An Architecture for Flexible, Evolvable Process-Driven User-Guidance Environments

Timothy J. Sliski, Matthew P. Billmers,
Lori A. Clarke, Leon J. Osterweil

CMPSCI Technical Report 00-63

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

An Architecture for Flexible, Evolvable Process-Driven User-Guidance Environments

Timothy J. Sliski*, Matthew P. Billmers, Lori A. Clarke, and Leon J. Osterweil
Laboratory for Advanced Software Engineering Research
Department of Computer Science
University of Massachusetts at Amherst
Amherst, MA 01003-9264
+1 413 545 2013
{tjs, billmers, clarke, ljo}@cs.umass.edu

ABSTRACT

Complex toolsets can be difficult to use. User interfaces can help by guiding users through the alternative choices that might be possible at any given time, but this tends to lock users into the fixed interaction models dictated by the UI designers. Alternatively, we propose an approach where the tool utilization model is specified by a process, written in a process definition language. Our approach incorporates a user-interface specification that describes how the user-interface is to respond to, or reflect, progress through the execution of the process definition. By not tightly binding the user-guidance process, the associated user-interfaces, and the toolset, it is easy to develop alternative processes that provide widely varying levels and styles of guidance and to be responsive to evolution in the processes, user interfaces, or toolset.

In this paper, we describe this approach for developing process-driven user-guidance environments, a loosely coupled architecture for supporting this separation of concerns, and a generator for automatically binding the process and the user interface. We report on a case study using this approach. Although this case study used a specific process definition language and a specific toolset, the approach is applicable to other process definition languages and toolsets, provided they meet some basic, sound software engineering requirements.

1 INTRODUCTION

As toolsets, collections of interrelated tools, become larger and more complex, users often find it difficult to understand how to use them. Toolsets should not frustrate users in their efforts to solve a problem, but should facilitate arriving at a solution. For example, users should not have to struggle in deciding where to start or where to go next after a result is received from a particular toolset component. Such ambiguities cause users to falter unnecessarily, can cause them to feel unproductive and frustrated, and may be one reason that CASEware often turns out to be SHELFware. This paper presents an approach that enables providing explicit user guidance in tool application. We refer to such a facility as a *user-guidance environment*.

Our approach entails the development of user-guidance processes to specify and help generate such environments. The user-guidance processes, written in a process definition language, are programmed to guide users away from inappropriate or illegal tool usage, while still giving users great flexibility in how they can apply tools. In addition, our approach augments the process definition with a user-interface binding specification, describing how the user interface is to respond to, or reflect, progress in using the toolset in executing the defined process.

Our approach supports flexibility, evolvability, and generality. It is flexible in that it supports a wide range of process definitions. For example, our work can be used to support either novices or experts and impose either strong or weak control over user actions. Our approach also supports evolution. With time, novices become more expert, and expert users find different ways in which they would like to utilize toolsets. In addition, the toolsets themselves evolve, with new tools being added and existing tools being modified. Thus, user-guidance processes must evolve over time to support both personal and toolset growth. Finally, the approach that we propose is quite general. Although to evaluate our approach we conducted a case study using a specific process definition language and a specific toolset, the approach is applicable to other process definition languages and toolsets, provided they follow some basic, sound software engineering practices.

The hypothesis of this work is that process definition languages can be used effectively to define user-guidance processes and that the execution of these processes can provide an effective user-guidance environment. More precisely, we have three specific subhypotheses. The first subhypothesis is that process definitions can be used to create environments that provide guidance to users through correct and effective execution of sequences of tools in complex

* Now at Alphatech, Inc., 50 Mall Road, Burlington, MA 01803

toolsets. Our second subhypothesis is that by augmenting a process definition with a user interface binding specification, our approach can automatically coordinate the process execution with the user-interface actions. Our third subhypothesis is that we can achieve considerable flexibility, evolvability, and generality by separating the process, the user interface, and the toolset and by providing technology that automatically coordinates all three under the umbrella of a user-guidance environment.

2 RELATED WORK

There has been a considerable amount of software engineering research aimed at helping users to be more effective in using collections of tools. Some of the earliest work simply addressed the need for tools to interoperate with each other. Early efforts suggested the use of a centralized database as a vehicle for managing the flow of artifacts among tools. For example, the Stoneman software environment specification proposed the use of such a centralized database to hold and distribute all of the artifacts needed by all the tools [7]. Thus, the artifacts were the primary form of communication among tools. Major problems with such an approach are the relatively rigid schema structures that are required and the insistence on artifact schema uniformity across all tools. Both of these are obstacles to environment evolution. A centralized database may suffice for a small collection of tightly integrated tools or as a conceptual architectural view, but experience showed that more loosely integrated architectures [20, 22, 25] are needed what to provide more flexibility.

A subsequent approach suggested that tool interoperability be supported by event based notification. With this approach tools are able to communicate artifacts and events as messages, where responsibility for delivery of these messages resides with a service that is separate from the various tools. Advantages of this approach are that the tools need not communicate directly with each other, nor do they need to be proactive in fetching needed artifacts from a central repository. This loose interaction model was pioneered in past software environment research [20, 25, 26], and is now widely supported by commercial products [6, 21-23] and programming languages [15].

While these efforts made progress in effecting tool interoperability, they did little to facilitate user interactions with large toolsets. Many tools boast superior user interfaces, designed to help users with the use of individual tools, but the total interface to integrated collections of tools is often uneven and confusing. To address this problem, some environments emphasized the use of a consistent look and feel to all tools, giving the user an impression of a seamlessly integrated toolset. Wasserman proposed one such environment [30], suggesting the use of a coordinated set of design diagrams as the vehicle for uniform access to all design tools, data, and artifacts needed to support design. Beaudouin-Lafon proposed another, using an iconic visual model for interacting with tools [5]. In this approach, icons are bound to application objects, and dragging these icons between windows or to other icons results in operations such as message passing or function invocation. Thery et al. propose a system for providing user interfaces for a collection of diverse theorem provers [29]. Experience with such environments suggested that the uniformity of the user interface was appealing, but, especially as toolsets became large and diverse, users needed guidance in deciding which tools to use under which circumstances and how to avoid tool incompatibilities. In some cases, where such guidance was incorporated as an integral part of the user interface, experience indicated that it was often difficult to respond to the varying levels of user expertise and to changing toolsets.

Provision of such guidance in the integration of tools to support software development was a key objective of process-centered environments. This approach was first suggested by Osterweil [24] and was a focus of the Arcadia project, in which processes were defined using a procedural-based language [17, 27] and were used to specify how tools were to be integrated in support of software development. The MARVEL/Oz family of environments provides [18] another good example of this approach. In MARVEL/Oz the process is described using rules, and specific tools are bound to the various rules so that the triggering of a rule could effect the invocation of one or more tools. Early versions of MARVEL triggered Unix tools, and later versions of this system have addressed increasingly broad and ambitious tool integration facilities. HFSP, a hierarchical functional language, was used in a similar way as a blueprint for the application of tools [28].

Process Weaver was an early example of a graphical process formalism, using data flow diagrams to represent processes, with activity boxes being used as the loci of tool bindings [14]. Similarly, SLANG [3, 4] and FunsoftNets [12, 16] both used modified Petri Net formalisms to define processes, with various of the transitions being bound to tools. All of these systems were successful in supporting specifications of tool integration, and the graphical depictions were particularly useful in helping designers understand and reason about the integration issues. While the processes drove the sequencing of tool invocations, the processes were not able to help users in dealing with the separate interfaces to the various separate tools.

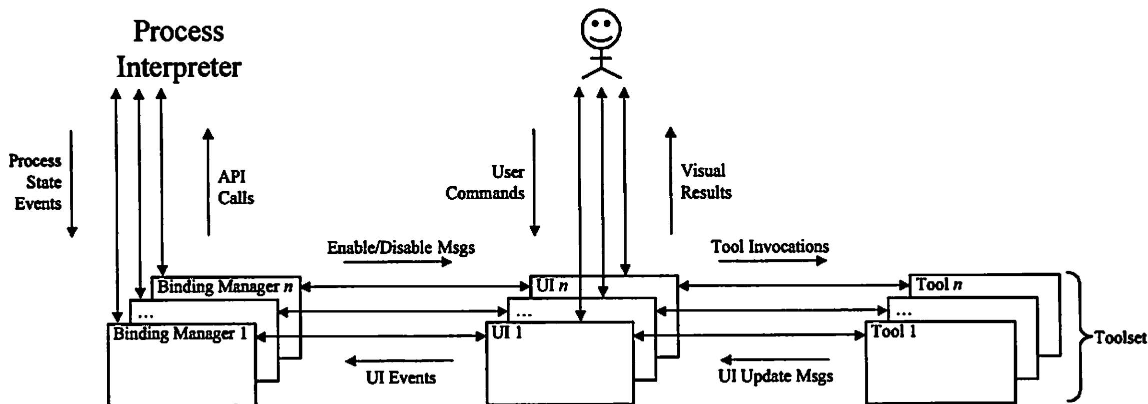

Figure 1: High Level Architecture

What seems needed is an approach combining the advantages of coordinated user interfaces with the advantages of process guidance. In particular an approach in using the process both to integrate tools and to manage interaction with human users seems promising, and is described here.

Finally, we note that coordinating process execution with user actions is somewhat reminiscent of work on coordinating help or tutorial information with user actions. This work, has primarily concentrated on recognizing and exploiting user models and not on the infrastructure for supporting that coordination [13, 19]. This approach is also reminiscent of work on process inconsistency management [2, 9, 10], where the goal is to identify when and where actual performance of a process diverges from the process specification and to tolerate the situation. That work, however, is not directed towards our goal of providing positive proactive assistance to users in the effective use of toolsets to support process performance.

Our approach proposes that the tools, user-interfaces and processes should all be independent first-class entities. We propose that a process be used to guide the execution of user interfaces and guide users through correct execution sequences of a complex toolset, while maintaining the independence of the individual components. The coordination of these components is facilitated by a binding manager that communicates with these components via event based notification.

3 ARCHITECTURE OF A PROCESS-GUIDED ENVIRONMENT

Our architecture consists of four major components: the tools that make up the toolset, the user interfaces (UIs) for the individual tools, a process interpreter, and the binding managers that interpret a set of bindings for connecting the UIs to the process interpreter. The data flow relationships among these components are shown in Figure 1. This architectural decomposition is driven by a desire to separate process, toolset, and UI concerns so they can evolve separately from one another. Typically, process, UI, and toolset code are tightly intertwined, making it difficult to modify any component independently. The architecture proposed here separates the process interpreter from the toolset and UI by inserting a "binding manager" between the process and each UI. The process interpreter sends process state change information to the binding managers, which use this information to enable and disable UI actions. Similarly, UI components communicate user interaction information to the process interpreter via the binding managers so that the process can respond appropriately.

This approach is most successful when the relevant steps in a process definition correspond closely to toolset API method calls. In addition, the user interfaces for the individual tools must be separated from the tools themselves. If all these conditions hold, the process, the UIs, and the bindings among these can all be defined separately from the toolset's tools. The binding managers have to be specialized to deal with the specific process definition and UI programming languages. While this somewhat limits the generality of our solution, it would be relatively easy to specialize the binding managers for several existing process and programming languages, and so we feel the overall approach is still relatively general.

In this depiction of our architecture, and in our case study, each UI in the toolset is associated with one tool and a separate binding manager. Conceptually, all binding managers have the same functionality, but a particular binding manager implementation is only concerned about controlling the UI with which it is associated. The approach that we are proposing does not limit the cardinality relationships between these three components. A single tool may have several user interfaces, or there may be several tools underneath one UI. Additionally, it may be possible to have one binding manager for all of the UIs. For the purposes of this discussion, we assume that there is one binding manager and UI per tool and that there are multiple tools in the toolset.

```
[infile <filename>
    [<UIComponentType> < UIComponentName >]*
    [parameter <parameterName> of <stepName> [comesfrom | goesto] <methodName>]*]*

[when <stepName> instate <state>
    [ do  <UIAction> on <UIComponentName> [with <options>] ]*
    [ if  <UIComponentName>     ReceivesEvent <UIActionPerformed> [and if
          <ClassName.methodName>] do
          <processReaction>     onstep <stepName> [with <exceptionName>] ]*]*
```

Figure 2. Syntax for the Binding Specification Language

Description of High-Level Architectural Components

Process Interpreter: The process interpreter is responsible for executing a process definition, which can be viewed as a set of steps that need to be completed, but perhaps in orders that may be intricate and highly dependent upon runtime conditions. The process interpreter is responsible for sequencing the steps in a process, assuring correct artifact flow into and out of steps, and securing all resources needed for step execution. The key resource needed by a step is the agent, either a human or automated system that is to be responsible for executing the step. In cases where a human is the execution agent for a step, the human is responsible for doing what is necessary to carry out the work of the step, but the process is responsible for providing needed resources and artifacts, and for coordinating the work of this step with substeps, sibling steps, and parent steps. In some cases, this coordination may entail restrictions on the agent's activities. Thus, a major challenge in our work is to assure that the process communicates with human agents in ways that are clear, supportive, and convenient. Further, the process must provide flexibility to the human user, while still preventing the human from doing things that are illegal, ineffective, or inappropriate. To some extent this is a process definition challenge, but our work has also indicated that the challenge is facilitated by a suitable process definition formalism.

Thus for example, humans need to be told clearly and gently when the process determines that the human is to be the agent for a tool application step. The assignment should be reflected in the user interface for that tool, rather than through a separate interface to the process. Thus in our architecture, the assignment of a tool usage task to a human agent is communicated through the tool's binding manager, which then communicates the assignment to the agent through the existing tool UI. Any flexibility, such as alternative choices of tools, is likewise communicated through existing UIs.

Other events of importance in step execution include the binding of needed resources, the passing of parameters, the invocation of substeps, the completion of substeps (and completion status), and the occurrence of exceptional conditions. In our architecture, the binding managers are responsible for filtering the interpreter's notification of these process state events and communicating them to the user through the appropriate tool UI, to assure that the user receives guidance that is both useful and inclusive of contextual information to help users understand why options are either provided or denied. This allows us to keep the process interpreter as generic as possible by keeping domain knowledge out of the interpreter, thus supporting our initial design goals of flexibility and generality.

The process interpreter must also provide an API by which the binding managers, and other components, can tell the process interpreter the state of the steps an agent is executing, such as when a human agent successfully completes an assigned step. The states of a step would depend on the process definition language, but would usually include states such as ready to execute, executing, execution completed normally, and execution completed abnormally.

UI: The UI component consists of the user interfaces to all of the tools in the toolset. These UI subcomponents usually employ a set of widget components, such as menu items, windows, checkboxes, data entry fields, and lists, that make it easy to design and implement the desired UI functionality for a particular tool. In general, a UI function can be enabled or disabled, thus allowing or preventing the user from performing some actions. When a UI function is enabled, it can be selected by the user, usually by clicking on a particular widget with a mouse button. When the user selects a UI function, the UI generates an event that encapsulates what the user did and notifies the function's observers about the event.

In our architecture, a UI has the additional responsibility of notifying the process interpreter when events of interest to the process occur. Clearly not all UI events are of interest to the process. Thus the event stream must be filtered. Here too, in the interests of flexibility, our architecture places the onus of determining the events to filter and the events to communicate upon the binding manager.

A UI's binding manager enables and disables the UI's functionality based on the current state of the executing process. Therefore, a UI must provide an API that allows UI functions to be enabled or disabled. In those cases where existing UIs do not provide mechanisms by which components external to the UI can register to receive notification about UI events or do not provide an API for enabling and disabling functions, it will be necessary to build special wrappers for these UIs.

Binding Manager: Our work has shown that the binding managers can be automatically generated from a specification that defines the relationship between the process and the UI, and vice-versa. This specification describes how to modify UI function states when the process state changes and how to change the process state when the user takes some action in the UI.

The syntax for the binding language we use is shown in Figure 2. The specification is divided into two sections. The first declares the UIs, component types and names, and parameters to be referenced in the specification. It also indicates the file for that UI, followed by the UI component types and names and parameters that will be referenced. The transfer of information between the UI and the process interpreter is done via parameters that are "passed" to the UI using the accessor or update method, *methodName*. The second section defines the bindings when the process is at a particular state of a step. This is done with a when clause that is potentially followed by lists of *UIAction* statements and lists of *processReaction* statements. A *UIAction* statement defines how to modify the state of a UI function when a step with the specified name enters a certain state. Thus, when a step named *stepName* progresses to state *state*, the binding manager is instructed to cause each *UIAction*, with *options*, to occur within the *UIComponentName* user interface. The options are used to give the UIComponent additional information where needed, such as the index of the tab of a JtabbedPane. When the process interpreter assigns a step to the user, an event will be generated and the binding managers will be notified of the step assignment via an appropriate event. Each binding manager then informs the UIs for which the event is relevant and determines what state change (e.g., enabling a menu item) to make to the appropriate UIs.

Similarly, the *processReaction* statement describes how to modify process state based on a UI state change. When a step named *stepName* progresses to state *state*, then the binding manager is instructed to react to the receipt of a *UIActionPerformed* event from *UIComponentName*, by doing *processReaction* on the step *stepName*, if the guard *ClassName.methodName* returns true (if specified). If an exception *exceptionName* is specified, it indicates that the process interpreter should throw that exception within the process. The guard *classname.methodname* provides a placeholder for an arbitrarily complex Boolean function that, if used, must be provided by the specifier. Thus, with such a specification, if the user clicks on a menu item while in step *stepName*, the UI will generate an event encapsulating the identity of the function that caused the event and will notify the binding manager. Upon receiving the event, the binding manager will check to make sure that the step is in the specified state. If the step is in that state, the binding manager will make the indicated change in the executing process step's state, otherwise it does nothing, making the assumption that the process has already moved on.

Figure 3 shows a data-flow diagram for the binding manager generation process. The parser recognizes the binding specification and creates five lists consisting of Process-UI bindings, UI-Process bindings, ProcessReaction bindings, Parameter bindings, and the UI components that will be referenced in the existing UI classes. The parser also uses the process definition to verify that the binding specification references step names that are actually declared in the process definition. These lists, along with the source code for the UI components, are input to a class creator, which generates new classes that include the binding manager code and instantiations of the binding managers in the new versions of the UI classes. The binding manager code consists of two sections, one that has knowledge of how to interact with the process interpreter and one that knows how to observe and respond to the declared UI widgets.

Toolset: The toolset is the collection of tools that actually perform the process steps based on what the user chooses to do. While this paper has emphasized the management of tools that are invoked by a human user, it is important to remember that some tools may be invoked directly from the process, without any user interaction. Under our architecture this could be done in two possible ways; by direct invocation from the binding managers or by direct invocation from the process interpreter. Direct invocation from the process interpreter requires tool domain knowledge be embedded in the process interpreter. Invocation through the binding managers, on the other hand, provides greater flexibility and generality by keeping concerns separate. In either case, the tool must provide either an appropriate API or be wrapped to provide an API at the same level of abstraction as the process definition.

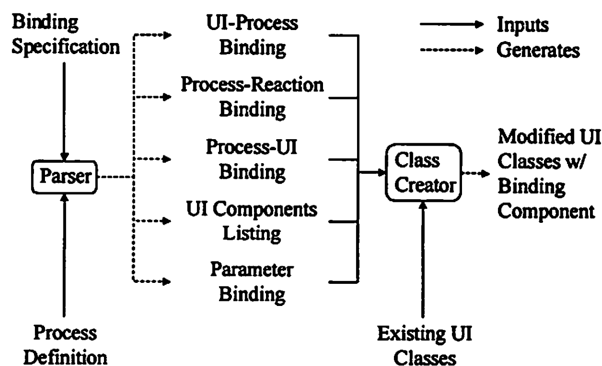

Figure 3: Generator Architecture

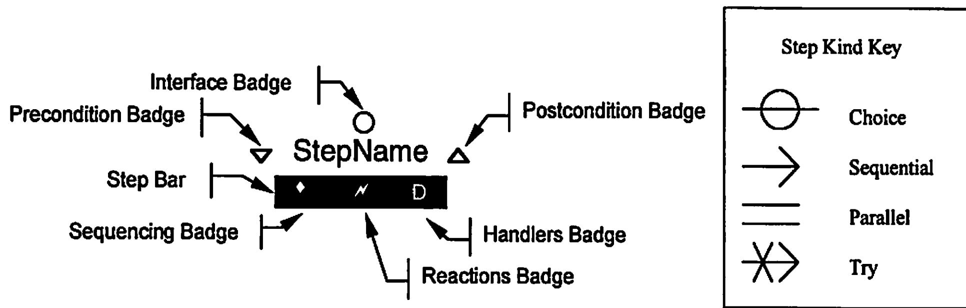

Figure 4 – A Little-JIL Step & Step Kinds

This architecture presents a relatively generic approach for the creation of flexible, evolvable user-guidance environments. We achieve these goals by adhering to the classic software engineering goal of separation of concerns by maintaining process, tool, and UI independence.

4 CASE STUDY

To study our hypotheses and the applicability of our architecture, we undertook a case study that consisted of integrating a complex toolset, called FLAVERS, with a process definition written in the Little-JIL process formalism. FLAVERS uses data-flow analysis to verify properties of software systems [11]. We believe that FLAVERS is an appropriate choice for our study because software analysis is a complex domain. It is unreasonable to expect that an inexperienced analyst could use FLAVERS effectively without training or guidance. Moreover, we found that even users that have knowledge of verification need some guidance in the use of FLAVERS. Little-JIL is a semantically rich, visual process definition language [31, 32] that offers process abstractions that are useful in constructing user-guidance processes. Little-JIL is supported by the Juliette interpreter that offers an interoperability layer having an API that allows external components to interact easily with an executing process.

Before describing the FLAVERS processes, it is important to have a rudimentary understanding of Little-JIL. Little-JIL represents processes as a hierarchical decomposition of steps. Little-JIL uses different step-kinds to orchestrate different execution orders for substeps. The step-kinds that we focus on here are sequential, choice, parallel, and leaf. A sequential step is executed by having its sub-steps performed sequentially in left-to-right order. A choice step presents the user with a set of sub-steps that represent options. When the user chooses one of the substeps, the others are retracted from the set. A parallel step is executed by having all of its substeps performed, but in no specified order, and potentially in parallel. Finally, leaf steps are not decomposed into sub-steps. Leaves are steps at which human agents apply tools to move a process forward. A generic step is represented in figure 4, along with the types of sequencing badges used during this discussion. Juliette interpretation is driven by a finite state machine, whose principal states include posted, started, retracted, and completed. The transitions among these states are the basis for the events that drive the UI.

The specific Little-JIL process for using FLAVERS that is shown in figure 5 defines a canonical way in which a user should be encouraged to perform an analysis. The steps of this process consist of choosing a system to verify, choosing properties to be verified on that system, choosing constraints, checking that the system is correctly annotated, running the analysis, and analyzing the results of that analysis. The details of these high level steps are also encoded as process definitions, but are not shown here because of lack of space The process describes how humans should use the FLAVERS toolset, but it also has implications for the UI for these tools. Figure 6 provides an example of a process-UI binding specification that describes how the process definition in Figure 5 should be coordinated with the user interface. In this example, when the "Choose Properties" step is posted, the process interpreter sends an event to the binding managers indicating that the user has been assigned a new step to perform. The Property Tool UI binding manager's job is to notify the user of this, in this case by enabling the appropriate tab in the tabbed panel of the tool UI. A similar effect is desired when "Choose Properties" enters the "Completed" state, except in this case, the job of the binding manager must assure that the tab is disabled. According to the specification, when the user selects the editPropButton, an event is sent to the binding managers. The Property Tool UI binding manager will receive this notification and process the event. If the executing process has a step "Edit Property" and it is in the state "Posted", then the "Edit Property" step will be executed (completed).

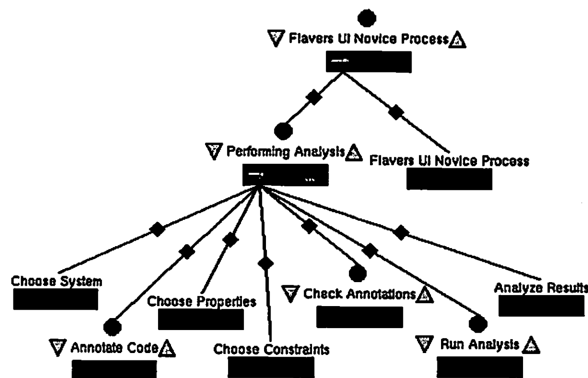

Figure 5 – The FLAVERS Novice Process

```
infile e:\tjs\...\FlaversUI.java
      JTabbedPanel mainTabbedPanel
infile e:\tjs\...\PropertyPanel.java
      JMenuItem editPropButton
infile e:\tjs\...\PropertyPanel.java
      JMenuItem doneProps
when "Choose Properties" instate Posted
      do Enable on mainTabbedPanel with 2
when "Choose Properties" instate Completed
      do Disable on mainTabbedPanel with 2
when "Edit Property" instate Posted
      do Enable on editPropButton
      if editPropButton receivesevent selected do CompleteStep onstep "Edit Property"
when "Edit Property" instate Retracted
      do Disable on editPropButton
when "DoneWithProperties" instate Posted
      do Enable on doneProps
      if doneProps receivesevent selected do StartStep onstep "DoneWithProperties"
when "DoneWithProperties" instate Started
      do Terminate on doneProps with ContinueException
```

Figure 6 An example process-UI binding specification

The implementation of the generator for the binding managers closely conforms to the description provided in Section 3. Here, we introduce the specifics of the implementation, given that the FLAVERS toolset and the Little-JIL process execution system are both written in JAVA and that the UIs for the FLAVERS tools are based on the JAVA SwingSet components.

During generation, each UI source file is modified to include a new additional class, which implements the interface ActionListener, and extends an abstract class AgendaMonitor. The ActionListener interface requires the method actionPerformed (Event) be implemented in the concrete class. The processStateChange (StepName, StepState) method is an abstract method, and sub-classes provide a concrete implementation. The generation process produces these concrete implementations. As part of generation, the original UI source code is modified so the original class instantiates the generated class, and the generated class is registered as an ActionListener with all the components in the original class that will be controlled via the process. When the user performs an action such as selection of a menu item, an event is generated and all the ActionListeners for the selected component are notified of the event, via an API call to the ActionListener's actionPerformed() method

The actionPerformed() method has an if-statement for every UI-Process binding related to the UI in the binding specification. In the example 2in Figure 6, if the user selects the "doneProps" menu item, when the "DoneWithProperties" step is still "Posted," the step will be started.

The generated class's super-class, AgendaMonitor, incorporates the functionality of observing the set of tasks that the user has been assigned and calls the processStateChanged() method when a process step changes state. In the example, we specified that the doneProps menu item should be enabled when "DoneWithProperties" is "Posted". Additionally, we indicated that we want to terminate "DoneWithProperties" and to throw an exception if the "DoneWithProperties" step enters the "Started" state.

Experiences

In studying the FLAVERS process, it became clear that we could potentially overly restrict the user if we were not careful in how we specified the process. Particularly, certain process constructs seemed to lead to a proliferation of dialog boxes, prompting the user for input that might be implicit under other circumstances. For example, the process construct shown in Figure 7 would have required that a user working with properties explicitly state an intention to keep working with properties after every selection of a menu item associated with properties. This excessive amount of dialog did not seem to be user friendly, nor was it consistent with our original goal of providing user-guidance. Our solution

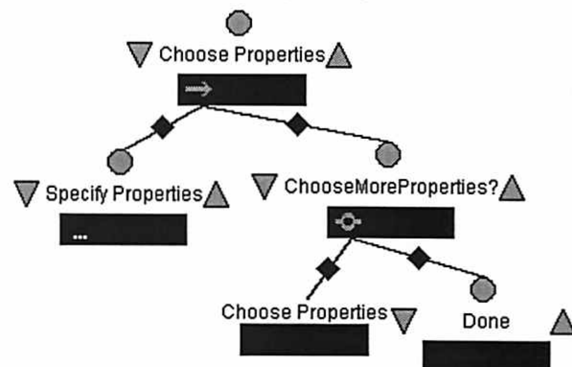

Figure 7 – Properties with Problematic Iteration

to this problem is shown in Figure 8, and involves adding a choice to the "Specify Property" step, which allows users to explicitly indicate when they are done working with properties. When the user selects a choice, the sequential step "PropertiesIterator" is invoked recursively, automatically making all the choices below "Specify Property" available to the user. When the user indicates no further desire to specify properties, an exception is thrown in the "ContinueToConstraints" step, causing the "PropertiesIterator" step to complete, and ultimately causing "Choose Properties" to complete as well. The user is then allowed to specify constraints. This "done problem" is a commonly occurring concern with event based notification, since the completion of processing is an important event that often should be observed and responded to but frequently is not reported by components.

To keep the UI user-friendly, we wanted to allow even a novice user to go back and fix something done in a previous step. Our original process did not allow for this, forcing users to specify a system, properties, constraints, and run the analysis before allowing users to go back and reconsider what had originally been specified. In the modified version of our process, we allow users to go back and revisit anything that had already been done.

In general, we found Little-JIL to be an effective language for programming FLAVERS processes. Little-JIL allows a task to be specified at arbitrary levels of detail, which we found very useful. When less guidance was desired, less process detail was programmed. We also found the Little-JIL choice step to be particularly effective in describing where and how users are to be offered flexibility of choice at key points in our processes. Finally, Little-JIL provides a well designed API that allows external components to interact with the process interpreter easily, which facilitated the construction of a generator for these environments.

We also encountered a few problems in using this architecture and the generated binding managers. One problem that we encountered was that some FLAVERS UIs had process decisions hardcoded into their implementations. For example, some UI components were originally implemented to communicate directly with each other, making decisions about what actions could happen next. This greatly reduced the flexibility of the processes that could be defined. In our case, we were able to recode these UIs to remove this process information, but clearly this would not always be the case. We learned that it takes considerable design discipline to avoid "process creep."

A related issue is the level of granularity of process control. For this project, process control was via the access methods of the tool components. To get finer grain control would require invading the integrity of the tool components. On the other hand, the tool component interfaces might not be designed to be amenable to support process control. For our experiment with FLAVERS, however, interacting at the component level seemed appropriate and effective.

Once we removed the "burned-in" process in some of the FLAVERS UI components, creating environments was very easy. The FLAVERS intermediate and expert processes required approximately 230 lines of specification to control about 30 different UI functions with the process. The binding specification for the novice process, which is partially shown in Figure 5, is approximately 320 lines long. Moreover, it was easier to debug the binding specification than the original manually developed binding managers.

5 CONCLUSIONS

We believe that our case study supports our original hypothesis that process definition languages such as Little-JIL can indeed be used to define user-guidance processes, whose execution can create effective user-guidance environments.

More specifically, our experience in developing the different Little-JIL process definitions clearly supports our first subhypothesis that such languages can indeed be used to create environments that provide users guidance for complex toolsets such as FLAVERS. We used Little-JIL to define a canonical process for using FLAVERS. Our work with this canonical process led to subsequent definitions that seemed to provide guidance that was decreasingly distracting and increasingly effective.

Our experience also supported our second hypothesis, namely that it is possible to augment such process definitions with user interface binding specifications that can then be used as input to a generator to help construct process guidance

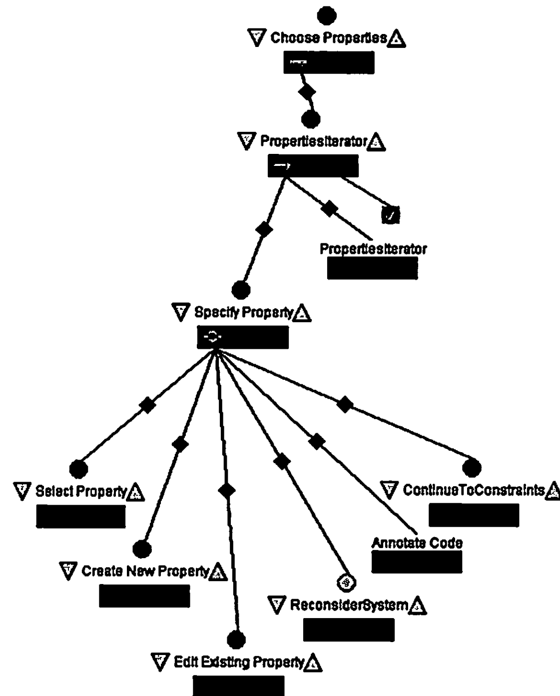

Figure 8 – Choosing Properties

environments. We demonstrated a language for defining such specifications and a generator that takes such specifications and creates interfaces to tool UIs. These then became key components of a user-guidance environment that consists of a user interface that exhibits behavior that is consistent with the specification that describes the relationship between the UI and the executing process.

Finally, our case study also supports our third subhypothesis, namely that we can achieve substantial flexibility, evolvability, and generality by separating concerns in the way dictated by our architecture. During our case study, we described several processes that modeled potential uses of the FLAVERS toolset. When different processes had the same basic steps and varied only in control-flow, it was possible to try several different processes without modifying the binding specification or the user interface. The user interface simply exhibited the control-flow behavior described by whatever process definition was used at a given time. When the steps in the process changed or varied, it was necessary to modify the binding specification and re-generate code from the new specification. However, this required no direct modification of the user-interface code by a programmer. Our experiences with expert users validate our claims of flexibility and evolvability, because it was so easy to adapt the system to expert system user requests for changes. We achieved generality via the separation of concerns that is proposed in our architecture. By separating out the process, the user interface, and the toolset, we were not bound to any particular type of instance of these three components, aside from the requirements that we have outlined earlier in this paper.

While achieving process flexibility and evolvability, we were also achieving evolvability and flexibility in the tools and the user interface. For instance, while we did not replace our UIs, we are confident that it is possible for the UI to be evolved, or even replaced, independent of the process definition. Similarly, as long as its API is identical, a new tool could replace an existing tool without requiring any modifications to the UI or the process.

There are several areas for future work. Our work with the FLAVERS process suggests applying our approach to wider classes of processes and toolsets. We have recently initiated a project to apply this approach to CVS configuration management system [8] using the jCVS user interface [1]. Configuration management seems particularly well suited to process control. Even with a tool, such as jCVS, it is relatively easy for programmers to violate agreed upon, but manual, configuration protocols. We expect that capturing these protocols with some well-designed executable processes that seamlessly guide the user via the user interface would be very valuable. Our initial work on this project seems very promising. To date we have defined one such process and the associated process-UI binding specification. It appears that the design of the jCVS toolset will be amenable to this approach but we intend to explore this further. We believe that this experiment will provide further evidence of the benefits of process guidance for helping users employ complex but important software tools.

The event-based notification within Java has many predefined event types, each with their own behavior. To date, we have implemented support for only two of these event types, ActionEvents and ChangeEvents. These were the only two that occurred in our two examples. This somewhat limits the flexibility of our binding managers, however, and we intend to add support for other event types and to examine general solutions that allow for handling arbitrary event types.

In developing our approach, we found it useful, especially during debugging or demonstrations, to be able to visualize the execution of the process steps. One could argue that the enabling and disabling of UI functionality provides a visualization of the process, but this is a very indirect, imprecise view. Since Little-JIL is a visual language, however, it seems relatively natural to superimpose execution visualization upon a visual process definition. We have an initial implementation of one such visualizer that seems very useful. We intend to explore other approaches to visualization and to determine if such visualization is useful to end users as well as to process designers.

ACKNOWLEDGEMENTS

Effort sponsored by the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0032 and by U.S. Department of Defense/Army under Contract DAAH01-00-C-R231. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, the U.S. Army or the U.S. Government."

REFERENCES

[1] "jCVS Home Page" Available online at http://www.jcvs.org/

[2] S. Arbaoui and F. Oquendo, "Managing Inconsistencies Between Process Enactment and Process Performance States," Proceeding of the Eighth Int'l Software Process Workshop, Wadern, Germany, 1993.

[3] S. Bandinelli and A. Fuggetta, "Computational Reflection in Software Process Modeling: the SLANG Approach," Fifteenth International Conference on Software Engineering, IEEE Computer Society Press. pp. 144-154, Baltimore, MD, 1993.

[4] S. Bandinelli, A. Fuggetta, and S. Grigolli, "Process Modeling in-the-large with SLANG," Second International Conference on the Software Process, IEEE Computer Society Press. pp. 75-83, Berlin

Germany, 1993.

[5] M. Beaudouin-Lafon, "User Interface Support for the Integration of Software Tools: An Iconic Model of Interaction," Third ACM SIGSOFT Symposium on Software Development Environments, pp. 143-153, 1988.

[6] M. Bever, K. Geihs, L. Heuser, M. Muhlhauser, and A. Scholl , "Distributed Systems OSF DCE and Beyond," in *DCE --- The OSF Distributed Computing Environment*, vol. 731, *Lecture Notes in Computer Science*, A. Schill, Ed. New York: Springer-Verlag, 1993.

[7] J. Buxton, "Requirements for Ada Programming Support Environments," : Department of Defense, 1980.

[8] P. Cederqvist, et al, "Version Management with CVS," : Signum Support AB, 1993.

[9] G. Cugola, "Tolerating Deviations in Process Support Systems via Flexible Enactment of Process Models," *IEEE Transactions on Software Engineering*, vol. 24, pp. 982-1001, 1998.

[10] E. Di Nitto and A. Fuggetta, "Open Issues In Managing Inconsistencies in Human-Centered Systems," Proceedings of the International Conference on Software Engineering (ICSE) '97, Workshop on Living with Inconsistency, Boston, MA, 1997.

[11] M. B. Dwyer and L. A. Clarke, "Data Flow Analysis for Verifying Properties of Concurrent Programs," Second ACM SIGSOFT Symposium on Foundations of Software Engineering, ACM SIGSOFT'94 Software Engineering Notes, Vol 19, ACM Press. pp. 62-75, New Orleans LA, 1994.

[12] W. Emmerich and V. Gruhn, "FUNSOFT Nets: A Petri-Net based Software Process Modeling Language," Thirteenthth International Conference on Software Engineering, pp. 175-184, 1991.

[13] L. M. Encarnacao and S. Stoev, "An Application-Independent Intelligent User Support System exploiting Action-sequence based on User Modelling," User Modeling: Proceedings of the Seventh International Conference, Springer Wien, New York. pp. 245-254, Banff, Canada, 1999.

[14] C. Fernstrom, "PROCESS WEAVER: Adding Process Support to UNIX," Second International Conference on the Software Process, pp. 12-26, 1993.

[15] J. Gosling, B. Joy, and G. Steele, *The Java(TM) Language Specification*: Addison-Wesley, 1996.

[16] V. Gruhn and R. Jegelka, "An Evaluation of FUNSOFT Nets," Second Eurpoean Workshop on Software Process Technology, Trondheim, Norway, 1992.

[17] R. Kadia, "Issues Encountered in Building a Flexible Software Development Environment: Lessons from the Arcadia Project," Fifth ACM SIGSOFT Symposium on Software Development Environment, pp. 169-180, Tyson's Corner , VA, 1992.

[18] G. E. Kaiser, P. H. Feiler, and S. S. Popovich, "Intelligent Assistance for Software Development and Maintenance," *IEEE Software*, vol. 5, pp. 40-49, 1988.

[19] F. Linton, D. Joy, and H.-P. Schaefer, "Building User and Expert Models by Long-Term Observation of Application Usage," User Modeling: Proceedings of the Seventh International Conference, Springer Wien, New York. pp. 129-138, Banff, Canada, 1999.

[20] M. J. Maybee, D. M. Heimbigner, and L. J. Osterweil, "Multilanguage Interoperability in Distributed Systems," 18th International Conference on Software Engineering, pp. 451-463, Berlin Germany, 1996.

[21] MicroSoft, "OLE 2 Programmer's Reference," MicroSoft, Manual 1994.

[22] OMG, *CORBA 2.0/Interoperability*, vol. OMG TC Document 95.3.xx, Revised 1.8 ed. Framingham MA: Object Management Group, 1995.

[23] OSF, "OSF DCE Application Development Guide Revision 1.0," Open Software Foundation, Manual 1993.

[24] L. J. Osterweil, "A Process-Object Centered View of Software Environment Architecture," Advanced Programming Environments, Lecture Notes in C.S., Vol 244, Springer-Verlag. pp. 156-174, Trondheim, 1986.

[25] S. P. Reiss, "Connecting Tools Using Message Passing in the FIELD Environment," *IEEE Software*, vol. 7, pp. 57-67, 1990.

[26] S. P. Reiss, *The Field Programming Environment: A Friendly Integrated Environment for Learning and Development*. Boston, MA: Kluwer Academic Publishers, 1996.

[27] S. M. Sutton Jr., D. Heimbigner, and L. J. Osterweil, "APPL/A: A Language for Software-Process Programming," *ACM Transactions on Software Engineering and Methodology*, vol. 4, pp. 221-286, 1995.

[28] M. Suzuki and T. Katayama, "Meta-Operations in the Process Model HFSP for the Dynamics and Flexibility of Software Processes," First International Conference on the Software Process, IEEE Computer Society Press. pp. 202-217, Redondo Beach, California, 1991.

[29] L. Thery, Y. Bertot, and G. Kahn, "Real Theorem Provers Deserve Real User-Interfaces," Fifth ACM SIGSOFT Symposium on Software Development Environments, pp. 120-129, 1992.

[30] A. I. Wasserman, "Tool Integration in Software Engineering Environments," Software Engineering Environments, pp. 137-149, France 1989, 1990.

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

[32] A. Wise, B. S. Lerner, E. K. McCall, L. J. Osterweil, and J. Stanley M. Sutton, "Specifying Coordination in Processes Using Little-JIL," Department of Computer Science, University of Massachusetts at Amherst, Technical Report 99-71, Revised November 1999.