Evaluation of Little-JIL 1.0 with ISPW-6 Software Process Example

Hyungwon Lee

Laboratory for Advanced Software Engineering Research
Computer Science Department
University of Massachusetts
Amherst, MA, 01003, USA
hlee@cs.umass.edu or lhw@knusun.kangnung.ac.kr

Abstract

Little-JIL is a new process language that focuses on the coordination aspects of processes and provides a formal yet graphical syntax and rigorous operational semantics. This report is to evaluate and analyze Little-JIL 1.0, the current version of Little-JIL. I present solutions to the ISPW-6 software process example with both Little-JIL and another process language HI-PLAN, and then compare both languages on a variety of process language issues. I propose the changes and additions to Little-JIL through the inclusion of some features of HI-PLAN.

1. Introduction

Process language research was an early emphasis of software process studies, but it has still remained vital and challenging and there have been two colliding approaches to the process language design [4, 9, 11]. First, many process languages are based on the premise that 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 languages have obvious limitations.

Little-JIL [4, 10, 11] is a new process language that attempts to resolve these two apparently conflicting objectives: semantic richness and ease of use. Little-JIL 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 [11].

This report evaluates and analyzes the current version of Little-JIL (Little-JIL 1.0 [10]). The applicability and efficiency of Little-JIL have been explored and demonstrated by writing of process programs from a variety of application areas: coordinating the actions of multiple designers doing Booch Object Oriented Design [5], guiding the use of the FLAVERS data flow analysis tool set, and coordinating the agent activities in knowledge discovery processes [1]. This report has some different viewpoints and approaches from the earlier publications on Little-JIL. First, in order to evaluate Little-JIL objectively, I use the ISPW-6 software process example (ISPW-6 example) [2], 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. Second, in order to draw easily the strengths and weaknesses of Little-JIL, I compare Little-JIL with another process language HI-PLAN that has different design philosophy and objective. HI-PLAN is based on modeling formalism using extended data flow diagram and Little-JIL is strongly rooted in the past research on process programming languages [7, 9], though it makes some important breaks with earlier work. HI-PLAN is relatively simple, limited and ambiguous, but has valuable features that have not yet been incorporated into Little-JIL. I believe both languages can benefit from each other. Last, in order to make Little-JIL more attractive and valuable, I try to identify potential drawbacks of Little-JIL and suggest solutions to those as far as possible.
The rest of this report is organized as follows. Section 2 provides a brief description of the ISPW-6 example. The Little-JIL and HI-PLAN solutions to ISPW-6 example are described in Section 3 and Section 4, respectively. Section 5 discusses the weaknesses and strengths of Little-JIL, and the conclusion is drawn in Section 6.

2. ISPW-6 Software Process Example

ISPW-6 example [2] is a standard software process used to compare various software process modeling approaches. The core problem focuses on the designing, coding, unit testing, and management of a rather localized change to a software system. This is prompted by a change in requirements, and can be thought of as occurring either late in the development phase or during the support phase of the life cycle. The problem begins with the project manager scheduling the change and assigning the work to appropriate staff. The example problem ends when the new version of the code has successfully passed the new unit tests. The elements of ISPW-6 example are as followings.

- Steps
  The entire process: Develop Change and Test Unit
  Component steps: Schedule and Assign Tasks
  Modify Design
  Review Design
  Modify Code
  Modify Test Plans
  Modify Unit Test Package
  Test Unit
  Monitor Progress
- Organization
  CCB (Configuration Control Board)
  Project Team: Project manager
  Design engineers
  QA engineers
  Software engineers
- Artifact
  Input + Source (file, agent, or step) + Physical communication mechanism
  Output + Destination (file, agent, or step) + Physical communication mechanism
- Constraints regarding step sequencing

3. Programming ISPW-6 Example with Little-JIL 1.0

3.1 Overview of Little-JIL 1.0

Little-JIL is a visual language derived from a subset of JIL, a process language developed for software development processes [9]. 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 [11]. Detailed language features and semantics are provided by the language report [10].

- 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.

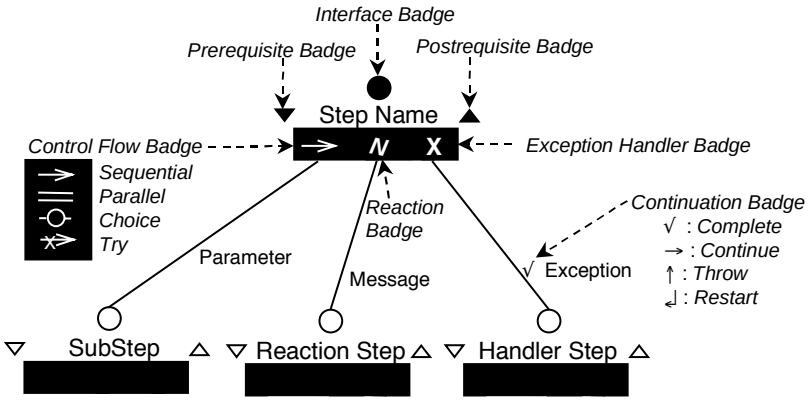


Figure 1 Little-JIL Legend

3.2 Little-JIL Solution to the ISPW-6 Example

In this section I illustrate the ISPW-6 example process in Little-JIL. From figure 2 to figure 8 shows the entire solution. To better describe the features and the use of Little-JIL, pairs of each important issue in ISPW-6 example and its solution are provided.

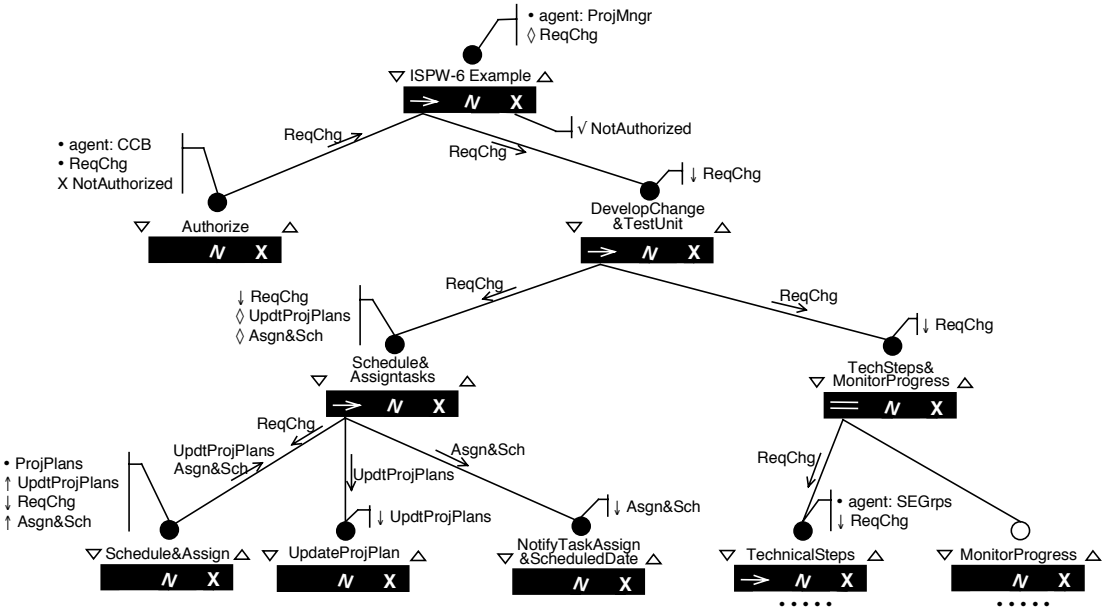


Figure 2 ISPW-6 in Little-JIL: Upper Steps

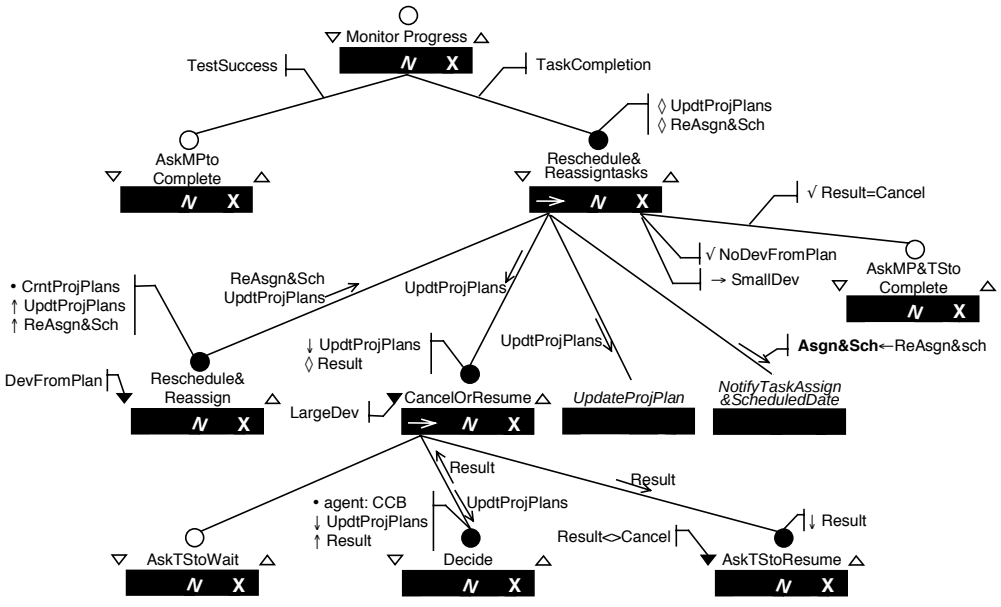


Figure 3 ISPW-6 in Little-JIL: Monitor Progress

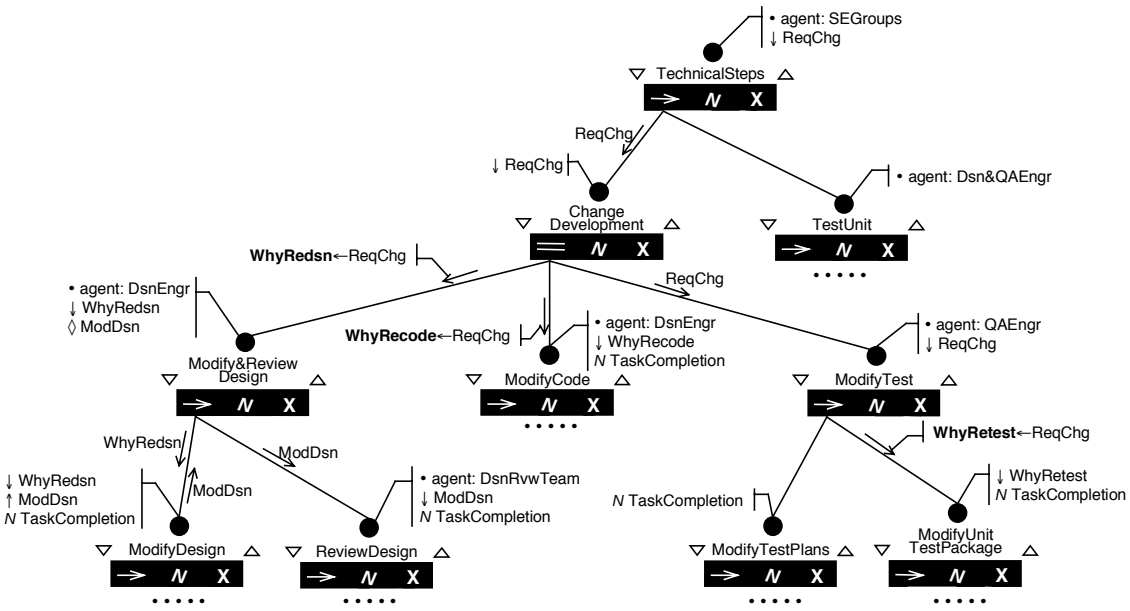


Figure 4 ISPW-6 in Little-JIL: Technical Steps

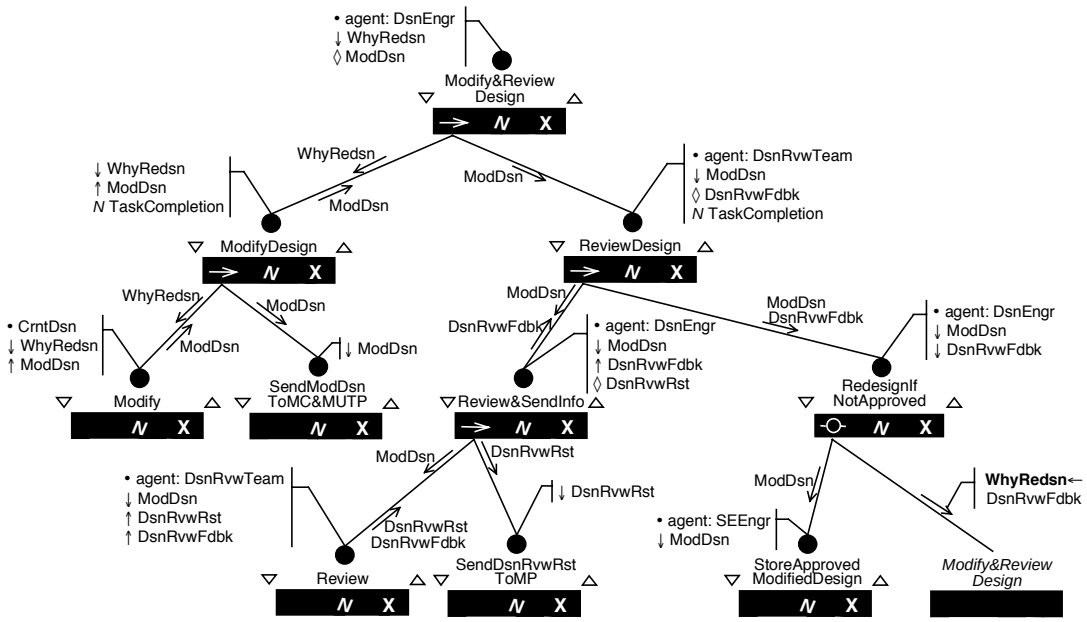


Figure 5 ISPW-6 in Little-JIL: Design

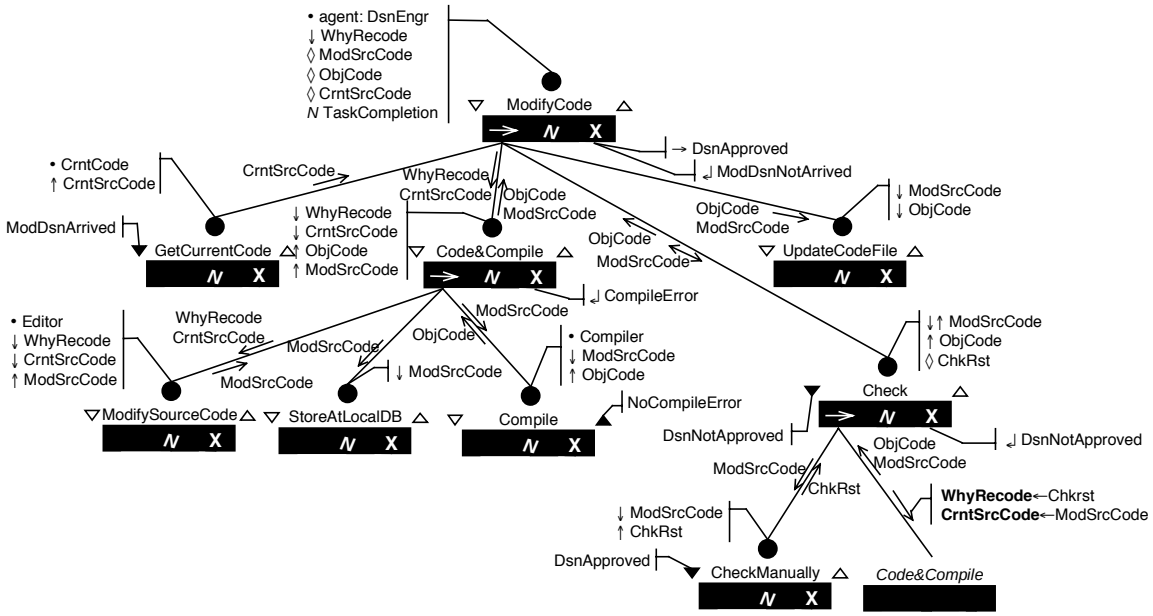


Figure 6 ISPW-6 in Little-JIL: Modify Code

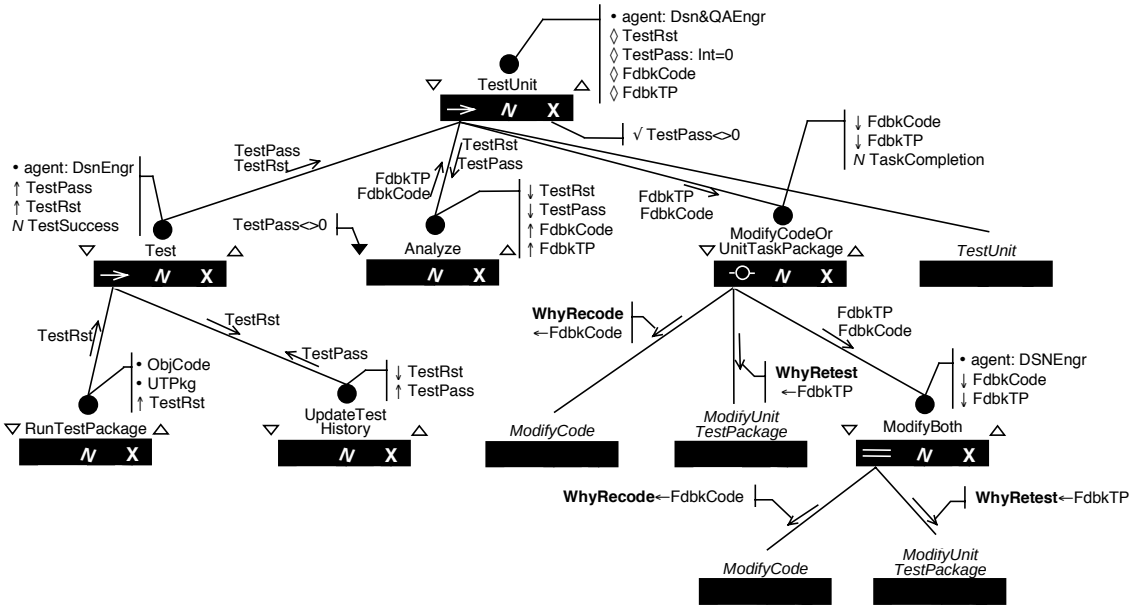


Figure 7 ISPW-6 in Little-JIL: Test Unit

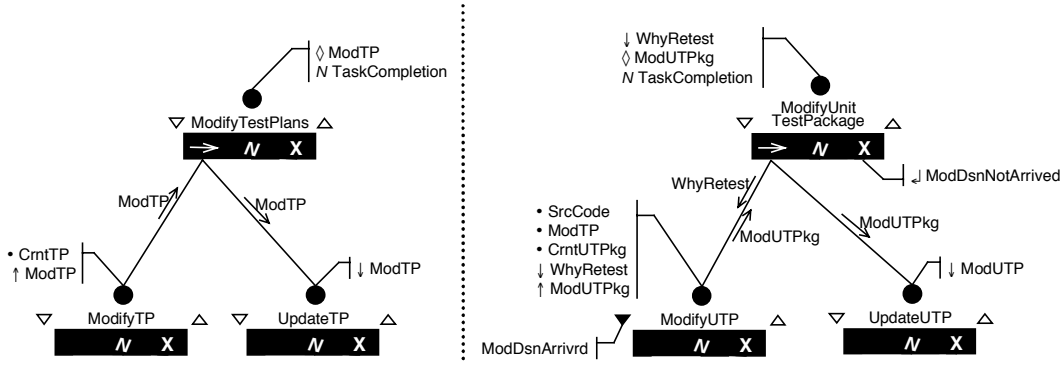


Figure 8 Modify Test Plans and Modify Unit Test Package

High-level abstraction and its 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.

Step ordering

The control flow badge of each step indicates how substeps should be executed. A sequential step's substeps are posted in left to right order until either all complete or one terminates. A parallel step's substeps are simultaneously posted and it completes when all its substeps complete. A try step's substeps are posted in sequence until one is completed. A choice step's substeps are posted simultaneously until one is started. In figure 2, the sequential step ISPW-6 Example is used to authorize the change before developing it. In figure 7, the choice step ModifyCodeOrUnitTestPackage is used to allow a design engineer and a QA engineer to choose modifying code, modifying unit test package, or both.

Input and source, output and destination, and agent

Local definitions associated with a step, such as resources, parameters, exceptions, and messages, 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; for example, the agent of the DevelopChange&TestUnit step in figure 2 is a project manager. Information is passed between steps via parameters. 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 arrows with parameters represents information flow; in figure 5, for example, the information ModDsn is passed from the ModifyDesign step to the ReviewDesign step via the parent of both steps.

Constraints regarding the step sequencing

A step's constraints in ISPW-6 example regard the sequencing of the step: when the step is started, ended, and iterated. Such constraints are classified into several categories as followings.

- The constraint on the precedence between steps can be simply represented by the control flow badge. For example, a sequential step is used for the constraint that the ModifyUnitTestPackage step can begin as soon as the ModifyTestPlans step has completed (figure 4).

- Iteration-related constraint is represented by a reference step with italicized text and without badges. Of course, a mechanism to end an iteration must be provided. The choice step in figure 5 reflects the constraint that subsequent iterations of the ModifyDesign step can begin when the ReviewDesign step is completed and the design is not approved.
- The constraint on the conditions to start or complete steps is represented by prerequisite or postrequisite. The postrequisite NoCompileError attached to the Compile step in figure 6 represents the constraint that the ModifyCode step ends when a clean compilation has been accomplished. If there are any errors after compilation, an exception CompileError (not shown) is thrown to the parent step Code&Compile, and then, after an attempt is made to match the exception with the handler exception specifications, the Code&Compile step is restarted as indicated by the continuation badge.
- Exceptional condition or error that can terminate a step during its execution is represented by an exception and handler; if the requirements change is not authorized during the Authorize step in figure 2, an exception NotAuthorized is thrown to the parent and the ISPW-6 Example step is terminated by the continuation badge.
- An execution step may be terminated by messages from other steps. Message and reaction solve this constraint in rather indirect manner. For example, consider the constraint that the MonitorProgress step ends when the unit testing has been successfully completed. If the message TestSuccess is sent by the Test step in figure 7, the MonitorProgress step in figure 3 responds to the message arrival and then the AskMPtoComplete step is executed to ask for the termination of the MonitorProgress step.
- Resource-related constraints, such that the TestUnit step can begin as soon as both the object code and unit test package are available, are represented naturally by the resources declaration. The RunTestPackage step (child step of TestUnit) in figure 7 cannot start, if acquiring the resources ObjCode and UTPkg fails.
- A number of constraints on the step completion in ISPW-6 example require that a step should end when its outputs have been provided. Normally, a step in Little-JIL program is terminated after generating its all outputs, because the value of a parameter in a step is copied out when the step completes.

4. Modeling ISPW-6 Example with HI-PLAN

4.1 Overview of HI-PLAN

HI-PLAN [3] is a structured process modeling language. The internal concepts and the external notations of HI-PLAN are based on the Ward and Mellor's extended structured analysis for the real time system. The design goals of HI-PLAN are as follows.

- *Ease of Use*: The complexity of process languages and the effort using these languages has hindered the adoption of process technology. Moreover, the individuals and organizations responsible for defining software processes are often not experienced at process modeling and/or programming. HI-PLAN is based on the idea that a simple and efficient approach for ease of use is to provide the syntax and semantics familiar to most software engineers.
- *Support of Multiple Perspectives*: Process language must provide four distinct but interrelated perspectives for analyzing and presenting process information. Functional perspective represents what steps are being performed and what information flows are relevant to these steps. Behavioral perspective represents when steps are performed and how they are performed through feedback loops, iteration, complex decision-making conditions, entry and exit criteria, and so forth. Organizational perspective represents when and by whom steps are performed, physical communication mechanisms used for

information transfer, and physical media and locations used for storing entities. Informational perspective represents both the structure of informational entities and the relationships among them. When combined, these perspectives will produce an integrated, consistent, and complete model of the process analyzed.

- *Visualization*: Most process languages are textual because visual representations are unsuitable for complex processes in general. But, visualization is essential to increase the understandability and communicability.
- *Relationships between entities*: The real world has many entities, such as products, processes, resources, and so on. Also, there are a variety of relationships between entities. Efficient process modeling languages must represent the entities and their relationships unambiguously and naturally.
- *Hierarchical Decomposition*: Hierarchical decomposition is very important to represent complex processes. It is desirable for all the entities to be decomposed into sub-entities, if necessary.

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. IFDs represent the functional and behavioral perspectives of problem process.
- *Information Definition Dictionary (IDD)*: An IDD is a form to define individual information entities represented in IFDs. IDDs enable to describe the organizational and informational perspectives of artifacts, their physical storages, and agents.
- *Task Specification (TSPEC)*: A TSPEC is used to describe a step in detail for supporting organizational and informational perspectives of steps.

The legend of IFD is shown in figure 9. IFD provides three entity types: *step*, *personnel*, and *deliverables store*. A step of an IDD is considered as a black box and its 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. After used, the artifacts neither transferred to other step nor stored at any deliverables store disappear.

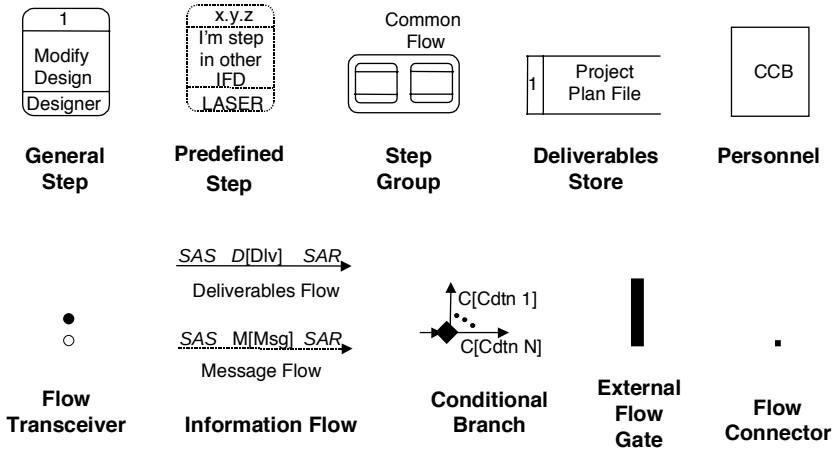


Figure 9 IFD Legend

IFD's most important strength is the representation scheme of the behavioral perspective. IFD allows representing both behavioral perspective and functional perspective together in a diagram and simulating or tracing the step behavior easily. The basic idea is that a step tends to produce some output and simultaneously its state is changed, which influences the executions of other steps. In HI-PLAN, there are four step states *NEWS*: N (Not wait), E (End), W (Wait), and S (Start). Possible state transitions are shown in figure 10. *Information flow* is the basic feature for the representation of both perspectives. An information flow connects two entities, where at least one of the entities must be a step and the information flow is connected to a flow transceiver attached to the step. An information flow can have three attachments: flow name, a *SAS* (*State After Sending*) indicator and a *SAR* (*State After Receiving*) indicator. The SAS and SAR indicators represent the state of a step after the step sends and receives the designated information. If SAS/SAR indicator is not specified, the state of the corresponding step is not changed after sending/receiving the designated information.

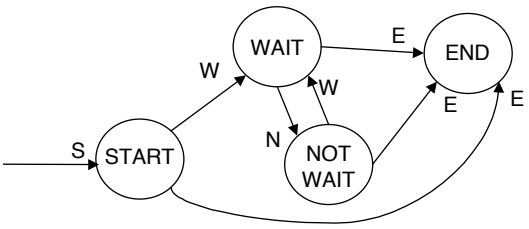


Figure 10 State Transitions in HI-PLAN

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

To represent step behavior more efficiently, two useful features are provided: *flow transceiver* and *conditional branch*. Flow transceivers are used to reduce the ambiguity and increase the traceability of step behavior when a number of information flows are connected between entities. An black flow transceiver of a step means that after the step sends or receives the designated information, the state of the step is changed. An white flow transceiver of a step means that after the step sends or receives the designated information, the state of the step is not changed. Flow transceivers can be utilized in several ways. One transceiver for an information flow represents a step state transition. 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) is (are) selected or iterated.

Both IDD and TSPEC are the forms to supplement the information represented in IFDs. The three types of IDD are shown in table 1. Table 1(a) is the form to describe artifacts and deliverables stores. Item "DEFINITION" is defined using six logical operators: + (Union), | (Selection), {} (Iteration), [] (Optional), @ (Key), and () (Grouping). Each element defined in "DEFINITION" item must be defined in another IDD, if it can be defined in detail. Table 1(b) and 1(C) is the form to describe messages and human resources, respectively. The format of TSPEC is shown in table 2. It includes information on a step for scheduling as well as execution.

Table 1 Format of IDD

(a) Artifacts and Deliverables Stores

NAME	Name of information
DEFINITION	Definition of components organization
AGENT	Agent responsible for the information
COMMUNICATION	Physical communication mechanism
STORAGE	Physical location or document name
COMMENT	Additional remark

(b) Messages

NAME	Name of message
DESCRIPTION	Meaning of message
AGENT	Agent responsible for the message
COMMUNICATION	Physical communication mechanism
COMMENT	Additional remark

(c) Human Resources

NAME	Name of individual or team
ORGANIZATION	Team members
RESPONSIBILITY	Assigned all steps
SPECIALTY	Major field
COMMENT	Additional remark

Table 2 Format of TSPEC

NAME	Name of step
OUTPUT UNIT	Unit of output deliverables type
OUTPUT SIZE	Output amount based on OUTPUT UNIT
COST	MM to perform step
START	Time to begin step
DURATION	Period between start and finish
AGENT	Agent responsible for step
PROCEDURE	Procedures to perform step
CONSTRAINTS	Precondition and postcondition
COMMENT	Additional remark

4.2 HI-PLAN Solution to the ISPW-6 Example

In this section I illustrate the ISPW-6 example process in HI-PLAN. The partial solution is composed of three IFDs, an IDD, and a TSPEC. It is less detailed than that of Little-JIL, but it shows sufficiently the characteristics of HI-PLAN. Same issues dealt in section 3.2 and solutions are discussed.

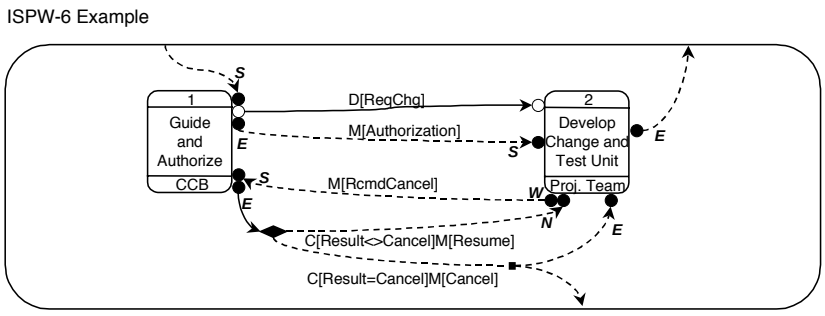


Figure 11 ISPW-6 in HI-PLAN: Highest Level IFD

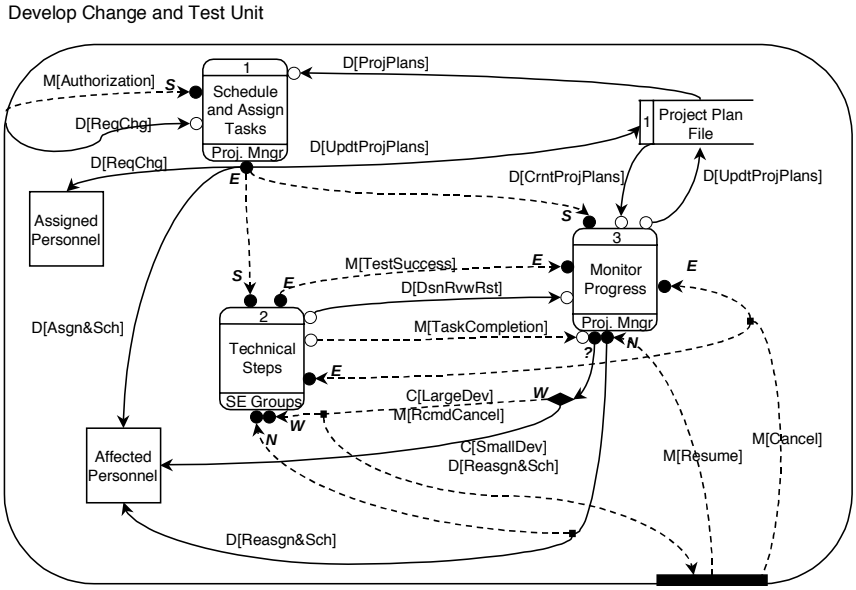


Figure 12 ISPW-6 in HI-PLAN: Upper Level IFD

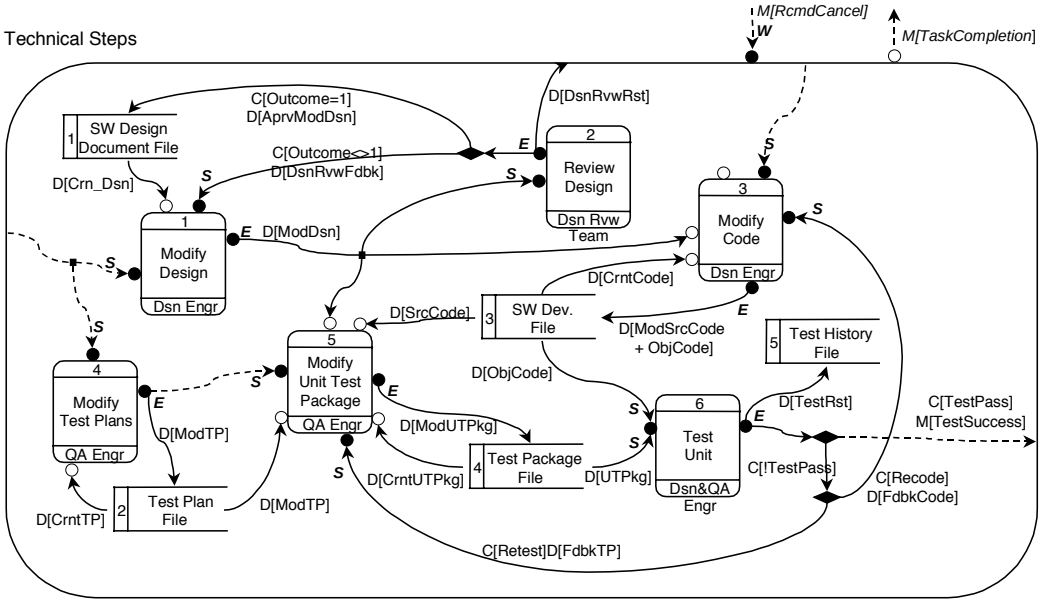


Figure 13 ISPW-6 in HI-PLAN: Technical Step IFD

Table 3 IDD for the artifact TestRst in figure 13

NAME	TestRst
DEFINITION	@(Unit_Test_Package_Version_ID + Object_Code_Version_ID) + Time_Stamp + Coverage_Level + [Indication_of_Test_Failed]
AGENT	Lee : Dsn Engr Kim : QA Engr
COMMUNICATION	Computer I/O
STORAGE	Test History File
COMMENT	This is a result of unit test

Table 4 TSPEC for the step Modify Code in figure 13

NAME	Modify Code	
OUTPUT UNIT	ModSrcCode	ObjCode
OUTPUT SIZE	1	1
COST	0.75 MM	
START	5/29/98	
DURATION	3 weeks	
AGENT	Lee : Dsn Engr	

PROCEDURE	Current source code is withdrawn
	If (compilation fails)
	Engineer makes additional modification
	Recompile
	If (clean compilation before design has been approved)
	Final manual check is made after design is approved
	Final version of code is placed into SW development files	

CONSTRAINTS	PRE : NONE	POST : Clean Compilation
COMMENT	This step involves the implementation of design changes into code and compilation of modified source code into object code.	

High-level abstraction and its decomposition into the details

In HI-PLAN, any entity can be decomposed. Also, artifact can be decomposed. IFDs in figure 11, 12, and 13 show the decompositions of the Develop Change and Test Unit step and its substep Technical Steps in turn. Table 3 shows the artifact Test Results is decomposed into several components. Of course, 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 11, the Develop Change and Test Unit step is started after the Guide and Authorize step ends. In figure 13, the Test Unit step is started after both Modify code and Modify Unit Test Package steps end. In figure 12, the Technical Steps and Monitor Progress steps are started simultaneously after the Schedule and Assign Tasks step ends. In figure 13, after the Review Design step ends, the Modify Design step, the Modify Code step, or both steps can be started according to the designated condition.

Input and source, output and 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 (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. For example, in figure 13, 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.

Constraints regarding the step sequencing

A step's constraints regarding the step are classified into following categories.

- The constraint on the precedence between steps can be represented by the information flows and the attached SAS/SAR indicators. For example, the constraint that step A can start after step B ends is represented by creating an information flow from step A to B and making the SAS and SAR indicators E and S respectively.
- The conditional branch can represent iteration-related constraints. Figure 13 shows the constraint that subsequent iterations of the Modify Design step can begin after the Review Design step ends, if the condition Outcome is not equal to 1, i.e., when the design is not approved.
- The constraint on the conditions to start or complete steps is represented in "CONSTRAINT" item of TSPEC; in table 3, the TSPEC for the Modify Code step shows that the ModifyCode step ends when a clean compilation has been accomplished.
- Message flow can represent the constraint that a step is started or terminated by a message from other step. For example, in figure 12, the Monitor Progress step ends after receiving TestSuccess message from the Technical Steps step.
- Resource-related constraints are represented by one flow transceiver with input information flow(s) from deliverables store(s). For example, the Test Unit step in figure 13 can begin as soon as both ObjCode and UTPkg are available from the deliverables store SW Dev. File and Test Package File, respectively.
- The constraints that a step should end when all its output have been provided are represented by multiple output information flows starting from one flow transceiver attached to the step, where an E SAS indicator is used for all information flows, and each information flow corresponds to each output. For example, in figure 12, the Schedule and Assign Tasks step ends after generating ReqChg, UpdtProjPlans, and Asgn&Sch. If the outputs are transferred to one entity, those can be combined with plus (+) operator and one information flow can be used; in figure 13, the Modify Code step ends after generating ModSrcCode and ObjCode.

5. Evaluation of Little-JIL

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

5.1 Fundamental Comparison

Both Little-JIL and HI-PLAN are graphical process languages, but have different design principles, objectives, and features (Table 5).
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 [11]. 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 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.

Table 5 Little-JIL vs. HI-PLAN: Fundamentals

	Little-JIL	HI-PLAN
Main Objective	Agent coordination	Process specification
Usage	Process execution	Process understanding and communication
Visualization Framework	Hierarchical tree structure	Hierarchical network structure
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

5.2 Step Decomposition

Focus on step decomposition

In Little-JIL, a step can be decomposed into one or more substeps whose execution sequence must be determined by a step kind. If the substeps can not be represented by one step kind, further decompositions have to occur. That is, 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 some problems. First, Little-JIL programs are likely to have long depth (hierarchy) 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 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 issue.).

In HI-PLAN, step decomposition is based on only process abstraction. Control abstraction is handled in each IFD through information flows. Therefore, step decomposition is simpler and IFDs have shorter depth than Little-JIL program has.

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 14 shows how Little-JIL solves example 1. From 1) and 2), step S seems to be a sequential step (figure 14(a)). From 3) and 4), step B must be decomposed into sequential substeps (figure 14(b)). Now, there are two possibilities to handle 5). One is to maintain process abstraction as in figure 14(c), where the starting step B1 will have to be delayed until step A is 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 14(d), where the program structure must be changed and the substeps of step B are scattered. If decomposition of these substeps is continued, restructuring may be repeated.

Figure 15 shows a HI-PLAN solution. Figure 15(a) is the top-level IFD from 1) and 2). Figure 15(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 15(c)).

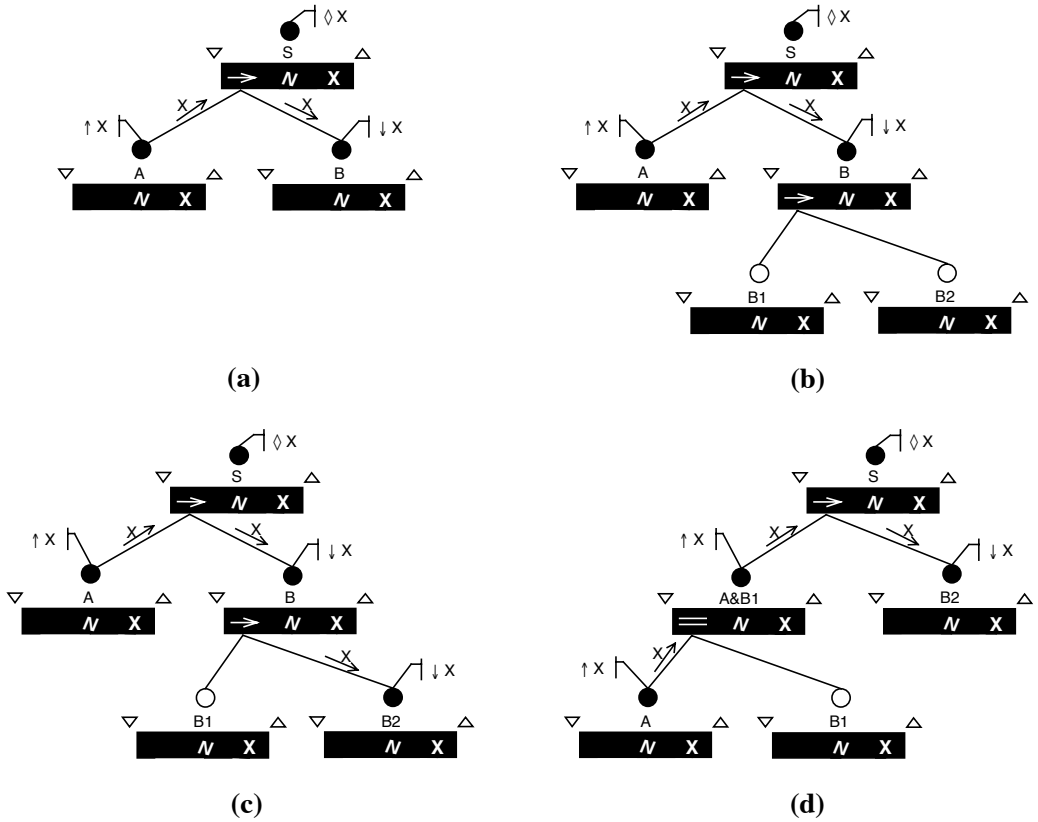


Figure 14 A Little-JIL Solution to Example 1

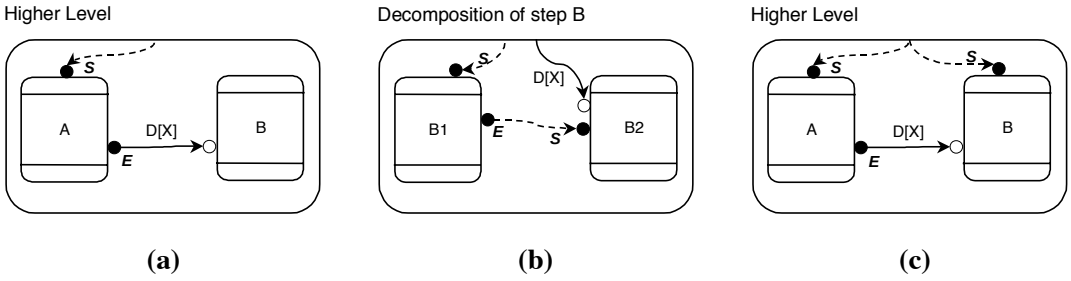


Figure 15 A HI-PLAN Solution to Example 1

5.3 Language Features and Semantics

This section focuses on the discussion of the issues to be supplemented or added to Little-JIL. For each issue, I describe why the issue is important, what is my suggestion to improve Little-JIL, if possible, and how HI-PLAN handles the issue.

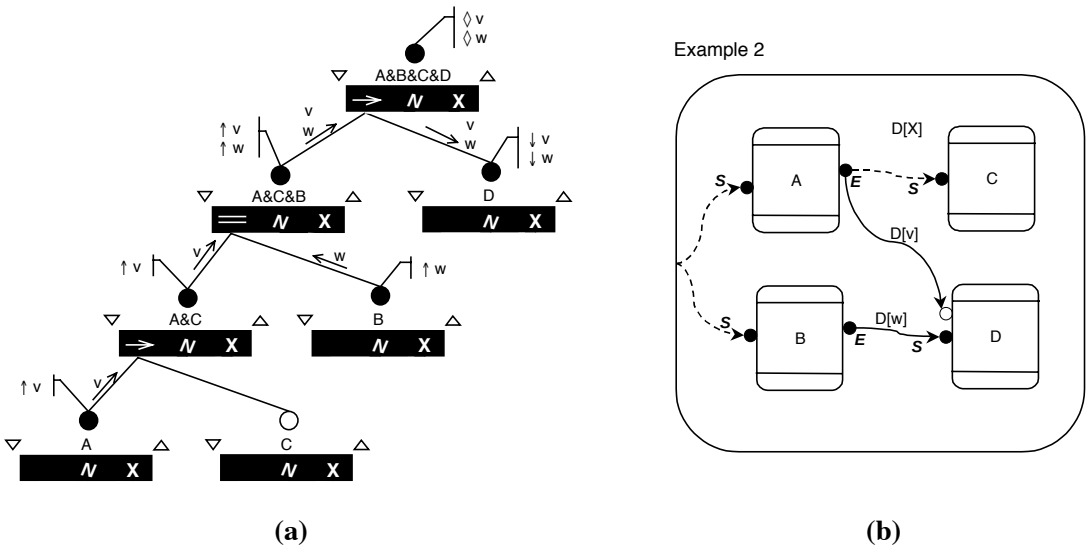
Parallelism

Needless to say, depicting parallelism is the essential feature of process language. Parallel step in Little-JIL is the explicit way of representing parallelism. However, some steps executed in parallel can cause inefficient situation. Let's consider following example.

Example 2. Parallelism

Step A and step B start simultaneously
Step C starts after completing step A
Step D starts after completing step B
Step A outputs v, step B outputs w, and step D uses v and w

Figure 16(a) shows a possible solution of Little-JIL. At a glance, it seems to have no problem. If step C is a small work, this program is still efficient. However, if step C requires longer execution time than that of step B, the starting time of step D has to be delayed, because the parallel step is completed when completing step A, B, and especially C that has no relationship with step D. Figure 16(b) shows a HI-PLAN solution, where step D can start as soon as step B is terminated.



(a) **(b)**

Figure 16 Solutions to Example 2

Other similar problem of parallelism is also found in ISPW-6 example programming. In ISPW-6 example, ModifyDesign, ModifyCode, and ModifyTest steps are executed in parallel and ReviewDesign step starts after completion of ModifyDesign step. ModifyDesign step outputs modified design (ModDsn), which is transferred to ReviewDesign, ModifyCode, and ModifyUnitTestPackage (substep of ModifyTest) steps. The overall control structure to handle this situation is simple (figure 4), however, the problem is how to transfer ModDsn as soon as ModifyDesign step produces it. Explicit parameter passing of Little-JIL is inadequate to represent that ModDsn is passed to ModifyCode and ModifyUnitTestPackage without delay.

The reason is that Modify&ReviewDesign step is completed when ReviewDesign step has been completed. Thus, SendModDsnToMC&MUTP step is used to transfer ModDsn immediately, which is implicit and unnatural. As seen in figure 13, HI-PLAN has no such a problem, though parallelism is not explicitly described.
Above situation affects the efficiency of step execution in Little-JIL seriously. I suggest a solution based on *self-reliant step* and *zombie step*. Self-reliant steps are the steps that may cause delaying the execution of other steps while having no relationship with those steps: for example, step C in figure 16 and step ReviewDesign in figure 5. The parent step of a self-reliant step assumes the self-reliant step is completed successfully at the same time when it is started. Started self-reliant steps go on their execution and may have substeps. To handle exceptions or messages from the self-reliant step and its substeps, the completed ancestor steps of the self-reliant step remains as a zombie step until the self-reliant step is actually completed. That is, when any ancestor step of an executing uncompleted self-reliant step is completed, the substeps of the ancestor step must not be cleaned up. Self-reliant steps can be distinguished from general steps by reverse coloring.

Artifact as a resource

Little-JIL has a notation to identify a resource that is acquired by a step. Resources are acquired (locked for exclusive use) when the agent begins the step and can be passed from one step to another or shared between steps. Resources are released (unlocked) when all steps using the resource are completed or terminated. Such features and semantics for resource are concise but insufficient to explicitly represent typical actions on resources, especially modifiable artifacts such as documents or codes, even though taking it into account that resource management is factored out of Little-JIL. In programming ISPW-6, for example, in order to represent that resource ProjPlans must be updated after scheduling, Little-JIL has no other way except using a separate step (UpdateProjPlan step in figure 2). Most artifact resources undergo similar actions. Also, resource creation can not be manipulated in Little-JIL. The simple method to complement existing resource representation is to add a description on the type of a resource's usage to resource declaration. This description is the typical action on the resource during execution of the step using the resource and has one of three types.

1) **UNCHG**: The contents of acquired resource are not changed.
2) **UPDT**: The contents of acquired resource are updated and stored.
3) **NEW**: New resource is created.

If the type of a resource is "**NEW**", resource acquisition of the resource is omitted. The addition of above types decreases the number of steps considerably and enhances understandability. The addition of above types can be textual or graphical.
HI-PLAN has explicit representation of acquiring and updating artifact resources. The information flows from a deliverables store to a step and from a step to a deliverables store represent acquiring and updating a resource, respectively.

Parameter passing

In Little-JIL, inputs and outputs of steps are treated as parameters and results. When a step is started, the values for *in* and *in/out* parameters are copied into the step. When the step is completed, the values for *out* and *in/out* parameters are copied out. This parameter passing is valuable and rigorous feature, but a problem exists if the results of a step are needed not by the parent but by some other steps. If a producer step is distant from a consumer step, every step on the path from producer to consumer step should have unnecessary parameter declaration for unusable parameter. I believe that a step must have minimal set of parameters indispensable to be

used for execution. To lessen such burden, additional functionality supplementing existing parameter passing is needed. One solution is providing additional notations called *direct-passing parameter* (for example **<=** and **=>**) to simplify long-distance parameter passing, which is similar to the global variable declaration of general-purpose programming languages. Direct-passing parameter is declared only at the step interface specifications of both producer step and consumer steps: for example, "**<=** parameter name" in producer step and "**=>** parameter name" in consumer steps. The value is copied into a direct-passing parameter when its producer step is completed, and it is not discarded.

In HI-PLAN, non-leaf step can receive information at any time before ended and send information at any time from started to ended. Information transfer and control of step execution are explicitly modeled with information flow. This clearly describes where the information is going, when the information may be transferred, and how step responds to the information arrival. Also, as mentioned earlier, this is useful for top-down decomposition.

Step reuse

Each step in a Little-JIL program is defined exactly once, however, it may be used multiple times. Such step reuse is represented by a reference with italicized text and without badges. The difficulty in reuse is that each step may have a prerequisite and a postrequisite; a reference to a step must reuse the requisites of the step. Reuse of a step with requisites may require different requisites or no requisites. Moreover, requisites are often used to represent conditionals such as "if-then-else" because choice step, another feature for conditionals, is inappropriate to express explicit conditional execution. Therefore, current feature of step reuse must reuse not only step but also control enclosing the step. Therefore, it is reasonable to exclude requisites from referencing and allow a reference to have its own requisites.

Iteration

In Little-JIL, recursion using references can be used to model iteration, which is often confusing even for experienced programmers and is not the natural way of representing iteration. Moreover, whenever a step is iterated it has to be terminated and restarted, which requires resource acquisition and parameter binding every time. Thus, an explicit representation for iteration must be provided. My suggestion is to add a step kind that continues to loop until it cannot satisfy entry or exit condition. Prerequisite (while-do) and postrequisite (repeat-until) of an iteration step represent the entry and exit condition. The additional step kind for iteration is shown in Figure 17, where step S is iterated while Condition is true. During iteration, step S is restarted but does not have to acquire resources and bind parameters with parent's.

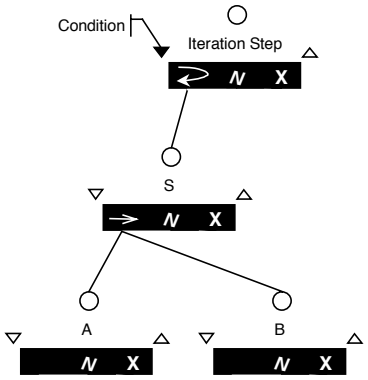


Figure 17 Iteration Step

State transition

A Little-JIL step is in one of five states: posted, started, completed, terminated, or retracted. The state of a step is transited implicitly. As in [10], Little-JIL has very clear and rigorous semantics on the step states and state transitions, but two deficiencies need to be complemented to describe real situations.

First, existing step states must be subdivided. If a Little-JIL step is started, the step should be completed or terminated. In real situations, however, a started step can be suspended and resumed. The suspension and resumption of a step is very important to prevent the agent of the step from vain effort and represent non-hierarchical communication. My suggestion is to add two step states: *suspended* and *resumed*. Revised state transitions are shown in figure 18. When a step is suspended, all of its started or resumed substeps are suspended, and when a step is resumed, all of its suspended substeps are resumed.

Second, feature for state transition by message must be provided. If once a Little-JIL step is started, the state of step can not be changed until the step is completed or an exception occurs in the step. Message to a step cannot affect the execution of the step. In a number of places of ISPW-6 example, messages cause state transition. For example, MonitorProgress step has to be completed when receiving TestSuccess message, which can be only represented by a message reaction step to ask for the completion of MonitorProgress step (Figure 3). This problem also disturbs non-hierarchical communication. I will propose a possible solution for non-hierarchical communication in the next issue.

A HI-PLAN step is one of four states: Not wait, End, Wait, or Start. The state transition of a step is represented explicitly. The state transition by message can be represented with an information flow, which enables easy and simple representation of state transition caused by non-hierarchical communication between steps. The suspension and resumption of a step can be represented by a group of overlapping transceivers; for example, two overlapping transceivers attached to the TechnicalSteps step in figure 12 represent that the step is suspended by RcmdCancel message and resumed by a message from Monitor Progress step.

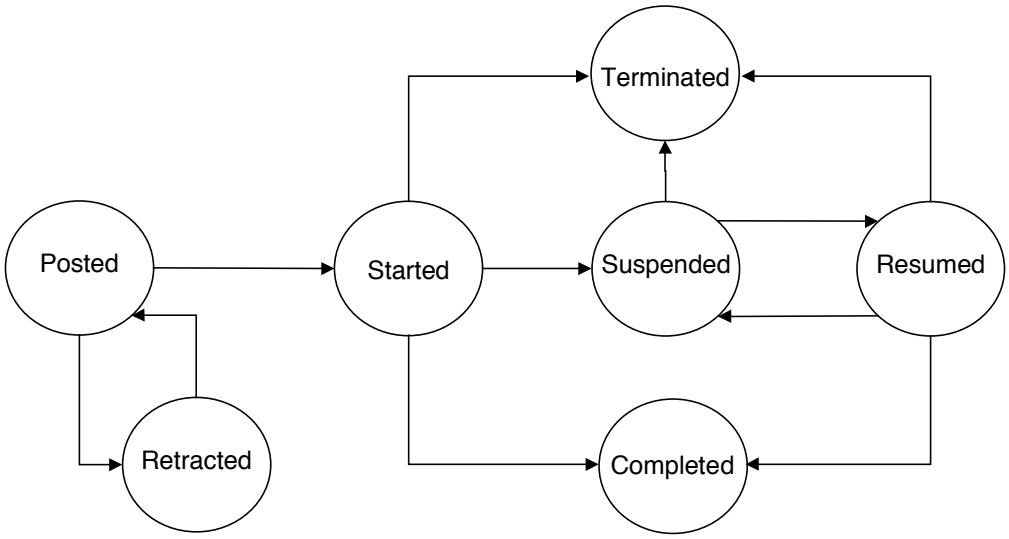


Figure 18 Revised State Transitions in Little-JIL

Non-hierarchical communication: extension of message

In Little-JIL, a step is posted according to the control flow badge of its parent step and then executed using its resources and parameters until it is completed or terminated. That is, when a step is started is determined by its parent step, and once a step is started, any other steps do not affect the execution of the step. Only internal troubles of the step such as exception can affect its execution. Even messages and reactions are not appropriate for non-hierarchical communication, because steps can only broadcast messages and message reaction steps of a step are executed independently of the step. This is one of the weakest points in Little-JIL. An explicit representation of how agents can share information and work cooperatively must be provided.

My suggestion is based on two premises. The first is that the state of a step can be transited when the step sends and receives a message with values or parameters. This requires syntax and semantics modification of message specifications of both message-sending step and message-receiving step. The new syntax of message specification for a message-sending step contains three elements: message name, time, and optionally parameter bindings.

N MessageName : *Time*
Parameter bindings

Time specifies when a message is sent by a step and has one of five values as followings:

- **STARTED** if the message is sent when the step is started
- **SUSPENDED** if the message is sent when the step is suspended
- **RESUMED** if the message is sent when the step is resumed
- **COMPLETED** if the message is sent when the step is completed
- **UNKNOWN** if the message may be sent at any or unknown time.

When a message is sent, objects can be sent together. The objects are passed via *parameter bindings* shown as indented under the message name. A binding copies the value of a parameter in a step sending a message into a parameter represented by an envelope (✉) in the step interface of the reaction steps matched with the message. Each binding is represented as "*parameter in receiving step* ← *parameter in sending step*".

The message specification of a message-receiving step is same as before except addition of *state-transition badges* similar to the continuation badges in exception handling. A step reacts to messages while the step is started, suspended, or resumed. Four kinds of state-transition badge are needed and the semantics for handling them are as followings.

- *"suspend" badge* (⌛): The step should be suspended if the step is in the started or resumed state, which means all of its started or resumed substeps should be suspended.
- *"resume" badge* (➠): The step should be resumed if the step is in the suspended state, which means all of its suspended substeps should be resumed.
- *"terminate" badge* (✞): The step should be terminated if the step is in the started, suspended, or resumed state, which means all of its started, suspended, or resumed substeps should be terminated.
- *"no transition" badge* (⊗): The state of the step is not affected.

In ISPW-6, for example, after a technical step such as ModifyDesign is completed, MonitorProgress step reschedules. In case of severe deviations from plan, MonitorProgress step puts the entire modification effort on hold and recommends to the CCB that this change effort be cancel. The CCB may either cancel the effort or indicate that it is to be resumed where it

left off. With Little-JIL, it is impossible to represent this situation without the steps that ask for suspension, resumption, or termination of related steps (Figure 3). Figure 19 and 20 show a simplified program in figure 3 and a revised program using state-transition badges, respectively.

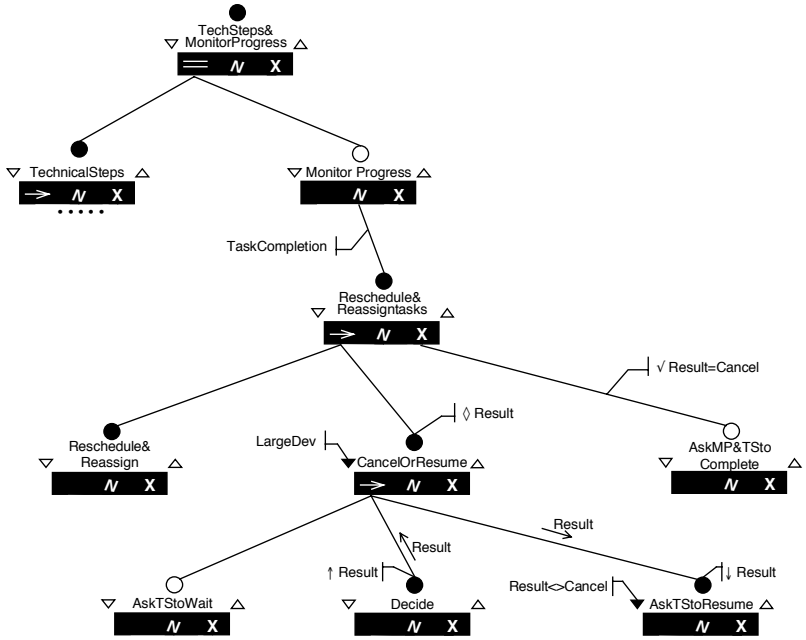


Figure 19 Simplification of figure 3

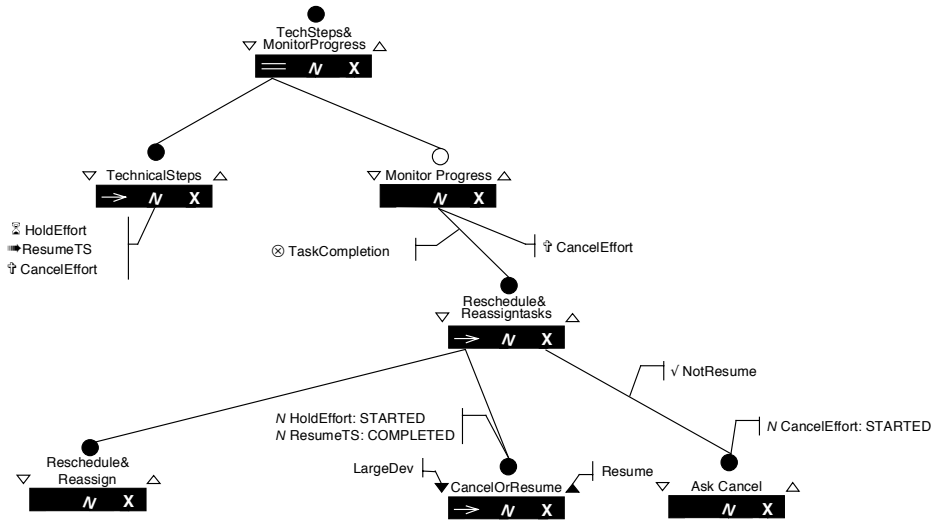


Figure 20 Revision of Figure 19

The second premise of the suggestion is that a step is triggered by either the control flow badge of its parent or a message from other step. Figure 21 is my suggestion of typical step icon for

trigger-by-message steps, which can be easily distinguished from trigger-by-parent steps and emphasizes that the steps are initiated by a message arrival. The message to trigger a step is attached to the reaction badge of the step like other general messages and represented by a "trigger" badge (➵). Whenever a step is posted, it is checked whether the step is a trigger-by-message step. If so, posting the step is delayed until the message with "trigger" badge is arrived to the step. If a "trigger" badge message arrives to a step again while the step is started,

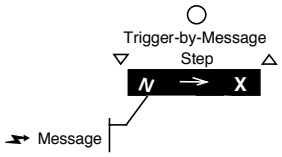
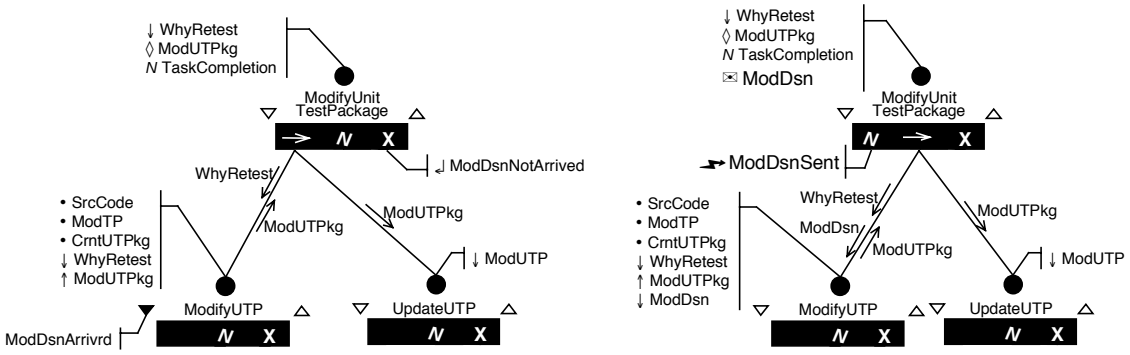


Figure 19 Trigger-by-Message step

suspended, or resumed, it is queued until the step is completed or terminated. In ISPW-6 example, ModifyUnitTestPackage step can start if ModDsn is arrived from ModifyDesign step, but these two steps are executed in parallel and ModDsn cannot be passed with parameter. This situation can be programmed with a pair of prerequisite and exception in Little-JIL (figure 8), however the ModifyUTP step has to be continuously restarted until ModDsn is arrived. Figure 22 shows a revised program using "trigger" badge, assuming that the ModifyDesign step sends ModDsn by message "*N* ModDsnSent(ModDsn):COMPLETED".
Above message specifications of message-sending step and message-receiving step can be a solution to non-hierarchical communication. In fact, these features can not force the state transition of a step but request the agent of the step to do so. Thus, each agent can be guided while maintaining autonomy.



(a) Figure 8 (Revisited) **(b) Revised Solution**

Figure 20 Example Using "trigger" Badge

Notification

Little-JIL is an agent coordination language. Every step is assigned to an agent and all agents are related with steps. All artifacts or messages are transmitted to steps. However, information with no relationship with step, such as project management data, may be transmitted to an individual, individuals, or organization. Such information is not used in any step but notified or carried to

personnel. For example, ISPW-6 specifies that Schedule&Assigntasks step must transfer task assignments and scheduled dates to all affected personnel and notify requirements change to all assigned personnel. Little-JIL does not support semantics for this situation because agent is an attribute of step.
HI-PLAN provides "personnel" entity that can be organizational group or individual. An information flow to a "personnel" means that it receives designated artifact or message (figure 12).

Choice step
A process language would be more powerful if it provides features for representing "if-then-else" conditional execution. It is desirable that "if-then-else" feature satisfies following criteria. First, it should be a distinguishable and concrete structure to maximize understandability and ease of use. Second, the condition should be evaluated earlier than step execution. Last, the condition may be omitted to maximize the autonomy of the execution agent when either any alternative would be fine or it is unable to offer any guidance to the execution agent. Little-JIL provides two abstractions that represent conditionals: requisites and choice step. Requisites can be used to express (more correctly, simulate) "if-then-else", but do not satisfy any criteria. Choice step will be better suited to describing "if-then-else", if it is reinforced properly. In Little-JIL, the only way to guide an execution agent to determine which alternative in a choice step to be selected is naming the substeps of the choice step in ways explaining when they should be selected. The focus of reinforcement is an explicit representation of when each substep of a choice step should be selected. The simplest method is to put labels on the edges from a choice step in a manner analogous to the way reactions and handlers are labeled. This causes two problems. First, all the labels may be evaluated in the worst case. The execution agent of a choice step should evaluate the labels one by one until finding true one. In general, all labels are different results of same condition evaluation that may be a simple expression evaluation or a complex step execution. Thus, same evaluations are repeated. Second, it is not expressed well that condition should be evaluated earlier than substep execution. My suggestion is to declare condition at the step interface specification of choice step and results, for example "true" and "false", are labeled on the edges from choice step. This can represent explicitly that condition is evaluated in choice step before posting any substep.
In HI-PLAN, conditionals are represented with conditional branch, which is different from choice step in that it is possible to select more than one step.

5.4 Qualitative Issues

Understandability
Basically, both 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. H-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 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* [11] 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 is possible for non-leaf steps to receive information before starting their executions. This flexibility enables top-down decomposition without having upper level IFDs affected. Also, any information transfer can cause state transition.

6. Concluding Remarks

Little-JIL is an executable, high-level process language for programming coordination in processes. Little-JIL has a formal yet graphical syntax and rigorously defined operational semantics. The development of Little-JIL has proceeded as a series of iterative cycles of design and evaluation. The applicability and efficiency of Little-JIL have been explored and demonstrated through the definition of processes from a variety of domains, implementation of an interpreter and supporting components, and use and analysis of the resulting processes.

This report is an evaluation result of Little-JIL 1.0, the current version of Little-JIL, but has different viewpoints from the earlier reports. After programming ISPW-6 example, a standard benchmark software process, in both Little-JIL and another process language HI-PLAN, I compared both languages on the primary issues of process language and attempted to improve Little-JIL through the inclusion of some features of HI-PLAN.

I believe that Little-JIL will rapidly gain very wide acceptance in the process community and be continuously refined to be better language. This report is a proposal of the changes and additions to Little-JIL that will be made for the next version of this language. I expect Little-JIL to benefit much from this experiment.

7. Acknowledgements

I wish to thank all the members of the LASER Process Working Group for their helpful comments and suggestions. 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. 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.
2. 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.
3. 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.
4. 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.
5. 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.
6. 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.
7. 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.
8. 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.
9. 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.
10. A. Wise. Little-JIL 1.0 Language Report. Technical Report 98-24, Department of Computer Science, University of Massachusetts at Amherst, April 1998.
11. 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 98-38, Department of Computer Science, University of Massachusetts at Amherst, April 1998.