The Design of a Next-Generation Process Language

Stanley M. Sutton, Jr.
Leon J. Osterweil

CMPSCI Technical Report 96-30
Revised January 22, 1997

Laboratory for Advanced Software Engineering Research
Computer Science Department
University of Massachusetts
Amherst, Massachusetts 01003

This work was supported in part by the Air Force Materiel Command, Rome Laboratory, and the Advanced Research Projects Agency under Contract F30602-94-C-0137.

The Design of a Next-Generation Process Language[1]

Stanley M. Sutton, Jr. and Leon J. Osterweil
Department of Computer Science
University of Massachusetts
Amherst, MA 01003-4610

CMPSCI Technical Report 96-30
Revised January 22, 1997

Abstract

Process languages remain a vital area of software process research. The first generation of process languages, mostly based on existing language paradigms, yielded mixed results. The second generation of process languages can build on lessons learned from these earlier languages and seek to address the issues raised by their definition and use.

Among the issues that we see as especially important for process languages are semantic richness, ease of use, appropriate abstractions, process composability, process and program visualization, and support for multiple paradigms. We see the balancing of the needs for both semantic richness and ease of use as one of the most critical aspects of process language design.

JIL addresses these issues in a number of innovative ways. It models processes in terms of steps with a rich variety of semantic attributes. JIL also offers a unique variety of control models. These combine proactive and reactive control, conditional control, and more simple means of control-flow modeling via step composition and simple step-execution constraints.

The combination of features in JIL supports a flexible and adaptable approach to process programming. Despite its semantic richness, the language facilitates ease of use. It does this through semantic factoring, the accommodating of incomplete step specifications, the fostering of simple sub-languages, and the ability to support visualizations. We believe that this approach will allow processes to be programmed in a variety of terms, and to a variety of levels of detail, suitable to the needs of particular processes, projects, and programmers.

1 Introduction

Process language research was an early emphasis of software process studies. It has remained vital for several reasons. First, no language has gained general acceptance or widespread use. This is not just a linguistic problem, as the use of languages depends also on organizational, methodological, and technological support. Second, languages developed during the first generation generally have obvious limitations. This is in part due to the fact that many of these languages were based on existing paradigms that were not particularly well adapted to the domain of software process [10, 20, 29, 22, 13, 4, 21]. Finally, research in other areas of software process has affected our ideas about what can and should be done with process languages. In this paper we report on the design of a "next-generation" process language that is intended to capitalize on lessons learned from first-generation languages, overcome limitations of those languages, and explore issues that are emerging from ongoing process research.

[1]This work was supported in part by the Air Force Materiel Command, Rome Laboratory, and the Advanced Research Projects Agency under Contract F30602-94-C-0137. Submitted to the Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering. The authors can be reached by email at {sutton, ljo}@cs.umass.edu.

Section 2 identifies our primary language design goals, which are based on our experience with first-generation process languages. Section 3 describes the design of JIL, our next-generation process language, including examples based on the Booch object-oriented design process. Section 4 discusses the multi-modal interpretation of JIL programs. An assessment of the JIL approach is presented in Section 5. Finally, Sections 6 and 7 report on our status and conclusions. Three appendices present supplemental information. Appendix A provides an overview of the Julia process environment, which is designed to support the execution of JIL programs; Appendix B discusses the visual representation of JIL programs; and Appendix C describes aspects of the JIL interpreter.

2 Language Design Goals

The original idea of process programming proposes that it is feasible and valuable to represent software processes using programs written in compilable, executable process coding languages [24, 25]. Our experience with APPL/A [29] has validated this proposal. There are many properties of coding languages that we now take as fundamental to representing software processes adequately. These include formal syntax and well-defined semantics, executability, analyzability, object management, and consistency management. These issues have been the focus of much previous work by us and others (for example, [27, 19, 10, 20, 6, 12, 29]), and they should continue to be addressed by second-generation process languages. Our focus, however, is on further issues as outlined below.

2.1 Semantic Richness

Software processes are complex and multi-faceted activities that represent a technically challenging application domain. To support this domain, a process programming language must provide a variety of kinds of interrelated semantics. This pressure for semantic richness is reflected in first-generation process languages. Many of these are based on elaborations or extensions of conventional programming languages or paradigms, including functional languages ([10, 22]), rule-based languages ([20, 19]), imperative languages ([29]), Petri nets ([4, 13]), and configuration management ([12, 6]). Conversely, where process languages have neglected certain areas of semantics and functionality, process programs have suffered. These areas include object management, reflexivity, and resource modeling, among others. Process language semantics must be both rich and rigorously based. They must cover an adequate range of process-related semantics, and they must do so with models that are appropriate for software processes and that support reasoning about them.

2.2 Ease of Use

The need for semantic richness notwithstanding, ease of use is also an important requirement for process programming languages. As a consequence of their semantic richness, significant software engineering skills are required to program effectively in these languages. However, the individuals and organizations responsible for defining software processes often are not programmers and do not act in a software development role. In these cases, the need for serious programming is an impediment to the use of process programming languages. To facilitate the adoption of process programming as a paradigm for process definition and implementation, process programming languages must be easy to use. A key issue in the design of process programming languages is thus balancing the need for technical rigor with this need for ease of use.

2.3 Appropriate Abstractions

The clear and concise representation of software processes requires appropriate kinds and levels of abstraction. The need for the user to construct process specific abstractions from lower-level abstractions, as is the case with APPL/A and other process languages that are based on general-purpose programming languages, complicates the development and maintenance of process programs. Process programming languages should provide built-in concepts and constructs that map naturally into the software process domain. (Some languages that do this with varying degrees of success include MVP-L [26], ProcessWeaver [15], LOTOS [27], and Oikos [23].)

2.4 Composability

Programming of software processes in general is difficult. Thus it is important to be able to readily compose larger processes out of smaller components and to support reuse-based process programming. This would greatly increase the value of existing process programs. In addition, it would be useful to allow processes to be programmed by the composition of elements having different language paradigms or representing different semantic aspects. This would introduce additional flexibility and incrementality into process program development.

2.5 Clarity through Visualization

Most traditional programming languages have been textual, as have many first-generation process languages. A few process languages support graphical representations of process control ([3, 13, 15, 18]). One great advantage of visual process representations is their aid to process understanding. Simple ideas are often most simply represented visually, and this can aid greatly in process design and verification and in process understanding and communication. On the other hand, more complex and dynamic process structures may be easier to express in textual languages, since visual representations can become cluttered and unwieldy and often tolerate ambiguity in order to cultivate simplicity of expression. Thus, strictly visual representations are likely to be unsuitable for complex processes in general. In light of these tradeoffs, our goals for a second generation process language give top priority to the expressive power afforded by textual languages, while supporting the use of visual representations, wherever they are workable, as a secondary (but still important) goal.

2.6 Multiple Paradigms

In our opinion, one of the most useful features of APPL/A is its incorporation of reactive triggers into a largely imperative language (Ada). This combination of proactive and reactive control was drawn on heavily in our process programming. Some combination of these two types of control is also found in many other process languages (Adele [6],AP5 [10], EPOS [11], HFSP [22, 30], Marvel [20], Merlin [19], and ProcessWeaver [15], to name a few). ALF [9] is another process project that is explicitly multiparadigmatic, and Pleiades demonstrates that multiple paradigms are also important in software object management [31]. In most of these process languages, however, one control paradigm typically predominates while the other is secondary. Thus, many languages primarily support one style of programming (such as rule-based programming in Merlin or Marvel, or functional programming in HFSP, among others). Our experience and observations suggest that

favoring one paradigm becomes problematic when a paradigm other than the favored one becomes most natural for the process to be programmed. Thus, one of our goals is to support multiple paradigms without favoring any of them. We seek to give these different paradigms coequal status in our language and thereby afford programmers flexibility in their choice of programming style.

3 Design of JIL

In this section we discuss the design of JIL, emphasizing features related to various aspects of the control model. These include process steps, the control paradigm, and exception handling. Several examples are shown, based on a process for programming the design of software following the principles of Booch Object-Oriented Design [8].

3.1 Process Steps

The central construct in JIL is the *step*. A JIL step is intended to represent a step in a software process. A JIL program is a composition of steps, each of which may contain a set of subunits (representing different aspects of the step) and supplementing units (such as separate procedures and packages). The elements of a step specification represent kinds of semantics that are important to process definition, analysis, understanding, and execution. Briefly, the elements of a step are:

- *Objects and declarations*: The parameters and local declarations for software artifacts used in the step
- *Resource requirements*: Specifications of resources needed by the step, including people, software, and hardware
- *Substep set*: The substeps of a step (which are themselves steps)
- *Step constraints*: These specify restrictions on the relative execution order of substeps
- *Proactive control specification*: An imperative specification of the order in which substeps are to be executed (representing direct invocation)
- *Reactive control specification*: A reactive specification of the conditions or events in response to which substeps are to be executed (representing indirect invocation)
- *Preconditions, constraints, postconditions*: Define artifact consistency conditions that must be satisfied (respectively) prior to, during, and subsequent to the execution of the step
- *Exception handlers*: Handlers for local exceptions, including handlers for consistency violations (e.g., precondition violations)

In defining a step not all of these elements are required but any of them may be used.

An example of a JIL step specification is shown in Figure 1. This specification represents the first step in a (simplified) Booch Object-Oriented Design process [8]. The step specification has a template-like syntax. The substeps are listed within the specification. The proactive control

```
STEP Identify_Classes_And_Object IS
    OBJECTS:  Reqts_Spec:
                  Requirements_Specification.Specification_Type;

    DECLARATIONS:  Class_Candidate_List, Object_Candidate_List:
                       Booch_Product_Definition.Name_List;

    -- Substeps
    STEPS:  Browse_Requirements,
            Extract_Class_Candidates,
            Identify_Classes,
            Extract_Object_Candidates,
            Identify_Objects,
            Edit_Class_Object_Dictionary;

    -- Proactive control specification
    -- [Separate subunit--see Figure 2]
    ACTIVITY:  Identify_Classes_And_Objects_Activity;

    -- Reactive control specification
    -- [Separate subunit--see Figure 3]
    REACTIONS:  Identify_Classes_And_Objects_Reactions;

    PRECONDITIONS:
        -- [Separate package]
        FROM Requirements_Consistency_Conditions USE
            Passed_Review(Reqts_Spec);

    POSTCONDITIONS:
        -- [Separate package]
        FROM Booch_Product USE
            Non_Empty(Booch_Product.Class_Diagram);
            Non_Empty(Booch_Product.Object_Diagram);
            Unique_Name_Per_Class(Booch_Product.Class_Diagram);
            Every_Object_Has_A_Class(Booch_Product.Object_Diagram,
                                     Booch_Product.Class_Diagram);

    -- Exception handlers
    -- [Separate subunit--see Figure 5]
    HANDLERS:  Handle_Class_And_Object_Errors;
END Identify_Classes_And_Objects;
```

Figure 1: Example of a JIL step specification.

specification, reactive control specification, and exception handlers are all contained in separate, named subunits (discussed in Sections 3.2.1, 3.2.2, and 3.3, respectively). The preconditions and postconditions are defined in separate units (that are not subunits of the step; see Section 3.2.3). Some parts of the step specification are unspecified. An example of a possible step constraint is shown later (Section 3.2.4). Some general comments on the language follow before specific features are discussed in more detail.

As the specification of a step suggests, it is our intention that JIL be a *factored* language. That is, it should provide independent representations for independent semantics, insofar as possible. Additionally, most of the elements of a step specification are optional. For example, a step need not have preconditions, postconditions, or constraints, or it may have only proactive or reactive control (or neither).

The orthogonality of semantic elements and the ability to select the elements used to define a step have several consequences. The various aspects of a step can be specified relatively independently of one another. For example, the substeps for a step can be given without indicating any proactive or reactive control flow, and control flow can be specified without regard to resources or preconditions and postconditions. (The elements of a step are not entirely independent, however. For example, consistency conditions will typically reference objects used by the step, and control specifications must refer to the substeps of the step.) The independence of elements in the step specification allows flexibility in process definition, and a step may be described in just those terms that are relevant to a particular purpose. However, the ability to selectively represent elements of a step imposes additional requirements on process interpretation, since various kinds and combinations of control specification may be present. Issues of interpretation are discussed in Section 4.

3.2 Control Paradigm

JIL affords a unique variety of control paradigms that enable alternative approaches to specifying process control flow. The JIL control paradigm is characterized by three primary features:

- The combination of proactive and reactive control, the value of which was demonstrated by first-generation process languages.
- The integration of preconditions and postconditions, which have also been widely used.
- The ability to specify loosely organized processes without requiring detailed programming.

Particularly important, though, is the flexibility that JIL affords in the use of these control paradigms: any or all may be used, within a single program, and even within a single step. These different aspects, and the resulting interpretation paradigm, are discussed below.

3.2.1 Proactive Control

The proactive control specification of a JIL step provides a context in which the execution of substeps can be imperatively or procedurally programmed. The step specification designates a

separate subunit to represent the proactive control specification for the step. This has two parts, a specification and body. The specification lists the entry calls and signals that can be received by the executing instance of the proactive part of the step. (These support interprocess communication.) The body provides the imperative code that controls the execution of substeps. The syntax of the imperative code is based on Ada, with some extensions such as a `parallel` command for representing certain sorts of concurrent activities. An example of a proactive control specification for the Booch process step of Figure 1 is shown in Figure 2.

```
ACTIVITY BODY Identify_Classes_And_Objects_Activity IS
BEGIN
    INVOKE SUBPROCESS Browse_Requirements;

    PARALLEL
        MUST DO BEGIN  -- Identify class candidates and classes
            LOOP       -- iteratively
                INVOKE SUBPROCESS Extract_Class_Candidates;
                INVOKE SUBPROCESS Identify_Classes;
                EXIT WHEN User_Okays_Classes;
            END LOOP;
        END DO;
    AND
        MUST DO BEGIN  -- Identify object candidates and objects
            PARALLEL   -- in parallel
                MUST DO BEGIN
                    INVOKE SUBPROCESS Extract_Object_Candidates;
                END DO;
            AND
                MUST DO BEGIN
                    INVOKE SUBPROCESS Identify_Object;
                END DO;
            END PARALLEL;
        END DO;
    AND
        MAY DO AWAIT User_Request BEGIN     -- Review requirements
            INVOKE SUBPROCESS Browse_Requirements;   -- optionally
        END DO;
    END PARALLEL;

    INVOKE SUBPROCESS Edit_Class_Object_Dictionary;
END Identify_Classes_And_Objects_Activity;
```

Figure 2: Example of the body of a JIL proactive control specification.

This example represents the first step of the Booch Object Oriented Design process, the purpose of which is to identify classes and objects. This is accomplished through five substeps. After the first substep, browse requirements, the next substeps are organized into three parallel branches: the first to identify class candidates and classes, the second to identify object candidates and objects, the third to allow further browsing of requirements. The first two branches are required; the

third is optional, pending user selection. The final substep follows the outer parallel statement. An explicit "invoke" command is used to distinguish the invocation of substeps from ordinary procedure invocation. These substeps are invoked as "subprocesses", which means that they are called synchronously by the invoker.

3.2.2 Reactive Control

The step specification also designates a separate subunit for the specification of reactions to events. The JIL event model recognizes and defines four distinct types of events, namely events related to product state, process state, resource state, and exceptions (Table 1). Most first generation process languages focus on events of one kind (e.g., product state events in APPL/A, AP5, and Marvel, process events in Adele). The definition of an event kind corresponding to exceptions is an important feature of JIL in that it allows for a generalization of the exception handling model (described in Section 3.3). The reactions triggered in response to these events can include commands of the same sorts as used in the proactive control specification; in particular, substeps can be invoked reactively.

Event Category	Examples
Product state events	Artifact updates
	Artifact state transitions
Process state events	Control events (e.g., step invocation)
	Signals (explicitly generated)
Resource state events	Resource access
	Resource access conflicts
Exceptions	Runtime exceptions
	Consistency violations (e.g., of preconditions)

Table 1: Categories of events in JIL.

An example reactive control specification is shown in Figure 3 (which continues to elaborate on the Booch process). This figure shows three types of reactions (selected for purposes of illustration). The first is to a violation of a postcondition for the step. The reaction is to re-invoke a substep to repair the violation. The second reaction is to a process event, the termination of the substep Identify Objects. The reaction is to restart the step if the class diagram is still being modified (since those modifications may outdate the object diagram). The third reaction is to an update of the requirements, upon which the work of this step depends. The reaction is to terminate the current step. Note that events generated by one step can be reacted to by another step.

3.2.3 Preconditions and Postconditions

A step may have both preconditions and postconditions. These are defined in separate packages that may be shared by multiple steps. The conditions are intended to help control the execution of the step in response to varying relations among the artifacts of the product state, but the conditions may also reference any state available in the environment, such as a reified process or resource

```
REACTIONS Identify_Classes_And_Objects_Reactions IS
-- Reactions for Booch Process Step Identify_Classes_And_Objects
BEGIN
    REACT TO VIOLATION
        OF Every_Object_Has_A_Class
        AS POSTCONDITION
        FOR Identify_Classes_And_Objects
    BY
        INVOKE SUBPROCESS Edit_Class_Object_Dictionary;
    END REACT;

    REACT TO COMPLETION
        OF Identify_Objects
    BY
        IF NOT Complete(Identify_Classes) THEN
            REDO STEP Identify_Objects;
        END IF;
    END REACT;

    REACT TO UPDATE
        OF Reqts_Spec
    BY
        TERMINATE Identify_Classes_And_Objects;
    END REACT;
END Identify_Classes_And_Objects_Reactions;
```

Figure 3: Example of a JIL reactive control specification.

state.[2] (*Constraints* can also be specified for a step. Constraints are syntactically like preconditions and postconditions, but they are enforced during the execution of the step. Constraints thus support the enforcement of intra-step consistency, while preconditions and postconditions support the enforcement of inter-step consistency.)

As the default, a step should not execute unless its preconditions are satisfied, and it should not terminate normally unless its postconditions are satisfied. However, we believe that this model is too restrictive for software processes in general: thus JIL includes several generalizations and extensions of it.

One generalization is that steps may be granted *variances* that allow them to be initiated before their preconditions are verified or to terminate before their postconditions are verified. Variances may be granted in cases where the conditions cannot be evaluated (e.g., due to contention for objects or other resources needed to evaluate the condition) or where there is good reason for overriding the programmed condition [28]. The granting of variances is supported through a runtime service (the Process Control Authority) that is accessed by the JIL interpreter (see Section A).

[2]Process and resource states are also accessed implicitly in the step specification through the step constraints and resource requirements, respectively.

A second generalization is that alternative responses may be made when violations occur. For example, when a step violates a postcondition the step may be aborted and its inconsistent results discarded. Alternatively, the step may be terminated abnormally but its results retained; this would interrupt the normal flow of the process but avoid the loss of work. In other cases, it may be more desirable to allow the step to terminate normally while leaving the product in an inconsistent state. This would allow the process to continue normally but with the product in need of some repair (this is somewhat analogous to the approach to handling inconsistency described in [2]). The coding of such approaches is done in the exception handlers (treated in Section 3.3).

At present, we are using Pleiades [31] as our product definition language and Pleiades constraints as our primary form of preconditions and postconditions. Pleiades generates an Ada package specification and the constraints represent arbitrary Ada functions. Other invokable functions (e.g., independently defined functions in Ada) may also be used as preconditions or postconditions.

3.2.4 Loose Process Organization

The proactive and reactive control specifications allow the flow of control within a step to be programmed in great detail. The preconditions and postconditions allow for further fine-grained conditional control. However, it is not always necessary in JIL to specify process control flow in great detail; it is often only necessary to indicate the composition of steps from substeps. This is important because it allows the execution agent (e.g., a human developer) to determine the order in which to attempt to execute the substeps (although preconditions and postconditions may restrict what the user can actually do). If appropriate, simple partial-ordering relations among the substeps of a step can be specified using the step constraint language.

An example of a step constraint specification is shown in Figure 4. This again uses the Booch process step Identify Classes and Objects, which was fully programmed in Figure 2. Here, a looser organization of the substeps is specified more simply. This step constraint subunit specifies two **constrain** blocks. Each of these imposes some order on a subset of the relevant substeps; both in combination impose a partial order on the whole set of substeps. The first step constraint allows the extraction of class and object candidates to proceed in parallel, followed by the identification of classes and objects in parallel. The second step constraint requires that the requirements be browsed before the class object dictionary is edited. The step constraints specified in different constraint blocks are independent in the JIL definition; so, for example, the browsing and editing substeps can occur before, during, or after the substeps involved in identifying classes and objects. The violation of a step execution constraint prevents the violating step from being executed and raises a runtime exception.

3.3 Exception Handling

Two main models of exception handling have been used in first-generation process programming languages (and in programming languages generally). These may be characterized as block-oriented and rule-based. The block-oriented model is represented by Ada and C++ and was used in APPL/A [29]. In this approach, an exception handling block is attached to the scope in which the exception may occur. This approach is especially appropriate for process-specific exception

```
STEP CONSTRAINTS Restrict_Identify_Classes_And_Objects is
BEGIN
    CONSTRAIN Extraction_and_Identification BY
        {  Extract_Class_Candidates;  |
           Extract_Object_Candidates;
        };

        {  Identify_Classes;  |
           Identify_Objects;
        };
    END CONSTRAIN;

    CONSTRAIN Browsing_and_Editing BY
        Browse_Requirements;
        Edit_Class_Object_Dictionary;
    END CONSTRAIN;
END STEP CONSTRAINTS Restrict_Identify_Classes_And_Objects;
```

Figure 4: Example of step-ordering constraints.

handling, where different occurrences of the exception should be handled in context sensitive ways. It is cumbersome, though, when the exception must be handled in a uniform way, regardless of where it arises. The alternative model is rule-based exception handling, in which exceptions trigger exception-handling rules. The consistency rules of AP5 [10] and Marvel [20] are examples. This approach is ideally suited to the case in which an exception can be handled uniformly regardless of where it originates, but it is much more cumbersome when exceptions must be handled according to the context in which they arise.

The JIL exception handling model combines these complementary approaches to exception handling. Global exception handling is provided through the reactive control mechanism, in which exceptions are treated like events (see Figure 3). This allows a single reactive mechanism to be used for both normal and exceptional reactive control. Local exception handling can be provided for a step through a subunit for local exception handlers. An example is shown in Figure 5, in which each **handle** statement represents an exception-handling block.

The exceptions shown in Figure 5 correspond to violations of preconditions and postconditions. The handling takes a variety of forms showing some of the possibilities for terminating or continuing the step. The `ABORT` command terminates the step abnormally, either with or without raising an exception. The `TERMINATE` command terminates the step normally. The `REDO` command terminates the current execution of the step and begins another. The `AWAIT REPAIR` command suspends the execution of the step pending the repair of the failed condition. The repair must be effected by some other step, which may be invoked, for example, as a reaction to the condition violation. Two compound forms of the handle statement are the `HANDLE UNLESS` and `HANDLE UNTIL`. These allow for the specification of primary and secondary handling actions; the primary action is taken, respectively, unless some special condition is met or until some special deadline is reached, in which

```
HANDLER Handle_Class_And_Object_Errors IS
BEGIN
    HANDLE FAILURE OF Requirements_Specification_Not_Empty
    AS PRECONDITION BY
        ABORT RAISE Requirements_Error;
    END HANDLE;

    HANDLE FAILURE OF No_Duplicate_Class_Names
    AS POSTCONDITION BY
        REDO STEP;
    END HANDLE;

    HANDLE FAILURE OF Every_Class_In_Relationship
    AS POSTCONDITION BY
        INVOKE SUBPROCESS Edit_Class_Object_Dictionary;
    UNLESS Close_To_Deadline(Identify_Classes_And_Objects)
    THEN
        Notify_Design_Director(...);
        TERMINATE STEP;
    END HANDLE;

    HANDLE FAILURE OF Every_Object_Has_A_Class
    AS POSTCONDITION BY
        AWAIT REPAIR;
    UNTIL Deadline(Identify_Classes_And_Objects) THEN
        ABORT;
    END HANDLE;
END Handle_Class_And_Object_Errors;
```

Figure 5: Example of JIL exception handlers.

case the secondary action is performed.

As with the variety of control-modeling mechanisms, the combination of local and global exception-handling mechanisms contributes to semantic richness and availability of alternative paradigms. It also allows flexibility that can contribute to ease of use.

3.4 Other Features

As noted, we are using the Pleiades [31] language to define our products and product consistency conditions. Pleiades provides several high-level type constructors that are especially appropriate for software products, including graphs, relations and relationships, and sequences.

A resource model and resource specification language are under development. The model includes both project-oriented and system-oriented representations of resources, including categories of human, software, and hardware resources. We believe that this model will be more general than those typically used in software systems and software processes to date.

Process state has been recognized as an important consideration in process control, management, and evaluation [16, 3, 12]. We plan to have the JIL runtime system maintain key components of the process state automatically. Additionally, the JIL event model defines events related to changes in process state; these can be used to trigger reflexive reactions.

The investigation of transaction modeling, including consistency management, was a major theme of APPL/A. In JIL, for simplicity and naturalness, steps provide a framework for defining units of concurrency control, atomicity, and consistency. However, for flexibility, as in the APPL/A model, these properties can be relaxed for a given step. Thus, for example, a step may be serializable without being atomic. Additionally, artifacts can be accessed in shared modes to allow collaborative work. Collaboration is further supported through an agenda management system, which allows group agendas and the sharing of items among individuals' agendas.

4 The Interpretation of JIL Programs

The interpretation of JIL programs is itself a process, for which the Julia environment provides the execution engine (see Appendix A). Since JIL interpretation is a process, the JIL interpreter is itself a process program. This program provides an operational specification of JIL semantics (it also provides an operational specification of aspects of JIL interpretation that are not defined by the language's semantics, such as the order of evaluation of preconditions and postconditions). To bootstrap our execution capabilities, we are programming a preliminary "level 0" JIL interpreter in Ada. Using that, we plan to program more sophisticated and flexible interpreters in JIL. This will provide us with a basis for experimentation with alternative interpretation strategies and also with alternative language semantics.

A full treatment of Julia is beyond the scope of this work and will be the subject of subsequent papers. To illustrate the Julia philosophy and approach, here we elaborate on one key issue in the interpretation of JIL, namely multi-modal interpretation.

JIL offers unprecedented flexibility in specifying process control flow, particularly in controlling substep invocation. Such flexibility imposes a corresponding requirement for flexibility in the interpretation of JIL programs. Depending on the elements present in a step specification, the step is interpreted in one of several modes. The choice of mode is determined primarily by three elements in the step specification:

- Commands: These include both proactive commands (the activity specification) and reactive commands (reactions). Substeps can be invoked explicitly by commands of both kinds.
- Execution constraints: These constrain the execution of substeps invoked by other means (e.g., commands), but they can also be interpreted as a plan for automatic substep invocation.
- Substep preconditions and postconditions: These guard the execution of individual substeps invoked by other means, but they can also provide a basis for inferring when substeps may be automatically invoked.

The presence or absence of these elements in various combinations dictate various modes of interpretation. Table 2 summarizes the combinations that determine particular modes; the modes are described briefly below.

Step Features			Interpretation Mode
Commands	Step Constraints	Substep Conditions	
Present	(Secondary)	(Secondary)	Programmed
Absent	Present	Absent	Guided
Absent	Absent	Present	Inferred
Absent	Present	Secondary	Guided/Guarded
Absent	Secondary	Present	Inferred/Constrained
Absent	Absent	Absent	Unconstrained

Table 2: Summary of applicability of JIL interpretation modes.

Programmed A step that has any command elements (activity specification or reactions) is interpreted in the programmed mode. In this mode it is assumed that substeps of the step are invoked by commands in the proactive or reactive parts of the step. (The programmed mode thus supports any combination of proactive and reactive styles of programming, without any preference for either.) If substeps have preconditions or postconditions, these guard the substeps invoked via commands. If execution constraints are also present, then the programmed substep invocations must conform to the constraints at runtime or an exception is raised. In both cases, the effect is to locally constrain the programmed execution flow.

Guided A step that has execution constraints but that lacks commands or substep conditions is interpreted in the guided mode. In this mode, the execution constraints are interpreted as a specification of an order (or partial order) in which substeps are to be automatically invoked.

Inferred A step that has substep conditions but that lacks commands and execution constraints is interpreted by inference. In this mode, the preconditions and postconditions are used to infer an order (or partial order) for the automatic invocation of substeps.

Guided/Guarded and Inferred/Constrained A step may lack commands but have both execution constraints and substep conditions. For such cases, there are two possible modes of interpretation. In the guided/guarded mode, the execution constraints are given priority and used to determine which substeps to invoke, as in the simple guided mode; the substep conditions are used to guard the invocations as in the programmed mode. In the inferred/constrained mode, the substep conditions are given priority and used to infer which substeps to invoke, as in the simple inferred mode; inferences are subject to runtime checking of the execution constraints, as in the programmed mode. The choice between the guided/guarded and inferred/constrained modes cannot be based just on the presence or absence of elements in a step specification. A default mode may be stipulated, but the alternative can be allowed via interpreter directives.

Unconstrained The simplest mode of interpretation is the unconstrained, in which a step has substeps but lacks commands, execution constraints, or substep conditions. In this case the steps are invoked automatically in some nondeterministic order, possibly in parallel. (This is equivalent in effect to the inferred mode when all preconditions evaluate to true.)

It should be noted that a JIL program can be composed of steps with heterogeneous interpretation modes. Regardless of how a step is invoked, the protocols for passing parameters, propagating exceptions, and so on are the same. The availability of multiple interpretation modes addresses the needs for semantic richness and alternative paradigms. The ability to specify process control flow in greater or lesser detail, in a form appropriate to the process or project, facilitates flexibility and ease of use.

5 Discussion

In this section we discuss how the features of JIL address our language design goals.

Fundamental Requirements JIL is a formally defined, executable programming language with semantics are based on Ada, APPL/A, and Pleiades. This directly supports the fundamental goals described in the introduction to Section 2. The language will support a variety of kinds of analyses related to control and data flow, concurrency control, exception propagation, resource usage, etc.

Semantic Richness The ability to provide technically rigorous support for software processes is generally a function of the semantic breadth and depth of a process language. JIL addresses a variety of semantic domains that is unusually wide. It represents process control, product artifacts, and project resources as first class semantic elements in the language. Maintenance of process state will also be supported through the language runtime system. The JIL semantic model builds on important lessons learned from first generation process languages (e.g., in integrating product and process representations, and in combining proactive and reactive control). However, JIL goes beyond first generation languages in several important respects such as the availability of alternative control paradigms, the degree of flexibility in consistency management, and the generality of the exception handling mechanism.

Ease of Use Despite (and in some cases because of) the semantic richness of JIL, our hope is that the ease of use of the language may be facilitated in several ways. The allowance for loosely specified models of process control means that process programs can be constructed and organized simply without requiring detailed programming. The factoring of step representations into relatively orthogonal elements allows these to be treated more or less individually. This allows specialists in particular areas (product definition, process modeling, resource specification) to work in their areas of expertise without requiring detailed understanding of all aspects of the language. The availability of alternative models for process control means that programmers can program in styles with which they are most comfortable or that are most appropriate to their process application. Support for visualization of processes and process programs should also facilitate process understanding and definition by technical specialists and non-specialists alike. We are committed to supplying templates, visual icons, and other high-level representations to facilitate the "coding" of JIL programs.

Appropriate Abstractions Compared to our previous efforts, JIL is at a level of abstraction that is better suited to the programming of important aspects of software processes. The central construct in the language is the step abstraction. This represents a generic process step and provides a linguistic context for organizing important kinds of process semantics. Specialized step attributes address essential aspects of the process, product, and resource models. Additionally, the variety of control paradigms allows process control flow to be expressed in terms that are more appropriate to a specific process or project. Although the language constructs are intended to be especially appropriate for software processes, they are still general purpose. The control model offers high-level control constructs, but imposes no particular control model. The Pleiades type model offers high-level type constructors, but imposes no required product model. This preserves the flexibility to program process-specific semantics in particular process programs. We expect to discover, implement, and embed a growing set of higher-level abstractions as our experience grows.

Composability Composability of JIL programs is provided by the ability to create a process step from existing substeps. It is further supported by the flexible interpretation model, which allows composed substeps to be interpreted in a way appropriate to their individual programming. It is also supported by the ability to attach resource specifications to steps, which enables analysis of their combined resource requirements and allows planning for their integrated execution.

Clarity through Visualization The JIL language is textual, but we hope to provide several sorts of visual windows into JIL programs and processes. We foresee the development of several kinds of adjunct visual programming languages, for example, for composing process steps, organizing control flow of substeps within a step, specifying step execution constraints, associating software objects to steps, associating resources to steps, and so on. Additionally we expect to support visualizations of process execution state (such as those provided by the ProcessWall [17]), resource usage, and other runtime concerns.

Multiple Paradigms JIL is especially rich in alternative control paradigms. It accommodates both simple and completely programmed representations of process control. It combines proactive and reactive mechanisms, and incorporates conditional control. Step execution constraints can be used to guide process execution directly or to constrain execution that is programmed using other mechanisms. JIL also takes advantage of Pleiades support for multiple paradigms for software object management, for example, the alternative views of data structures, and the provision of navigational and associative access to data.

6 Status

We have been experimenting with preliminary versions of the JIL language for about one year, writing prototype process programs and refining the syntax and semantics of JIL as indicated by the needs of our prototypes. The JIL definition has progressed to a stable initial version with which we are continuing development of process programs, language support technology, and environment infrastructure. We have defined the BNF for the JIL grammar and constructed a parser that translates JIL source code into an IRIS ([1] internal representation. We are developing an interpreter and a JIL-to-Ada command translator (see Appendix C). Our primary process programming efforts are directed at a design process based on Booch Object Oriented Design [8] and a dataflow-analysis

process based on iterative, incremental improvement of analytic accuracy [14]. Other processes are also the subject of experimentation. These include software processes, such as requirements specification, and non-software processes, such as cooking.

7 Conclusions

Experience with the development and use of first-generation process languages yielded many valuable results. Languages based on conventional paradigms have useful aspects, but they generally lack the variety of features and kinds of abstractions that are required to most effectively support process applications. Our observations of these limitations have lead us to identify a number of issues that we now believe are key to second generation process languages. Among these are the needs for semantic richness, ease of use, appropriate abstractions, process composability, visualization, and multiple paradigms. We see the balancing of the demands for both semantic richness and ease of use as one of the most critical aspects of process language design.

JIL addresses these issues in a number of innovative ways. It models processes in terms of steps with a rich variety of semantic attributes. Despite this semantic richness the language facilitates ease of use. It does this through semantic factoring, the accommodating of incomplete step specifications, the fostering of simple sub-languages, and the ability to support visualizations.

An important characteristic of JIL is the variety and flexibility of control models that it supports. JIL combines proactive and reactive control, conditional control, and more simple means of control-flow modeling via step composition and simple step-execution constraints. These mechanisms may be used alternately or in combination, both within a process and within a step.

The combination of features in JIL supports a flexible and adaptable approach to process programming. We believe that this approach will allow processes to be programmed in a variety of terms, and to a variety of levels of detail. These may be simple and high-level, or complex and elaborate. The needs of particular processes, projects, and programmers can be accommodated. We will be continuing an ongoing program of experimentation through programming processes with JIL in order to determine the extent to which JIL successfully addresses these goals.

Acknowledgements

The Julia/JIL project reflects the work of many people. The Julia-to-IRIS translator was built by Peri Tarr. The resource model has been developed by Rodion Podorozhny. The agenda-management system has been programmed by Eric McCall. The Booch product server was programmed in Pleiades by Jin Huang and Arvind Nithrakashyap. Elements of the user interface have been programmed by Sandy Wise. Process programs in JIL have been written by Peri Tarr, Rodion Podorozhny, Jin Huang, Dan Rubenstein, Chris Prosser, and Todd Wright.

References

[1] D.A. Baker, D.A. Fisher, and J.C. Shultis. *The Gardens of Iris.* Incremental Systems Corporation, Pittsburgh, PA, 1988.

[2] Robert Balzer. Tolerating inconsistency. In *Proc. of the 13th International Conference on Software Engineering*, pages 158 - 165, May 1991.

[3] Sergio Bandinelli and Alfonso Fuggetta. Computational reflection in software process modeling: the SLANG approach. In *Proc. of the 15th International Conference on Software Engineering*, pages 144-154, 1993.

[4] Sergio Bandinelli, Alfonso Fuggetta, and Sandro Grigolli. Process modeling in-the-large with SLANG. In *Proc. of the Second International Conference on the Software Process*, pages 75–83, 1993.

[5] Naser S. Barghouti. Supporting cooperation in the MARVEL process-centered SDE. In Herbert Weber, editor, *5th ACM SIGSOFT Symposium on Software Development Environments*, pages 21–31, Tyson's Corner VA, December 1992. Special issue of *Software Engineering Notes*, 17(5), December 1992.

[6] Noureddine Belkhatir, Jacky Estublier, and Melo L. Walcelio. ADELE-TEMPO: An environment to support process modeling and enaction. In Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh, editors, *Software Process Modelling and Technology*, pages 187 - 222. John Wiley & Sons Inc., 1994.

[7] Gregory A. Bolcer and Richard N. Taylor. Endeavors: A process system integration infrastructure. In *Proc. of the Fourth International Conference on the Software Process*, pages 76 - 85, December 1996.

[8] Grady Booch. *Object-Oriented Analysis and Design with Applications.* The Benjamin/Cummings Publishing Company, Inc., second edition, 1994.

[9] Gérome Canals, Nacer Boudjlida, Jean-Claude Derniame, Cladue Godart, and Jaques Lonchamp. EPOS: Object-oriented cooperative process modelling. In Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh, editors, *ALF: A Framework for Building Process-Centred Software Engineering Environments*, pages 153 - 185. John Wiley & Sons Inc., 1994.

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

[11] R. Conradi, M. Hagaseth, J.-O. Larsen, M. N. Nguyên, B. P. Munch, P. H. Westby, W. Zhu, M. L. Jaccheri, and C. Liu. EPOS: Object-oriented cooperative process modelling. In Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh, editors, *Software Process Modelling and Technology*, pages 33 - 70. John Wiley & Sons Inc., 1994.

[12] Reider Conradi, Christer Fernström, and Alfonso Fuggetta. Concepts for evolving software processes. In Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh, editors, *Software Process Modelling and Technology*, pages 9 - 31. John Wiley & Sons Inc., 1994.

[13] Wolfgang Deiters and Volker Gruhn. Managing software processes in the environment melmac. In *Proc. of the Fourth ACM SIGSOFT Symposium on Practical Software Development Environments*, pages 193–205, 1990. Irvine, California.

[14] Matthew B. Dwyer and Lori A Clarke. Data Flow Analysis for Verifying Properties of Concurrent Programs. In *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering, New Orleans*, pages 62–75, December 1994.

[15] Christer Fernström. PROCESS WEAVER: Adding process support to UNIX. In *Proc. of the Second International Conference on the Software Process*, pages 12 - 26, 1993.

[16] Dennis Heimbigner. Experiences with an Object-Manager for A Process-Centered Environment. In *Proceedings of the Eighteenth International Conf. on Very Large Data Bases*, Vancouver, B.C., 24-27 August 1992.

[17] Dennis Heimbigner. The ProcessWall: A Process State Server Approach to Process Programming. In *Proc. Fifth ACM SIGSOFT/SIGPLAN Symposium on Software Development Environments*, pages 159–168, Washington, D.C., 9-11 December 1992.

[18] H. Iida, K.-I. Mimura, K. Inoue, and K. Torii. Hakoniwa: Monitor and navigation system for cooperative development based on activity sequence model. In *Proc. of the Second International Conference on the Software Process*, pages 64 – 74, 1993.

[19] G. Junkermann, B. Peuschel, W. Schäfer, and S Wolf. MERLIN: Supporting cooperation in software development through a knowledge-based environment. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modelling and Technology*, pages 103 – 129. John Wiley & Sons Inc., 1994.

[20] Gail E. Kaiser, Naser S. Barghouti, and Michael H. Sokolsky. Experience with process modeling in the MARVEL software development environment kernel. In Bruce Shriver, editor, *23rd Annual Hawaii International Conference on System Sciences*, volume II, pages 131–140, Kona HI, January 1990.

[21] Gail E. Kaiser, Steven S. Popovich, and Israel Z. Ben-Shaul. A bi-level language for software process modeling. In Walter F. Tichy, editor, *Configuration Management*, number 2 in Trends in Software, chapter 2, pages 39–72. John Wiley & Sons, 1994.

[22] Takuya Katayama. A hierarchical and functional software process description and its enaction. In *Proc. of the 11th International Conference on Software Engineering*, pages 343 – 353, 1989.

[23] Carlo Montangero and Vincenzo Ambriola. OIKOS: Constructing process-centered sdes. In Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh, editors, *Software Process Modelling and Technology*, pages 33 – 70. John Wiley & Sons Inc., 1994.

[24] Leon J. Osterweil. A Process-Object Centered View of Software Environment Architecture. In R. Conradi, Didriksen T, and D. Wanvik, editors, *Advanced Programming Environments*, pages 156–174, Trondheim, 1986. Springer-Verlag.

[25] Leon J. Osterweil. Software processes are software, too. In *Proc. Ninth International Conference on Software Engineering*, 1987. Monterey, CA, March 30 – April 2, 1987.

[26] H. D. Rombach and M. Verlage. How to assess a software process modeling formalism from a project member's point of view. In *Proc. of the Second International Conference on the Software Process*, pages 147 – 159, 1993.

[27] Motoshi Saeki, Tsuyoshi Kaneko, and Masaki Sakamoto. A method for software process modeling and description using LOTOS. In *Proc. of the First International Conference on the Software Process*, pages 90 – 104, 1991. Redondo Beach, California, October, 1991.

[28] Stanley M. Sutton, Jr. Preconditions, postconditions, and provisional execution in software processes. Technical Report CMPSCI TR 95-77, University of Massachusetts at Amherst, Computer Science Department, Amherst, Massachusetts 01003. August 1995.

[29] 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.

[30] Masato Suzuki and Takuya Katayama. Meta-operations in the process model HFSP for the dynamics and flexibility of software processes. In *Proc. of the First International Conference on the Software Process*, pages 202 – 217, 1991. Redondo Beach, California, October, 1991.

[31] Peri L. Tarr and Lori A. Clarke. PLEIADES: An Object Management System for Software Engineering Environments. In *Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 56–70, December 1993.

A The Julia Environment

The execution of JIL programs takes place in the Julia environment ("JIL" stands for "Julia Input Language"). Julia provides a number of services for the support of software process execution. These are used by the JIL interpreter and are also available to other agents in the environment (including human users in various roles).

The logical architecture of the Julia environment is shown in Figure 6. In the interpretation of JIL programs, the general behavior of Julia is as follows. An interpreter reads JIL code (in an internal representation) from program libraries. In interpreting the JIL code, the interpreter communicates with an object manager to acquire needed software artifacts and with a resource manager to acquire needed resources. The object manager and resource manager in turn make use of a lock manager (not shown) to assure proper concurrency control.

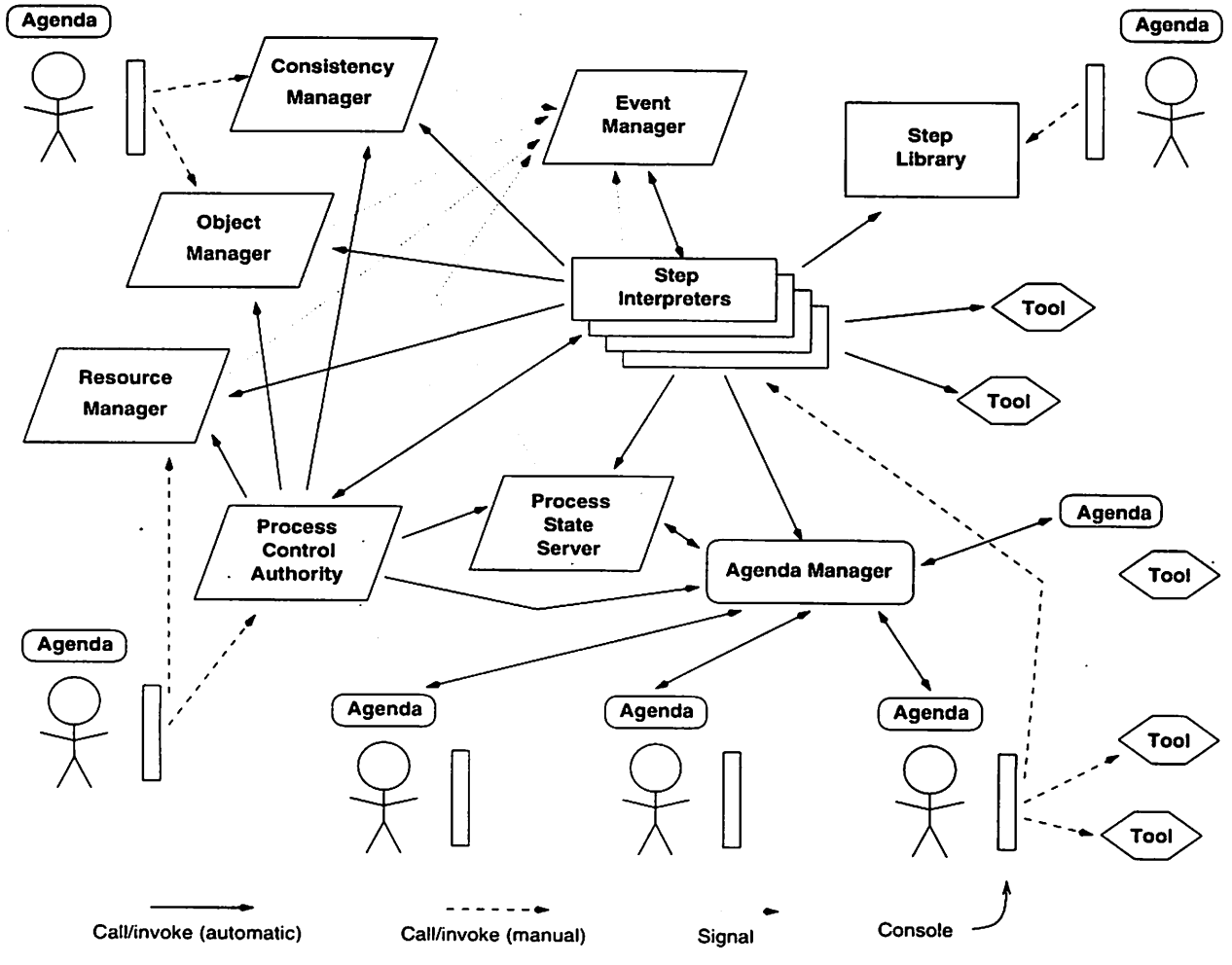

Figure 6: Architecture of the Julia environment.

Conflicts in the acquisition of objects and resources are mediated through interaction with the Process Control Authority (PCA), which provides the opportunity for human management of process interpretation. For example, suppose the execution of a step is blocked because a needed object is held by some other step; this leads the interpreter to call out to the PCA. The project manager, acting through the PCA, can direct the interpreter to respond in any of several ways. The interpreter may be directed to abort the step, to wait for the object, or to proceed without it. The PCA may also release the object from the step that initially holds it so that the later step can acquire it. While the PCA allows a human manager to exercise such control, it is also possible to automate conflict resolution, for example, according to strategies used in advanced database systems [5].

As the interpreter executes a process, and as object and resource state are updated, various event signals are generated. These are propagated via an event manager back to the executing process, which can react to these events. Such reactions, since they can occur in response to events in the process, provide one approach to process reflection. Process reflection is also supported through access to artifact state, resource state, and process state, including meta-data such as object, resource, and process status.

As a process is interpreted, the atomic steps in the process are executed by various agents, both human and automated. A JIL program can invoke external tools directly (like UNIX "exec"). Alternatively, tasks can be exported for execution by external agents. This is done through an interface that posts tasks to an agenda management system, analogous to a "Process Wall" [17]. The PCA, with support from the resource manager and scheduler (not shown), coordinates the assignment, scheduling, and distribution of tasks from multiple concurrent processes. The tasks are then distributed, according to schedules and assignments, to the agendas of particular agents, which may be human or automated, individual or group. Users, in turn, have agendas that combine events from multiple processes (and possibly other sources). Users are free to select agenda items to work on according to their own criteria, subject to the priorities and schedules attached to the items. An agenda item for a step typically provides the artifacts, resources, description, etc., for the task to be performed. Alternatively, in a "late binding" mode, the choice of tools or acquisition of artifacts may be left up to the user at the time the agenda item is selected.

B Visual Representation of JIL

JIL, with its Ada-like command syntax, allows for sophisticated programming of process control structures. However, the language is highly structured, elements may be "factored out" from step specifications, and many steps have straightforward control logic. This admits the opportunity for graphical programming of JIL steps using iconic representations of standard control constructs.

Visual representations have been widely used for both software processes and workflow in general, including especially flow-graphs of various forms (for example, SLANG [4], Melmac [13], Endeavors [7], and Flowmark (www.software.ibm.com/ad/flowmark/exmndemo.htm). These typically provide some representation for common control structures such as sequencing, iteration, and parallelism. Flow-graphs can be used to represent these features in JIL, as well.

Another visual representation for JIL processes is the phase map, in which sequential phases are represented horizontally and concurrent phases vertically. Control within a phase is indicated iconographically. Figure 7 shows the phase map for the activity specification of the step `Identify Classes and Objects` from the Booch process (The corresponding JIL code is shown in Figure 2.) The activity of the step is shown broken down into three sequential phases. The first entails the invocation of a substep, the second is subdivided into three parallel sub-phases, and the third entails the invocation of another substep. The three parallel sub-phases include one iterative branch (with iteration over a sequence), one parallel branch, and one branch with a substep invocation. The branches of the parallel phase are annotated with "must" or "may." Other icons are used to represent additional aspects of the program, for example, conditional branching, and "code" elements (where JIL source code must be filled in).

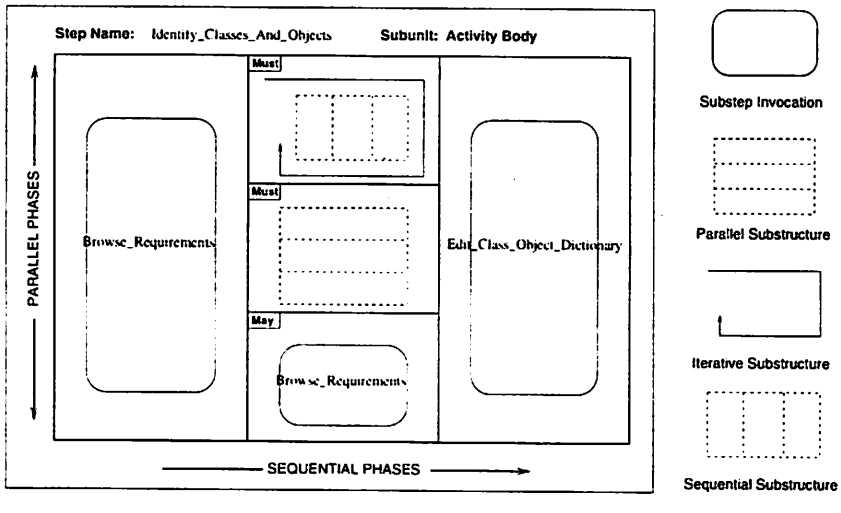

Figure 7: Phase map of the activity body for step `Identify Classes And Objects`.

We expect to develop tools to support graphical programming of JIL, leading to generation of standard JIL source code. The ability to program JIL either visually or textually (or both) should allow process programming by programmers with varied levels of skill, in a wide variety of contexts, and according to a diversity of life cycles.

C Design of the JIL Interpreter

JIL is primarily an interpreted language. Interpretation is step-oriented; that is, each step to be executed is allocated its own instance of an interpreter. Abstractly, the type of the interpreter is that of a variant process. The variants are discriminated by control category (i.e., execution mode) of the step to be interpreted. (See Section 4 for a discussion of interpretation modes.) The variant parts contain an interpreter task of the appropriate type for that control category of step. Interpretation is recursive in that the interpreter for a step invokes interpreters for the substeps invoked by that step. The interpreters for the steps of a process are linked in that each interpreter (except for that of the root step) holds a reference to its parent (the interpreter that allocated it) and (if it has any) to its children (the interpreters that it allocated). Interactions among the interpreters for a process are mediated by a server known as the Interpreter Manager.

It is our goal that JIL be a fully interpreted language. As an strategy in building our "level 0" base version of the interpreter, we are adopting the expedient of translating the command-containing subunits of JIL programs into an equivalent Ada program. These subunits include the activity specification, the reactions, and the exception handlers. The interpretation of a step that includes any of these subunits thus involves the execution of the associated Ada program. During interpretation, then, the interpreters for the various steps may be involved in communication with both other interpreters (e.g., their parents and children) and with their separate subunits. Interactions between interpreters and subunits are managed by the Subunit Manager.

The interpreter operates on an abstract interface to the IRIS internal representation of a JIL program. The basic control architecture of an interpreter task is shown in Figure 8. This figure shows the design of the interpreter task for JIL steps of the programmed control mode (for which commands are translated to Ada subunits).[3] For the given step, the interpreter follows an initiation protocol that comprises acquiring the parameter objects, evaluating specified preconditions, acquiring specified resources, and invoking the Ada program representing the translated subunits. At that point, the execution of the step is driven primarily by the execution of the subunits.

Once the step is initiated, the interpreter handles service requests. These come from three general sources, and reflect normal and abnormal behaviors. Initially, requests come primarily from the executing subunits (proactive or reactive activities in the step). Normally, these include requests to invoke substeps, evaluate constraints, and send signals. They also include "requests" that indicate that the subunits have terminated normally or abnormally, that they are raising a step exception (i.e., a process exception), and that they are raising an interpreter exception (i.e., an exception in the subunit runtime system, not in the process itself). As substeps are invoked, the interpreter will receive requests from the substep interpreters. These will concern termination of the substep and propagation of exceptions. Additionally, the interpreter may receive requests from external agents, such as the Event Manager or the Process Control Authority.

Due to the complexity of software processes, the acquisition of objects and resources, and the evaluation of preconditions and postconditions, are not guaranteed to succeed. In light of that, those

[3]This is a preliminary version that includes checking of consistency conditions but does not incorporate evaluation of substep execution constraints.

```
task body Programmed_Interpreter_Task is
-- Interpret a step for which the control information is programmed
    -- Declarations; principal subroutines:
    -- * shutdown substeps
    -- * terminate normally
    --   -- includes evaluation of postconditions
    -- * terminate with step exception
    -- * terminate with interpreter exception
    -- * terminate from shutdown
    -- * take action (from external control directive):
    --   -- retry step, terminate step, propagate exception, abort step
Begin
    -- Initiation
    -- * acquire objects (parameters)
    -- * evaluate preconditions
    -- * acquire resources
    -- * initiate subunit execution (Ada translations)

    -- Let subunits execute; handle requests from subunits,
    -- substeps, and external agents
    -- loop
            -- Handle requests from subunits
            -- * invoke substep
            -- * evaluate constraint
            -- * send signal
            -- * subunit terminated normally
            -- * subunit terminated abnormally
            -- * subunit interpreter exception
            -- * subunit step exception

            -- Handle requests from interpreters for substeps
            -- * substep terminated normally
            -- * substep interpreter exception
            -- * substep step exception

            -- Handle requests from external agents
            -- * receive signal
            -- * shutdown

            -- Exit following normal or abnormal termination
    -- end loop
exception
        -- Abnormal shutdown
        -- * release objects and resources
        -- * signal interpreter exception
End Programmed_Interpreter_Task;
```

Figure 8: Overview of interpreter logic for programmed JIL steps.

operations are mediated by the Process Control Authority (PCA), a separate process-management service that can provide direction in the event of problems (see Section A). Figure 9 illustrates the protocol for object acquisition, which is representative.

```
if Step_Has_Objects then      -- try to acquire objects
    ...  -- initialize request specification and timeout;
         -- set initial control-action to null

    while Control_Action /= Continue loop
        if Control_Action /= Wait then
            ...  -- request objects
        end if;

        if Objects_Acquired then
            Control_Action := Continue;  -- proceed from loop
        else
            -- call PCA to find out how to proceed
            ...      -- call returns a Control_Action

            case Control_Action is
                when Continue =>
                    null;  -- let step continue without all objects
                when Wait =>
                    ...  -- call back and wait indefinitely
                when Retry =>
                    ...  -- prepare a request for missing objects;
                         -- go around loop and submit request
                when Abort_Step =>
                    -- terminate step, signaling exception
                    Terminate_With_Step_Exception(...);
                    -- terminate interpreter
                    return;
            end case;
        end if;
    end loop;
end;  -- if step has objects
```

Figure 9: The object-acquisition protocol, showing mediation by the Process Control Authority.

The object-acquisition process is iterative. It continues until the objects are acquired, the step is allowed to proceed without some objects, or the step is aborted. Objects are requested initially with a timeout. If the request is satisfied, the interpreter can proceed, otherwise it calls out to the PCA. The PCA (which may represent a human or automated agent), returns with one of four control directives: *continue*, *wait*, *retry*, or *abort*. *Continue* directs the interpreter to proceed without all of the requested objects (the step will either have to do without these or request them separately later on). *Wait* directs the interpreter to call back to the object manager and wait indefinitely for access to the objects to be granted. With *continue*, the interpreter puts together a request for the outstanding objects and submits this request in a second iteration of the loop. With *abort*, the

interpreter terminates the step with the propagation of an exception (which will be received by its parent step), and then returns (completing its own execution). (A portion of the code for handling exceptions propagated from substeps is shown in Figure 10.) The protocols for acquiring resources, evaluating preconditions, and evaluating postconditions are analogous to that for acquiring objects.

```
if STEP.Has_Handler(Given_Step, The_Step_Exception) then
  -- Invoke exception handler
  SUBUNIT_MGR.Handle_Exception(
    Self, The_Step_Exception, Step_Action);

  -- Take action called for by handler
  Take_Action(The_Step_Exception, Step_Action);
  case Step_Action is
    when Continue =>
      -- a substep has just terminated; was this the last
      -- substep so that the current step should terminate?
      if INTERP_SET.Is_Empty(Substep_Interpreters) and
        Subunits_Terminated
      then
        Terminate_Normally;
        exit;
      end if;

    when Retry =>
      null;   -- retry this step in any case
    when others =>
      exit;   -- terminate this step in any case
  end if;

else -- No handler for exception
   -- Abort this substep, re-raising exception
  Shutdown_Substeps(Substep_Interpreters);
  SUBUNIT_MGR.Shutdown_Subunits(Self);
  Terminate_With_Step_Exception(The_Step_Exception);
  exit;
end if;
```

Figure 10: Detail of handling of exceptions propagated from a substep to a step.

Termination of step interpretation entails release of held objects and resources. If the step is shutdown, then any active substeps and subunits must also be shutdown recursively. Additionally, if the step is terminating normally, postconditions must also be evaluated. The completion of normal termination depends on the satisfaction of postconditions (unless the PCA allows the step to terminate even though postconditions have not been validated).