The Design of a Next-Generation Process Language*

Stanley M. Sutton, Jr. and Leon J. Osterweil

Department of Computer Science
University of Massachusetts
Amherst, MA 01003-4610

Abstract. Process languages remain a vital area of software process research. Among the important issue for process languages are semantic richness, ease of use, appropriate abstractions, process composability, visualization, and support for multiple paradigms. The need to balance semantic richness with ease of use is particularly critical.
JIL addresses these issues in a number of innovative ways. It models processes in terms of steps with a rich variety of semantic attributes. The JIL control model combines proactive and reactive control, conditional control, and more simple means of control-flow modeling via step composition and execution constraints. JIL facilitates ease of use through semantic factoring, the accommodation of incomplete step specifications, the fostering of simple sub-languages, and the ability to support visualizations. This approach allows processes to be programmed in a variety of terms, and to a variety of levels of detail, according 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, first-generation languages generally have obvious limitations. This is in part because many of these languages were based on existing paradigms that were not particularly well adapted to the domain of software process [9, 22, 31, 24, 13, 4, 23]. 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 emerging from ongoing process research.

Section 2 identifies our primary language design goals, which are based on our experience with first-generation process languages. Section 3 describes the

* 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.
Published in *Software Engineering - ESEC/FSE '97*, LNCS 1301, M. Jazayeri and H. Schaure (eds.), Springer, 1997, pp. 142-158.

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, and our status is discussed in Section 6.

2 Language Design Goals

Process programming proposes that it is feasible and valuable to represent software processes using programs written in compilable, executable coding languages [26, 27]. Our experience with APPL/A [31] has validated this proposal. We now take many properties of coding languages as fundamental to representing software processes, including formal syntax, well-defined semantics, executability, analyzability, object management, and consistency management. These issues have been the focus of much previous work (e.g., [29, 21, 9, 22, 5, 10, 31]), and they should continue to be addressed by second-generation process languages. Our focus here, however, is on the issues outlined below.

2.1 Semantic Richness

Software processes are multi-faceted and technically challenging applications. To support this domain, a process programming language must provide many kinds of interrelated semantics. This pressure for semantic richness is reflected in first-generation process languages. Many of these are based on extensions of conventional programming languages or paradigms, including functional languages ([9, 24]), rule-based or reactive languages ([22, 21, 10, 5]), imperative languages ([31]), and Petri nets ([4, 13]). Conversely, where process languages have neglected certain areas of semantics (e.g., reflexivity, resource modeling), process programs have suffered. Process language semantics must be both rich and rigorous. They must cover an adequate range of process semantics, and they must do so with appropriate models that support reasoning about processes.

2.2 Ease of Use

Ease of use is an important requirement for process programming languages because the individuals and organizations responsible for defining software processes are often not experienced at programming. The semantic richness of process languages means, however, that significant software engineering skills are required to program in them effectively. This is an impediment to the widespread adoption of process languages. A key issue for process languages is thus balancing the need for technical rigor with this need for ease of use.

2.3 Appropriate Abstractions

The clear and concise representation of software processes requires appropriate kinds and levels of abstraction. The development and maintenance of process programs is complicated if the user must construct process-specific abstractions from lower-level abstractions, as with process languages that are based on

general-purpose programming languages (e.g., APPL/A). Process programming languages should provide built-in concepts and constructs that map naturally into the software process domain. (Languages that do this with varying degrees of success include MVP-L [28], ProcessWeaver [16], LOTOS [29], and Oikos [25].)

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. Also, the ability to program processes by composing elements having different language paradigms or representing different semantic aspects would introduce additional flexibility and incrementality into process program development. Composition is also recognized as important in the subject-oriented view of object-oriented programming [17].

2.5 Clarity through Visualization

Many first-generation process languages are textual. A few process languages support graphical representations of process control (e.g., Slang [3], Melmac [13], Process Weaver [16], and Hakoniwa [20]). Visual process representations greatly aid understanding and communication of some processes. Simple ideas are often most simply represented visually, and this can aid greatly in process design and verification. 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 (especially if they are complex enough to support process execution [15]). 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 where they are workable.

2.6 Multiple Paradigms

In our opinion, one of the most useful features of APPL/A is its incorporation of triggers into a largely imperative language (Ada). We relied heavily on this combination of proactive and reactive control in our process programming. Some combination of these two types of control is also found in many other process languages (e.g., Adele [5],AP5 [9], EPOS [11], HFSP [24, 34], Marvel [22], Merlin [21], and ProcessWeaver [16]). ALF [8] is another process project that is explicitly multi-paradigmatic in combining proactive and reactive control along with preconditions and postconditions, and Pleiades demonstrates that multiple paradigms are also important in software object management [35]. And multiple paradigms have also been found useful in requirements specification [12]. In most 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 emphasizing one paradigm becomes problematic when another paradigm is more natural for a particular process. Thus, one of our goals is to support multiple paradigms without favoring any of them.

3 Design of JIL

In this section we discuss the design of JIL, emphasizing process steps, the control paradigm, and exception handling. Examples are based on a process for software design following the principles of Booch Object-Oriented Design [6].

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 include:

- *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*: Restrictions or prescriptions on the relative execution order of substeps
- *Proactive control specification*: An imperative specification of the order in which substeps are to be executed (direct invocation)
- *Reactive control specification*: A reactive specification of the conditions or events in response to which substeps are to be executed (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)

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 [6]. The step specification has a template-like syntax (i.e., it is composed of various fields). The substeps are listed within the specification. The proactive control specification (Section 3.2), reactive control specification (Section 3.2), and exception handlers (Section 3.3) are all contained in separate, named subunits. The preconditions and postconditions are defined in separate units (see

```
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
            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;
```

Fig. 1. Example of a JIL step specification.

Section 3.2). Some parts of the step specification are unspecified. An example of a possible step constraint is shown later (Section 3.2). Some general comments on the language follow before specific features are discussed in more detail.

As the division of elements in a step specification may suggest, JIL is a *factored* language. That is, it provides independent representations for independent semantics, insofar as possible. This has several consequences. First, 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.) Second, the relative independence of elements in the step specification allows steps to be defined using just a subset of the elements (in other words, many elements are optional). This promotes flexibility in process specification, since just those elements that are relevant to a particular purpose need be used. However, this imposes additional requirements on process interpretation, since various kinds and combinations of elements may be present in a step. Implications for 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 (each step interpreted independently according to the elements it contains). Additionally, elements may be combined in a single step (which is interpreted according to the particular combination of elements). These different aspects, and the resulting interpretation paradigm, are discussed below.

Proactive Control The proactive control specification of a JIL step provides a context in which the execution of substeps can be imperatively 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, including loop and conditional commands, entry-call accept statements, and a new `parallel` command with mandatory and optional branches. (Space limitations preclude presentation of an extensive example here, but see [33]).

An explicit "invoke" command is used to distinguish substep invocation from ordinary procedure invocation. A substep can be invoked as a "subprocess" or "process." In the former case, the substep executes as a child process of the

calling step; in the latter case, the substep executes as an independent process (i.e., as a separately invoked program). Subprocesses, in turn, can be invoked synchronously, like a procedure call, or asynchronously, as a parallel thread of control (like an Ada task).

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 types of 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 2. This figure shows two types of reactions. The first 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 second is to an update of the requirements, upon which the work of this step depends. The reaction is to terminate the current step.

Preconditions and Postconditions A step may have 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 according to varying aspects of product, process, and resource state.[2] (*Con-*

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

```
REACTIONS Identify_Classes_And_Objects_Reactions IS
-- Reactions for Booch Process Step Identify_Classes_And_Objects
BEGIN
    REACT TO COMPLETION OF Identify_Objects BY
        IF NOT Complete(Identify_Classes) THEN
            INVOKE SUBPROCESS 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;
```

Fig. 2. Example of a JIL reactive control specification.

straints 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 intra-step consistency, while preconditions and postconditions support 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) or where there is good reason for overriding the programmed condition [30]. The granting of variances is supported through a runtime service.

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 (Section 3.3).

At present, we are using Pleiades [35] 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.

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 control relations among the substeps of a step can be specified using the step constraint functions. These are comparable to the control specifications of ALF [8], which represent path expressions [7].

An example of a step constraint specification is shown in Figure 3. This constraint allows the extraction of class and object candidates to proceed in parallel, followed by the identification of classes and objects in parallel. (Additional step constraint functions include `Unordered` (arbitrary sequence), `Any` (nondeterministic), and `Alternate` (choice).) If step constraints are given along with an activity specification, then the step constraints are used to constrain the programmed behavior of the activity (violation of a step constraint by the activity leading to an exception). If step constraints are given without an activity specification, they are used instead to drive the execution of substeps according to the indicated control pattern.

```
STEP CONSTRAINTS Restrict_Identify_Classes_And_Objects IS BEGIN
    Ordered(Parallel(Extract_Class_Candidates,
                     Extract_Object_Candidates),
            Parallel(Identify_Classes,
                     Identify_Objects));
END STEP CONSTRAINTS Restrict_Identify_Classes_And_Objects;
```

Fig. 3. Example of step-ordering constraints.

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 [31]. 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 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 [9] and Marvel [22] 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.

Exception handling in JIL combines these complementary approaches to exception handling. Global exception handling is provided through the reactive control mechanism, in which exceptions are treated like "normal" events outside the process in which they occur. This allows one process to react in a normal way to an exception in another process. Local exception handling can be provided for a step through exception handlers. An example is shown in Figure 4, in which each **handle** statement represents an exception-handling block.

The exceptions shown in Figure 4 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` (not shown) and `HANDLE UNTIL`. These allow for the specification of primary and secondary handling actions; the primary action is taken, respectively, unless some given condition is met or until some given deadline is reached, in which case the secondary action is performed.

As with the variety of control models, the combination of local and global exception handling 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 [35] 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 [18, 3, 10]. We plan to have the JIL runtime system maintain key components of the process state automatically. Addition-

```
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_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;
```

Fig. 4. Example of JIL exception handlers.

ally, 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 shared agenda items.

4 The Interpretation of JIL Programs

The interpretation of JIL programs is itself a process, for which the Julia environment provides an execution engine. Since JIL interpretation is a process, the JIL interpreter is itself a process program. This program provides an operational specification of JIL semantics. 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 paper (but see [32]). To illustrate the Julia philosophy and approach, we elaborate here on one key issue

in the interpretation of JIL, namely multi-modal interpretation.

JIL offers great flexibility in specifying process control flow, particularly for substep invocation. Such flexibility imposes a corresponding requirement for flexibility on the JIL interpreter. 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 (activity specifications) and reactive commands (reactions). Substeps can be invoked by both.
- *Execution constraints*: These constrain the execution of substeps invoked by other means (e.g., commands), but they can also be interpreted directly to drive 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.

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, programmed control flow is locally constrained.

Guided A step with execution constraints but no commands or substep conditions is interpreted in the guided mode. In this mode, the execution constraints are interpreted as a specification of an order for automatic invocation of substeps.

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.

Inferred A step with substep pre- and postconditions but no commands or execution constraints is interpreted by inference. In this mode, the conditions are used to infer an 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; 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; 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.

JIL programs can be composed of steps with heterogeneous interpretation modes. 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 facilitates flexibility and ease of use.

5 Discussion

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 JIL addresses an unusually wide variety of semantic domains, including process control, product artifacts, and project resources. 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 JIL facilitates ease of use in several ways. The allowance for loosely specified process control means that process programs can be constructed and

organized simply, without requiring detailed programming. The factoring of step representations into relatively independent elements allows these to be treated more or less individually. Thus specialists in particular domains (e.g., product definition or resource modeling) can work in their areas of expertise without needing a detailed understanding of the whole language. The availability of alternative control models 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 JIL is at a level of abstraction that is directed to the programming of important aspects of software processes. The central construct in the language is the step abstraction. Specialized step attributes address essential semantics of the process, product, and resource models. Additionally, the variety of control paradigms allows process control flow to be expressed in terms that are 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.

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 [19]), 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, writing process programs and refining the syntax and semantics. 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 generated 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. We are also developing visual language (implemented in Java) for a subset of JIL. Our primary process programming efforts are directed at a design process based on Booch Object Oriented Design [6] and a dataflow-analysis process based on iterative, incremental improvement of analytic accuracy [14].

Acknowledgments

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. Baker, D. Fisher, and J. Shultis. *The Gardens of Iris.* Incremental Systems Corporation, Pittsburgh, PA, 1988.
2. R. Balzer. Tolerating inconsistency. In *Proc. of the 13th International Conference on Software Engineering*, pages 158 – 165. IEEE Computer Society Press, May 1991.
3. S. Bandinelli and A. Fuggetta. Computational reflection in software process modeling: the SLANG approach. In *Proc. of the 15th International Conference on Software Engineering*, pages 144–154. IEEE Computer Society Press, 1993.
4. S. Bandinelli, A. Fuggetta, and S. Grigolli. Process modeling in-the-large with SLANG. In *Proc. of the Second International Conference on the Software Process*, pages 75–83. IEEE Computer Society Press, 1993.
5. N. Belkhatir, J. Estublier, and M. L. Walcelio. ADELE-TEMPO: An environment to support process modeling and enaction. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modelling and Technology*, pages 187 – 222. John Wiley & Sons Inc., 1994.
6. G. Booch. *Object-Oriented Analysis and Design with Applications.* The Benjamin/Cummings Publishing Company, Inc., second edition, 1994.

7. R. H. Campbell and A. N. Haberman. The Specification of Process Synchronization by Path Expressions. In *Operating Systems – Proc. of an Int. Symposium, Rocquencourt, France*, volume 16 of *Lecture Notes in Computer Science*, pages 89–102. Springer, 1974.
8. G. Canals, N. Boudjlida, J.-C. Derniame, C. Godart, and J. Lonchamp. ALF: A framework for building process-centred software engineering environments. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modelling and Technology*, pages 153 – 185. John Wiley & Sons Inc., 1994.
9. D. Cohen. *AP5 Manual*. Univ. of Southern California, Information Sciences Institute, March 1988.
10. R. Conradi, C. Fernström, and A. Fuggetta. Concepts for evolving software processes. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modelling and Technology*, pages 9 – 31. John Wiley & Sons Inc., 1994.
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 A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modelling and Technology*, pages 33 – 70. John Wiley & Sons Inc., 1994.
12. R. Darimont and A. van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 179–190, Oct. 1996.
13. W. Deiters and V. Gruhn. Managing software processes in the environment melmac. In *Proc. of the Fourth ACM SIGSOFT Symposium on Practical Software Development Environments*, pages 193–205. ACM Press, 1990. Irvine, California.
14. M. B. Dwyer and L. 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. ACM Press, December 1994.
15. W. Emmerich, S. Bandinelli, L. Lavazza, and J. Arlow. Fine grained process modelling: An experiment at british airways. In *Proc. of the Fourth International Conference on the Software Process*, pages 2–12. IEEE Computer Society Press, Dec. 1996.
16. C. Fernström. PROCESS WEAVER: Adding process support to UNIX. In *Proc. of the Second International Conference on the Software Process*, pages 12 – 26, 1993.
17. W. Harrison and H. Ossher. Subject-Oriented Programming: A Critique of Pure Objects. In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 411–428, October 1993. Published as ACM SIGPLAN Notices 28(10).
18. D. 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.
19. D. 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.
20. 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. IEEE Computer Society Press, 1993.

21. 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.
22. G. E. Kaiser, N. S. Barghouti, and M. H. Sokolsky. Experience with process modeling in the MARVEL software development environment kernel. In B. Shriver, editor, *23rd Annual Hawaii International Conference on System Sciences*, volume II, pages 131–140, Kona HI, January 1990.
23. G. E. Kaiser, S. S. Popovich, and I. Z. Ben-Shaul. A bi-level language for software process modeling. In W. F. Tichy, editor, *Configuration Management*, number 2 in Trends in Software, chapter 2, pages 39–72. John Wiley & Sons, 1994.
24. T. Katayama. A hierarchical and functional software process description and its enaction. In *Proc. of the 11th International Conference on Software Engineering*, pages 343 – 353. IEEE Computer Society Press, 1989.
25. C. Montangero and V. Ambriola. OIKOS: Constructing process-centered sdes. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modelling and Technology*, pages 33 – 70. John Wiley & Sons Inc., 1994.
26. L. J. Osterweil. A Process-Object Centered View of Software Environment Architecture. In R. Conradi, D. T, and D. Wanvik, editors, *Advanced Programming Environments*, pages 156–174, Trondheim, 1986. Springer-Verlag.
27. L. J. Osterweil. Software processes are software, too. In *Proc. Ninth International Conference on Software Engineering*, pages 2–13. IEEE Computer Society Press, 1987. Monterey, CA, March 30 – April 2, 1987.
28. 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. IEEE Computer Society Press, 1993.
29. M. Saeki, T. Kaneko, and M. 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. IEEE Computer Society Press, 1991. Redondo Beach, California, October, 1991.
30. S. 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.
31. S. M. Sutton, Jr., D. Heimbigner, and L. J. Osterweil. APPL/A: A language for software-process programming. *ACM Trans. on Software Engineering and Methodology*, 4(3):221–286, July 1995.
32. S. M. Sutton, Jr. and L. J. Osterweil. The design of a next-generation process language. Technical Report CMPSCI Technical Report 96-30, University of Massachusetts at Amherst, Computer Science Department, Amherst, Massachusetts 01003, May 1996. revised January, 1997.
33. S. M. Sutton, Jr. and L. J. Osterweil. Programming parallel workflows in JIL. In *Proceedings of the 9th International Conference on Parallel and Distributed Computing and Systems*, 1997. To appear.
34. M. Suzuki and T. 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. IEEE Computer Society Press, 1991. Redondo Beach, California, October, 1991.

35. P. L. Tarr and L. 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. IEEE Computer Society Press, Dec. 1993.