Experience Using the JIL Process Programming Language to Specify Design Processes

Stanley M. Sutton Jr., Barbara Staudt Lerner, and Leon J. Osterweil
Computer Science Department
University of Massachusetts
Amherst, Massachusetts 01003

ABSTRACT

Software design is a complex process that requires significant human involvement, collaboration, and coordinated use of tools and artifacts. Software design methods describe software design in general terms but neglect many details that are important to executing specific design processes. A process program that defines a design process clearly and precisely should be an important aid to supporting and reasoning about the process. The demands placed on a process programming language in defining a software design process are great, including the need for flexible control flow, error handling, resource management, agent coordination, and artifact consistency management. This paper describes the use of JIL, a process programming language, in the definition of a process supporting Booch object-oriented design. The paper illustrates the need for precision and clarity in defining software processes, and it indicates how various of the features of JIL are effective in meeting these needs.

1 Introduction

Software development and maintenance are widely agreed to be particularly complex undertakings. They entail the development and management of diverse software artifacts, through the coordination of various human and automated agents, under the control of numerous complex processes. We believe that effective processes can be key in successfully coordinating agents to develop the needed artifacts. Our work aims to produce processes that are demonstrably effective in these ways. A major obstacle in presenting demonstrably effective processes has been the lack of languages that are adequate vehicles for this.

In our past work we have presented languages designed to support the clear and precise exposition of process. Most recently we have proposed that the JIL [16] process programming language incorporates a mix of features and abstractions that seem to offer promise of supporting the clear and precise exposition of complex software process ideas. While JIL seems promising, its success has certainly not yet been demonstrated.

In this paper we describe an effort to validate whether JIL can be used to clearly and precisely express complex software processes. We selected a popular software design method, Booch Object Oriented Design (BOOD), and attempted to determine which language features are needed in order to express the process for performing BOOD. We also evaluated the degree to which the features and abstractions incorporated into JIL support these needs.

JIL currently exists as a prototype language specification that is more complete and detailed in some areas than in others. One major goal of the work described here was to identify areas in which further elaboration is needed, and areas in which evolution of the current language notions is indicated.

The remainder of this paper is organized as follows. Section 2 provides background on object-oriented design processes and indicates some of the challenging aspects of these processes. Section 3 provides an overview of the JIL process programming language. Section 4 then discusses in detail some specific semantic complexities of design processes, drawing on BOOD for examples, and shows how JIL can be used to address those complexities. Section 5 presents an evaluation of our experience, and Section 6 describes our status and future work.

2 Object Oriented Design Process

Design is an inherently complex activity that is becoming increasingly critical as society and its infrastructure become increasingly computerized. Numerous researchers have proposed object-oriented design methods to help designers approach design processes in a systematic fashion with the goal of achieving higher quality designs as a result [3, 11, 5, 19, 7]. Numerous tools, such as Rational Rose, Graphical Designer, and Paradigm Plus[1] have been developed to support these methods,

This work was supported by the Advanced Research Projects Agency under grant F30602-94-C-0137 and F30602-97-2-0032.

[1] Rational Rose–http://www.rational.com/products/rose/; Graphical Designer–http://www.advancedsw.com/; Paradigm Plus–http://www.platinum.com/products/appdev/pplus_ps.htm.

but these tools are primarily aimed at assuring that design artifacts are expressed in correct notation. They do not aim to support the processes by which these artifacts are developed, evaluated, and evolved. A process program can provide the framework that allows such tools to be used effectively within the guidelines of these methods [12].

Even when a specific method is used, object-oriented design remains an activity that is inherently iterative, has a flexible flow of control, and, perhaps most importantly, is largely driven by human innovation. Unlike workflow applications, which are often more mechanical in nature, design processes must allow for a great deal of flexibility, reactivity, and exception management. Moreover, the users of a process must feel that it is providing support and guidance rather than authoritarian control.

In this paper, we use Booch Object Oriented Design (BOOD) [3] as an example method to demonstrate how the constructs in JIL provide the functionality to meet the demands of design process specification. BOOD consists of four main steps: identifying classes and objects, identifying the semantics of the classes and objects, identifying the relationships between the classes and objects, and implementing the classes and objects. These steps are repeated iteratively, typically starting with high levels of abstraction and iteratively adding lower levels of abstraction. Each of these major steps is further subdivided into smaller steps. We show how JIL's step composition and proactive control model can meet these needs.

In many cases, the choice of which substeps to use and the order in which they are applied is left to the designer. For example, during the identification of classes and objects, a designer can apply any of the techniques of classical object-oriented analysis, behavior analysis, or use case analysis. We show how JIL can support this freedom of choice. In some cases, however, a project manager might require the use of a particular technique, in which case a refinement of the BOOD process program could require the use of that technique rather than making it optional. By modifying the process program, these variations in the method can be specified and enforced.

The successful completion of a BOOD step is characterized by well-formedness conditions on design artifacts such as those representing the definition of classes, objects, and their relationships. These are milestones of the BOOD process and are represented as postconditions in the JIL process program. JIL's reaction and exception-handling mechanisms support precise specification of actions to take in the event that postconditions do not hold.

It should be clear from this discussion that the role of humans in the design is central even when the design is formalized with a process program. Humans are responsible for performing many of the steps, often choosing which steps to perform, checking most of the postconditions, and fixing problems that result in failures of the postconditions. JIL supports humans in these tasks, but it also provides the ability to track the progress of the design process, to assist in the assignment and reassignment of steps to agents, both human and automated, and to manage contention for resources among the agents. The value of these capabilities increases with more complex designs and larger design teams.

3 Overview of JIL

JIL [16] is a process-programming language intended to support the development of high-level process abstractions through a collection of powerful and easy to use semantic constructs. JIL represents processes as compositions of steps. The specification of a step is defined in terms of a number of elements. Each element defines a specific aspect of step semantics, such as data, control, resource usage, or consistency requirements. For most elements, the actual element definitions appear in a separate body. In this section, we give a flavor of JIL by presenting an example step specification and describing the elements that appear in step specifications.

Figure 1 contains an example step specification in JIL. This specification represents the first step of the BOOD process. While this example does not include all possible elements of a step specification, it is representative of their basic style. Elements not present in this example will be seen in later examples. Briefly, the elements of a step specification are as follows:

Objects and declarations: Parameters and local data used by the step.

Resources: Specifications of resources needed by the step, including people, software, and hardware. People may be represented as specific individuals or by roles and skills. Software may include tools, services, and systems. Hardware may include computer systems and non-computer objects.

Steps: Identification of the substeps of a step (which are themselves steps). The substeps are used within the control flow elements described below.

Activity: Identification of the unit containing an imperative definition of the actions the step takes when it is invoked. This is typically an embedding of substep invocations into traditional programming constructs.[2]

[2]Note that *activity* as used in this paper generally refers to a JIL language construct, not to elements of a design method as in Booch [3].

```
STEP Booch_Process IS
  OBJECTS
    Req_file:              FILE;
    Data_dictionary:       DICTIONARY;
    Interface_file:        FILE;
    ...

  STEPS:
    Identify_Classes_and_Objects;
    Identify_Class_and_Object_Semantics;
    Identify_Class_and_Object_Relationships;
    Implement_Classes_and_Objects;
    Propagate_Requirements_Update;

  ACTIVITY: Booch_Process_Activity;

  REACTIONS:  Booch_Process_Reactions;

  PRECONDITIONS:
    FROM Requirements_Constraints USE
      Approved(Requirements_File);
      ...

  POSTCONDITIONS:
    FROM Design_Constraints USE
      Interface_Files_Complete(
        Data_Dictionary, Interface_Files);
      ...

  HANDLERS:  Booch_Process_Handlers;
END Booch_Process
```

Figure 1: Specification for Step Booch_Process

Reactions: Identification of the unit that specifies programmed reactions to triggering events. Events may include artifact updates, process control events, and non-local exceptions, while the reactions typically include substep invocations.

Step execution constraints: Restrictions on the relative execution order of substeps. These can be used to constrain the runtime behavior of the activities and reactions or to drive substep execution directly.

Preconditions, constraints, postconditions: Consistency conditions that must be satisfied (respectively) prior to, throughout, and subsequent to the execution of the step. Conditions may refer to product, process, or resource state.

Handlers: Identification of handlers for local exceptions. Handlers can invoke substeps, thus exception handling can use the full power of JIL to recover from errors.

The elements in a step specification represent a variety of semantics that are important to the definition, analysis, understanding, and execution of software processes in general. Steps are important not only as a process modeling construct in and of themselves, but also as a mechanism for binding and scoping the semantics represented by these various elements.

As suggested by the example in Figure 1, JIL is a *factored* language. By this we mean that each element may be included or omitted from a step specification as is appropriate for the step at hand, leading to a specification with or without the corresponding factor. To give just some examples, resource specifications or step execution constraints may be omitted (as in the example), or the activity or reactions may be omitted, or preconditions and postconditions may be omitted. This factored approach allows processes to be specified in just enough detail, and in just those terms, that are relevant to the process and to the needs of the people and the organization for which it is being defined. The combination of powerful semantic constructs with a factored language design leads to a language that is both expressive and flexible, thereby promoting precision, adaptability, and ease of use.

The Julia process-execution engine is designed to support the execution of JIL process programs (JIL stands for Julia Input Language). Julia will provide a variety of services related to the execution of JIL process programs and management of JIL processes. The Julia architecture is discussed in more detail in [15]; relevant aspects are discussed here where appropriate.

4 Supporting Design Processes

Design methods are typically described in relatively general terms, omitting many details so as to be adaptable to, and adoptable by, a wide variety of organizations and projects. In effect, they define generic process architectures [13]. Design methods also typically emphasize nominal behavior, that is, they describe how the design process would proceed under ideal conditions. They tend to omit aspects related to handling errors, propagating changes, and accommodating inconsistencies. Additionally, design methods tend to focus on the actions of an individual designer and do not address collaboration among designers or the allocation of resources to individual designers. Thus, the actual execution of design processes requires addressing many issues and performing many activities that are not defined by design methods. The handling of these additional issues and activities is a significantly complicating factor in actual software design processes.

If actual software design processes (as opposed to process architectures) are to be defined, then the range of relevant concerns must be addressed. Moreover, pro-

cess definitions should be clear and precise, so as to promote understandability and facilitate execution consistent with the intended definition. However, design processes have a number of features that make them too complex to be described effectively using simple workflow models or standard programming languages. These features of design processes revolve around the human creative element, the need to cope with a changing environment, the iteration and backtracking that are commonplace yet unpredictable in design, the management of teams of designers contending for limited resources, and the need for ensuring consistency of design artifacts even though they are being developed by multiple designers in parallel.

Programming languages are computationally powerful but their abstractions are relatively low level and do not speak directly to many process concerns. Workflow languages tend to be higher level but relatively limited computationally. For example, they typically lack support for some important kinds of process semantics (e.g., exception handling). Many languages have been proposed for software process programming, but many of these bear close resemblance to programming languages (e.g., [4, 14, 10, 9, 8, 1]) or to workflow languages (e.g., [6, 2]), and thus they tend to suffer from the same limitations. In this section, we elaborate on selected aspects of design processes with examples from BOOD, providing explanation of how JIL supports these aspects of the design process. The particular aspects of design processes we address are flexible control flow, coordination of people and tools, coping with errors, and artifact consistency.

4.1 Flexible Control Flow

One major challenge in defining a design process lies in providing enough clarity and precision to assure the creation of a design that has the specific quality attributes that have been selected for the particular project. At the same time, though, designers must retain enough flexibility within the approved process to use their creative talents effectively. A good design process must also be able to react to events that might affect the normal flow of control through the process, such as changes in the requirements or identification of an error in the design. In this section, we provide some examples of the need for clear, precise, yet flexible control flow in design processes, including the nominal process flow and reactions to events, and describe how JIL addresses these needs.

4.1.1 Describing the Nominal Process BOOD is described in terms of a sequence of steps. The steps are decomposed into activities. The order in which the activities are performed and the manner in which each activity is performed are left unspecified. For example, the first step of BOOD is to identify the classes and objects for the design. The activities include classical object-oriented analysis, behavior analysis, and use-case analysis. BOOD does not specify an order for these activities, or even, in fact, whether all must be carried out. Such questions may be determined by the organization, project, or individual designer responsible for the design.

JIL similarly describes a process as a collection of steps, which may be divided into substeps. JIL provides a variety of constructs for specifying the control flow between steps and their substeps. By means of these constructs process control flow may be described simply or in detail, strictly or flexibly. Decision points can be explicitly represented, and those decisions may be made by human or automated agents.

JIL supports simple specifications of process control flow through the *step execution constraints*. These are step composition operators that take as their arguments steps or nested step composition operators. The `ORDERED` operator specifies that steps must be executed in the given order; this is simple but rigid. Three other operators introduce various degrees of flexibility into process execution. The `UNORDERED` operator specifies that steps must be executed in some sequence, but one that is not defined by the process program. The `PARALLEL` operator specifies that the given steps must be executed according to some nondeterministic serial or concurrent interleaving. An example showing the composition of operators is shown below:

```
Ordered(Parallel(Classical_OO_Analysis;
                 Behavior_Analysis;
                 Use_Case_Analysis),
        Review_Class_Diagram);
```

Another useful step composition operator is the `TRY`, which specifies that the given substeps are to be executed in the given order, but stopping when one succeeds. In BOOD, the need for this control construct is apparent when considering the substeps involved in implementing classes and objects. If the chosen standard strategy for implementing a class fails, then a custom implementation is tried next. This effect is achieved with the `TRY` operator:

```
TRY(UNORDERED(
        Look_for_Inheritance,
        Look_for_Objects_to_Delegate_to,
        Look_for_Parameterized_Class),
    Custom_Implementation);
```

The step execution constraints offer simple, clear, precise constructors for specifying process control flows

with strict or flexible behaviors and the opportunity for human involvement in guiding process control. However, they are not computationally general. For detailed process programming, JIL offers `ACTIVITIES` and `REACTIONS` (the latter are discussed in Section 4.1.2).

The activity body of a JIL step provides for imperative programming of *proactive* process control flows, that is, what the step should do when it is invoked. The JIL command syntax includes familiar forms of iteration and branching statements. JIL also adds a `PARALLEL` command and commands by which substeps can be invoked as threads or separate programs. (The programming of parallel workflows in JIL is discussed at length in [17].)

An example of an activity body representing an implementation for step Identify_Classes_And_Objects is presented in Figure 2. The process program for this activity, although more complex than the examples of step execution constraints, is still straightforward. Moreover, it is precise and clear about many important aspects of the process. The classes and objects are iteratively identified and reviewed. The identification is performed according to a preferred analysis technique; the technique used may vary from iteration to iteration within a single execution of the step. The analysis technique is selected by a programmed step that is defined as part of the process. The implementation of that step is not shown here, but, depending on the implementation, the preferred technique might be determined by querying the project manager, querying the design engineer, or querying a project database. In any case, the precise function used in selecting the preferred analysis technique is defined explicitly by the process programmer, and different selection functions may distinguish different design processes. The class diagram that results from analysis is reviewed at the end of each iteration. Iteration continues until the class diagram passes review or until the deadline for the step is near (in which case an exception is raised).

It is easy to imagine other significant details of the process that might be programmed differently. For example, the preferred analysis technique might be obtained once per invocation of the step instead of once per iteration of the loop, or when the deadline is imminent an extension might be requested instead of raising an exception, or the iteration that is now within the step might be moved outside the step. None of these specific issues is significant for the Booch Method, but they are indicative of the wide range of concerns that must be addressed in an executable process, and all can be stated precisely and clearly in a JIL step activity body.

Although the process program for this activity is straightforward, it would be difficult to capture this process and its variants using just the step execution constraints or other comparably simple execution models. The difficulty is that even traditional programming language constructs such as JIL includes (iteration, branching, etc.) come in a variety of forms and readily enable the programming of an even wider variety of composite control structures. Simple control languages can be elaborated to address more and more features, but at some point they cease to be simple. While the JIL step constraint language may continue to evolve to improve the usefulness of its constructs, we will continue to keep them simple. Activities are available for complicated processes that can be most clearly expressed using traditional programming commands.

```
ACTIVITY BODY
  Identify_Classes_And_Objects_Activity IS
BEGIN
  Get_Deadline(Deadline);

  Identify_And_Review:  LOOP
    INVOKE SUBPROCESS
      Get_Preferred_Analysis_Technique(Technique);

    CASE Technique IS
    WHEN Classical =>  INVOKE SUBPROCESS
      Classical_OO_Analysis(...);
    WHEN Behavioral => INVOKE SUBPROCESS
      Behavior_Analysis(...);
    WHEN Use_Case =>   INVOKE SUBPROCESS
      Use_Case_Analysis(...);
    END CASE;

    INVOKE SUBPROCESS Review_Class_Diagram(
      Class_Diagram_Approved);

    exit when Class_Diagram_Approved;

    IF Deadline_Near(Deadline) THEN
      RAISE Deadline_Violation;
    END IF;
  END LOOP Identify_And_Review;
END Identify_Classes_And_Objects_Activity;
```

Figure 2: Example of JIL Activity Body

4.1.2 Reacting to Events During design, there are typically events that occur outside of an individual designer's activity that affect the activity. For example, a change to the requirements of a system may impact the design that is in progress. BOOD does not describe such external events or how to deal with them. An executable process program, however, must be able to do so. At a minimum, a requirements change must be evaluated to determine its impact on the design. If a design is af-

fected by the change, the current design must be halted and modified to accommodate the change. Changing one component of the design might have ripple effects on other parts of the design and perhaps even the budget and schedule. Again, evaluating the effects of the change and adjusting the design to accommodate the change require extensive human involvement. The process can assist in sending notifications to the designers working on related parts of the design.

JIL provides the `REACT` control construct for programming reactions to events. Reactions are attached to steps. A reaction specification identifies the event to which the reaction occurs and the action to take in response to that event. Reactions may be used in combination with, or instead of, an activity body. If a step contains both an activity and reactions, the reactions can execute in parallel with the activity. Reactions can respond to events based on changes in artifact state (e.g., product updates), changes in process state (e.g., step initiation and termination), changes in resource state (e.g., a resource becomes available), and programmer-defined events (which must be explicitly signaled).

The following shows a reaction from the top-level BOOD step (Figure 1):

```
REACT TO Requirements_File.Update BY
    INVOKE SUBPROCESS
        Propagate_Requirements_Change;
  END IF;
END REACT;
```

The reaction occurs in response to the update of the requirements file. The reaction is to invoke a substep to propagate the update through the design product. This subprocess is itself a step. In a simple form, this might just send electronic mail to all the designers. A more complex form of this step might analyze the portion of the requirements that changed, and, using information about the relationships between the requirements and the design, inform only the affected designers.

4.2 Coordination of People and Tools

One limitation of BOOD and other design methods is that they focus on the activities that a single designer undertakes. In real design processes, however, teams of designers work concurrently to complete a design. As a result, a practical design process must augment a design method with management activities to schedule and control the use of limited resources and to assign tasks to individual designers. In this section, we describe how JIL supports these management activities.

4.2.1 Resource Management To complete a design, a designer requires various resources such as the artifacts that serve as inputs to the design step and tools and hardware used to assist in the completion of the step. The designer may produce additional artifacts that serve as inputs to other design steps. From a manager's (or process's) perspective, each designer is also a resource bringing unique skills to the development team. When multiple designers work together, it is clear that resource contention becomes an important issue. For example, concurrent modification of design artifacts must be controlled, the workload should be balanced across the available designers, individual designers should be assigned tasks consistent with their skills, computer resources cannot be overloaded, and use of software tools must adhere to licensing constraints.

Julia provides a resource management component that allows a project manager or process programmer to define a resource model in terms of a set of programmer-defined attributes. The resources are organized into a classification hierarchy. A JIL program can reserve and acquire resources either by naming a specific resource or by requesting a resource that has particular attribute values (including resources of a particular type). By using attribute-value specifications to acquire resources instead of explicit naming, the resource management system can assign resources intelligently to balance the load across the resources and adjust for changes in the available resources, allowing the definition and exact execution of the process to be guided by resource requirements and availability.

When the step is executed, the JIL interpreter in Julia processes the resource specifications to obtain the indicated resources from the Julia resource manager. Julia's resource manager is a module that could be implemented in the form of a human, or an automated system, or a synergistic combination of the two.. If the specified resources cannot be granted, the JIL interpreter raises an exception, as discussed in Section 4.3.

The scarcity or abundance of resources has important effects on execution of a process. Yet the process definition cannot be specific to the details of resource availability. Thus, for example, abundant resources (e.g.,. skilled designers) may well enable considerable parallelization of effort. But it is unreasonable to write different process programs to depict different degrees of parallelization. The separation of resource specification in JIL, from resource binding in Julia, enables the writing of one single clear, precise process program that will execute differently under different resource availability situations.

4.2.2 External Execution Agents At some point in any design process, it is necessary to rely upon the talents of the designers and the capabilities of the software tools to complete the task. The development of a design

is substantially a human, creative activity that is facilitated by a variety of software tools. The consistency of a design is evaluated with respect to conditions that require a mix of manual and automated evaluation. To a significant degree, then, the purpose of a design process and process program is to define, coordinate, and support the work of people and tools. JIL supports the incorporation of people and software tools through the concept of external *agents*, which can be human or automated. Activities in a JIL process can be exported for execution outside the process by an external agent. There is a uniform interface in JIL for dealing with both human and automated agents. This enables process programs to ignore the differences between the two types of agents except when process semantics otherwise require that those differences should be manifested in the process program text.

Agents and resource management are related in the sense that certain resources, namely people and software tools, can serve as agents for a step. Binding agents to a step is therefore performed by acquiring resources to serve as the agents and then posting steps to the agendas of those agents using the agenda manager component of Julia.

Figure 3 shows the specification of agent resources, that is, designers, who will be assigned the responsibility of completing the class and object identification step of BOOD. The specification requests a number of designers determined by the parameter `Team_Size`; these designers may be skilled at any of the analysis methods that might be used in the process.

4.3 Coping with Errors

In addition to reacting to events, a process must also be able to cope with errors that arise during the execution of a step. For example, a design tool might be a resource required to perform a step. The process must be able to deal with errors related to the inability to acquire resources when they are needed or to the unexpected loss of a resource after it has been acquired. The former might occur if there are a limited number of licenses allowing concurrent use of a tool, while the latter might occur if the license expires or the machine on which the tool executes becomes unavailable.

Exception handling is not usually described as part of any design method. Nevertheless successful exception handling is critical to the execution of any design process. JIL provides an exception handling mechanism to cope with errors that arise in invoking, executing, and finalizing a step. Often humans will need to intervene to fix the problem identified by an exception before the step can proceed, so JIL exception handlers can invoke further JIL steps, and the exception-handling process can be defined in as much detail as appropriate. For example, if a postcondition on the step Identify_Classes_And_Objects fails because some part of the requirements file was overlooked, the exception handler may reinvoke the entire step on that part of the requirements.

```
STEP Identify_Classes_And_Objects IS
  RESOURCES:  Identification_Resources;
  ...
End;


RESOURCE SPECIFICATION
    Identification_Resources IS
BEGIN

  Engineers:  Agent_Resource_Handle :=
    Resource_Manager.Acquire_Agents(
      Kind =>      Designer,
      Number =>    Team_Size,
      Skill =>     (Use_Case_Analysis,
                    Behavioral_Analysis,
                    Classical_OO_Analysis),
      Duration =>  Unlimited);

END Identification_Resources;
```

Figure 3: Example of JIL resource specification.

The following is an example of a handler that copes with the failure to acquire a needed resource during the initialization of a step:

```
    HANDLE Resource_Acquisition_Error BY
      IF Resource_Of(
           Resource_Acquisition_Error)
             = Design_Tool
      THEN
        AWAIT EVENT
            Resource_Available(Design_Tool);
        CONTINUE;
      END IF;
    END HANDLE;
```

(Additional structured forms of handler are defined [16].)

Besides resource management errors, exception handlers are also useful for dealing with typical programming errors and with exceptions raised by violations of consistency conditions on design artifacts or process state (discussed in Section 4.4).

If a step handles an exception successfully, the step may be terminated successfully or retried. If the exception was raised during step initialization or finalization,

those activities may also be continued. For example, suppose an exception is raised during the initialization of a step, after the step has acquired half of the resources it needs; if the exception can be handled, then initialization can proceed without loss of the previously acquired resources. If the intervention is unsuccessful, the step may be terminated abnormally, in which case it fails but its results are preserved in case they can be reused in subsequent iterations of the step. Alternatively, the step may be aborted, in which case it fails with the discarding of the results. (Propagating an exception is equivalent to abort.) Thus, there are a variety of responses that can be made to an exception, the most appropriate of which will depend on the overall process and on the particular circumstances in which the exception occurs.

4.4 Artifact Consistency

BOOD is described in terms of artifacts and activities with equal status. That is, to fully understand BOOD, it is necessary both to understand the various design diagrams that are the artifacts produced by the process as well as the activities that need to be performed to create and manipulate those diagrams. In this section, we briefly discuss artifact definition and illustrate the way in which JIL integrates artifact consistency management into the executable aspects of process programs.

4.4.1 Artifact Definition In JIL, artifact definition and management is done using Pleiades [18], a persistent object management system. Artifacts are decomposed into collections of interrelated objects. This decomposition allows for simple concurrent manipulation of an artifact by allowing designers to work on non-intersecting sets of objects. Pleiades also supports the specification of both inter-artifact and intra-artifact relationships, both of which are useful in ensuring artifact consistency. For example, when a requirement changes, these relationships can be used to identify the pieces of the design that are affected. The potential ripple effect of the requirements change can be determined by identifying the transitive closure of the relationships between and within the artifacts emanating from the object capturing the requirement change.

4.4.2 Assessing Milestones The ultimate goal of BOOD is to create the artifacts that represent a high quality design for the given requirements. Each step in BOOD identifies milestones. Reaching a milestone is indicative of successful completion of the step to which the milestone is attached. For a process to describe BOOD accurately, it is necessary to evaluate the artifacts to determine if the milestones have been reached.

JIL allows a programmer to attach postconditions to steps. These postconditions can be used to determine if the milestones have been reached. Upon completion of a step, all the postconditions for the step are evaluated. If a postcondition fails, an exception is raised that can be dealt with in an exception handler. A common action to take on failure of a postcondition is to repeat some or all of the activities in the step to correct for the failed postcondition. In the context of design, many postconditions require human evaluation and exception handlers for failed postconditions require human action. For example, a postcondition for the class and object identification step of BOOD is that the classes and objects are consistently named. Evaluating naming consistency is not likely to be something that can be automated in any generally meaningful way and thus should be the responsibility of a human reviewer to verify. The action to correct for naming inconsistencies involves renaming the inappropriately-named classes and objects. A software tool can ensure that the new names are applied everywhere they occur in the design, but a human must be involved in selecting the new names. JIL quite comfortably supports the specification of these sorts of interactions between tools and humans.

4.4.3 Artifact Flow Artifacts created in one design step are typically required as input for other design steps. JIL explicitly represents the artifact flow between steps using parameters. Artifact flow often implicitly constrains the order in which steps can be executed and thus the amount of concurrency that is possible between steps. For example, Step 2 of BOOD is to define the semantics of classes and objects. Obviously, a class's semantics cannot be identified until the class itself has been identified in Step 1. Thus, the output of Step 1 becomes the input of Step 2. Note, however, that a certain amount of concurrency is possible, even expected, between these two steps. In particular, it is not necessary to identify all classes before defining the semantics of some of them. As a result, Step 2 can be started as soon as some classes or objects have been identified, but no sooner.

JIL provides several ways of specifying concurrency between steps that would allow the semantics of some classes to be defined while other classes are still being defined. Premature execution of Step 2 is prevented by attaching a precondition to the step requiring that the data dictionary must contain at least one class or object whose semantics have not yet been defined. Failure of the precondition raises an exception whose handler delays initiation until the precondition becomes true, as shown below. In this manner, the control flow through the design process can be constrained by the availability of the appropriate artifacts.

```
HANDLE VIOLATION OF
  Some_Undefined_Classes_Or_Objects
AS PRECONDITION BY
    AWAIT EVENT
        Artifact_Created(Class) or
        Artifact_Created(Object);
    CONTINUE;
  END IF;
END HANDLE;
```

4.4.4 Optimistic Design The evaluation of preconditions and postconditions for design processes may be complex and prone to delays due to the need for human evaluation or conflicts over access to other required resources. These problems may be especially severe for processes that are long-lived and highly concurrent, as design processes may be. In the worst case, it may be impossible to evaluate preconditions and postconditions in a timely manner, leading to a state of indeterminacy, introducing potentially unnecessary delays in the design process. For example, there is often a delay between the completion of a portion of a design and the formal review meeting to evaluate the design so that the reviewers have adequate time to become familiar with the proposed design. If a project is under an ambitious schedule, it might be unreasonable to force the developer to wait until the review is complete before continuing with more detailed design or implementation. Instead, a manager might want to trust the designer to identify the least risky aspects of the design and continue their development prior to completion of the review.

To help address these problems, the JIL execution model includes the notions of *consistency variances* and *at-risk execution*. A consistency variance is a variance from the usual enforcement of preconditions and postconditions. In cases where a condition cannot be evaluated due to conflicts over access to needed resources, a step may be granted a variance from enforcement of the condition, that is, the step may begin to execute before its preconditions are verified, or it may complete before its postconditions are verified. This entails subsequent at-risk execution, since the step may later be aborted, or a significant amount of rework may be required if the condition is found to be violated. The benefits of increased concurrency and efficiency may outweigh the risks of a possible violation, however. We do not expect that variances will be granted automatically in general. Rather, we see this as another opportunity for people to be involved, this time in the role of process managers guiding process interpretation.

5 Evaluation

In this project, we evaluated the adequacy of JIL for the specification of the BOOD process. The two areas of need that we had identified as being of primary interest were precision and clarity of the process specification. Two additional goals were process flexibility and the ability to accommodate people in the process. The ability to support clarity and foster ease of use through visualizations of process programs was a further goal.

5.1 Precision and Clarity

We believe that the foregoing section strongly supports our hypothesis that the sorts of linguistic features contained in JIL are quite effective in enabling the precise specification of BOOD processes. We have indicated numerous ways in which process details are needed in order to distinguish among several alternative ways of performing BOOD. These distinctions seem essential if we are to be able to provide strong computer support for execution of these processes, and to be able to reason definitively about various important aspects of them. We have demonstrated that JIL generally has specific features that enable us to make these very precise (and crucial) distinctions.

We also believe that JIL supports clarity in the process descriptions we have developed and examined. One key way in which clarity is achieved is through the use of higher-level, process specific abstractions. While the number of constructs in JIL may be greater than in a lean programming language, the constructs have now been shown to be present to meet demonstrated needs. We have sought to add to the language constructs that succeed in reducing the size of process programs by expressing needed higher level semantic features in a direct and terse fashion. In addition we have striven to make these features cleanly orthogonal, and to assure that they have clear semantics.

Our work has also indicated some areas in which JIL did not readily enable us to be sufficiently clear or precise. In particular, while the step composition operators were useful, it would be beneficial to extend this set to include some simple looping constructs and to better integrate the step composition operators into the procedural portions of the step descriptions. In addition, the resource specification capabilities in JIL were found to be less precise than might be desired. These findings have helped us to pinpoint areas in which further development of JIL should be focussed.

5.2 Flexibility and Accommodation of People

The goals of process flexibility and accommodation of people in the process are closely related in that flexibility is partly necessary for and partly exercised by people in the process. Flexibility was achieved in a number of ways. Some of the step composition operators (e.g., `UNORDERED` and `PARALLEL`) contain an aspect of nondeterminism where human decision making can occur.

Through the integration of resource management into process execution, process programs can be decoupled from bindings to specific agents and resources. The use of preconditions and postconditions allows the execution of process steps to be controlled according to dynamically changing product state. Humans also serve as agents for step execution, carrying out step activities, reacting to process events, and handling process exceptions. These features proved quite useful in specifying alternative BOOD processes.

We also noted some limitations in these areas. Experimentation is needed with alternative strategies for making nondeterministic control decisions. Some linguistic means of specifying how those decisions should be made would also be helpful, especially for explicitly delegating control decisions to human agents.

5.3 Visual JIL

Another way in which the need for clarity is being addressed is through the use of visualization. A key aspect of the JIL development project has been the concurrent development of Visual-JIL. Visual-JIL is an interactive graphical system that supports the development of process programs written in Little-JIL, a subset of JIL that has been carefully selected for clear visualizability and significant effect. Visual-JIL emphasizes features related to agent coordination and concentrates on the simpler, yet still flexible, process control structures. It is hoped that the visual representation, coupled with the relatively simple but useful subset semantics, will foster the adoption of process programming by making it easy to write process programs that are still clear and precise. Indeed, our experience with the BOOD process is based on programming in both full JIL and in Little-JIL using the Visual-JIL system. The precise specification of many aspects of these examples was possible through the use of Visual-JIL, which offers distinct visualization and clarity advantages in those areas of the language it supports.

We believe that JIL, Visual-JIL, and Julia are examples of a language and system that balance the level of expressive power needed to support sufficient precision with necessary clarity and usability. We believe this claim is supported by the existence of BOOD process programs that are precise, expressive, and clear.

6 Status and Future Work

As noted earlier in this paper, the development of JIL, Visual-JIL, and Julia are ongoing activities. Evaluation activities such as the one described here are an integral part of the development of these languages and system. From these evaluations we are gaining key insights into the strengths and weaknesses of these languages and systems. This particular evaluation has reinforced our belief that JIL and Visual-JIL are likely to be broadly useful in continuing efforts to support process execution and powerful reasoning about real processes. Thus, we expect to begin releasing JIL and Visual-JIL specifications and prototypes in the coming year or so.

Simultaneously, evaluation activities such as the one described here will continue. For example, we are extending the BOOD process program described here to support collaborative design involving multiple designers sharing the same set of artifacts and resources. In doing this, we expect to evaluate further the effectiveness of JIL's support for resource, agenda, and artifact management and to use this experience to help us refine the abstractions required during these management activities.

We are also planning to use JIL and Visual-JIL to develop and evaluate process programs for such other software engineering activities as regression testing, configuration management, and incremental dataflow analysis. We expect to continue to learn about JIL and Visual-JIL strengths and weaknesses from these activities. Based on these experiences, we expect to add new features for which there is a demonstrable need, delete existing features for which the need seems slight, and thereby continue to enhance the clarity and precision of the language. Increasingly, moreover, we also expect that the process programs themselves will become increasingly interesting, useful, and valuable artifacts.

Acknowledgments Work on JIL, Julia, and JIL process programming has been performed by many people, whose contributions we gratefully acknowledge. Alexander Wise is the primary architect and developer of Visual-JIL. Eric McCall is the designer and developer of the agenda management system and a contributor to Visual-JIL. Rodion Podorozhny is developing the resource management system and has contributed JIL process programs. Jin Huang and Arvind Nithrakashyap contributed to the implementation of a Booch-product object-management system in Pleiades. Peri Tarr provided Pleiades support and developed the initial JIL parser. Additionally, Chris Prosser, Dan Rubenstein, and Todd Wright have all contributed JIL process programs, and we have benefited from the comments of numerous colleagues at the University of Massachusetts and elsewhere.

REFERENCES

[1] 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. IEEE Computer Society Press, 1993.

[2] 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. IEEE Computer Society Press, December 1996.

[3] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings, Redwood City, CA, second edition edition, 1994.

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

[5] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The FUSION Method*. Prentice-Hall, Englewood Cliffs, NJ, 1994.

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

[7] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. ACM Press, New York, 1992.

[8] 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.

[9] 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.

[10] 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. IEEE Computer Society Press, 1989.

[11] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[12] Xiping Song and Leon J. Osterweil. Engineering software design processes to guide process execution. In *Proc. of the Third International Conference on the Software Process*, pages 135 – 152, 1994.

[13] Xiping Song and Leon J. Osterweil. Experience with an approach to comparing software design methodologies. *IEEE Trans. on Software Engineering*, 20(5):364–384, May 1994.

[14] 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.

[15] Stanley M. Sutton, Jr. and Leon 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.

[16] Stanley M. Sutton, Jr. and Leon J. Osterweil. The design of a next-generation process language. In *Proceedings of the Joint 6th European Software Engineering Conference and the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*. Springer-Verlag, 1997. To appear.

[17] Stanley M. Sutton, Jr. and Leon J. Osterweil. Programming parallel workflows in JIL. In *Proceedings of the 9th International Conference on Parallel and Distributed Computing and Systems*, 1997. To appear.

[18] 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. IEEE Computer Society Press, December 1993.

[19] R. Wirfs-Brock, B. Wilkerson, and L. Weiner. *Designing Object-Oriented Software*. Prentice-Hall, Englewood Cliffs, NJ, 1990.