Patterns for Orchestration Environments*

Dragos A. Manolescu
mailto:dragos@thoughtworks.com

Draft of July 26, 2004

Introduction

Service oriented architecture (SOA) is an emerging architectural style. SOA builds on insights gained from previous attempts at building systems out of distributed components, using the pull of the Web to succeed where these failed.

Service orientation revolves around orchestration. Orchestration deals with the coordination of services comprising SOAs. It provides the means for aggregating services into applications. Orchestration is not a panacea. It involves tradeoffs. As it typically happens, orchestration solves some problems while adding its own problems.

Although the fundamental ideas of orchestration are several decades old, orchestration represents uncharted territory for many people. This collection of patterns aims at shedding light over the problems you have to deal with when you use orchestration in your enterprise architecture.

The patterns target people who would like to learn about orchestration, people who are evaluating orchestration-related products, and people who are building orchestration-related middleware. In effect, the patterns aim at educating users, helping them make decisions, and communicating with others.

1 Orchestration Engine

Aliases

Process engine, workflow engine, orchestration server

Context

Imagine that you are designing a large business application that handles mortgage applications. The business process takes a mortgage application as its input. Data elements provide information about

the applicant (name, current address, taxpayer identification number, income, etc.), the property address, the product the applicant is interested in (e.g., 15-year fixed rate mortgage), and so on. The process starts by obtaining a credit report from a credit vendor (i.e., a process actor). Should the applicant's credit score be above a given threshold, the application process continues by ordering an appraisal and a flood certification from other specialized vendors.

Once all this information is available the business process fires off a decision engine (another process actor) that assesses the risk and approves or rejects the application. Risk assessment entails running the application data through a risk model developed by the bank. The model measures the likelihood of repayment of the loan, as well as how easy it will be to sell it to one of the investors the bank works with (its salability).

The business process involves several process actors, such as services (e.g., credit reporting, appraisal, flood certification) or components (e.g., the decision engine). The process invokes these actors at the appropriate time, with the appropriate data. Often times this logic is implemented with glue code that handles invocations and data flow between actors.

Changing the business process translates into changing the glue code. How feasible is it to rewrite the glue code each time the business analyst changes the process? In addition, what happens if the customer wants to query the status of their mortgage application? Answering such queries requires implementing more than glue code. They require that you also deal with instrumentation, querying, and reporting. Suddenly your glue code experiment is much more complex than you initially thought.

Problem

How do you implement a business process such that you can easily change the activities it entails, the data that flows into and out of the process actors, as well as answer queries about their execution?

Forces

Providing a solution to the above problem requires balancing the following forces:

- Business process changes typically translate into changing the process actors, the information supplied to them, and their sequencing. For example, streamlining a loan application process changes the order of the component invocations and the data passed to them. However, generally it doesn't require the credit reporting agency to perform the credit check in a different way.
- The actors within a business process are unaware of the global business context. As long as their pre-requisites are satisfied, the actors don't know what happened before them, nor do they know what invocations follow afterward.
- Analysts want to be able to modify business processes quickly. This agility is critical to adapting to changing business environments or staying ahead of the competition.

- Users are interested in various operational aspects of their business processes. For example, they want to know how many processes are executing, or what is the status of a particular process instance.

Solution

Factor the "coordination aspect" out of the glue code and hand it to an orchestration engine. Factoring out entails finding the glue code that implements the business process (typically a difficult task). An orchestration engine (i.e., a component specialized in dealing with the coordination aspect) deals solely with the coordination aspect, delegating business processing to the process actors.

Composition through glue code typically mixes the coordination aspect with business functionality. The code dealing with coordination ends up scattered among various components. For example, the mortgage application system could have coordination logic in the front end, the decision engine, and the components that communicate with the external vendors.

Using an orchestration engine prevents this problem. The process definition gathers service composition in a single place. The separation of concerns imposed by the orchestration engine improves maintainability, testability, and modifiability. It also facilitates answering questions about the executing processes. The runtime information resides in the orchestration engine; it is no longer distributed throughout the entire application.

Factoring out the coordination aspect has operational consequences as well. The orchestration engine handles errors reported by the participating services. This allows it to treat in a uniform way errors such as not being able to reach a process actor.

Typically applications dealing with business processes have to manage the process state and the long-running transactions associated with the process. Dealing with these aspects is tedious and error-prone. An orchestration engine frees developers from dealing with these chores.

Resulting Context

Orchestration Engine brings along the following benefits and liabilities:

Benefits:

Improved modifiability Changing the business process translates into changing the process definition. Business analysts can change the process themselves. In other words, using an orchestration engine improves the application's agility.

Simpler application code The application code becomes simpler. Developers no longer write glue code. In addition, they no longer implement the operational functionality required to answer queries about the process' state, etc.

Improved reusability The components and services comprising the application make fewer assumptions about the control context where they're operating. Consequently they're more reusable.

Domain independence The orchestration engine deals only with control- and data-flow. You can use it to orchestrate business processes from any application domain (e.g., financial, insurance, health care, telecommunications, etc.).

Liabilities:

Potential architectural mismatch An orchestration engine coordinates actors that make few assumptions about the control context where they operate. Unfortunately many software components in general and services in particular make conflicting assumptions about the locus of control. In fact, most components assume that they're in control.

Increased implementation complexity Orchestration engines solve problems at the intersection of programming languages, transaction management, component software, messaging, and so on. Building an orchestration engine entails tackling problems from all these domains. This increases the implementation complexity [4]. Consequently most people buy rather than build orchestration engines.

Uncertain process definition language Using an orchestration engine requires a process definition language for defining the coordination aspect. The agreement on a process definition language is critical to the acceptance of an orchestration engine and the interoperability of multiple engines.

Known Uses

- Microsoft BizTalk Server 2004 (BTS) implements an orchestration engine. The engine compiles XLANG schedules (based on a process algebra known as Pi-calculus [3]) into .NET assemblies. The Microsoft Intermediate Language (MSIL) then executes within the context of the BizTalk Server Execution Engine. In effect, the execution engine acts as an application server for XLANG schedules.
- FiveSight's implementation of this pattern in their Process Execution Engine (PXE) also revolves around Pi-calculus. PXE users define their orchestrations in BPEL or another process markup language (PML). PXE reduces the PML into a process algebra and then executes the process by reduction in a reliable virtual machine.
- TIBCO IntegrationManager.
- Intalio | n3 Server.

Related Patterns

- The *Interpreter* pattern [2] allows developers to solve particular problems by first defining a language for the problem and then interpreting it. You can regard *Orchestration Engine* as an *Interpreter* specialized in actor coordination

2 Orchestration Language

Aliases

Process definition language, control structures, activity network

Context

You are using an *Orchestration Engine* for orchestrating your business processes. The engine is generic; it provides the mechanisms for coordinating the actors participating in the business process but has no intrinsic knowledge about what behavior each actor encapsulates. You must tell the engine how to coordinate the actors. Specifically, you must specify:

- the order in which it should transfer control to (i.e., invoke) each actor,
- what data to pass into and retrieve from each actor, and
- how to handle exceptions

For example, consider a business process that quotes insurance premiums for homeowners. The process uses customer data such as name, address, SSN, property type, and so on. It begins with invoking a credit check service. If the credit service reports an acceptable credit rating, the process gathers several rating factors involved in the premium computation. For instance, a rate territory service provides a rating factor that reflects the location, such as proximity to fire stations, large body of waters, etc. Another service provides a rating factor that reflects the risk of damage due to various perils, such as fires, winds, floods, etc. The premium computation uses these factors as well as the data supplied by the customer.

Implementing the quoting process with an orchestration engine makes coordinating the actors supplying the various factors a snap. You are no longer using ad-hoc code to invoke and pass data between them. Instead, you specify how the coordination should happen and the engine takes care of it all.

Problem

How do you program the orchestration engine with the business process?

Forces

Providing a solution to the above problem requires balancing the following forces:

- You need a means for unambiguously defining the coordination of the actors involved in the business process.

- Orchestrations involve different abstractions than the ones provided by traditional programming languages such as Java or Ruby. While the former are concerned with actors and coordination, the latter deal with generic concepts such as variables and objects. However, having access to constructs such as variables, objects and messages from orchestrations increases their expressiveness.
- A small set of simple control structures (sequence, repetition, conditional, fork, and join) can represent a wide range of coordination behaviors.
- Some complex control flow constructs (particularly those involving concurrency) are hard or impossible to represent just with simple control structures.
- The people who work with business processes are non-technical users. While some tinker with programming languages and development tools, from their perspective these are too complex for routine use. Typically they prefer to draw orchestrations rather than describe them in specialized languages.

Solution

Use a specialized language to define how the orchestration engine should coordinate the entities participating in the business process. The *Orchestration Language* departs from general-purpose programming languages, focusing on the abstractions required to describe the coordination aspect. The engine transfers control to and data between the business process actors according to this definition.

At minimum the language must provide a means for transferring control to a business process actor and receiving data from it. Activity execution represents a point where the control flow crosses the boundary between the orchestration and application realms. Receiving data from an actor allows the engine to bring information into the orchestration. Let's call this a Primitive.

However, business processes involve more than a single interaction with one actor. Therefore an Orchestration Language must also provide a means of composing Primitives, thus allowing its users to define more realistic business processes.

Sequencing is the simplest form of composition. A Sequence defines a temporal ordering between two or more interactions with actors. In other words, it enables users to specify orchestrations such as "obtain a credit score before ordering an appraisal."

Sequence implements a linear flow of control; the language needs a way to break this linearity. A Conditional activity provides another way of composing activities. It has two children activities and passes control only to one of them, depending on how a test condition evaluates. This enables users to specify orchestrations such as "if the credit score is greater than 700 then grant skip the pay bonus verification."

Yet another form of composition typical of business processes is repetition. A While activity provides a way of repeating one or several steps. Like the Conditional, a While has a condition that determines the flow of control. The orchestration engine evaluates the condition and executes the

While's body only if the condition be true. The engine repeats this process until the condition is no longer satisfied.

Fork and Join add support for concurrency and synchronization. For example Fork lets users specify things like "this actor does this and that actor does that concurrently." A Join provides a synchronization point; OR-, XOR- and AND-Joins synchronize the incoming concurrent orchestrations in different way. For example, an OR-Join continues as soon as the control flow from any one of the incoming orchestrations reaches it. In contrast, an AND-Join waits for all the concurrent orchestrations to reach it before it continues execution.

The above control structures represent a basic set. Many other possibilities exist and various orchestration languages implement different constructs [5]. However, the fundamental idea is the same: provide a set of abstractions that allows people to define the coordination aspect.

A key assumption in the above discussion is that the various control structures are interchangeable. In other words, if you consider the orchestration language comprised of activities, then the grammar would look like the following:

```
activity := Primitive|Sequence|While|Fork|Join
Sequence := Primitive*
While := Condition activity activity
Fork := activity *
Join :=  activity *
Condition := expression evaluating to a Boolean
```

Resulting Context

Benefits:

Improved reusability The specialized language makes the engine generic. You can use it to implement a wide range of orchestrations.

Liabilities:

High complexity Developers can read code. Most business users cannot, and will be turned off by the syntactic noise.

Impedance mismatch Some compositions cannot be easily described in terms of the control flow. For example, the control flow of an application where the user can at any time perform one of many possible actions (e.g., adding an item to the cart, logging out, updating the payment information, etc.) develops dynamically and thus is cumbersome to prescribe with control structures.

Insufficient expressivity Typically the definition language users need control structures that the language doesn't provide. This is a problem if the language prevents them from synthesizing these structures out of existing ones.

Known Uses

- Microsoft BizTalk designer provides many control structures, including action, decision, while, fork, and join. BizTalk translates the XLANG schedule into MSIL, which then executes within the execution engine.
- The BPEL 1.1 specification [1] provides several types of activities. Basic activities deal with invoking and providing web service operations, signaling faults, or doing nothing (NOP). Structured activities deal with "ordinary sequential control" (sequence, switch, and while), concurrency and synchronization (flow), and non-deterministic choice (pick).

Variants

Micro-workflow is an open-source lightweight workflow engine. Micro-workflow targets software developers. It doesn't need to insulate them from the programming language. Therefore micro-workflow doesn't provide an orchestration language. Instead, developers define orchestrations through instantiating and connecting objects that represent the process definition. In other words, they create the abstract syntax tree directly.

Related Patterns

- Wil van der Aalst has collected a set of workflow modeling patterns [5]. The patterns focus on control structures and group them in 6 classes: basic control, advanced branching, structural, multiple instances, state-based, and cancellation.

3 Orchestration Builder

Aliases

Process editor

Context

You are using an orchestration engine. Users specify the coordination aspect through the orchestration language. Typical business users find the syntactic noise and syntax of the definition language too cumbersome.

Problem

How can non-technical users get involved with creating and modifying business processes without having to become proficient with the orchestration language?

Forces

Providing a solution to the above problem requires balancing the following forces:

- The availability of orchstration languages does not eliminate programming. People who define orchestrations in orchestration languages are still programming, albeit not with generic programming languages.
- Doodleware aims at hiding complexity behind graphical tools. However, without a way that provides direct access to the underlying programming model it inevitably gets in the way when you're trying to do things that its designers haven't thought about.

Solution

Create a builder tool that provides a level of abstraction above the orchestration language. The builder allows non-technical users to define the coordination aspect through combining graphical representations of the elements of the orchestration language. Figure 1 shows an example of an orchestration builder, the Oracle BPEL designer. Users define orchestrations by dragging activities from the BPEL palette (right side) and connecting them in the process map (left side).

Resulting Context

Benefits:

Lower interface complexity Typically the builders allow non-technical users to define orchestrations graphically, through dragging, connecting, and configuring graphical elements.

Liabilities:

Increase the implementation complexity Building the development environment is a more complex production. In addition to the orchestration engine, the development team must also design the graphical builder. Building these components involves dealing with different concerns and requires different skills.

Known Uses

- Microsoft BizTalk server 2004
- Oracle BPEL server

Variants

Instead of developing a stand-alone orchestration builder, extend an existing IDE. Collaxa developed a BPEL designer plugin for the Eclipse IDE. Likewise, Microsoft developed the BizTalk 2004 development environment inside the Visual Studio .NET IDE.

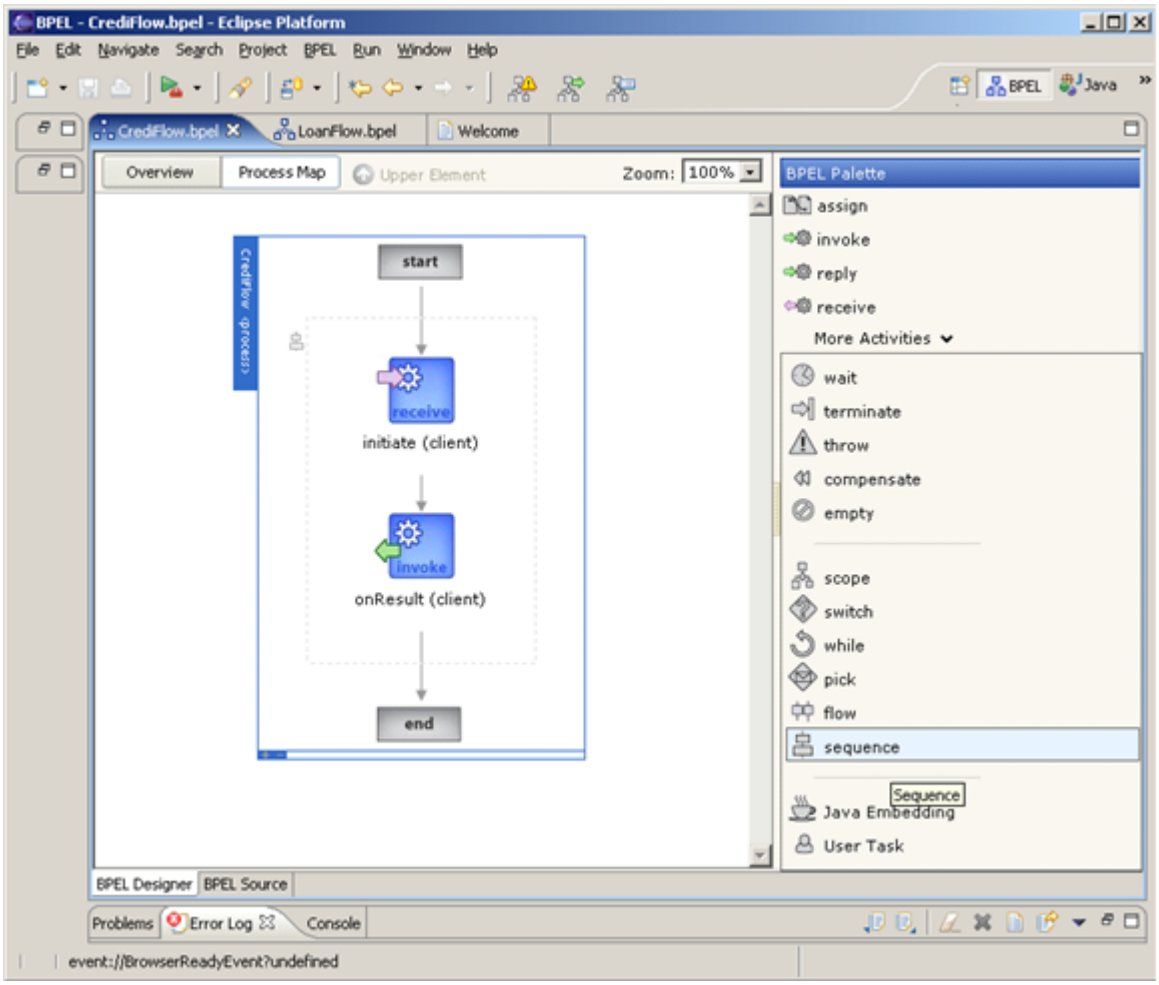

Figure 1: Oracle BPEL Designer

Related Patterns

- *Orchestration Engine*

4 Evaluator

Context

You are using an *Orchestration Engine.* Test conditions are associated with control structures of the *Orchestration Language* (such as conditional and repetition).

Problem

How does the orchestration deal with test conditions?

Forces

Providing a solution to the above problem requires balancing the following forces:

- Orchestration definitions can involve a wide range of test conditions, ranging from simple arithmetic through function calls through table lookups
- You need an unambiguous way of expressing the test conditions
- The constructs of the implementing programming language are typically too complex for expressing the test conditions
- Using the implementing programming language for the test conditions may require recompilation
- Defining a language is hard; learning a new language is demanding

Solution

Define a specialized language for defining test conditions. Delegate their processing to an evaluator, i.e., a component specialized in evaluating conditions.

Resulting Context

Benefits:

Distance from general-purpose programming languages Orchestration engine users can express test conditions with less syntactic noise than what it would take when using a general-purpose programming language.

Liabilities:

Increased complexity The users of the orchestration engine must learn the condition specification language as well as how to debug them. Developers must build the evaluator, or integrate the orchestration engine with an existing one.

Variants

The complexity of this pattern can vary widely. At one end of the spectrum a simple evaluator implements a few operations, such as logical expression. At the other end of the spectrum, a full-fledged rule engine allows users to define complex rule sets and execute multi-step rule engine cycles.

Known Uses

- BPEL uses XPath expressions
- BizTalk 2004 rules

5 Rule Builder

Context

You are using an *Orchestration Engine*. Non-technical users have difficulties writing conditions in the specialized language available for this purpose.

Problem

How can non-technical users get involved with conditions without having to master the intricacies of the definition language?

Forces

Providing a solution to the above problem requires balancing the following forces:

- The availability of rules languages does not eliminate programming. People who define rules with specialized languages are still programming, albeit not with generic programming languages.
- Doodleware aims at hiding complexity behind graphical tools. However, inevitably it gets in the way when you're trying to do things that its designers haven't thought about.

Solution

Create a builder tool that provides a level of abstraction above the rules specification language.

Example

Resulting Context

Benefits:

Lower the complexity of defining rules Typically the rule builder allows non-technical users to define and compose rules graphically.

Liabilities:

Increase the tool complexity Users have to learn a new tool rather than a new language. Developers have to build orthogonal functionality.

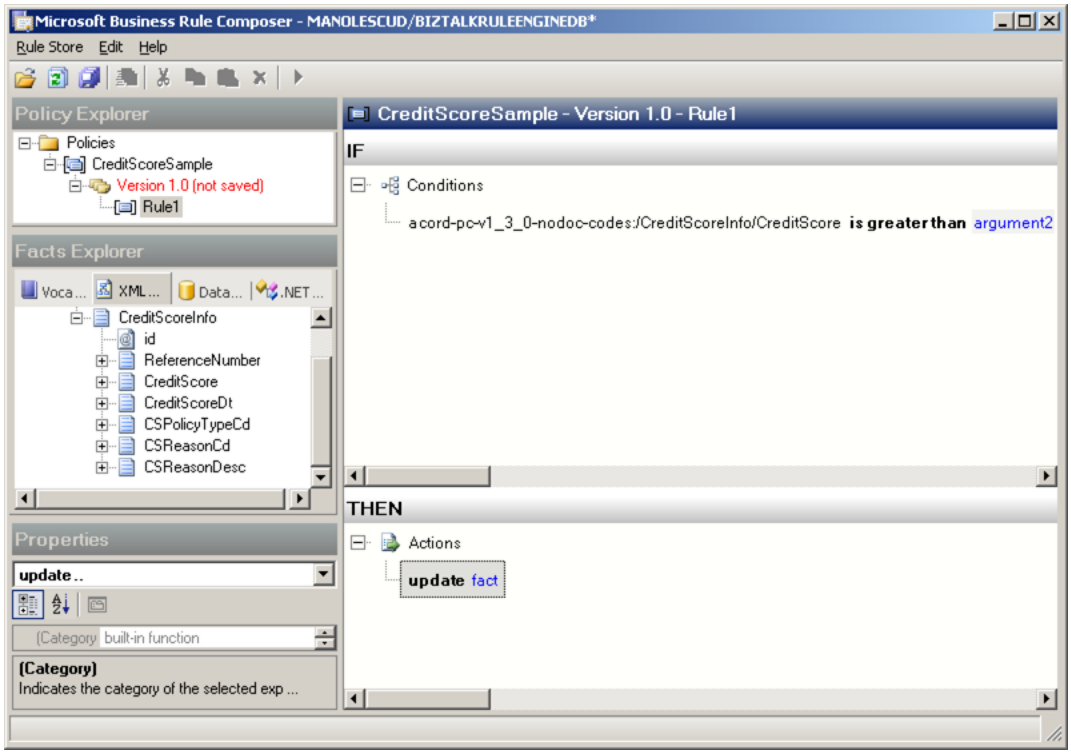

Figure 2: Microsoft Business Rules Composer

Known Uses

- Microsoft Business Rules Composer for BizTalk Server 2004

6 Orchestration Context

Context

You are using an *Orchestration Engine* to orchestrate your business processes.

For example, consider a process that decisions a loan application. The process involves several activities, each of which refines the decisioning result. It begins with a credit decisioning activity that uses a credit model to make a decision. If credit decisioning passes, the process proceeds to a product evaluation activity, followed by salability and finally pricing activities.

Problem

How does an orchestration pass the outputs from each activity to the next, i.e., how does the data flow in the orchestration?

Forces

Providing a solution to the above problem requires balancing the following forces:

- Some activities only need data provided by the requester. Other activities need information produced by the process.
- Some outputs are used by other activities; other outputs comprise the output of the process
- Need knowledge about the downstream activities in order to discard

Solution

The *Orchestration Engine* automatically creates a context for each orchestration, and maintains it throughout the orchestration's lifetime. The context carries orchestration relevant information . The engine uses this context to carry information between the actors and store other orchestration-relevant data.

Example

BPEL scope

Resulting Context

Benefits:

Key to data flow Cannot orchestrate with control flow only

Improve security Localize sensitive information

Liabilities:

Increase interface complexity Special constructs that define the scope

Increase implementation complexity Persist the context

Known Uses

- BPEL

Related Patterns

- *Orchestration Engine*
- *Orchestration Language*

7 Conditional Transitions

Context

You are using an *Orchestration Engine*. You described your orchestration in the *Orchestration Language*. The definition specifies temporal dependencies between the orchestration's steps and uses specialized control structures to directly alter the control flow. Sometimes the transition from one activity to another depends on a condition. Using conditionals to express these dependencies clutters the orchestration definition, making it hard to understand the control flow.

Problem

How do you express activity transitions that depend on a condition without polluting the control flow?

Forces

Providing a solution to the above problem requires balancing the following forces:

- Most definition languages provide structures that alter the flow of control based on conditions
- Expressing transitions between activities with control flow structures makes the definition hard to understand
- State transitions represent transitions with conditions between states

Solution

Augment the activities with conditions that determine when they start execution.

The orchestration engine no longer transfers control from one activity to another as soon as the execution completes. Instead, once the execution of A completes the engine evaluates the condition associated with the transition from A to B and waits until the condition is satisfied.

For example, assume a sequence with three steps. Each step invokes an actor, and each step must wait until a condition is satisfied. A system using solely control structures must represent each step with a loop that guards the service invocation. In effect, the loop suspends execution until its condition is satisfied. In contrast, a system supporting conditional links no longer needs guarding

loops. The steps of the sequence are simple invocations. Instead of residing on control-flow altering structures, the conditions reside on the transitions between the steps.

This mechanism could easily handle activities that do not need conditional transitions. The default transition has a condition that is always satisfied. In other words, if you don't a conditional transition.

Resulting Context

Benefits:

Simple orchestrations Using conditional transitions leaves the orchstration definition simpler. You no longer have to rely on specialized control flow.

Increased expressiveness Control structures with conditional transitions provide a hybrid approach between activity netwroks and state transitions. The hybrid approach can express a wide range of control flows and dependencies.

Liabilities:

Increased implementation complexity Conditional transitions require evaluating conditions. Triggering evaluations and evaluating conditions in a scalable way are hard problems.

Potential performance penalty The conditional evaluation mechanism may incur overhead for transitions that do not have conditions associated with them.

Known Uses

- TriGSFlow (Trigger System for GemStone)
- The micro-workflow framework represents conditional transitions as Event-Condition-Action (ECA) rules. The event corresponds to completing the execution of the preceding activity, and the action corresponds to the execution of the current activity.

Related Patterns

- *Orchestration Engine*
- *Orchestration Language*

8 Engine Monitoring and Control

Context

You are using an engine to orchestrate services. The orchestration engine could tell how many orchestrations it has executed, how many are running, how long they take to complete, and so on.

Engine-level orchestration information is valuable to business users. For example, the number of executed orchestrations over a given period of time provides them with insight into the popularity of a business process. Likewise, the time elapsed between an orchestration's steps helps them identify potential bottlenecks.

Engine-level orchestration information also helps the administrators who maintain the system. For example, administragtors could monitor the number of running orchestrations and perform software and/or hardware maintenance when the impact is minimum (e.g., no orchestrations are executing).

Problem

How do users obtain information about executing orchestrations and control their execution?

Forces

Providing a solution to the above problem requires balancing the following forces:

- The orchestration engine is in a unique position to provide information about and control the orchestrations it executes
- Business users and administrators want to be able to abort, stop, pause, and resume orchestration execution
- An orchestration engine designed without instrumentation in mind cannot easily provide orchestration information
- Engine-level orchestration information provides useful metrics for business users and administrators
- Capturing information about the running orchestrations intereferes with their execution

Solution

Design the engine with instrumentation and control points. Instrumentation points let you capture engine-level orchestration information. Control points let you control orchestration execution.

Designing instrumentation points requires that you first decide the information you want to capture. You then find the appropriate places in the orchsetration engine and implement data capture hooks. The hooks should allow users to customize the data capture mechanism or even provide

new implementations. Ideally users should be able to control what information they want captured by selecting the instrumentation points, as well as how this information is captured by customizing the data capture hooks.

For example, consider an instrumentation point providing access to code that executes each time the engine fires off an activity. Let's assume that the default implementation records this information to a file. A hook designed with modifiability in mind allows users to hook in a custom implementation that augments the information with a timestamp and then writes it to a database.

You can draw an analogy between an orchestration engine and a virtual machine. A virtual machine can tell you how many class instances it has created, how many instances it has garbage-collected, how large the heap and stack are, and so on. The orchestration engine deals with orchestrations rather than classes. Therefore it could answer similar questions about orchestrations.

In addition to instrumentation points you could also implement control points in the engine. These points provide a programmatic way to control the orchestration engine, such as abort execution of an orchestration, stop the engine, resume the engine, checkpoint the engine state, and so on. Business users and administrators exercising the control points need a way of interacting with the engine. You must either expose the control points as an API or build tools that allow people to exercise them.

Example

Oracle BPEL server has a web-based console. The console provides several views inside the orchestration engine. Users can visualize the executing as well as executed (i.e., completed) orchestration instances, orchestration definitions (BPEL and WSDL), audit trails, orchestration activities, and so on. Figure 3 shows the instance view and figure 4 shows the audit trail corresponding to an orchestration instance.

Resulting Context

Benefits:

Simplifies development Implementing engine monitoring and control is a one time activity. Once the engine has these features all orchestrations it executes can benefit from them without programming or any other additional work.

Provides data for CPI Instrumentation data is the critical ingredient to improving the processes corresponding to the orchestrations.

Improves maintainability The ability to perform controlled shutdowns and startups facilitates the administrators' work, thus improving the maintainability of the system.

Liabilities:

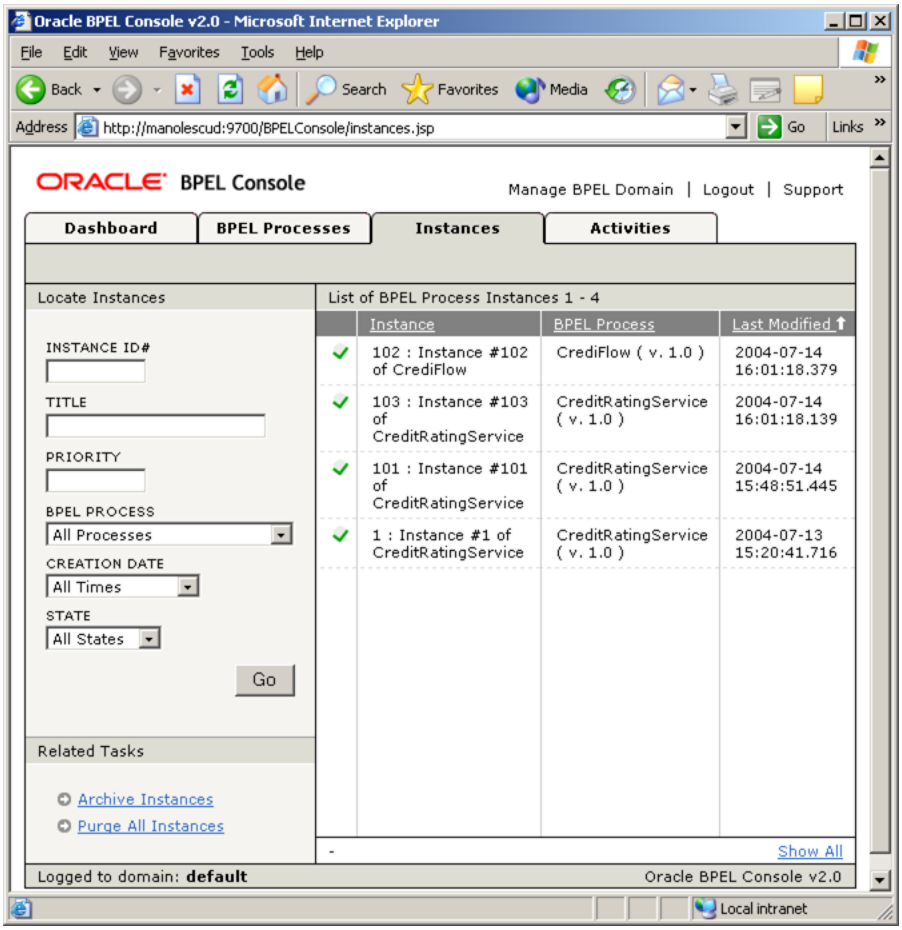

Figure 3: Oracle BPEL console showing orchestration instances

Increases implementation complexity Implementing instrumentation and control points complicates the orchestration engine. In addition, building tools for accessing the orchestration data typically further increases the complexity.

Lowers performance Instrumentation code incurs additional overhead. This lowers performance of the engine. Typically the overhead is not a problem since orchestrations have time scales orders of magnitude larger than instrumentation code.

Increases the maintenance effort for the collected data The engine could generate large volumes of instrumentation information. You need to store it in a repository (file system, database, and so on). More importantly, you need to be able to sift through it and quickly find the pieces you're interested in.

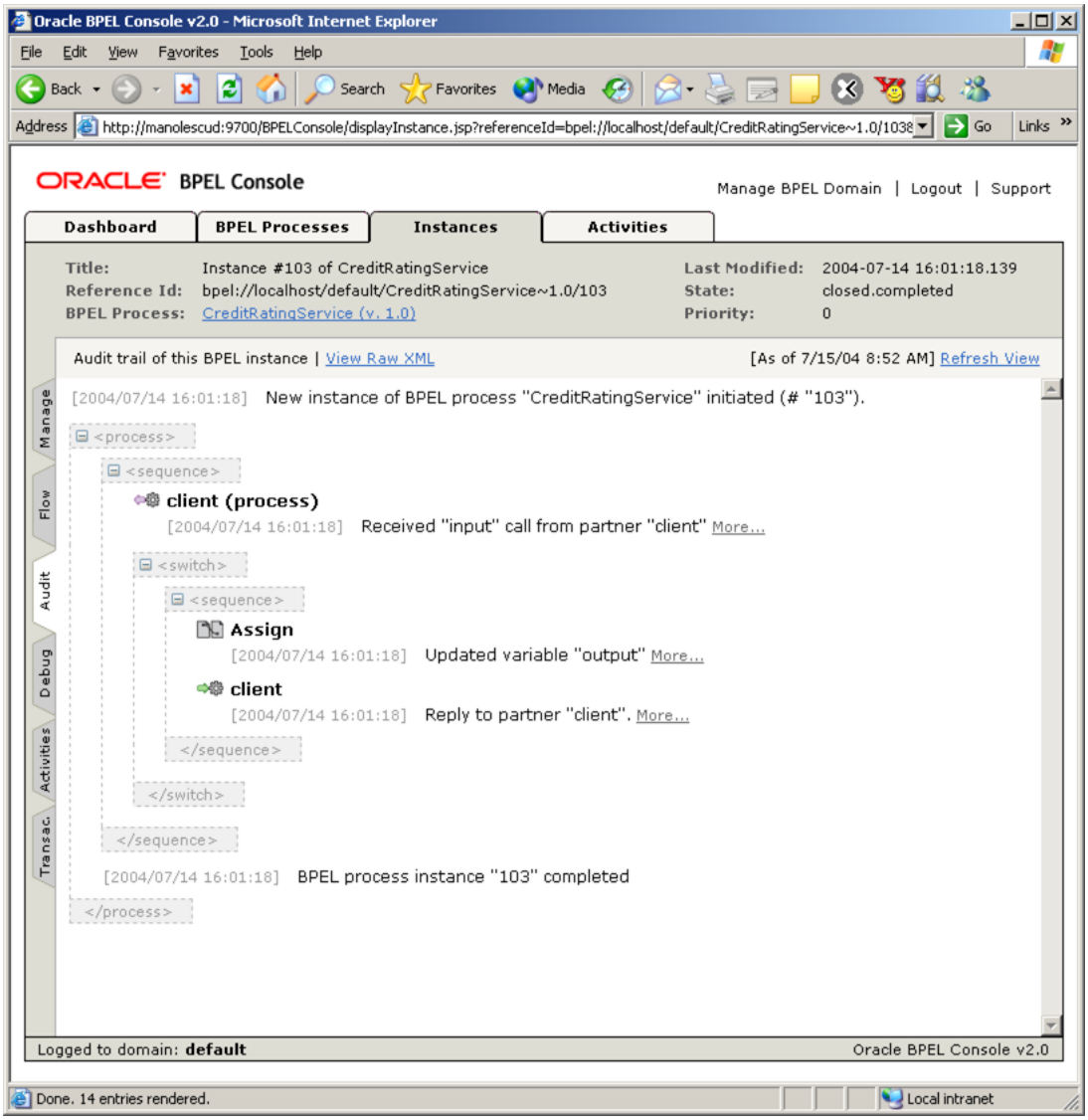

Figure 4: Oracle BPEL console showing the audit trail for an orchestration instance

Provides a limited set business metrics The engine doesn't have the context required to understand the semantics of the orchestrations that it executes. In other words, the engine can tell you how many instances of an orchestration it created, or how long they took to execute. However, the engine cannot answer questions that require additional context.

Known Uses

- BizTalk Server 2004 records in a database information about the documents it processes. The Health and Activity Tracking (HAT) tool, a specialized tracking tool uses this data to provide information about messages and orchestrations. BAM provides business information from instrumentation hooks embedded within orchestrations.
- Oracle BPEL server

Related Patterns

- *Orchestration Engine*

9 Compensating Action

Context

You are using an orchestration engine to orchestrate actors. At run time you need to cancel some orchestration activities. Typically this follows the discovery of invalid input data, changes in the environment, or user abort. The work affected by these discoveries needs to be undone.

Consider a travel web service. The service employs several actors that provide specialized services, such as air travel, hotel reservations, car rentals, credit card billing, and marketing. Based on the customer's itinerary and preferences, an orchestration first uses the air, hotel, and car rental services to assemble a set of itineraries. Once the customer selects an itinerary, the orchestration invokes the credit card billing service. Finally, the orchestration invokes the marketing service if the travel agent has agreements with the selected providers (airline, lodging facility, and car rental company). Assume that just after the billing service has been invoked, the customer realizes that he has made travel arrangements for the wrong month. He cancels the process and corrects the date. However, the travel web service must undo the completed reservation process. This entails removing the credit card charge and canceling the reservations.

Problem

How do you cancel the effects of selected actions within an orchestration?

Forces

Providing a solution to the above problem requires balancing the following forces:

- Canceling activities of an orchestration may require undoing the effects of actors' work
- The undo may span several actors, from one to several
- Not all actors are transactional. For example, an actor that issues a check to a customer cannot be rolled back
- Undoing an invocation is actor-dependent

Solution

Supplement the orchestration definition with compensating actions. In other words, defining an orchestration involves specifying compensating invocations, as well as composing the services that achieve the goal of the orchestration.

For example, consider an actor that writes checks. Adding it to an orchestration definition requires instructing the engine how to undo issuing a check. As long as check writing is in process the check writing actor could abort check printing. Alternatively, if the check is in the mail a different actor could cancel it and issue an explanation letter.

Users defining the orchestration must choose the granularity of the compensating action. Each orchestration activity may have an inverse that cancels its effects. Cancelling a sequence of 3 activities translates into invoking the corresponding compensating actions, in reverse order (3rd , 2nd, and 1st). However, this granularity may be too fine. The effect of several invocations may be undone with a single compensating action. Typically adjacent activities form logical groups. Instead of a compensating action for each activity you could have a compensating action for each group of activities. The larger granularity reduces the number of compensating actions. Note, however, that typically you cannot group together transactional and non-transactional serevice invocations.

The orchestration engine can implement the compensating actions in different ways. For transactional invocations the compensating action involves the rollback of the group. If this is the case the definition must instruct the orchestration engine how to trigger a rollback. However, not all activities are transactional. For example, issuing a refund check or mailing out a policy ammendment cannot be rolled back. To undo the effects of a non-transactional activity the orchestration engine executes a semantically compensating action.

Resulting Context

Benefits:

Improved consistency Compensating service invocations improves the consistency. Aborting an invocation cancels its effects.

Liabilities:

Increases interface complexity The orchestration definition becomes more verbose. In addition to the normal flow, the definition also contains the flow required to undo its effects.

Increases implementation complexity Users must define the logical groups. Developers must implement transaction rollback or semantically compensating services.

Known Uses

- BPEL uses `<scope>` to group service invocations and attach compensation handlers to them.

Related Patterns

- *Orchestration Engine*
- *Orchestration Language*

Acknowledgments

Paul Brown, Brian Marick (my PLoP 2004 shepherd), Kristin Stout and Ulrich Roxburgh have provided feedback on various drafts of these patterns. I am grateful to them all.

References

[1] Microsoft Corporation SAP AG Siebel Systems BEA Systems, International Business Machines Corporation. *Business Process Execution Language for Web Services Version 1.1*, May 2003. Available on the web from `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbizspec/html/bpel1-1.asp`.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns—Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[3] Robin Milner. *Communicating and Mobile Systems, The Pi Calculus*. Cambridge University Press, 1999.

[4] Eric S. Raymond. *The Art of UNIX Programming*. Addison-Wesley, 2003. Available on the web from `http://www.faqs.org/docs/artu/`.

[5] Wil van der Aalst. Workflow patterns. Available on the web from `http://www.workflowpatterns.com/`.