methods to Support Real Design Processes. In *9ᵗʰ IEEE International Workshop on Software Specification and Design*, pp. 159-161. IEEE Computer Society Press, April 1998.

8. Leon J. Osterweil. Software Processes Are Software Too, Revisited. In *Proceedings of the Nineteenth International Conference on Software Engineering*, pp. 540-548, May 17-23, 1997, Boston, MA.

9. Stanley M. Sutton Jr., Dennis Heimbigner, and Leon J. Osterweil. APPL/A: A Language for Software-Process Programming. *ACM Trans. on Software Engineering and Methodology*, 4(3):221-286, July 1995.

10. Stanley M. Sutton Jr., Barbara Staudt Lerner, and Leon J. Osterweil. Experience Using the JIL Process Programming Language to Specify Design Processes. Technical Report 97-68, Department of Computer Science, University of Massachusetts at Amherst, September 1997.

11. Stanley M. Sutton Jr. and Leon J. Osterweil. The Design of a Next-Generation Process Language. In *Proceedings of the Joint 6th European Software Engineering Conference and the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering,* pp. 142-158, Springer-Verlag, 1997.

12. A. Wise. Little-JIL 1.0 Language Report. Technical Report 98-24, Department of Computer Science, University of Massachusetts at Amherst, April 1998.

13. A. Wise, Barbara Staudt Lerner, Eric K. McCall, Leon J. Osterweil, and Stanley M. Sutton Jr.. Specifying Coordination in Processes Using Little-JIL. Technical Report 99-71, Department of Computer Science, University of Massachusetts at Amherst, November 1999.