A Pattern Language for Process Execution and Integration Design in Service-Oriented Architectures

Carsten Hentrich[1], Uwe Zdun[2]
[1] CSC Deutschland Solutions GmbH, Abraham-Lincoln-Park 1, 65189 Wiesbaden, Germany, chentrich@csc.com
[2] Distributed Systems Group, Information Systems Institute, Vienna University of Technology, Argentinierstrasse 8/184-1, A-1040 Vienna, Austria, zdun@infosys.tuwien.ac.at

Abstract. Process-driven SOAs are using processes to orchestrate services. Designing a non-trivial process-driven SOA involves many difficult design and architectural decisions. Examples are: Different kinds of processes exist: long-running, business-oriented and short-running, technical processes. How to best integrate them and how to map them to execution platforms? A SOA has many different stakeholders, such as business analysts, management, software designers, architects, and developers, as well as many different types of models these stakeholders need to work with. How to present each of them with the best view on the models they need for their work? A realistic process-driven SOA contains many systems that need to be integrated, such as various process engines, services, and backend systems, running on heterogeneous technologies and platforms. How to perform integration in a way that is maintainable and scalable? This article introduces a pattern language that deals with process modeling, execution, and integration. Its main goal is to help solution architects, as well as process and service designers, to master the challenges in designing a stable and evolvable process-driven SOA.

Keywords: Process-Driven Architecture, SOA, Integration Architecture

1 Introduction

Service-oriented architectures (SOA) can be defined as an architectural concept or style in which all functions, or services, are defined using a description language and have invokable, platform-independent interfaces that are called to perform business processes [Channabasavaiah 2003 et al., Barry 2003]. Each service is the endpoint of a connection, which can be used to access the service, and each interaction is independent of each and every other interaction. Communication among services can involve simple invocations and data passing, or complex activities of two or more services. Though built on similar principles, SOA is not the same as Web services, which is a collection of technologies, such as SOAP and XML. SOA is more than a set of technologies and runs independent of any specific technologies.

Even though this definition and scoping of SOA gives us a rough idea what SOA is about, many important aspects are still not well defined or even misleading. For instance, the definition is centered on SOAP-style services (so-called WS-* services) and seems to exclude other service technologies such as REST. More importantly, the

definition does not explain the main purposes of SOAs in an organization such as supporting business agility or enterprise application integration, to name a few. To get a clearer picture on what SOA is about and which proven practices exist, we provide in this article a pattern language describing proven knowledge for an important part of many SOAs: the process execution and integration design in SOAs.

A SOA is typically organized as a layered architecture (see Figure 1), both on client and server side [Zdun et al. 2006]. At the lowest layer, low-level communication issues are handled. On top of this layer, a Remoting layer is responsible for all aspects of sending and receiving of remote service invocations, including request creation, request transport, marshalling, request adaptation, request invocation, etc. Above this layer comes a layer of service client applications on the client side and a layer of service providers on server side. The top-level layer is the Service Composition Layer at which the service clients and providers from the layer beneath are used to implement higher-level tasks, such as service orchestration, coordination, federation, and business processes based on services.

In this article we view the SOA concept from the perspective of a Service Composition Layer that is process-driven. That is, the Service Composition Layer introduces a process engine (or workflow engine) which invokes the SOA services to realize individual activities in the process (aka process steps, tasks in the process). The goal of decoupling processes and individual process activities, realized as services, is to introduce a higher level of flexibility into the SOA: Pre-defined services can flexibly be assembled in a process design tool. The technical processes should reflect and perhaps optimize the business processes of the organization. Thus the flexible assembly of services in processes enables developers to cope with required changes to the organizational processes, while still maintaining a stable overall architecture.

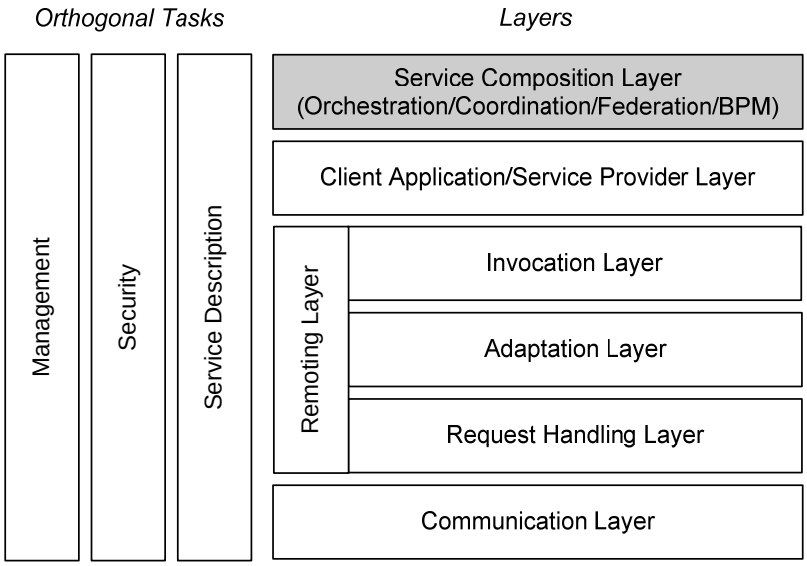

Fig. 1. SOA Layers.

In a process-driven SOA the services describe the operations that can be performed in the system. The process flow orchestrates the services via different activities. The operations executed by activities in a process flow thus correspond to service invocations. The process flow is executed by the process engine. In SOAs different communication protocols and paradigms, such as synchronous RPC, asynchronous RPC, messaging, publish/subscribe, etc. can be used and are supported by SOA technologies, such as Web Service frameworks or Enterprise Service Bus implementations. For a process-driven SOA, it can generally be assumed, however, that mainly asynchronous communication protocols and paradigms are used. This is because it cannot generally be assumed that a business process blocks until a service invocation returns. In most cases, in the meantime other sensible activities can be performed by the process. In addition, there are many places in a process-driven SOA where invocations must be queued (e.g. legacy systems that run in batch mode). It is typically not tolerable that central architectural components of the process-driven SOA, such as a central dispatcher, block until an invocation returns. Hence, synchronous service invocations are only used in exceptional cases, where they make sense.

This article is structured as follows. In Section 2 we give an overview of the pattern language presented in this article. Section 3 introduces the challenges in modeling and executing business-driven and technical processes. We present two conceptual and two architectural patterns in this context. Integration and adaptation issues in process-driven SOAs are introduced in Section 4, and four architectural patterns are presented. In Section 5 we provide a literature review and overview of related patterns. Finally, in Section 6 we conclude.

2 Pattern Language Overview

The pattern language presented in this article basically addresses conceptual and architectural design issues in the Service Composition Layer, when following a process-driven approach to services composition.

The patterns and pattern relationships for designing a Service Composition Layer are shown in Figure 2. The MACRO-/MICROFLOW[1] pattern conceptually structures process models in a way that makes clear which parts will be run on a process engine as long running business process flows (below called *macroflows*) and which parts of the process will be run inside of higher-level business activities as rather short running, technical flows (below called *microflows*). The DOMAIN-/TECHNICAL-VIEW pattern explains how to split models in a SOA into two views: A high-level, domain-oriented view and a low-level, technical view. This pattern solves the problem that executable models sometimes must be designed so that both technical and non-technical stakeholders can participate in model creation and evolution. This problem is especially in the context of long-running process flows. In this context, we require – at some point in the design – a link or translation between conceptual flows and

[1] We use SMALLCAPS font to highlight pattern names.

executable flows. The pattern can also be applied for other models in a process-driven SOA, such as business object models or component models.

The PROCESS INTEGRATION ARCHITECTURE pattern describes how to design a MACRO-/MICROFLOW architecture in detail. It is based on a number of tiers. In particular, two kinds of process engines can be used in a PROCESS INTEGRATION ARCHITECTURE. With regard to the macroflows, you can delegate the business process execution to a dedicated MACROFLOW ENGINE that executes the business processes described in a business process modeling language. The engine allows developers to configure business processes by flexibly orchestrating execution of macroflow activities and the related business functions. With regard to microflows, you can delegate the execution to a dedicated MICROFLOW ENGINE that allows developers to configure microflows by flexibly orchestrating execution of microflow activities and the related services.

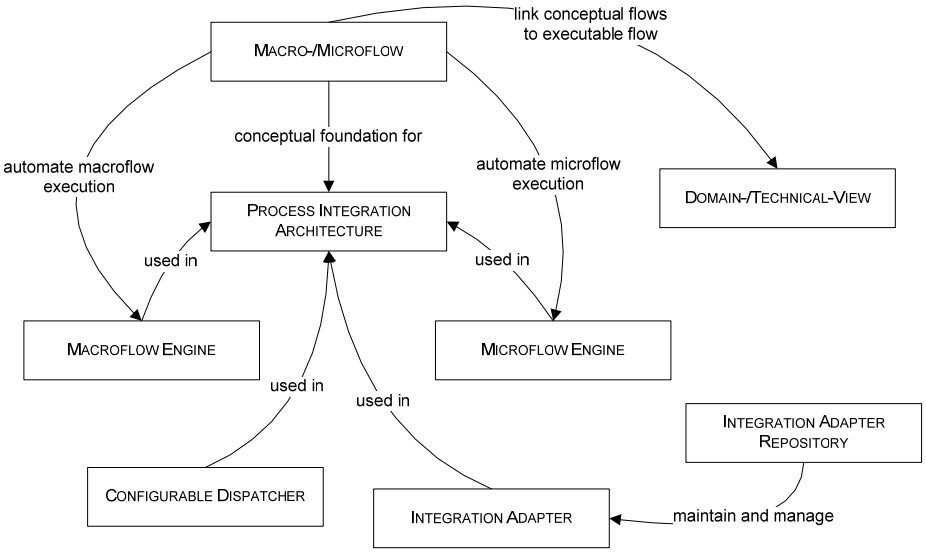

Fig. 2. Pattern relationships overview.

The INTEGRATION ADAPTER pattern explains how to connect the various parts of the SOA, such as process engines and backend systems, in a maintainable and evolvable fashion. The INTEGRATION ADAPTER REPOSITORY pattern describes how to manage and maintain INTEGRATION ADAPTERS.

The CONFIGURABLE DISPATCHER pattern explains how to connect client and target systems using a configurable dispatch algorithm. Hence, it enables us to postpone dispatch decisions until runtime. It uses configurable dispatching rules that can be updated at runtime.

3 Modeling and Executing Business-Driven and Technical Processes

In many business domains, there is a need to model the processes of the business. A process model defines the behavior of its process instances. The process model is the type of the process instances. That is, process instances are instantiations of the same kind of behavior. Process models are usually expressed in a process modeling language or notation. There are more high-level, domain-oriented languages and notations, such as BPMN, EPC, Adonis process flows, UML activity diagrams, and so on. These focus on expressing the behavior of a business or a domain. In addition, there are technical process modeling or workflow languages that define how a process behavior can be executed on a process or workflow engine. Examples are the Business Process Execution Language (BPEL), the jBPM Process Definition Language (JPDL), Windows Workflow Foundation models, or the XML Process Definition Language (XPDL)[2]. In both cases, the process modeling languages define which elements can be used in the process models.

Consider the process model example in BPMN depicted in Figure 3. It shows a very simple order handling process in which first an order is received, then the credit card is verified, and only if it is valid, the process proceeds. Otherwise, the customer is informed of the invalid credit card. Next, in parallel, the order shipment and charging for the order happens. Finally, the order status is reported to the customer.

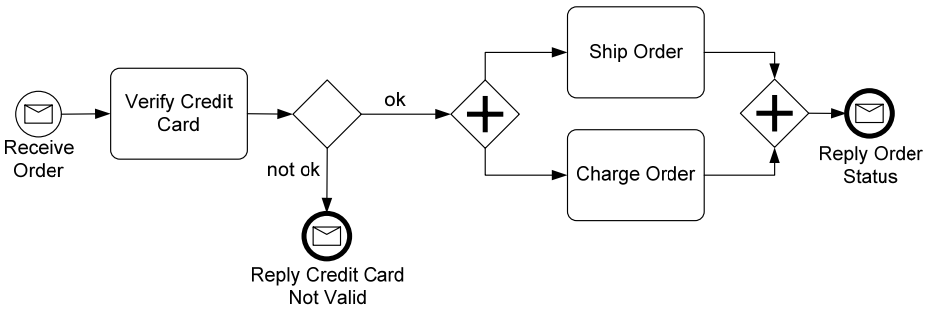

Fig. 3. Example BPMN process.

This process model can describe different things with regard to an organization, depending on the purpose of the business process modeling, such as:

- The process model describes how order handling should be handled, as a guideline or documentation of the business. The people involved in the execution of the process instances can deviate from the predefined process, where it makes sense. For instance, in some exceptional cases, shipment might be postponed, but once the shipment date is fixed, the order status reply can already be sent. This makes for instance sense in a small business,

[2] XPDL actually is not an execution language, but a process design format that can be used to store and exchange process diagrams. However, XPDL elements can have attributes which specify execution information. Some process engines, such as Enhydra Shark [Enhydra 2008], use XPDL directly as their execution language.

where people fulfill the order handling process and have an overview of the orders they handle.

- The process model defines how exactly the process instances must behave. An automated process management system ensures that each process instance follows the process model. This makes for instance sense in an organization where a high volume of similar orders with only a few exceptions must be processed, and the activities in the process are mostly automated. People only handle exceptional cases.
- The process model defines how the process instances should behave in the future. Process modeling is part of a business change initiative, for example with the goal to improve the business performance. This is one goal of many SOA initiatives. Such initiatives aim to make the business processes explicit, optimize them, and then support them through technology.
- The process model has explanatory purposes, such as the following. It defines the rationale for what happens in an information system. It links to the requirements of the IT system. It defines the data of the process that can be used for reporting purposes. It enables management or other non-technical stakeholders to analyze and plan the IT system.

Many combinations of these reasons for process modeling are possible and many other reasons exist. This section deals with the situation that you model your business processes and also want to implement them using IT support.

The first issue that must be addressed is the semantic difference between a domain-oriented business processes like the one depicted above and an executable process. *Executable* in this context means that the process contains all technical information that is needed to run it on a computer. The BPMN order handling example is not executable because many technical details are omitted. Some examples in the sample process are:

- It is unclear how the activities of the process are realized. For instance, the credit card verification could be realized as a service, a sub-process, a piece of programming language code, a script, etc. For a service, for example, we need the endpoint information that is needed to access it, such as host and port, but this technical information is missing in the BPMN example process.
- It is unclear how the data is passed from the incoming message to the process activities. For instance, which credit card information is provided and how is it used in the process?
- It is unclear how the data is mapped to the interface of a component or service that performs activities, such as credit card verification? How are interface and data differences handled? It is also unclear how the results are mapped into the process, so that the process' control structures, such as the decision node following the credit card verification in the BPMN example process, can use it.

All this information is provided in technical modeling languages, such as BPEL. Please note that some executable processes include human tasks. Others are machine-executable, meaning that no human tasks are part of the process. In both cases, we need to add the technical details to the domain-oriented processes executable.

For instance, below you see a very small excerpt from a simplistic BPEL process, implementing the BPMN process above. The example excerpt just shows the code

needed to receive the initial message, copy some variables to an input type for the `VerifyCreditCard` service, and then invoke that service.

```
<sequence>
  <receive name="ReceiveOrder" createInstance="yes"
    partnerLink="Customer"
    operation="OrderHandlingOperation"
    xmlns:tns=
      "http://j2ee.netbeans.org/wsdl/OrderHandling"
    portType="tns:OrderHandlingPortType"
    variable="OrderHandlingOperationIn"/>
  <assign name="AssignCreditCardInfo">
    <copy>
      <from>$OrderHandlingOperationIn.
        orderHandlingInputMessage/ns0:creditCardNumber
      </from>
      <to>$IsValidIn.parameters/number</to>
    </copy>
    <copy>
      <from>$OrderHandlingOperationIn.
        orderHandlingInputMessage/ns0:creditCardHolder
      </from>
      <to>$IsValidIn.parameters/holder</to>
    </copy>
    <copy>
      <from>$OrderHandlingOperationIn.
        orderHandlingInputMessage/
        ns0:creditCardSecurityCode
      </from>
      <to>$IsValidIn.parameters/securityNumber</to>
    </copy>
  </assign>
  <invoke name="VerifyCreditCard"
    partnerLink="VerifyCreditCard"
    operation="isValid" xmlns:tns="http://orderHandling/"
    portType="tns:VerifyCreditCard"
    inputVariable="IsValidIn"
    outputVariable="IsValidOut"/>
```

In addition to the BPEL code, we require the WSDL files that describe the interfaces of the services and of this process, and the XML Schema definitions of the data types that are passed.

All these technical specifications are even hard to understand and complex for technical experts. For that reason, many modeling tools exist. For example, the following Figure 4 shows a simplistic BPEL process implementation of our order handling process modeled in the BPEL modeler of the NetBeans IDE. In addition to modeling BPEL graphically, designer tools offer support for designing WSDL interfaces, XML Schema definitions, and data mappings. The typical tooling around

process engines has been described in pattern form by Manolescu (see [Manolescu 2004]).

In cases where process execution is the main goal of the process modeling, it seems to make sense to model the processes directly in BPEL using such a modeling tool, instead of modeling in BPMN. However, it is rather seldom the case that process execution is the only goal that process modeling is needed for. Usually, the technical experts are not domain experts and hence need to discuss the processes with the domain experts to incorporate the domain knowledge in the right way. BPEL is usually not a good representation for tasks that involve domain experts because BPEL processes are overloaded with technical details. This is certainly valid for the BPEL code itself. But it is also the case for what is shown in BPEL modeling tools: While these technical process modeling tools are a very helpful aid for developers, the models they expose are still pretty technical and complex. It is awkward to use them for the discussion with domain experts and usually impossible to let domain experts themselves work with these tools. For the same reasons, they are also not the best solution for technical stakeholders, if their work requires only getting a quick overview of the existing processes. The technical process code or the technical process modeling tools should only be used for in-depth technical work.

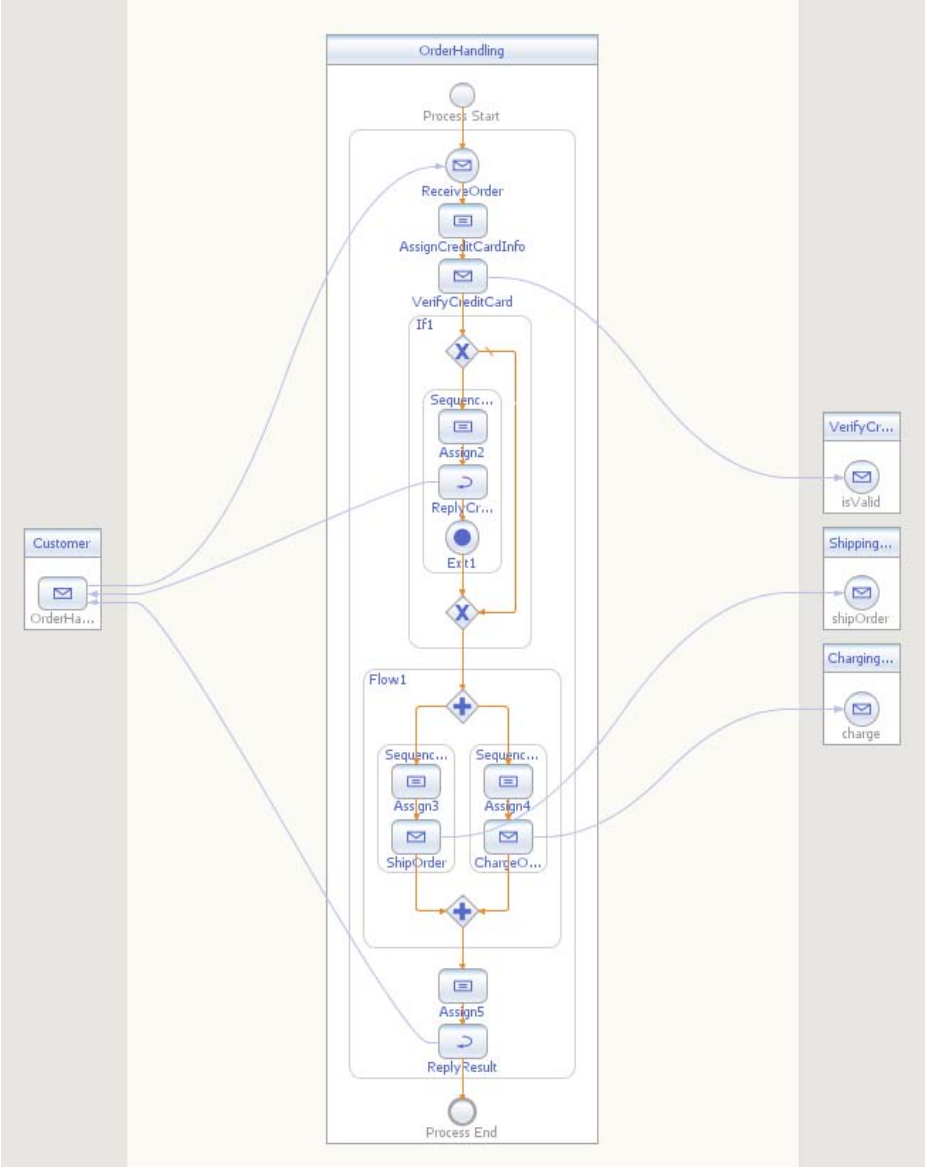

Fig. 4. Example BPEL Process.

The DOMAIN-/TECHNICAL-VIEW pattern solves this problem by splitting the models into two views:

- A high-level, domain-oriented view that represents the process in a technology-independent fashion leaving away all details not needed for the domain task.
- A low-level, technical view that contains the elements of the domain view and also contains additional technical details.

This pattern is not only applicable for process models, but also for all other kind of models that must be shown in two views. An example is a data model that has a logical data view and a technology-dependent view. Here, the technology-dependent view would model the mapping of the logical view to a database (access) technology. Another example is a class model that represents a high-level domain model and an implementation model showing how the domain model elements are realized using e.g. a component technology.

So far, we did not distinguish different kinds of processes. However, in a typical SOA in the enterprise field we can observe different kinds of behaviour that could be modelled as process flows. For instance, there are strategic, very high-level business processes that are hard to automate or support through technology. These are often broken down – sometimes in a number of steps – into more specific business processes, such as the order handling example above. The result is long-running business processes, perhaps with human involvement, which can possibly be mapped to a supporting technology. Finally, when implementing these processes, we observe also more short-running and technical processes. For instance, the verification of the credit card in the example above could consist of three steps, each calling an external Web service. Or the shipping of an order could require to a few steps guiding a human operator through a number of GUI dialogs for approving the automatically selected inventory items, approving the sending of an invoice, and so on.

The distinction between long-running, business-oriented and short running, technical processes is an important conceptual distinction that helps us to design process activities at the right level of granularity. In addition, the technical properties of the two kinds of processes are different. For instance, for long-running processes it is typically not appropriate to use ACID (Atomicity, Consistency, Isolation, Durability) transactions because it is infeasible to lock resources for the duration of the whole process, while this might be perfectly feasible for more short running processes of only a few service invocations.

The MACRO-/MICROFLOW pattern provides a clear guideline how to design process models following these observations. In the pattern, we refer to the long-running process using the term *macroflow*. We use the term *microflow* to refer to the short running, technical processes. The pattern advises to refine macroflows in a strictly hierarchical fashion – starting from high-level, strategic process to long-running, executable processes. The activities of these executable macroflows can further be refined by microflows. Microflow activities can be refined by other microflows. That is, an activity of a higher-level process model is refined by a lower-level process model in form of a sub-process.

The pattern is closely related to the DOMAIN-/TECHNICAL-VIEW pattern: The highest level macroflows usually only have domain views. The lowest level microflows often only have a technical view. But the models in between – in particular the executable macroflows as in the example above – have both views as they are relevant to both technical and non-technical stakeholders.

Refinements, as described in MACRO-/MICROFLOW and DOMAIN-/TECHNICAL-VIEW patterns, can be performed in general following processes such as or similar to Catalysis [D'Souza and Wills 1999]. Catalysis is a method for component-based development that defined traceable refinements from business requirements through component specifications and design, down to code.

The MACRO-/MICROFLOW pattern advises to use a suitable technology for realizing macroflows and microflows. Of course, it is possible to implement both macroflows and microflows using ordinary programming language code. But often we can provide better support. For instance, macroflows often should be support with process persistence, model-based change and redeployment, process management and monitoring, and so on. The MACROFLOW ENGINE pattern describes how to support a macroflow using process or workflow technology. An example MACROFLOW ENGINE that could be used in the example above is a BPEL process engine.

For microflows, supporting technology is more seldom used. However, if rapid process change or reuse of existing functionality is needed, MICROFLOW ENGINES can be very useful. We distinguish MICROFLOWS ENGINES for microflows containing human interactions, such as pageflow engines, and MICROFLOWS ENGINES for microflows supporting automatic activities, such as message brokers.

3.1 Pattern: DOMAIN-/TECHNICAL-VIEW

Context
Various stakeholders participate in the development, evolution, and use of a SOA. Typical technical stakeholders are the developers, designers, architects, testers, and system administrators. Typical non-technical stakeholders are the domain experts of the domain for which the SOA is created, the management, and customers.

Problem
How should executable models be designed if both technical and non-technical stakeholders need to participate in model creation and evolution?

Problem Details
Designing one model for a number of distinct stakeholders is challenging because the different stakeholders require different information for their work with the model, as well as different levels of detail and abstraction. A typical – often problematic – case in this context is that a model depicts a concern from the domain for which the SOA is created. So the model is, on the one hand, important for the communication with and among the domain experts. But, on the other hand, in a SOA often such models should be automatically processed and executed.

"Executable model" means that a model is interpreted by an execution engine, or the model representation is compiled and then executed on an execution engine. Here are some examples of executable models:

- A BPEL business process model that is executed on a BPEL engine.
- A role-based access control model that is interpreted by an access control enforcement component.
- A UML or EMF model that is transformed by the generator of a model-driven development (MDD) solution into executable code (see [Stahl and Völter 2006]). In this case the model is interpreted at design time of the SOA, but at runtime of the generator. An alternative is using an executable

model, such as Executable UML [Mellor and Balcer 2002] or the UML virtual machine [Riehle et al. 2001].

In order to be executable, a model must contain all technical information needed for execution. The original intent of modeling is often different, however: to serve as a means of communication among stakeholders and for system understanding. The information needed for execution is usually targeted only at one type of stakeholders: the technical developers. This, in turn, makes the models hard to understand for the domain experts and, in general, hard to use for tasks that require getting an overview of the design. The reason is the executable models are simply overloaded with too many technical details.

Just consider business processes as one example of SOA models. Domain experts usually design with tools that use diagrams such as or similar to diagrams in BPMN, EPC, Adonis process flows, UML activity diagrams, and so on. The diagrams usually contain only the information relevant for the business. Often such models are hard to automate because they miss important technical information, such as how data is passed or transformed, or where a service to be invoked is located and with which interface it is accessed. Technical process modeling languages such as BPEL, jPDL, Windows Workflow Foundation models, or XPDL, in contrast, contain all this information. This makes them useful for technical execution, but also complex and hard to understand for domain-oriented tasks.

Solution

Provide each model that it is required both for domain-oriented tasks, such as getting an overview of the design, and technical tasks, such as execution of the model, in two views: a domain view and a technical view. All elements from the domain view are either imported or mapped into the technical view. The technical view contains additional elements that enrich the domain model elements with the technical details necessary for execution and other technical tasks.

Solution Details

Figure 5 illustrates the solution of the DOMAIN-/TECHNICAL-VIEW pattern.

To realize the DOMAIN-/TECHNICAL-VIEW pattern, the elements from the domain view must be imported or mapped into the technical view. This can be done in various ways. Basically, the differences between these variants of the pattern are the mechanisms used for the import and the mapping, and the degree of automation.

- The simplest variant of the pattern is to perform a manual translation to map the domain model elements into the technical view. First of all, for each domain model element, the most appropriate modeling construct for representing the domain model element in the technical view is chosen, and then the translation is performed. Next, the technical model is enriched with all information needed for technical tasks such as execution and deployment. This variant of the pattern has the benefit of flexibility: Any modeling languages can be mapped in any suitable way. As a creative design step is needed for the mapping, no formal link or mapping between the modeling languages is required, but the translation can happen on a case by case basis. This variant incurs the drawback that – for each change – manual effort is

required for performing the translation, and consistency between the models must be ensured manually.

- Sometimes translation tools between a modeling language for domain models and a modeling language for technical models exist, such as a BPMN to BPEL mapping tool. In the mapping process, somehow the additional technical information must be added. For instance, it can be given to the translation tool using an additional configuration file. Using this additional information and the domain view model, the translation tool generates a technical view corresponding to the domain view. This variant of the pattern can potentially reduce the manual mapping effort and ensure consistency automatically. It can be realized for two distinct modeling languages. However, not always an automatic mapping provides the best mapping. Especially for two languages with highly different semantics, the automatic mapping can cause problems. For instance, technical models can get hard to understand and debug, if they are automatically generated from a domain model with different semantics.
- If both models, technical view and domain view, are described based on a common meta-model, ordinary model extension mechanisms, such as package import, can be used. Extensions in the technical view are then simply added using ordinary extension mechanisms for model elements, such as inheritance or delegation. This variant makes it easy to maintain consistency between the view models. Modeling tools can for instance allow designers to start modeling in the domain view and enrich it with technical details in property views. As an alternative, you can generate the domain view from the technical view model. That is, you just strip the technical information provided in the technical view.

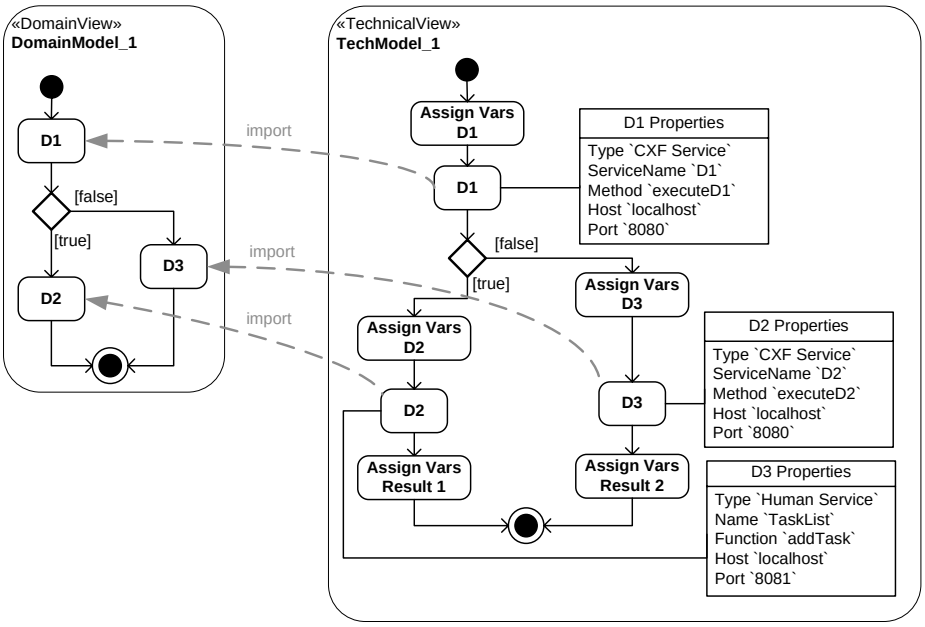

Fig. 5. Illustration of DOMAIN-/TECHNICAL-VIEW.

If the domain view is not generated from the technical view, for all the imported domain model elements, all changes should be performed in the domain view. They should then be propagated to the technical view. This means for the three variants of the pattern:

- If manual translation is used, translating the changes made to the domain models is required. Change propagation is a main drawback of this variant: If changes are made to one model and they are forgotten to be propagated or incorrectly propagated, the models are getting inconsistent.
- If an automatic translation tool is used, the tool must be re-run, and it regenerates the domain elements in the technical view. An alternative is a round-trip translation: Changes to the technical view can be translated back into the domain view. Round-trip translation is often not advisable, as tools tend to generate hard to read source code and creating a well-working round-trip tool set is a substantial amount of work.
- If integration based on a common meta-model and package imports are used, changes to the domain model are reflect automatically in the technical view. Hence, no efforts for change propagation are required in most cases. Only if changes cause incompatibilities in dependent models, the models must be adapted accordingly.

Example: Manual BPMN to BPEL translation
We have seen a simple example of a manual BPMN to BPEL translation in the introduction of Section 3.

Example and Known Use: View-based Modeling Framework

The View-based Modeling Framework (VbMF) [Tran et al. 2007] is a model-driven infrastructure for process-driven SOAs. It uses the model-driven approach to compose business processes, services, and other models that are relevant in a SOA. VbMF abstracts each concern in its own view model. Each VbMF view model is a (semi)-formalized representation of a particular SOA concern. The view model specifies the entities and relationships that can appear in a view.

In particular, there is a Core view model from which each other view model is derived. The main task of the Core view model is to provide integration points for the various view models defined as part of VbMF, as well as extension points for enabling the extension with views for other concerns.

The view models derived from Core are *domain views*. Example domain views are: Collaboration, Information, Control-Flow, Long-Running Transactions, Data, and Compliance Metadata. In addition, to these central concerns, many other concerns can be defined. Each of these view models is either extending the core model or one of the other view models. These view models contain no technology-specific information, but information understandable to the domain expert.

In addition, VbMF defines also a second level of extensional view models, derived from these domain views models – the *technical views*. For specific technologies realizing the domain views, such as BPEL, WSDL, BPEL4People, WS-HumanTask, Hibernate, Java services, HTML, or PDF, VbMF provides technical view models, which add details to the general view models that are required to depict the specifics of these technologies.

Figure 6 provides an overview of the VbMF view models and their relationships.

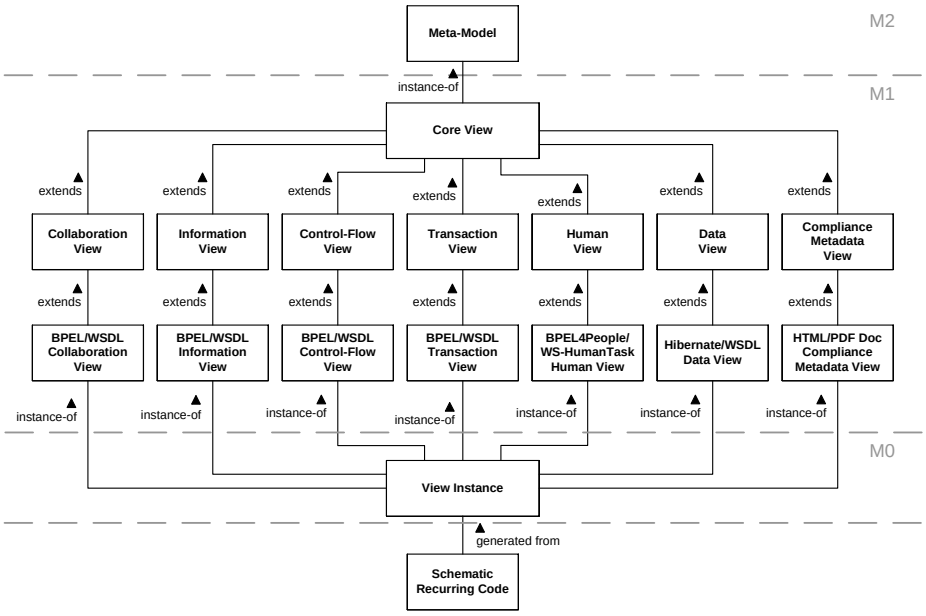

Fig. 6. VbMF view models.

The integration of view elements is done using modeling abstractions, such as inheritance and associations, as well as matching algorithms, such as name-based matching. Integration is performed in the transformation templates of the code generator.

The separation of view abstraction levels helps in enhancing the adaptability of the process-driven SOA models to business changes. For instance, the business experts analyze and modify the domain views to deal with change requirements at the level of the business. The technical experts work with technical views to define necessary configurations so that the generated code can be deployed into the corresponding runtime (i.e., the process engines and Web service frameworks). This view-based separation into two view layers, domain views and technical views, also helps to better support the various stakeholders of the SOA: Each stakeholder views only the information necessary for their work. Hence, the view-based approach supports involving domain experts in the design of SOAs.

Known Uses

- The BPMN2BPEL tool [BPMN2BPEL 2008] is an Ecplise plugin for transforming BPMN processes, modeled in Eclipse, to BPEL. Like many other such tools, the tool can only translate the information that is present in BPMN, which might mean that technical details are not considered and semantic differences between BPEL and BPMN are not translated in the best way.
- In [Dikmans 2008] it is discussed how to transform a BPMN model from the Oracle Business Process Analysis Suite to an executable BPEL process. The article also discusses the semantic differences between BPMN and BPEL. If processes are not translatable using the tool, the article advises to change the BPMN process by removing arbitrary cycles that are valid in BPMN, but not in BPEL.
- Sculptor [Fornax 2008] is a cartridge for openArchitectureWare, a model-driven software development infrastructure. Sculptor enables developers to focus on the business domain view, which is designed in a textual, domain-specific language using concepts from Eric Evans' book Domain-Driven Design [Evans 2004], such as Service, Module, Entity, Value Object, Repository, and so on. The code generator is used to generate Java code for well-known frameworks, such as Spring Framework, Spring Web Flow, JSF, Hibernate and Java EE. The technical view is added using configurations and manually written code.

3.2 Pattern: MACRO-/MICROFLOW

Context

If your system is or should be described using process models, it makes sense to think about automating the processes using process technology. Usually, if an organization decides to use business processes to depict their business, high-level and mostly business-oriented models are created.

Problem
How can conceptual or business-oriented process models be implemented or realized?

Problem Details
One important aspect to consider when implementing or realizing business processes is the nature of the processes to be executed. For instance, many typical business processes are long running flows, involving human tasks. Such a business process can run for many hours, days, or even weeks before it is finished. In such cases, the process technology must support persisting the process instances, as the process states should not get lost if a machines crashes. The process instance should not occupy memory and other system resources, when it is not active. It should be possible to monitor and manage the process instance at runtime. Also, the processes should be interruptible via a process management interface. Such functionalities are supported by process or workflow engines. Process engines usually express processes in a process execution language, such as BPEL, jPDL, Windows Workflow Foundation models, or XPDL.

In contrast to the long-running kind of flows, also short-running flows need to be considered. These often have a more technical nature and rather transactional or session-based semantics. One example is a process in which a number of steps are needed to perform a booking on a set of backend systems. Another example is guiding a human user through a pageflow of a few Web pages. In these examples, process instance persistence is not really needed and typical process execution languages make it rather awkward to express these flows. Hence, it makes sense to realize them using special-purpose modeling languages and technology. For instance, a message flow model and message broker technology or a pageflow model and pageflow engine technology could be used to realize the two examples.

Please note that the named technologies are just examples: The distinction of short-running and long-running flows is in first place a conceptual distinction. Any suitable technology can be chosen. For instance, it is also possible to implement both short-running and long-running flows using ordinary programming language code (e.g., Java source code or code written in a scripting language) – maybe provided as a service. However, as in both cases some features, such as process persistence, wait states, concurrency handling, and so on, are needed over and over again, reusing existing technologies often makes sense.

Finally, in some cases, it turns out that automating the process is not useful or feasible. Then an entirely manual process fulfilled by people with no automation whatsoever can be chosen as a realization of a process as well. Another example is high-level processes, such as strategic business processes of an organization, which would need concretization before they could be implemented.

Unfortunately, in practice often long-running and short-running flows, as well as technical and non-technical concerns, are intermixed. Often concerns with different semantics are modeled in one model. This practice often causes confusion as business analysts do not understand the level of technical detail, and technical modelers do not have the expertise to understand the business issues fully. Thus, these models tend to fail their primary purpose – to communicate and understand the processes and the overall system design.

In addition, models are sometimes mapped to the wrong technology. For instance, a short-running transactional flow should not be realized using a process engine for long-running flows, and vice versa, as the different technologies exhibit significantly different technical characteristics.

Solution

Structure a process model into two kinds of processes, macroflow and microflow. Strictly separate the macroflow from the microflow, and use the microflow only for refinements of the macroflow activities. The macroflows represent the long-running, interruptible process flows which depict the business-oriented process perspective. The microflows represent the short-running, transactional flows which depict the IT-oriented process perspective.

Solution Details

Figure 7 illustrates the solution of the MACRO-/MICROFLOW pattern.

The MACRO-/MICROFLOW pattern provides a conceptual solution in which two kinds of flows are distinguished:

- *Macroflows* represent the business-oriented, long-running processes.
- *Microflows* represent the IT-oriented, short-running processes.

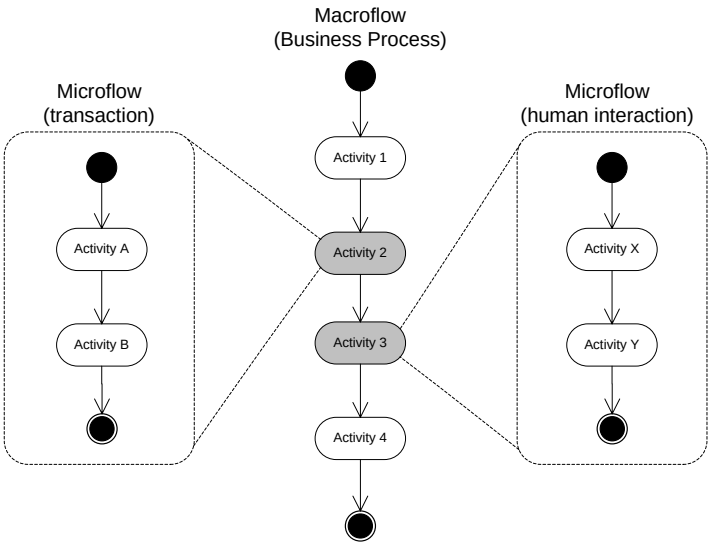

Fig. 7. Illustration of MACRO-/MICROFLOW.

The MACRO-/MICROFLOW pattern interprets a microflow as a refinement of a macroflow activity. That is, a microflow represents a sub-process that runs within a macroflow activity. This separation of macroflow and microflow has the benefit that modeling can be performed in several steps of refinement. First the higher level macroflow business process can be designed, considering already that business

process activities will further be refined by microflows. Vice versa, if certain microflows already exist, the business process can be modeled accordingly, so that these IT processes fit in as business process activities at the macroflow level. However, this also incurs the drawback that the conceptual separation of the MACRO-/MICROFLOW pattern must be understood and followed by modelers, which requires additional discipline.

The refinement concepts of the MACRO-/MICROFLOW often require adjusting IT processes and business processes according to the concerns of both domains – business and IT – in order to bring them together. The modeling effort is higher than in usual business modeling, as more aspects need to be taken into consideration and, at all refinement levels, activities must be designed at the right level of granularity.

A microflow model can be linked to one or many macroflow activities. The consequence is that the types of relationships between macroflow and microflow are well-defined. Microflows and macroflows both have a defined input and output, i.e., a well-defined functional interface. However, the functional interfaces between IT processes and business processes must be understood and considered by all stakeholder manipulating process models.

Multiple levels of both macroflows and microflows can be modeled. That is, high-level macroflows can be refined by lower-level macroflows. The same is possible for microflows. The refinement is strictly hierarchical: Always an activity in the high-level process is refined by a low-level sub-process, realizing the activity. Never a microflow is refined by a macroflow. Figure 8 illustrates two levels of macroflow refinement and two levels of microflow refinement.

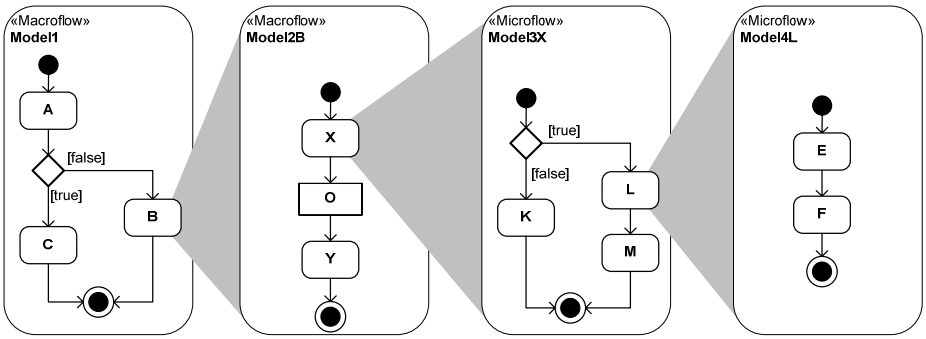

Fig. 8. Illustration of MACRO-/MICROFLOW.

The microflow can be directly invoked as a sub-process that runs automatically, or it can represent an activity flow that includes human interaction. As a result, two types of links between a macroflow activity and a microflow exist:

- *Link to a microflow for an automatic activity (transaction):* A short-running, transactional IT process defines a detailed process model of an automatic activity in a higher-level business process. It represents an executed business function or transaction at the business process level.
- *Link to a microflow for human interaction*: In case an activity of a business process is associated to a user interface, the IT process is a definition of the

coherent process flow that depicts the human interaction. This process flow is initiated if a human user executes the business process activity.

The microflow level and the macroflow level distinguish conceptual process levels. Ideally, both levels should be supported by suitable technology. An example for a macroflow technology is a process execution engine, such as a BPEL engine. An exemplary microflow technology for automatic activities is a message broker which provides a message flow modeling language. For short-running human interactions technologies such as pageflow engines can be used.

Both types of microflows are often hard-coded using ordinary code written in a programming language. Sometimes an embedded scripting language is used to support flexible microflow definition in another language such as Java or C#. If this implementation option is chosen, a best practice is to provide the microflows as independent deployment units, e.g., one service per microflow, so that they can be flexibly changed and redeployed. Microflow implementations should be architecturally grouped together, e.g., in a SERVICE ABSTRACTION LAYER [Vogel 2001], and not scattered across the code of one or more components, which also realize other tasks.

In the ideal case, the modeling languages, techniques, and tools should support the conceptual separation of macroflows and microflows, as well as the definition of links between macroflow activities and microflows using the two types of links described above.

This pattern is strongly related to the DOMAIN-/TECHNICAL-VIEW pattern because typically, at the point where the macroflows are mapped to technologies, we need both views. That is, the macroflows require in any case a domain view (e.g., modeled in BPMN, EPC, or Abstract BPEL), as macroflows need to be discussed with domain experts from the business. At the point where macroflows are mapped to technologies, we also need a technical view of the macroflow (e.g., modeled in BPEL, jPDL, Windows Workflow Foundation models, or XPDL). The same duality can be observed for microflows. Here, in any case, a technical view is needed, as all microflows are executable. Sometimes, an additional domain view is needed, for instance, if microflow models should be designed together with domain experts. Just consider a pageflow model: a domain view would just graphically show the pageflow, and a technical model adds the technology-dependent details.

The most common solution for combining MACRO-/MICROFLOW and DOMAIN-/TECHNICAL-VIEW is:

1. High-level macroflows that depict strategic business processes and that are not implemented are designed only using a domain view (this step is optional).
2. The high-level macroflows are refined by lower-level macroflows that get implemented and offer a domain view as well as a technical view.
3. The macroflows invoke microflows which only have a technical view.

Example: Structural model of macroflow and microflow

Figure 9 shows an exemplary model for explicitly supporting the MACRO-/MICROFLOW pattern. The model shows different kinds of macroflows and microflows, and the relationships between them. The MACRO-/MICROFLOW pattern

generally provides a conceptual basis for the development of such models, which could for instance serve as a foundation for model-driven software development.

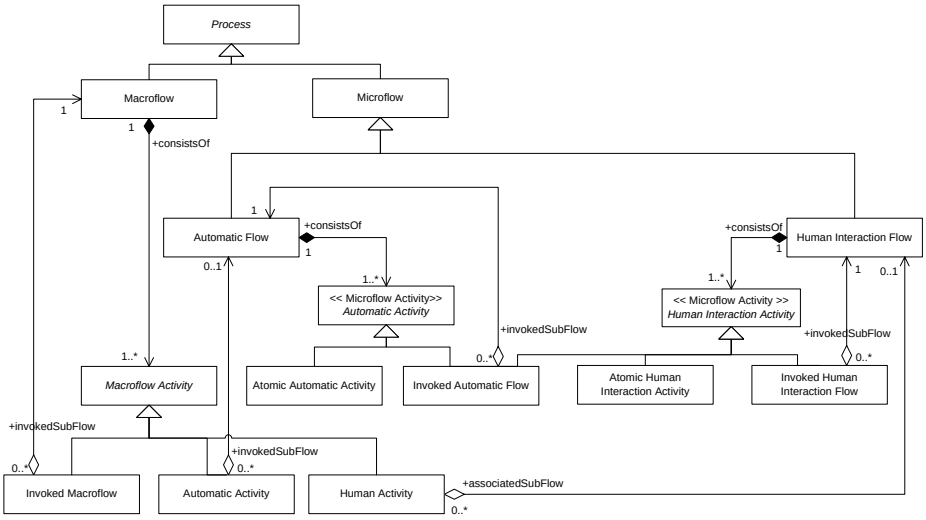

Fig. 9. Structural model of macroflow and microflow.

Known Uses

- In IBM's WebSphere technology [IBM 2008] the MACRO-/MICROFLOW pattern is reflected by different technologies and methodologies being used to design and implement process-aware information systems. Different kinds of technologies and techniques for both types of flows are offered. On the macroflow level, workflow technologies are used that support integration of people and automated functions on the business process level. An example is IBM's WebSphere Process Choreographer, which is a workflow modeling component. The microflow level is rather represented by transactional message flow technologies that are often used in service-oriented approaches. Examples are the WebSphere Business Integration Message Broker and the WebSphere InterChange Server. At the macroflow level, a service is invoked that is designed and implemented in detail by a microflow that performs data transformation and routing to a backend application. Moreover, aggregated services are often implemented at the microflow level using these kinds of message flow technologies.
- GFT's BPM Suite GFT Inspire [GFT 2007] provides a modeler component that uses UML activity diagrams as a notation for modeling the macroflow. Microflows can be modeled in various ways. First, there are so-called step activities, which allow the technical modeler to model a number of sequential step actions that refine the business activity. In the step actions, the details of the connection to other systems can be specified in a special purpose dialog. This concept is especially used to invoke other GFT products, such as the document archive system or a form-based input.

Alternatively, the microflow can be implemented in Java snippets, which can be deployed to the server – together with the business process. Finally, services can be invoked that can integrate external microflow technologies, such as message brokers.

- JBoss' jBPM engine [JBoss 2007] follows a slightly different model, as the core component is a Java library and hence can be used in any Java environment. The jBPM library can also be packaged and exposed as a stateless session EJB. JBoss offers a graph-based designer for the macroflow process languages, and works with its own proprietary language, jPDL. A BPEL extension is also offered. The microflow is implemented through actions that are associated with events of the nodes and transitions in the process graph. The actions are hidden from the graphical representation, so that macroflow designers do not have to deal with them. The actions invoke Java code, which implements the microflow. The microflows need not be defined directly in Java, but can also be executed on external microflow technology, such as a message broker or a pageflow engine.
- Novell's exteNd Director [Novell 2008] is a framework for rapid Web site development. It provides a page flow engine implementing microflows for human interaction. A workflow engine realizes long-running macroflows. A pageflow activity in the workflows is used to trigger pageflows. This design follows the MACRO-/MICROFLOW pattern.

3.3 Pattern: MACROFLOW ENGINE

Context
You have decided to model parts of your system using macroflows to represent long-running business processes, for instance following the MACRO-/MICROFLOW pattern. The simplest way to implement and execute your macroflow process models is to manually translate them into programming language code. But, as many tasks and issues in a macroflow implementation are recurring, it would be useful to have some more support for macroflows.

Problem
How can macroflow execution be supported by technology?

Problem Details
One of the main reasons to model macroflows is to enable coping with business change. The reasoning behind this idea is that if you model your processes explicitly, you understand the implementation of your business better and can more quickly react to changes in the business. Changes to business are reflected by changes in the corresponding macroflows. Today a lot of IT systems support business processes, and the required changes often involve significant changes in IT systems with high costs and long development times. In a dynamic business environment, these costs and long development times are often not acceptable, as conditions might have already changed when old requirements are implemented.

One of the major reasons for this problem is that business process logic is hard-coded in the program code. The required changes thus imply to change program code in various systems. The result is a fragmentation (or structural gap) of business processes and IT systems that support them.

Often a lot of different skills are required to achieve this, as the systems are implemented on varying platforms with different technology, programming paradigms, and languages. The heterogeneity of systems and concepts lead also problems for end-users, who have to roughly understand the adaptations of the changed systems. Often the desired business process, as it was originally designed, cannot be realized due to limitations of existing systems or because of the high efforts required to implement the changes.

The complexity generated by this heterogeneity and the interdependencies between the systems let projects fail even before they have started, as the involved risks and the costs may be higher than the estimated benefit of the business process change. Thus incremental evolution cannot be achieved. As a result, IT has gained the reputation of just being a cost driver but not a business enabler. In many cases, this problem causes a situation in which no significant and innovative changes are made, and solving prevalent problems is postponed as long as possible.

Hard-coding business processes also means that recurring functionality required for executing macroflows, such as process persistence, wait states, process management, or concurrency handling, need to be manually coded, too. That is, effort is required to develop and maintain these functionalities.

Solution

Use a dedicated MACROFLOW ENGINE that supports executing long-running business processes (macroflows) described in a business process modeling language. Integrate business functions as services (or modules) that are invoked by the macroflow activities. Changes of macroflows are supported by changing the macroflow models and deploying the new versions to the engine at runtime.

Solution Details

Figure 10 illustrates the solution of the MACROFLOW ENGINE pattern.

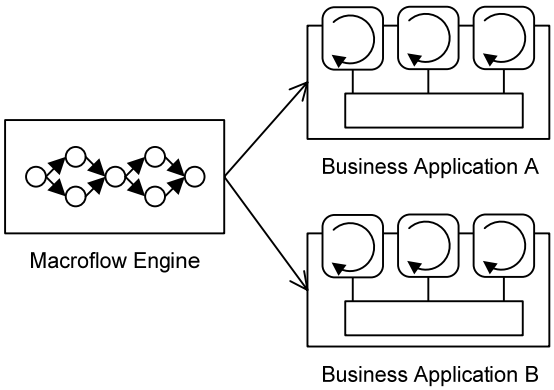

Fig. 10. Illustration of MACROFLOW ENGINE.

The MACROFLOW ENGINE pattern's main participant is the engine component that allows developers to describe the business process logic by changing the business process definitions. Using a MACROFLOW ENGINE in an architecture means to decouple the business process logic from the IT systems. However, effort is needed to introduce a MACROFLOW ENGINE based architecture. The pattern has best effects if applied as a long term approach to architecture design and application development. Short term goals may not justify the efforts involved.

Macroflow definitions are defined using a process modeling language. The engine executes the models, written in that modeling language. Changes occur by modifying the macroflow models and deploying the changed versions on the engine. Using this architecture, business process definitions can be flexibly changed, and the corresponding processes in IT systems can be adapted more easily than in hard-coded macroflow implementations in programming language code.

The models executed by a MACROFLOW ENGINE represent a technical view of a macroflow, as described in the DOMAIN-/TECHNICAL-VIEW pattern, and are usually expressed in business process execution languages, such as BPEL, XPDL, Windows Workflow Foundation models, or jPDL. In many cases, a domain view of the models is defined as well, for instance in high-level process modeling languages, such as BPMN or EPC.

Applications are understood as modules that offer business functions, e.g., as services. If the MACRO-/MICROFLOW pattern is applied, of course, a service can internally realize a microflow implementation. The MACROFLOW ENGINE does not see these internal details, however, but only the service-based interface. The business functions are orchestrated by the business process logic described in the MACROFLOW ENGINE'S modeling language. Business functions are either completely automatic or semi-automatic, representing a human interacting with a system.

Business functions are represented in the macroflow as macroflow activities. They are one of a number of different activity types, supported by the engine. Other example activity types are control flow logic activities, data transformation activities, and exception handling activities. The MACROFLOW ENGINE concentrates on orchestration issues of macroflow activities but not on the implementation of these activities. The actual implementation of macroflow activities is delegated to functionality of other systems that the engine communicates with.

The MACROFLOW ENGINE offers an API to access the functionality of the engine, i.e., processing of automatic and semi-automatic tasks. It further offers functions for long-running macroflow execution, such as process persistence, wait states, and concurrency handling. It also offers an interface for managing and monitoring the processes and process instances at runtime.

Various concepts exist for orchestration of macroflow activities in a MACROFLOW ENGINE. Two common examples are:

- Strictly structured process flow, e.g., in terms of directed graphs with conditional paths (this is most common variant in commercially used products and tools)
- Flexibly structured flow of activities, e.g., by defined pre- and post-conditions of macroflow activities

If business processes have already been hard-coded, developers must first extract the implemented business process logic from the systems and model them in the MACROFLOW ENGINE'S modeling language. Hence, in such cases, introducing the MACROFLOW ENGINE pattern has the drawback that efforts must be invested to extract the business process logic out of existing systems implementations and to modify the architecture.

Example: Structural model of a MACROFLOW ENGINE
Figure 11 shows a simple example model of a MACROFLOW ENGINE design that could be used as a starting point if one would realize a MACROFLOW ENGINE from scratch. The `Macroflow Engine` supports a simple management interface for `Macroflows`. These have a number of `Macroflow Activities`. A `Macroflow Activity` is assigned to a `Resource`, where a `Resource` can be some virtual actor like an IT system acting in a certain role, or a human actor who interacts with an IT system. As far as a human actor is concerned, constraints may be applied to make the macroflow activity only accessible to a defined set of users, e.g., by roles and rights that a user must have in order to be able to process a macroflow activity. The `Macroflow Activities` always invoke a `Business Function`, whether the `Business Function` is executed with support of a human being or whether it is completely automatic. Control data, such as process variables, is transformed during the execution of `Macroflow Activities.`

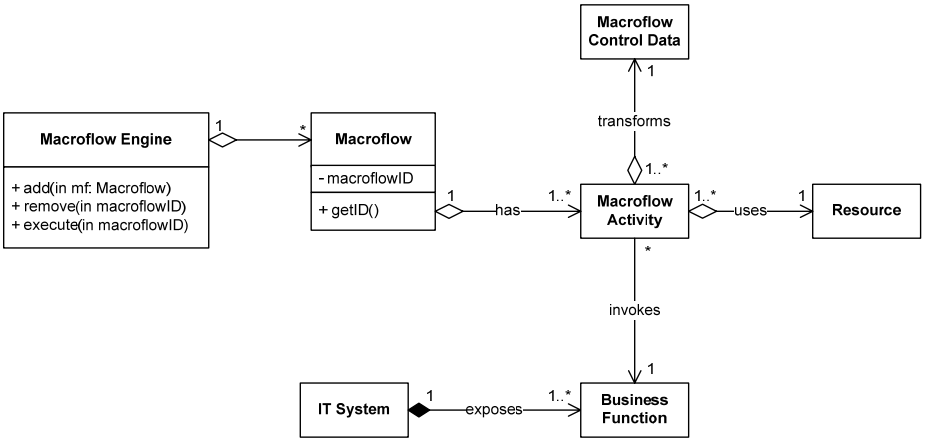

Fig. 11. Structural model of a MACROFLOW ENGINE.

Known Uses

- IBM's WebSphere Process Choreographer [IBM 2008] is the workflow modeling component of WebSphere Studio Application Developer Studio, Integration Edition, which provides a MACROFLOW ENGINE. The workflow model is specified in BPEL.

- In the latest WebSphere product suite edition [IBM 2008], the two products WebSphere Process Choreographer and WebSphere InterChange Server have been integrated into one product which is called WebSphere Process Server. Consequently, this new version offers both, a MACROFLOW ENGINE and a MICROFLOW ENGINE.
- GFT's BPM Suite Inspire [GFT 2007] provides a designer for macroflows that is based on UML activity diagrams. The business processes can be deployed to an application server that implements the MACROFLOW ENGINE for running the business processes. The engine also offers an administrator interface for monitoring and management of the processes.
- JBoss' jBPM [JBoss 2007] is an open-source MACROFLOW ENGINE for graph-based business process models that can be expressed either in jPDL or BPEL as modeling languages. jBPM offers a Web-based monitoring and management tool.
- ActiveBPEL [Active Endpoints 2007] is an open-source BPEL engine that acts as a MACROFLOW ENGINE for business processes modeled in BPEL.
- Novell's exteNd Director [Novell 2008] is a framework for rapid Web site development. It provides a workflow engine realizing long-running macroflow.

3.4 Pattern: MICROFLOW ENGINE

Context

You have realized that the business functions (services) that are orchestrated by macroflows in your system can be understood as short-running, technical processes. Following the MACRO-/MICROFLOW pattern, you introduce microflow models for representing these processes. In many cases, the "conceptual" notion of microflows is useful and sufficient, and microflows are implemented without supporting technology, for instance, using ordinary programming language code or scripts written in a scripting language.

You can further support microflows in hard-coded solutions: A best practice for realizing hard-coded microflows is to place them in their own coding units that can be independently deployed, e.g., each microflow is implemented as its own service in a distinct microflow SERVICE ABSTRACTION LAYER [Vogel 2001]. Support of embedded scripting or dynamic languages for defining microflows can even more support the flexibility of microflow definition and deployment. For many cases, this solution is absolutely good enough. In some cases, however, you would like to get more support for microflow execution.

Problem

How can microflow execution be supported by technology to avoid hard-coded microflow solutions and offer benefits for microflows akin to workflow technologies?

Problem Details

It takes considerable time and effort to realize and change processes, if the technical microflow details are hard-coded in programming language code. Consider you implement a microflow for human interaction. If you realize a hard-coded implementation using a UI technology, you could write a thin client Web UI or a fat client GUI, hard-code certain microflows for human interactions in a layer on top of the UI, and provide a service-based interface to that microflow layer, so that the hard-coded microflow implementations can be accessed from macroflows. Consider you want to perform simple changes to such a design, such as adding or deleting an activity in the microflow. Every change requires programming efforts. In a dynamic environment, where process changes are regular practice, this might not be acceptable.

The design described in this example incurs another drawback: It requires discipline from the developers. Developers must place every microflow in the SERVICE ABSTRACTION LAYER for microflows. If developers do not strictly follow such as design guidelines, the consequence is that microflow code is scattered through one or many components, and hence changes are even more difficult and costly to implement.

For these reasons, in highly dynamic business environments, a similar level of support for changing and redeploying microflows as provided for the macroflow models in a MACROFLOW ENGINE might be needed.

Even though rapid changes and avoiding scattered microflow implementation are the main reasons for requiring a better support for microflows, some other requirements for technology support exist, such as:

- In integrated tool suites, to provide a uniform user experience, tool vendors would like to provide a tooling that is similar to the macroflow tooling, including a modeling language for microflows.
- Even though microflows are short running processes, in some cases it might be necessary to monitor and manage the microflows. To provide monitoring and management for hard-coded microflows usually requires a substantial amount of work.
- Microflows also require recurring functionalities, such as realizing transaction semantics, accessing databases, or handling page flows. Hence, to reuse existing components providing these functionalities is useful.

Solution

Apply the business process paradigm directly to microflow design and implementation by using a MICROFLOW ENGINE that is able to execute the microflow models. The MICROFLOW ENGINE provides recurring tasks of the microflows as elements of the microflow modeling language. It supports change through changing of microflow models and redeployment to the engine. All microflows of a kind are handled by the same microflow engine.

Solution Details

Figure 12 illustrates the solution of the MICROFLOW ENGINE pattern.

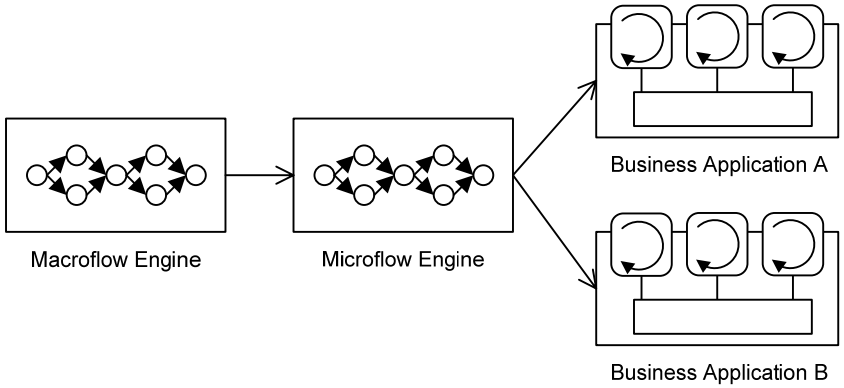

Fig. 12. Illustration of MICROFLOW ENGINE.

If a MICROFLOW ENGINE is used, the microflow processes are defined in a microflow modeling language. Processes can be flexibly changed through microflow deployment. The microflow logic is architecturally decoupled from the business applications and centrally handled in one place. The MICROFLOW ENGINE concentrates on orchestration issues of microflow activities but not on implementation of these activities. The actual implementation of microflow activities is delegated to functionality of integrated systems the engine communicates with or to human users.

There are two main kinds of MICROFLOW ENGINES corresponding to the two kinds of microflows:

- MICROFLOW ENGINE for automatic activities: These engines support full-automatic and transaction-safe integration processes. Hence, they offer functions for short-running transactional microflow execution. As integration processes usually must access other technologies or applications, many MICROFLOW ENGINES for automatic activities also support technology and application adapters, such as ODBC, JDBC, XML, Web service, SAP, or Siebel.
- MICROFLOW ENGINE for human interactions: These engines support pageflow handling functionalities. A pageflow defines the control flow for a set of UI pages. The pages usually display information, and contain controls for user interaction. Many pageflow engines focus on form-based input.

The microflow modeling language is a technical modeling language. In many cases, only a technical view of these models is exposed, but some tools also expose a high-level view of the integration processes. If this is the case, the DOMAIN-/TECHNICAL-VIEW pattern is realized by the microflow models. The goal could for instance be to enable designers and architects to gain a quick overview of the microflows. That is, here the domain view depicts a technical domain: either the integration behavior of the systems or the human interactions. Hence, the domain experts are software designers and architects.

Defining executable microflow models using a modeling language does not mean a MICROFLOW ENGINE must be used. An alternative is for instance to generate microflow execution code in a programming language using a model-driven code generator. Using a MICROFLOW ENGINE should be carefully considered, as it has some

disadvantages as well. Usually, it is not possible to define a custom microflow modeling language for existing engines, and many existing languages are much more complex than needed for very simple microflow orchestrations. This means additional effort, as developers, designers, and architects must learn the microflow modeling language. The MICROFLOW ENGINE is an additional technology which must be maintained. The additional engine component adds complexity to the system architecture.

The MICROFLOW ENGINE has the advantage that the models are accessible at runtime, e.g., for reflection on the models, and can be manipulated by redeployment. Management and monitoring of running processes is possible – either through an API or a management and monitoring tool. A tool suite similar to the macroflow tools can be provided. Recurring functionalities can be supported by the MICROFLOW ENGINE and reused for all microflows.

Example: Structural model of a MICROFLOW ENGINE

Figure 13 shows a simple example model of a MICROFLOW ENGINE design that could be used as a starting point if one would realize a MICROFLOW ENGINE from scratch. The basic feature of this MICROFLOW ENGINE design is execution of defined microflow integration process logic by orchestrating `Microflow Activities`. Analogous to the similar MACROFLOW ENGINE example presented before, each activity transforms `Control Data` that is used to control the orchestrations of microflow activities and invokes a `Function` of an `IT System`. The main difference to the previous example is: Here the functions are services exposed by IT Systems, not business-related functions (see the BUSINESS-DRIVEN SERVICE pattern for guidelines how to design the IT services). The invocations are performed automatically and in a transaction-safe way.

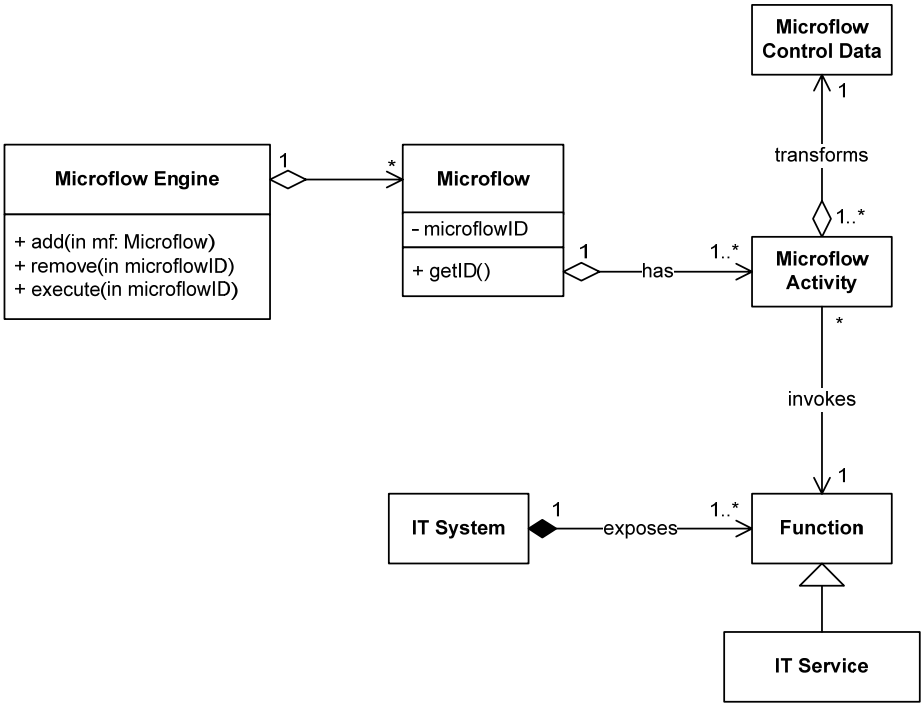

Fig. 13. Structural model of a MICROFLOW ENGINE.

Example: Java Page Flow Architecture

The previous example mainly illustrates a schematic design of a MICROFLOW ENGINE for automatic activities. A similar design could also be used as a core for a MICROFLOW ENGINE for human interactions. But additionally, we must define how to integrate the MICROFLOW ENGINE component into the UI architecture. Many UIs follow the MODEL-VIEW-CONTROLLER pattern (MVC) [Buschmann et al. 1996].

We want to illustrate one example design for the Java Page Flow Architecture which provides an implementation of a MICROFLOW ENGINE for human interactions. A Java Page Flow consists of two main components: controllers and forms. Controllers mainly contain a control flow, defined by so-called actions and forwards. The forms associated to the actions and forwards are mainly JSP pages.

In Figure 14, you see an example from [Mittal and Kanchanavally 2008], which shows a mapping of the Java Page Flow Architecture to MVC, as implemented in the Apache Beehive project. The main engine component, executing the microflows, is used as the MVC controller. JSP and the NetUI tag libraries are used to display the information in the view. Any model layer can be used, it is not determined by the pageflow engine. In this example architecture, the Controls technology from the Apache Beehive project is used as a model layer technology.

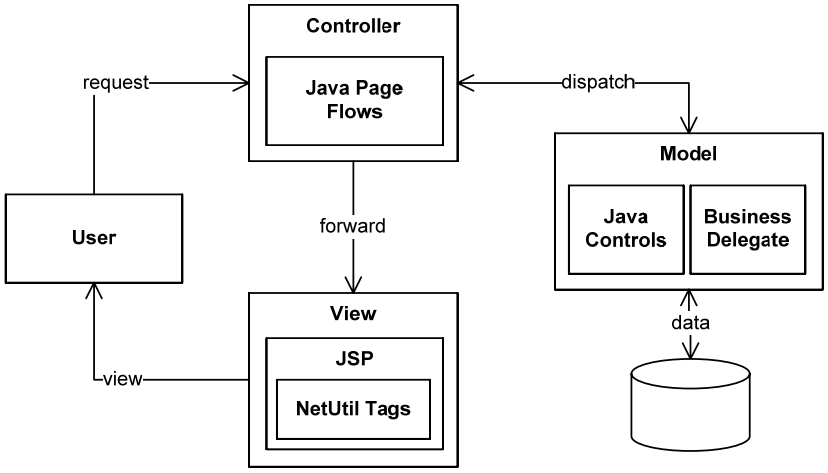

Fig. 14. Java Page Flow Architecture and MVC [Mittal and Kanchanavally 2008].

Known Uses

- The WebSphere Business Integration Message Broker and the WebSphere InterChange Server [IBM 2008] are both realizing MICROFLOW ENGINES. Both middleware products can also be used in conjunction. The WebSphere Business Integration Message Broker is used for simpler functions, such as adapter-based integration or dispatching. The product offers support for off-the-shelf adapters, message routing, and transformation. WebSphere InterChange Server offers transaction safe integration process execution. Process definition is done via a GUI, and the product also offers a very large set of INTEGRATION ADAPTERS for most common technologies and applications.
- webMethods' Integration Server (now integrated in the Fabric BPM suite) [webMethods 2007] provides a MICROFLOW ENGINE that supports various data transfer and Web services standards, including JSP, XML, XSLT, SOAP, and WSDL. Its offers a graphical modeler for microflows that models the microflow in a number of sequential steps (including loop steps and branching), as well as a data mapping modeler.
- iWay's Universal Adapter Suite [iWay 2007a] provides an Adapter Manager [iWay 2007b] for its intelligent, plug-and-play adapters. The Adapter Manager is a component that runs either stand-alone or in an EJB container and executes adapter flows. The basic adapter flow is: It transforms an application-specific request of a client into iWay's proprietary XML format, invokes an agent that might invoke an adapter or perform other tasks, and transforms the XML-based response into the application specific response format. The Adapter Manager provides a graphical modeling tool for assembling the adapters, the Adapter Designer. It allows developers to specify special-purpose microflows for a number of adapter-specific tasks, such as various transformations, routing through so-called agents, encryption/decryption, decisions, etc. Multiple agents, transformations, and

decisions can be combined in one flow. The Adapter Manager hence provides a special-purpose MICROFLOW ENGINE focusing on adapter assembly.

- The Java Page Flow Architecture, explained before, is a technology defining MICROFLOW ENGINES for human interactions. Apache Beehive is a project that implements the Java Page Flow Architecture using Java metadata annotations. The implementation is based on Struts, a widely-used MVC framework. BEA WebLogic Workshop is another implementation of the Java Page Flow Architecture, which is provides a declarative pageflow language.
- Novell's exteNd Director [Novell 2008] is a framework for rapid Web site development. It provides a page flow engine that orchestrates pageflows consisting of XForm pages.

4 Integration and Adaptation in Process-Driven SOAs

In the previous section we mainly discussed how to realize various types of executable process flow, macroflows and microflows, and how to connect them to services that realize functions in the processes. In a real-world SOA, usually not all services are implemented by the SOA developers, but in most cases a number of existing (legacy) systems, such as custom business applications, databases, and off-the-shelf business applications (such as SAP or Siebel), must be integrated.

Consider a typical starting point: Your organization uses two primary business applications. The first step to build a SOA orchestrating functions provided by those legacy applications is to provide them with a service-oriented interface. This is usually an incremental and non-trivial task. But let's assume we are able to find suitable business services to access these applications. In order to support orchestration through executable business processes, we will design high-level macroflows representing the business processes of the organization – from the business perspective. Following the MACRO-/MICROFLOW and DOMAIN-/TECHNICAL-VIEW patterns, the high-level macroflows are step-by-step refined into executable macroflows. Next, we realize the executable macroflows in a macroflow engine and use the macroflow activities to invoke the services exposed by the business applications. The result is an architecture as shown in the sketch below in Figure 15.

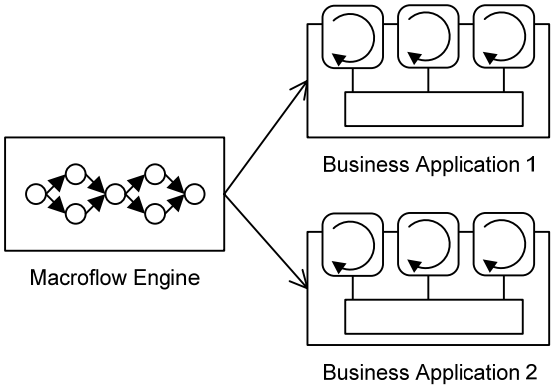

Fig. 15. Macroflow Engine and Business Applications.

Unfortunately, often the interfaces provided by the legacy business applications are not identical to what is expected in the business processes. The business application services expose the – often rather technical – interfaces of the legacy business applications. The macroflow processes, in contrast, require interfaces that correspond to the business activities in the processes. Changing the macroflows to use the technical interfaces does not make sense because we want to keep the macroflows understandable for business stakeholders. In addition, hard-wiring process activities to the volatile interfaces of backends is not useful, because for each change in the backend the process designs would have to be changed.

For these reasons, it makes sense to introduce INTEGRATION ADAPTERS for process integration, exposing the interfaces that the macroflows require (as shown in Figure 16). The INTEGRATION ADAPTER pattern translates between the interfaces of two systems connected using asynchronous (or if needed synchronous) connectors. The pattern also enables maintenance of both the connected target system and the adapter, by being suspendable and stoppable. Macroflow engine technology often provides such INTEGRATION ADAPTERS for connecting the processes to backend services. These adapters perform interface adaptations and data transformations, as well as data mapping tools to design the transformations.

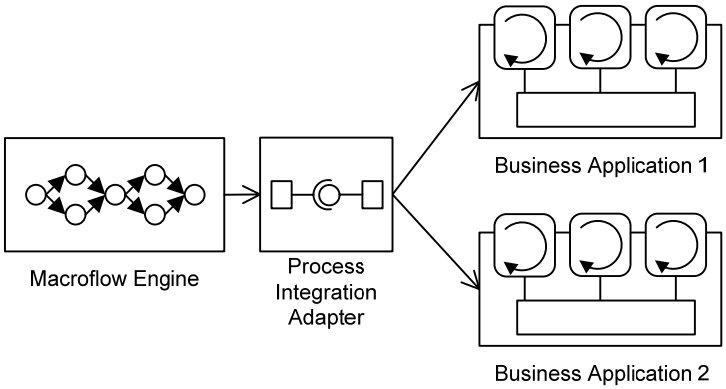

Fig. 16. Introducing a Process Integration Adapter.

In many cases, the abstraction of business application services through an adapter is not enough. Still, the macroflows contain technical issues that go beyond simple adaptations and data transformations, but rather deal with orchestration tasks. As explained in the MACRO-/MICROFLOW pattern, these technical flows should not be realized in a MACROFLOW ENGINE, but strictly distinguished from the macroflows – and realized as microflows. For such a small-scale architecture, it is usually enough to provide a few hard-coded services in a distinct microflow tier, as shown in Figure 17.

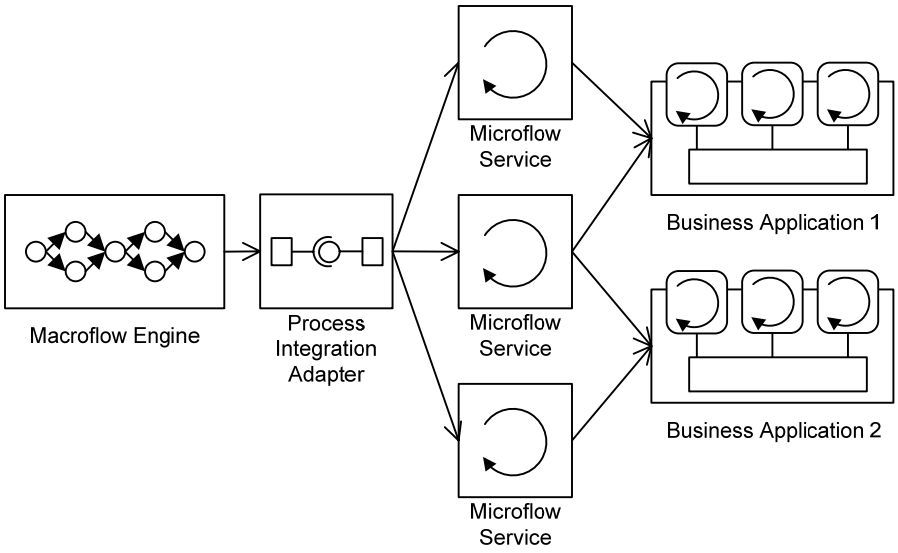

Fig. 17. Introducing a Microflows as Services.

In this architecture, the business applications are hard-wired in the service implementations. That means, if the applications need to be stopped for maintenance, the whole SOA must be stopped. If the application service interfaces need to be changed, all dependent services must be changed, too. This is ok for small SOAs with limited maintenance and availability requirements. But consider we require the SOA to continue to run, while new versions of the business applications are deployed. This can be resolved by applying the INTEGRATION ADAPTER pattern again: We provide INTEGRATION ADAPTERS for the business applications as illustrated in Figure 18.

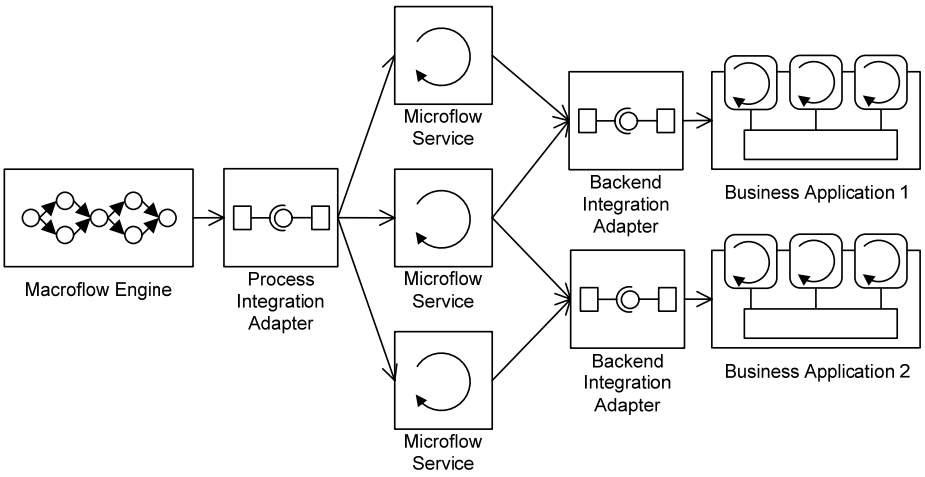

Fig. 18. Introducing Backend Integration Adapters.

Now consider we run this SOA for a while and our organization merges with another organization. That means the information system of that other organization needs to access our SOA. If the other organization uses explicit business processes as well, it is likely that it runs its own MACROFLOW ENGINE. We can perform the integration of the two systems by providing that other MACROFLOW ENGINE with a process integration adapter that integrates the microflow services of our SOA with the business activity interfaces required by the other organization's macroflows. The resulting architecture is sketched in Figure 19.

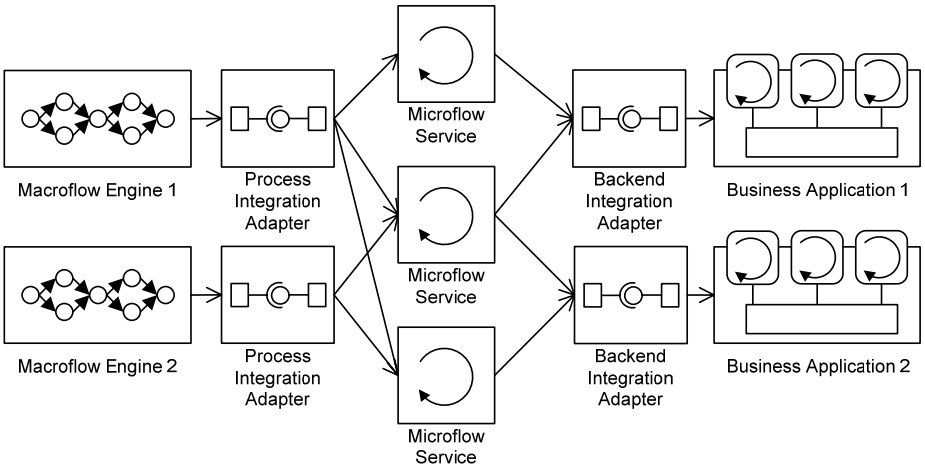

Fig. 19. Introducing multiple Macroflow Engines.

The macroflow tier is currently hard-wired to the technical tiers. If dynamic content-based routing to microflows and backends is needed or load balancing to multiple servers hosting the services should be provided, the introduction of a

CONFIGURABLE DISPATCHER (as shown in Figure 20) between macroflows tier and technical tiers can be beneficial to provide more configurability and flexibility. The CONFIGURABLE DISPATCHER pattern connects client and target systems using a configurable dispatch algorithm. Hence, it enables us to postpone dispatch decisions till runtime. It uses configurable dispatching rules that can be updated at runtime.

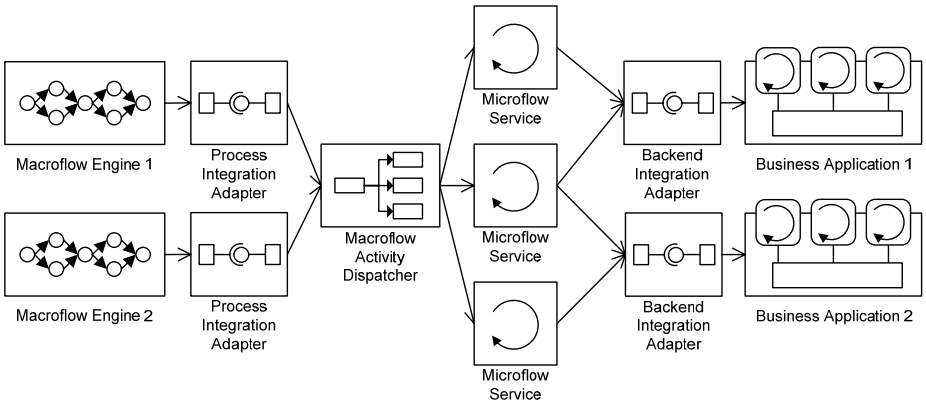

Fig. 20. Introducing a Macroflow Activity Dispatcher.

Over time, we might realize that more and more microflows are needed and more and more recurring tasks are performed in the microflows. In addition, it might make sense to make the microflow orchestrations more configurable. Hence, as a last step, we replace the microflow service tier by two MICROFLOW ENGINES: a page flow engine to realize the human interaction microflows and a message broker to realize the automated microflows. This is illustrated in Figure 21.

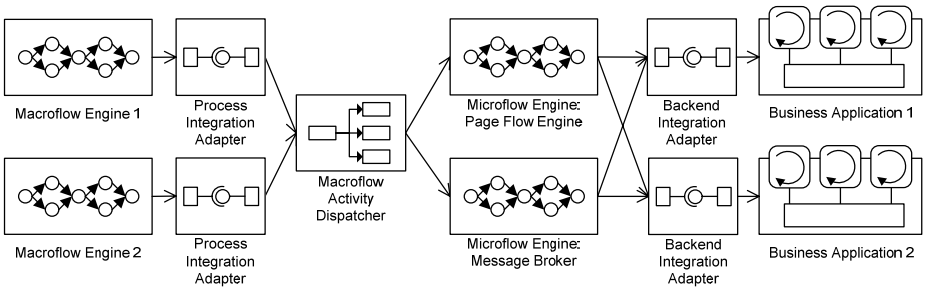

Fig. 21. Introducing Microflow Engines.

In our SOA, we have applied multiple INTEGRATION ADAPTERS that must be maintained and managed. Consider further the organization develops other SOAs for other business units that use similar technologies and must operate on similar backends. Then it makes sense to introduce an INTEGRATION ADAPTER REPOSITORY for the backend adapters. The INTEGRATION ADAPTER REPOSITORY pattern provides a central repository and maintenance interface for INTEGRATION ADAPTERS that supports management, querying, and deployment of adapters. It hence facilitates reuse of adapters.

The sketched, incrementally built architecture in this example follows the PROCESS INTEGRATION ARCHITECTURE pattern. This pattern is an architectural pattern that defines a specific configuration using a number of other patterns. It explains a specific architectural configuration of how the other patterns can be assembled to a flexible and scalable SOA.

One of the primary advantages of following the PROCESS INTEGRATION ARCHITECTURE pattern is that is enable architects to build up a SOA incrementally – just as in this example walkthrough. A process-driven SOA initiative is usually a large-scale project in which multiple new technologies must be learned and integrated. Hence, step-by-step introduction of extensions, following an approach that is known to scale well to larger SOAs, is highly useful.

4.1 Pattern: INTEGRATION ADAPTER

Context
In a SOA, various systems need to be connected to other systems. For instance, in a process-driven SOA, among others, the MACROFLOW ENGINES, MICROFLOW ENGINES, business services, and backend systems must be connected. The systems in a process-driven SOA are heterogeneous systems, consisting of diverse technologies running on different platforms and communicating over various protocols. When different systems are interconnected and the individual systems evolve over time, the system internals and sometimes even the public interfaces of these systems change.

Problem
How can heterogeneous systems in a SOA be connected and the impacts of system and system interface changes kept in acceptable limits?

Problem Details
Connecting two systems in a SOA means that a client system must be aligned with a target system that captures the requests, takes over the task of execution, and generates a response. For instance, if a MACROFLOW ENGINE or MICROFLOW ENGINE is the client, it acts as a coordinator of activities. Some of these activities are tasks that need to be executed by some other system. But, in many cases, the target systems are different to what is expected in the process engine. For instance, different technology, different synchronization mechanisms, or different protocols are used. In addition, in case of asynchronous communication, we must provide a way to connect heterogeneous technologies in such a way that the client can correlate the response to the original request.

One important consideration, when connecting systems in a SOA, is the change impact. Changes should not affect the client of a system, if possible. For instance, changes to integrated systems should not have effects on the processes that run in process engines.

In many change scenarios, downtimes of the whole SOA for maintenance are not tolerable. That is, changing a system should not mean that the other systems of the SOA must be stopped, but they should be able to continue to work, as if the changed

system would still be functioning. Apart from this issue, internal system changes can be tolerated in a SOA as long as the public interfaces exposed as services do not change.

However, many changes include interface change. Often the public interface of a system changes with each new release of the system. In this context of ongoing change and maintenance, the costs and efforts of changes should be kept at a minimum level. The impact of changes and the related testing efforts must also be kept within acceptable limits.

If your organization is in control of the system that must be changed, sometimes it is possible to circumvent these problems by avoiding changes that influence other systems. However, in a SOA usually many systems by external vendors or open source projects are used. Examples are backend systems, such as databases, SAP, or Siebel, as well as SOA components, such as MACROFLOW ENGINES and MICROFLOW ENGINES. Changes cannot be avoided for these systems. Migration to a new release is often forced as old releases are not supported anymore, or the new functionality is simply required by the business.

Apart from migration to a new version, the problem also occurs if a system shall be replaced by a completely different system. In such cases, the technology and functional interfaces of the new system are often highly different, causing a significant change impact.

Solution

If two systems must be connected in a SOA and keeping the change impacts in acceptable limits is a goal for this connection, provide an INTEGRATION ADAPTER for the system interconnection. The adapter contains two connectors: One for the client system's import interface and one for the target system's export interface. Use the adapter to translate between the connected systems, such as interfaces, protocols, technologies, and synchronization mechanisms, and use CORRELATION IDENTIFIERS to relate asynchronous requests and responses. Make the adapter configurable, by using asynchronous communication protocols and following the COMPONENT CONFIGURATOR pattern [Schmidt et al. 2000], so that the adapter can be modified at runtime without impacting the systems sending requests to the adapter.

Solution Details

Figure 22 illustrates the solution of the INTEGRATION ADAPTER pattern.

The core of the solution of the INTEGRATION ADAPTER pattern is the same as in the classical, object-oriented ADAPTER pattern [Gamma et al. 1994]: An adapter connects the interfaces of a client and a target, and translates between the two interfaces. For instance, if the client is a process engine, it acts as a sender in terms of sending out requests for activity execution, which are received by the adapter and transformed into a request understood by the target system. The INTEGRATION ADAPTER pattern adds to the solution of the ADAPTER pattern by supporting integration at the architectural level of connecting distributed and heterogeneous systems.

In the ADAPTER pattern, invocations are mainly synchronous, object-oriented message calls in the local scope. An INTEGRATION ADAPTER must consider in first place distributed requests and responses, which can be send either synchronously or

asynchronously. Receiving a request or response can work via push or pull mechanisms. The request contains an identifier for the function to be executed and input parameters. The INTEGRATION ADAPTER transforms the request into a format that can be understood by the target system's interface and technology. The request will be forwarded to the target system after the transformation is performed. After the adapter has received a response of the target system, the response is transformed back into the format and technology used by the interface of the client.

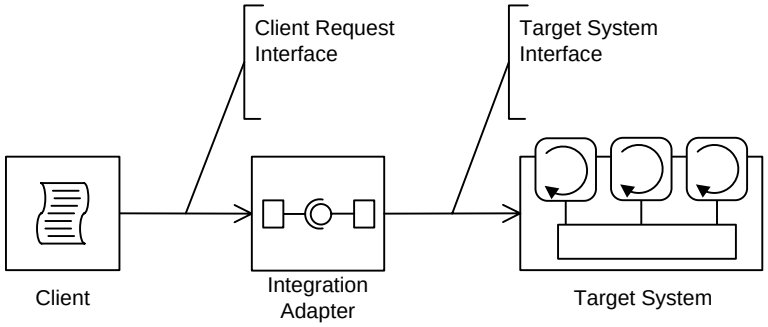

Fig. 22. Illustration of INTEGRATION ADAPTER.

To make the INTEGRATION ADAPTER maintainable at runtime, the COMPONENT CONFIGURATOR pattern [Schmidt et al. 2000] should be applied. That is, the adapter offers a configuration interface, which supports stopping and suspending the adapter. The adapter is stopped, when new versions of the adapter must be deployed. The adapter is suspended, when new versions of the target system are deployed or the adapter is configured at runtime. Later on, after maintenance activities are finished, the adapter can resume its work and process all requests that have arrived in the meantime. The INTEGRATION ADAPTER can also offer a finalization function such that it finishes all ongoing activities properly and then terminates itself.

To realize an adapter with a COMPONENT CONFIGURATOR interface, the adapter must be loosely coupled to other systems, which is achieved by using connectors to client and target systems, as well as asynchronous communication protocols. As requests must be accepted at any time, no matter whether an adapter is at work or temporally suspended, an asynchronous connector should be used to receive the requests and to send the responses. That is, the connector must be decoupled from the adapter to still accept requests in case the adapter is not active.

Basically, asynchronous communication is only required on client side, i.e., for systems that access the adapter. The target system does not necessarily need to be connected asynchronously. For instance, a connected system might only offer a synchronous interface, or the system is a database which is connected via synchronous SQL. That also means, the connector may accept requests and queue them until they are processed by the adapter.

In case of asynchronous communication, requests and responses are related by applying the CORRELATION IDENTIFIER pattern [Hohpe et al. 2003]. That is, the client sends a CORRELATION IDENTIFIER with the request. The adapter is responsible for putting the same CORRELATION IDENTIFIER into the respective response, so that the

client can relate the response to its original request. For instance, if the client is a process engine, the CORRELATION IDENTIFIER identifies the activity instance that has sent the request.

If supported by the target system, the CORRELATION IDENTIFIER will also be used on the target system's side to relate the response of the target system back to the original request. Consequently, the target system will have to send the CORRELATION IDENTIFIER back in its own response so that the adapter can re-capture it. The response will also contain the result of the execution. If CORRELATION IDENTIFIERS cannot be used with the target system, for instance, because it is a legacy system that we cannot change, the INTEGRATION ADAPTER must implement its own mechanism to align requests and results.

The transformations performed by the adapter are often hard-coded in the adapter implementation. In some cases, they need to be configurable. To achieve this, the adapter can implement transformation rules for mapping a request including all its data to the interface and request format of the target system. Transformation rules can also be used for the response transformations. Data mapping tools can be provided to model such transformation rules.

An INTEGRATION ADAPTER is very useful for flexible integration of business applications from external vendors. It also gets more popular to provide interconnectivity by supporting generic adapters for common standards, such as XML and Web Services. That is the reason why many vendors deliver such adapters off-the-shelf and provide open access to their APIs. As standard adapters can be provided for most common standards or products, solutions following the INTEGRATION ADAPTER pattern are usually reusable.

One drawback of INTEGRATION ADAPTERS is that potentially many adapters need to be managed, if many systems exist where adapters for different purposes, systems, or technologies are required. Hence, maintenance and deployment of adapters might become problematic and must be done in a controlled way. The INTEGRATION ADAPTER REPOSITORY offers a way to manage adapters in a centralized and controlled way.

If an adapter is suspended for a long time or if the amount of requests sent to a suspended adapter is very high, then the request queue may contain large amounts of requests that take a long time to be processed or the requests may even have timed out. The workload of requests and the amount of requests that an adapter can process must be in balance. Middleware is required to queue the requests.

Example: Simple example structure of an INTEGRATION ADAPTER

Figure 23 shows an exemplary model for the internal design of INTEGRATION ADAPTER. This adapter receives asynchronous requests from a client system and translates them into synchronous requests for a target system. While waiting for the response, the adapter stores the CORRELATION IDENTIFIER sent by the client and adds it to the respective response message that is sent back to the client. The INTEGRATION ADAPTER offers an API for adapter configuration:

- The adapter can be initialized with `init`.
- The adapter can be stopped with `finalize`.

- The connected target system can be maintained. Then the adapter must be suspended using the `suspend` operation, and after the maintenance it can `resume`.
- The adaptation status can be queried with `info`.

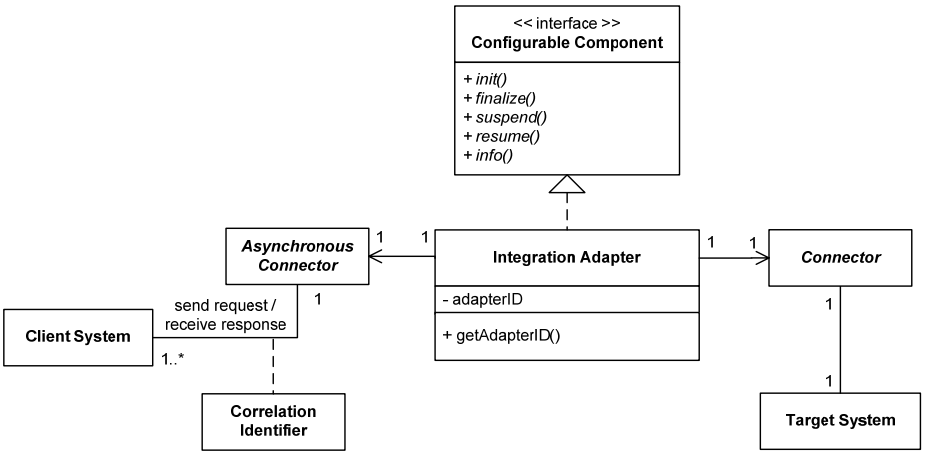

Fig. 23. Example structure of an INTEGRATION ADAPTER

Example: Process Integration Adapter

Let us consider now a slightly more complex example of an INTEGRATION ADAPTER: An adapter that connects a process engine (i.e., a MACROFLOW ENGINE or MICROFLOW ENGINE) to a target system. Using INTEGRATION ADAPTERS for process integration has the benefit of a clear model for the communication between a process engine and the connected target systems.

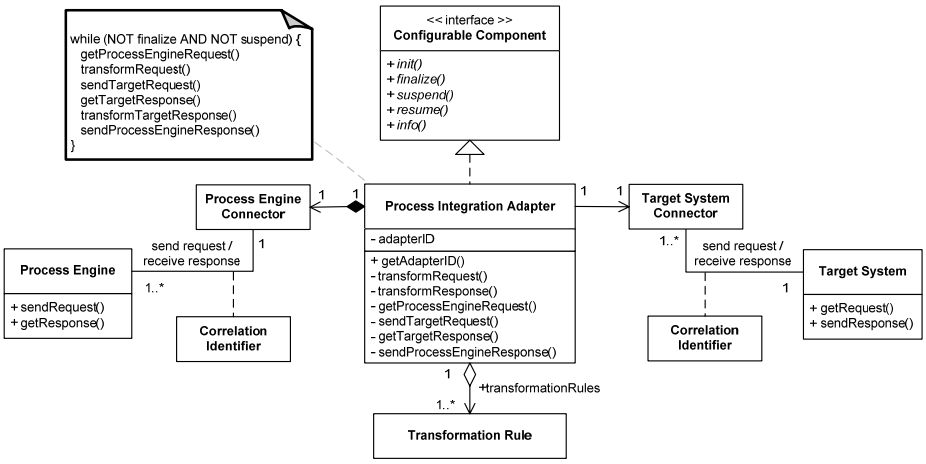

Fig. 24. Example structure of a process integration adapter

In this example, both connectors are asynchronous. The adapter must translate between the two CORRELATION IDENTIFIERS. The adapter uses the same interface for configuration, as in the previous example. It follows a predefined protocol of a few operations to perform the adaptation.

Both request and response message are transformed using transformation rules. Many process engines offer data mapping tools for graphical design of the transformation rules. Figure 24 illustrates the structure of the process integration adapter.

The process integration adapter has a straightforward adaptation behavior, as shown in the following sequence diagram in Figure 25.

Known Uses

- WebSphere InterChange Server [IBM 2008] offers a very large set of INTEGRATION ADAPTERS for most common technologies and applications. Users can extend the set of adapters with self-defined adapters.
- The transport providers of the Mule ESB [Mule 2007] provide INTEGRATION ADAPTERS for transport protocols, repositories, messaging, services, and other technologies in form of their connectors. A connector provides the implementation for connecting to an external system. The connector sends requests to an external receiver and manages listeners to receive responses from the external system. There are pre-defined connectors for HTTP, POP3/SMTP, IMAP, Apache Axis Web Services, JDBC, JMS, RMI, and many other technologies. Components can implement a common component lifecycle with the following lifecycle interfaces: Initialisable, Startable, Callable, Stoppable, and Disposable. The pre-defined connectors implement only the Disposable and Initialisable interfaces.
- iWay's Universal Adapter Suite [iWay 2007a] provides so-called intelligent, plug-and-play adapters for over 250 information sources and broad connectivity to multiple computing platforms and transport protocols. It provides a repository of adapters, a special-purpose MICROFLOW ENGINE for assembling adapters called the Adapter Manager, a graphical modeling tool for adapter assembly, and integration with the MACROFLOW ENGINE and EAI frameworks of most big vendors.
- WebSphere MQ Workflow [IBM 2008] offers a technical concept called a User-Defined Program Execution Server (UPES), which implements this pattern for process integration. The UPES concept is a mechanism for invoking services via XML-based message adapters. Basis of the UPES concept is the MQ Workflow XML messaging interface. The UPES concept is all about communicating with external services via asynchronous XML messages. Consequently, the UPES mechanism invokes a service that a process activity requires, receives the result after the service execution has been completed, and further relates the asynchronously incoming result back to the process activity instance that originally requested execution of the service (as there may be hundreds or thousands of instances of the same process activity).

- CSC offers within their e4 reference meta-architecture the concept of INTEGRATION ADAPTERS for process integration. For an insurance customer in the UK the e4 adapter concept has been used to integrate FileNet P8 Business Process Manager with an enterprise service bus based on WebSphere Business Integration Message broker.
- Within the Service Component Architecture (SCA) concept of IBM's WebSphere Integration Developer various INTEGRATION ADAPTERS are offered off-the-shelf, e.g., for WebSphere MQ, Web services, or JMS.

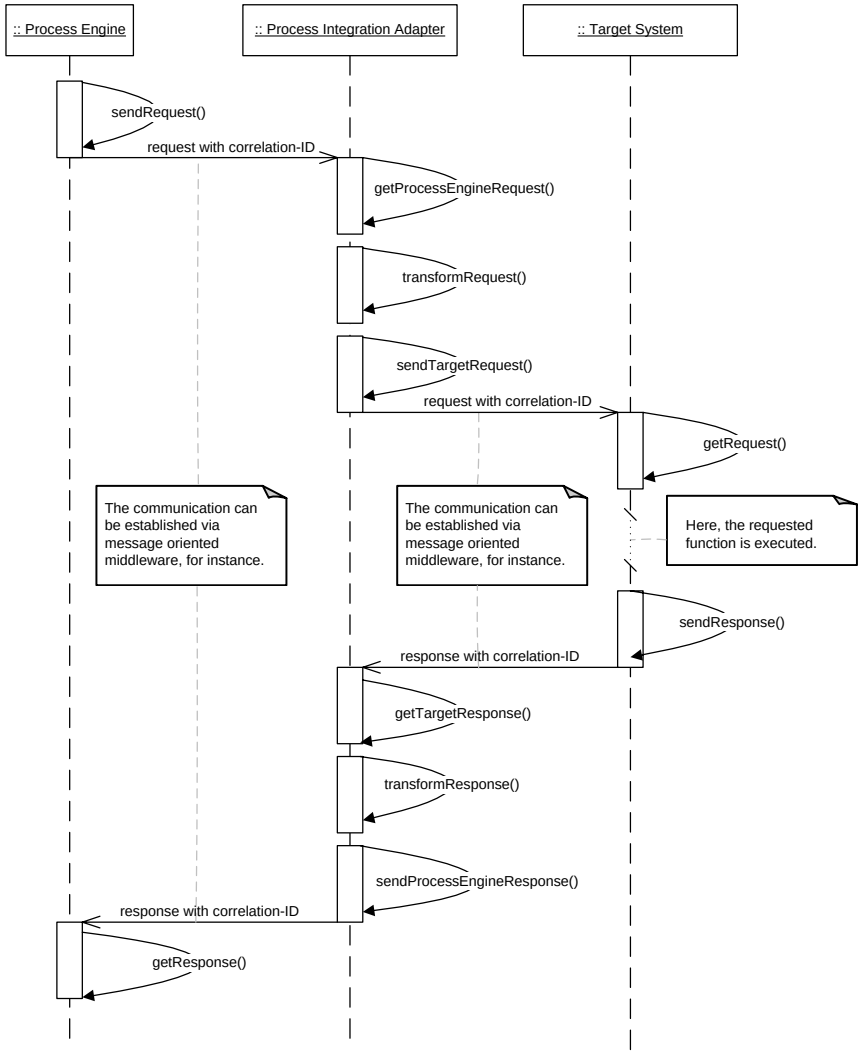

Fig. 25. Behavior of a process integration adapter

4.2 Pattern: INTEGRATION ADAPTER REPOSITORY

Context
Various systems shall be connected via INTEGRATION ADAPTERS. That means, a large number of adapters is used or can potentially be used in a SOA.

Problem
How can a large number of INTEGRATION ADAPTERS be maintained and managed?

Problem Details
INTEGRATION ADAPTERS are important to connect systems that have incompatible interfaces and to minimize the change impact, when multiple systems are integrated. But with each system integrated into a SOA, the number of adapters to be maintained grows. In addition, when the adapters evolve, new adapter versions need to be supported as well, meaning that actually multiple versions of each adapter need to be maintained and managed.

Not always the organization running the SOA also provides the adapters. Especially for standard software, vendors offer INTEGRATION ADAPTERS. The result is often a large set of reusable standard adapters. Reusable adapter sets can also be built inside an organization, for instance, if the organization builds multiple SOAs and wants to reuse the adapters from previous projects. To facilitate reuse of adapters, it should be possible to search and query for an adapter or an adapter version in such a larger adapter set.

Managing multiple INTEGRATION ADAPTERS also introduces a deployment issue: Usually connected systems should not be stopped for deploying a new adapter or adapter versions. Instead it should get "seamlessly" deployed at runtime. That means, tools should support seamless deployment.

The problem of INTEGRATION ADAPTER maintenance and management especially occurs in larger architectural contexts, where different systems have to communicate and larger sets of adapters exist. The problem does not have such a great impact within the boundaries of one closed component or application, as the whole component or application needs to be redeployed if changes are made.

Solution
Use a central repository to manage the INTEGRATION ADAPTERS as components. The INTEGRATION ADAPTER REPOSITORY provides functions for storing, retrieving, and querying of adapters, as well as adapter versioning. It also provides functions for automatic deployment or supports automatic deployment tools. The automatic deployment functions use the COMPONENT CONFIGURATOR interface of the INTEGRATION ADAPTERS to suspend or stop adapters for maintenance. The functions of the repository are offered via a central administration interface.

Solution Details
Figure 26 illustrates the solution of the INTEGRATION ADAPTER REPOSITORY pattern.

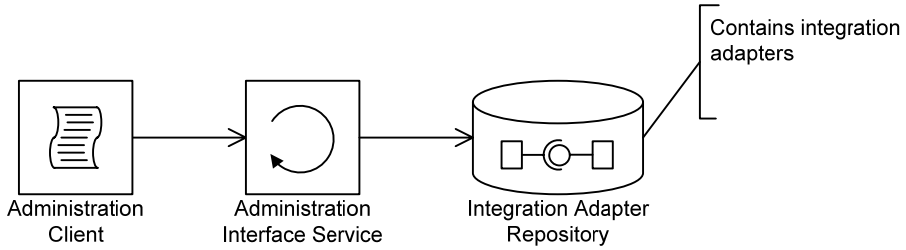

Fig. 26. Illustration of INTEGRATION ADAPTER REPOSITORY.

The INTEGRATION ADAPTERS are stored in a central repository that offers operations to add, retrieve, and remove adapters in multiple versions. Optionally, the repository can provide functions to search for adapters and adapter versions by given attributes.

In the simple case, the INTEGRATION ADAPTER REPOSITORY just identifies the adapter by adapter ID (name) and version. More sophisticated variants support metadata about the adapters as well.

The INTEGRATION ADAPTER REPOSITORY can be used to support adapter deployment. In the simplest form it fulfills tasks for external deployment tools, such as delivering the right adapter in the right version. But it can also provide the deployment functions itself.

The automatic deployment functions use the COMPONENT CONFIGURATOR interface of the INTEGRATION ADAPTERS. That is, maintenance or deployment tasks are supported because each single adapter can be stopped and restarted, new adapters or adapter versions can be deployed, and old adapters can be removed via a centralized administration interface.

It is important that requests sent to adapters are processed asynchronously (see INTEGRATION ADAPTER pattern) to bridge maintenance times when the adapters are modified. The requests are queued while the adapter is suspended. The pending requests can be processed when the adapter restarts work after maintenance, or after an adapter is replaced by a new adapter. The deployment functions must trigger this behavior of the adapters.

The INTEGRATION ADAPTER REPOSITORY can pattern addresses the flexible management of adapters at runtime. Following the pattern, changes to adapters can be deployed rather quickly and easily.

However, the pattern requires changing the adapters because a configuration interface is necessary for maintaining the adapters. As all adapters must implement the interface needed by the repository, putting third-party adapters with a different interface into the repository is not trivial. In some cases, it is impossible to add the required configuration functions to the third-party adapter; in other cases, writing a wrapper for the third-party adapter's interface is required.

Example: Simple integration adapter repository design

Figure 27 shows the simplest INTEGRATION ADAPTER REPOSITORY design. In this design the INTEGRATION ADAPTERS are just managed and retrieved using the adapter ID.

This design can easily be extend with more sophisticated search and query options. For instance, we could add metadata about the adapters. Using simple versioning we could further improve this repository design.

At the moment the provided administration interface only supports deployment by delivering the adapter using get. More sophisticated deployment functionality could be added that can stop a running adapter, deploy a new adapter, and initialize that adapter then.

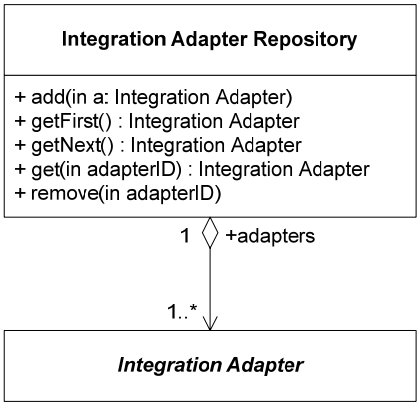

Fig. 27. Illustration of INTEGRATION ADAPTER REPOSITORY.

Known Uses

- WebSphere InterChange Server [IBM 2008] offers an INTEGRATION ADAPTER REPOSITORY in which a pre-defined large set of INTEGRATION ADAPTERS is provided. Self-defined adapters can also be added.
- The connectors of transport providers of the Mule ESB [Mule 2007] are, like all other components in Mule, managed either by the Mule container or an external container like Pico or Spring. The container manages the lifecycle of the connectors using the component lifecycle interfaces, which the components can optionally implement. Thus the container acts as an INTEGRATION ADAPTER REPOSITORY for the connectors.
- iWay's Universal Adapter Suite [iWay 2007a] provides a repository of adapters in the Adapter Manager [iWay 2007b]. The graphical modeler of iWay, the Adapter Designer, is used to define document flows for adapters. The Adapter Designer can be used to maintain and publish flows stored in any Adapter Manager repository. The adapters in the repository can be deployed to the Adapter Manager, which is the MICROFLOW ENGINE used for executing the Adapter flows.

4.3 Pattern: CONFIGURABLE DISPATCHER

Context
In a SOA, multiple systems need to be integrated. Not always you can decide at design time or deployment time, which service or system must execute a request.

Problem
How to decide at runtime which service or system has to execute a request?

Problem Details
There are numerous issues that require a decision about request execution at runtime. Some examples are:

- As system architectures usually change over time, it is necessary to add, replace, or change systems in the backend for executing process activities. In many process-driven systems, this must be possible at runtime. That is, it must be dynamically decided at runtime which component has to execute a request, e.g., sent by a macroflow activity. If the architecture does not consider these dynamics, then modifications to the backend structures will be difficult to implement at runtime.
- Scalability can be achieved through load balancing, meaning that multiple services on different machines are provided for serving the same type of requests. Depending on the load, it must be dynamically decided using a load balancing scheme or algorithm which service is invoked.
- Sometimes for the same functionality, multiple systems are present in an organization. For instance, if two or more organizations have merged and the information systems have not yet been integrated, then it is necessary to decide based on the content of a request to which system the request must be routed. For instance, if multiple order handling systems are present, orders can be routed based on the product IDs/categories.
- If some functionality is replicated, for instance to support a hot stand-by server, requests must be sent to all replicas.

All these issues actually point to well known issues in distributed architectures and can be conceptually classified as dimensions of transparency [Emmerich 2000]: access transparency, location transparency, migration transparency, replication transparency, concurrency transparency, scalability transparency, performance transparency, and failure transparency. The core problem is thus how to consider those dimensions of transparency appropriately.

One important aspect of handling dynamic request execution decisions properly is that the rules for these decisions can also change at runtime. For instance, consider we change the system architecture, add more servers for load balancing, require different content-based routing rules, or add additional replicas. In all these cases, the rules for routing the requests change.

Solution
Use a CONFIGURABLE DISPATCHER that picks up the incoming requests and dynamically decides on basis of configurable dispatching rules, where and when the request should be executed. After making the decision, the CONFIGURABLE DISPATCHER forwards the requests to the corresponding target system that handles the request execution. New or updated dispatching rules can be deployed at runtime.

Solution Details
Figure 28 illustrates the solution of the CONFIGURABLE DISPATCHER pattern.

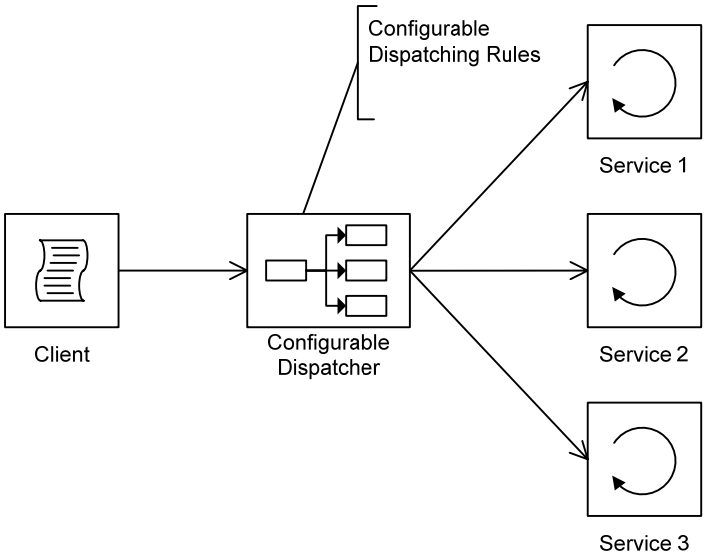

Fig. 28. Illustration of CONFIGURABLE DISPATCHER.

The dispatcher decides based on dispatching rules. The term "rule" is not very strictly defined, however. Any directive that can decide – based on an incoming request – how to handle the request can be used. For instance, the rules can be implemented as event-condition-action rules and a rule engine can be used to interpret the rules. Another implementation variant is to embed a scripting language interpreter and execute scripts that perform the decision.

In any case, the rules must be triggered upon dispatching events (mainly incoming requests). They must be able to evaluate conditions. That is, the rule engine or interpreter must be able to access the relevant information needed for evaluating the conditions. For instance, if content-based routing should be supported, the content of the request must be accessible in the rule implementation. If a round-robin load balancing should be implemented, the accessible target systems as well as a state of the round-robin protocol need to be accessed. Finally, functionality to realize the

decision is needed, such as a command that tells the dispatcher to which target system it should dispatch the request.

The CONFIGURABLE DISPATCHER pattern supports the flexible dispatch of requests based on configurable rules. These dispatching rules can be changed at runtime. Dynamic scripting languages or rule engines enable developers to update dispatching rules on the fly. If this is not possible, the dispatcher can apply the COMPONENT CONFIGURATOR pattern [Schmidt et al. 2000] to suspend dispatching, while the rules are updated. In any case, the dispatcher should provide a dynamic rule maintenance interface.

The dispatcher also has the task to pick up the request result from the component and send it back to the adapter. It is optionally possible to apply dispatching rules for the results as well. If asynchronous communication is used, a CORRELATION IDENTIFIER [Hohpe et al. 2003] is used to correlate the requests and responses.

The CONFIGURABLE DISPATCHER pattern can be used to make the workload in a SOA manageable by scaling the architecture in terms of adding instances of services/systems to execute the requests. However, a central component like a CONFIGURABLE DISPATCHER is a single-point-of-failure. It might be a bottleneck and hence have a negative influence on the performance of the whole system.

Example: Simple asynchronous CONFIGURABLE DISPATCHER

Figure 29 shows an exemplary model for an asynchronous CONFIGURABLE DISPATCHER. The dispatching rules are simply stored in aggregated objects. The dispatcher design uses a CONFIGURABLE COMPONENT interface to suspend the dispatcher while the dispatch rules are updated. The dispatcher follows a simple linear algorithm to forward requests and responses (of course, this algorithm can also be parallelized). The CORRELATION IDENTIFIER pattern is used to correlate asynchronous requests and responses.

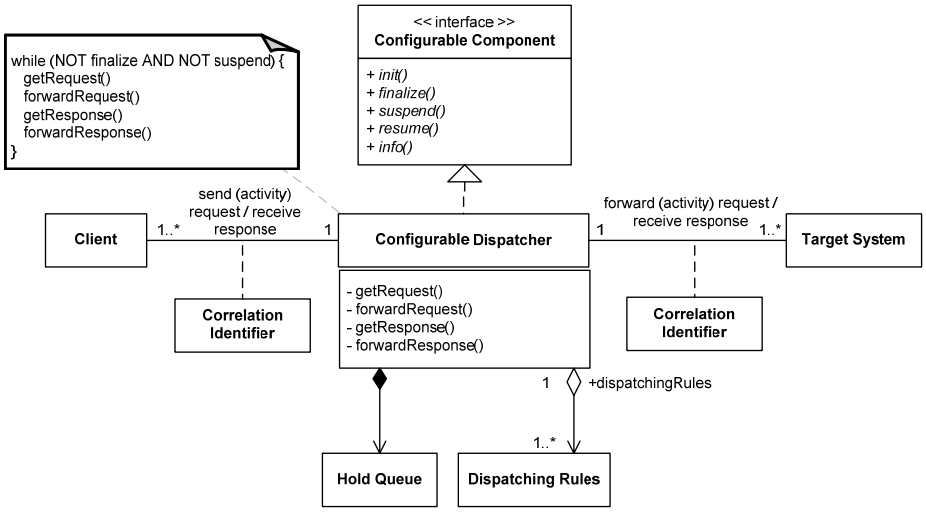

Fig. 29. Simple asynchronous CONFIGURABLE DISPATCHER.

Known Uses

- Using IBM's WebSphere Business Integration Message Broker [IBM 2008] a CONFIGURABLE DISPATCHER can be implemented with a message flow definition that represents the dispatching logic. The dispatching rules are stored in a database and are accessed via a database access node in the flow.
- The Service Container of the Mule Enterprise Service Bus [Mule 2007] offers support for content-based and rule-based routing. Inbound and outbound message events, as well as responses, can be routed according to declarative rules that can be dynamically specified. A number of predefined routers are available (based on the patterns in [Hohpe et al. 2003]). Pre-defined (or user-defined) filters, like a payload type filter or an XPath filter, can be used to express the rules that control how routers behave.
- Apache ServiceMix [ServiceMix 2007] is an open source Enterprise Service Bus (ESB) and SOA toolkit. It uses the rule-language Drools to provide rule-based routing inside the ESB. The architecture is rather simple: A Drools component is exposed at some service, interface, or operation endpoint in ServiceMix and it will be fired, when the endpoint is invoked. The rule base is then in complete control over message dispatching.

4.4 Pattern: PROCESS INTEGRATION ARCHITECTURE

Context

Process technology is used and the basic design follows the MACRO-/MICROFLOW pattern. Process technology is used at the macroflow level, and backend systems need to be integrated in the process flow. The connection between the macroflow level and the backend systems needs to be flexible so that different process technologies can (re-)use the connection to the backend systems. The architecture must be able to cope with increased workload conditions, i.e., it must be scalable. Finally, the architecture must be changeable and maintainable to be able to cope with both changes in the processes and changes in the backends. All those challenges cannot be mastered without a clear concept for the whole SOA.

Problem

How to assemble a process-driven SOA in way that is flexible, scalable, changeable, and maintainable?

Problem Details

To properly consider the qualities attributes flexibility, scalability, changeability, and maintainability a number of issues must be addressed. First, there are technology specifics of the process technology being used at the macroflow level. In principle, implementations of macroflow activities represent reusable functions that are not restricted to one specific process technology but which can rather be used with different types and implementations of process engines. If the process technology is tightly coupled to implementations of activities, changes in the process technology

may potentially have larger impact on the corresponding activity implementations which means a loss of flexibility.

Activities at the macroflow level are usually refined as microflows following the MACRO-/MICROFLOW pattern. Thus, one has to consider where and how these microflows are executed. Aspects of scalability must be considered to cope with increasing workload. As requests for activity execution are permanently initiated and business will usually go on day and night, we additionally have to deal with the question: What further mechanisms are necessary to maintain the whole architecture at runtime?

Changes to the microflow and macroflow should be easy and of low effort. Actual backend system functionality will be invoked at the microflow level, and it is obviously an issue how this can be achieved, as those backend systems are in principle independent and are subject to individual changes themselves. The impact of these changes must be kept within acceptable limits, in a way that those changes can be managed.

Solution

Provide a multi-tier PROCESS INTEGRATION ARCHITECTURE to connect macroflows and the backend systems that need to be used in those macroflows. The macroflows run in dedicated macroflow engines that are connected to the SOA via INTEGRATION ADAPTERS for the connected services. Microflows are realized in a distinct microflow tier, and they either run in dedicated MICROFLOW ENGINES or are implemented as microflow services. The backend systems are connected to the SOA using INTEGRATION ADAPTERS, too. To cope with multiple backends, multiple microflow engines, as well as for replication and load balancing, CONFIGURABLE DISPATCHERS are used.

Solution Details

Figure 30 illustrates the solution of the PROCESS INTEGRATION ARCHITECTURE pattern.

The PROCESS INTEGRATION ARCHITECTURE pattern assumes service-based communication. That is, the systems connected in PROCESS INTEGRATION ARCHITECTURE are exposed using service-oriented interfaces and use services provided by other systems in the PROCESS INTEGRATION ARCHITECTURE to fulfill their tasks. In many cases, asynchronous communication is used, to facilitate loosely coupling. Then usually CORRELATION IDENTIFIERS are used to correlate the requests and results. Sometimes it makes sense to use synchronous communication, too, for instance because blocking on a results is actually required or a backend in batch mode can only work with synchronous invocations.

The PROCESS INTEGRATION ARCHITECTURE provides a flexible and scalable approach to service-oriented and process-driven architectural design. The main architectural task of a PROCESS INTEGRATION ARCHITECTURE is to connect the macroflows, representing the executable business processes, to the backend systems and services providing the functions needed to implement these processes. In a naïve approach to architectural design, we would simply invoke the services (of the backend systems) from the macroflow activities running in the MACROFLOW ENGINE. But this only works well for small examples and very small-scale architectures. The benefit of

the PROCESS INTEGRATION ARCHITECTURE is that we can start from this very simple architecture and step-by-step enhance it, as new requirements emerge. The enhancements are described by the other patterns of this pattern language.

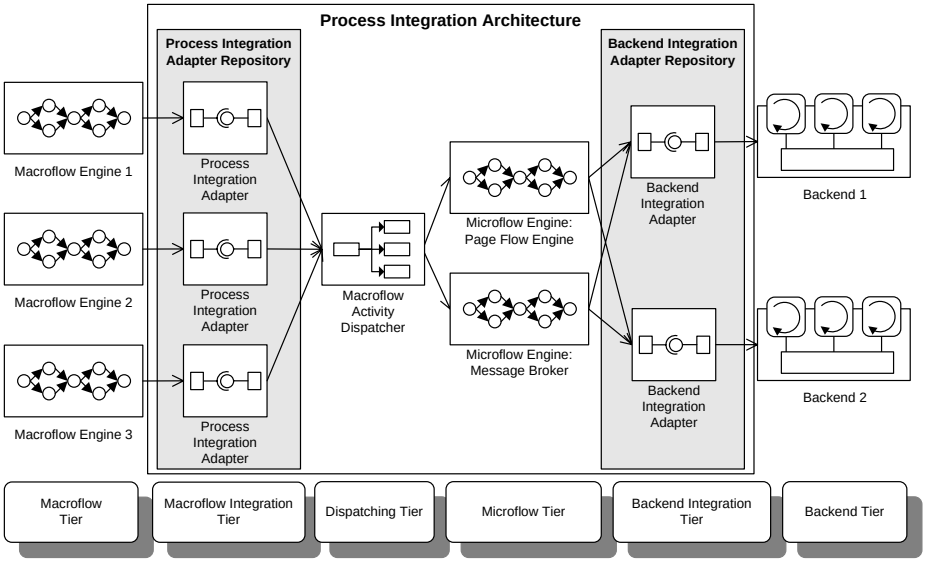

Fig. 30. Illustration of PROCESS INTEGRATION ARCHITECTURE.

The process integration architecture introduces multiple tiers:

- The *Macroflow Tier* hosts the implementations of the executable macroflows. Usually MACROFLOW ENGINES are used to execute the macroflows.
- The *Macroflow Integration Tier* is a common extension to the Macroflow Tier. It introduces one INTEGRATION ADAPTER for the processes per MACROFLOW ENGINE. This adapter integrates the process activities with the technical functions provided by the SOA. That is, it connects the business-oriented perspective of the business activities in the macroflows to the technical perspective of the services and microflows.
- The *Dispatching Tier* is an optional tier that can be added to the PROCESS INTEGRATION ARCHITECTURE if content-based routing, load balancing, or other dispatching tasks are needed for connecting the macroflow requests to the microflow or service execution.
- The *Microflow Tier* is a common tier, if the PROCESS INTEGRATION ARCHITECTURE design follows the MACRO-/MICROFLOW pattern. This makes sense, if short-running, technical orchestrations of services are needed. In the simplest version, a number of hard-coded services can be provided for microflow execution. A more sophisticated realization introduces MICROFLOW ENGINES.
- The *Backend Integration Tier* is an optional tier which is used to provide backends with an interface that is needed by the SOA. As this tier uses INTEGRATION ADAPTERS it to enable independent maintenance of backend

systems, it is highly recommended for SOAs that need to continue operating when connected systems are maintained.

- The *Backend Tier* contains the systems that are connected to the SOA and perform the functions required to execute the business processes. Typical backend systems are custom or off-the-shelf business applications (such as SAP or Siebel), custom business applications, databases, services, and so on. The backend systems are usually connected to the SOA via service-based interfaces that expose the API of the backend system without great modifications. For providing a specific interface to the SOA, INTEGRATION ADAPTERS in the Backend Integration Tier should be used.

The PROCESS INTEGRATION ARCHITECTURE pattern provides a systematic way to scale up a process-driven SOA. It can be applied for a single macroflow engine, and multiple engines can be added later one. Similarly, only one a few services or business applications can be initially provided, and later on more services or business applications can be added. In both cases, the INTEGRATION ADAPTER pattern provides a clear guideline how to perform the connection in a maintainable fashion. The INTEGRATION ADAPTER REPOSITORY pattern should be used, if a larger number of adapters must be maintained.

The various systems connected in the PROCESS INTEGRATION ARCHITECTURE are treated as exchangeable black-boxes. The business applications, macroflows, and microflows can be maintained as independent systems as long as the service interfaces do not change. Load balancing and prioritized or rule-based processing of requests can be supported, for instance via the CONFIGURABLE DISPATCHER. Many existing off-the-shelf engines can be used in a PROCESS INTEGRATION ARCHITECTURE, which might reduce the necessary in-house development effort.

The pattern has the drawback that greater design effort might be necessary compared to simpler alternatives, because of the multi-tier model with corresponding loosely coupled interfaces. To buy (and customize) different off-the-shelf engines or system can be costly, just like inhouse-development of these engines or systems. Hence, for small, simple process-driven SOAs, it should be considered to start-off with a single process engine and follow the MACRO-/MICROFLOW pattern only conceptually. A more sophisticated PROCESS INTEGRATION ARCHITECTURE can then still be introduced later in time, when requirements for higher flexibility, scalability, changeability, and maintainability arise.

Even though the PROCESS INTEGRATION ARCHITECTURE pattern concerns the design of the services used as backends, it does not solve problems of service design.

In various parts of the PROCESS INTEGRATION ARCHITECTURE pattern business objects (or business data) must be accessed. The business objects relevant to microflows and macroflows essentially form a CANONICAL DATA MODEL [Hohpe et al. 2003] for storing process relevant business data. The BUSINESS OBJECT REFERENCE [Hentrich 2004] pattern is used to keep the references to the business objects in the process flows (macroflows and microflows) and services.

Example: Step-by-step design of a PROCESS INTEGRATION ARCHITECTURE

A schematic example for a step-by-step design of a PROCESS INTEGRATION ARCHITECTURE has been given in the introduction of this chapter.

Known Uses

- In a supply chain management solution for a big automotive customer in Germany this architectural pattern has been applied. WebSphere MQ Workflow has been used as the MACROFLOW ENGINE. The integration adapters, the dispatching layer, and the microflow execution level have been implemented in Java. The application services are implemented using MQ messaging technology. In this realization of the pattern, a Java architecture has been implemented to represent the CONFIGURABLE DISPATCHER, a MICROFLOW ENGINE, and the application adapters. No off-the-shelf middleware has been used.
- For a telecommunications customer in Germany, the pattern has been used in a larger scale variant. The MICROFLOW ENGINE has been implemented by an enterprise service bus based on WebSphere Business Integration Message Broker. WebSphere MQ Workflow has been used as the process engine at the macroflow layer. The off-the-shelf MQ Workflow adapters provided by the message broker served as the process integration adapters. The architecture has been laid out initially as to support different instances of MQ Workflow engines to cope with growing workload using a dispatcher represented as a routing flow that routes the messages received by the adapter to another message broker instance. New message broker instances have been created according to the growing workload.
- A simple variant of the pattern is implemented in IBM's WebSphere Integration Developer [IBM 2008], which includes WebSphere Process Server, a process engine that represents both the micro- and macroflow levels. It further offers an architectural concept called Service Component Architecture (SCA) to wire up services, including the corresponding adapters.

5 Literature Review and Overview of Related Patterns

A lot of related work taking process perspectives in conjunction with patterns can be found in the workflow and process domains. Many languages have been proposed for the design and specification of workflow processes. Similarly, languages and tools have been proposed for business process modeling (e.g., the extended EPCs in ARIS and the various stereotypes in UML). Also in other domains such as ERP, CRM, PDM, and Web Services, languages have been proposed to model processes and other perspectives such as the organization and data perspective. Some of these languages are based on well-known modeling techniques such as Petri Nets and UML. Other languages are system specific.

To the best of our knowledge the work on workflow patterns conducted by van der Aalst et al. was the first attempt to collect a structured set of patterns at the level of process-aware information systems (summarized in [Workflow Patterns 2008, van der

Aalst et al. 2003]). Several authors have used these workflow patterns to evaluate existing workflow management systems or newly designed workflow languages. The work has also been augment with other pattern collections, such as service interaction patterns [Barros at al. 2005]. These works on patterns strongly focuses on the workflow perspective and does not take an overall architectural perspective. The workflow patterns rather address finer grained structural elements of workflows than software patterns in their actual sense of emergent design solutions.

Other authors have coined the term workflow patterns but addressed different issues. In [Weigand et al. 2000] a set of workflow patterns inspired by Language/Action theory and specifically aiming at virtual communities is introduced. Patterns at the level of workflow architectures rather than control-flow are given in [Meszaros and Brown 1997]. Collaboration patterns involving the use of data and resources are described in [Lonchamp 1998].

Patterns for exception handling in process execution languages are introduced in [Russel et al. 2006b]. [Schümmer and Lukosch 2007] provide patterns for human-computer interaction, and some of them include process or service perspectives. [Buschmann et al. 2007] describe a summary of the most successful emerging software architecture patterns and integrate patterns from different sources in a consistent manner, as to provide a comprehensive summary on architecture patterns. However, these architecture patterns do not address SOAs specifically.

The POSA 1 book introduces a number of general architectural patterns [Buschmann et al. 1996]. These are implicitly used in our pattern language. For instance, it is assumed in a SOA that a BROKER architecture is used. The CONFIGURABLE DISPATCHER pattern resembles the general solution of CLIENT/DISPATCHER/SERVER. The INTEGRATION ADAPTER pattern implicitly resembles the general solution of FORWARDER/RECEIVER.

Enterprise integration patterns [Hohpe et al. 2003] are also related to this work, as they mainly describe asynchronous messaging solutions. This communication paradigm is often used in process driven SOAs.

Specific architectural guidance for SOA construction is given in [Josuttis 2007]. However, this book does neither focus on process-driven SOAs nor patterns in specific, and hence can be seen as complementary to our pattern language.

In his work on micro-workflows [Manolescu 2000, 2002], Manolescu provides a workflow approach that is used to realize mainly workflows for object-oriented compositions. The work is also based on patterns. Please note that the term micro-workflows in Manolsecu's work has a different meaning than microflow in our work. Micro-workflows can be microflows but could also exhibit macroflow characteristics. We chose to use the macroflow/microflow terminology despite the overlap in terminology because this terminology has already been established and proven to be intuitive to pattern language users in our experience.

Evans identified that it is not just design patterns but also many different types of patterns that are influential when developing systems in a certain domain [Evans 2004]. This work does not yet combine aspects of organizational flexibility with a SOA and pattern-based approach.

The typical tooling around process engines has been described in pattern form by Manolescu (see [Manolescu 2004]). These patterns can be used to link our rather

architectural patterns on process engines, MACROFLOW ENGINE and MICROFLOW ENGINE to the tooling provided by concrete technologies.

Some patterns are directly related and referenced in the patterns in this work. These patterns and their sources are summarized in Table 1.

Pattern	Problem	Solution
GENERIC PROCESS CONTROL STRUCTURE [Hentrich 2004]	How can data inconsistencies be avoided in long running process instances in the context of dynamic sub-process instantiation?	Use a generic process control data structure that is only subject to semantic change but not structural change.
BUSINESS OBJECT REFERENCE [Hentrich 2004]	How can the management of business objects be achieved in a business process, as far as concurrent access and changes to these business objects is concerned?	Only store references to business objects in the process control data structure and keep the actual business objects in an external container.
ENTERPRISE SERVICE BUS [Zdun et al. 2006]	How is it possible in a large business architecture to integrate various applications and backends in a comprehensive, flexible, and consistent way?	Unify the access to applications and backends using services and service adapters, and use message-oriented, event-driven communication between these services to enable flexible integration.
CORRELATION IDENTIFIER [Hohpe et al. 2003]	How does a requestor that has received a response know to which original request the response is referring?	Each response message should contain a CORRELATION IDENTIFIER, a unique identifier that indicates which request message this response is for.
CANONICAL DATA MODEL [Hohpe et al. 2003]	How to minimize dependencies when integrating applications that use different data formats?	Design a CANONICAL DATA MODEL that is independent from any specific application. Require each application to produce and consume messages in this common format.
COMPONENT CONFIGURATOR [Schmidt et al. 2000]	How to allow an application to link and unlink its component implementations at runtime without having to modify, recompile, or relink the application statically?	Use COMPONENT CONFIGURATORS as central components for reifying the runtime dependencies of configurable components. These configurable components offer an interface to change their configuration at runtime.
SERVICE ABSTRACTION LAYER [Vogel 2001]	How do you develop a system which can fulfill requests from different clients communicating over different channels without having to modify your business logic each time a new channel has to be supported or a new service is added?	Provide a SERVICE ABSTRACTION LAYER as an extra layer to the business tier containing all the necessary logic to receive and delegate requests. Incoming requests are forwarded to service providers which are able to satisfy requests.

Table. 1. Related Patterns Overview.

6 Conclusion

In this article we have documented the fundamental patterns needed for an architecture that composes and orchestrates services at the process level. The patterns explain two important kinds of design and architectural decisions in this area:

- Modeling and executing business-driven and technical processes
- Integration and adaptation in process-driven SOAs

The individual patterns can be used on their own to address certain concerns in a process-driven SOA design, but the general architecture following the PROCESS-INTEGRATION ARCHITECTURE pattern – in first place – aims at larger architectures. The pattern language as a whole focuses on separating business concerns cleanly from technical concerns, in macroflows and microflows. All integration concerns are handled via services, and macroflows and microflows are used for flexible composition and orchestration of the services.

Acknowledgements

We like to thank Andy Longshaw, our EuroPLoP 2006 shepherd, for his useful comments. We also like to thank the participants of the EuroPLoP 2006 writers' workshop for their valuable feedback. Finally, we could like to thank the anonymous reviewers of the Transactions on Pattern Languages of Programming journal for their in-depth comments that helped us to improve this article.

References

[van der Aalst et al. 2003] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, Alistair P. Barros: Workflow Patterns. Distributed and Parallel Databases 14(1): 5-51, 2003.

[Active Endpoints 2007] Active Endpoints. ActiveBPEL Open Source Engine. ttp://www.active-endpoints.com/active-bpel-engine-overview.htm, 2007.

[Barros at al. 2005] Alistair P. Barros, Marlon Dumas, Arthur H. M. ter Hofstede: Service Interaction Patterns. Business Process Management, 3rd International Conference, BPM 2005, Nancy, France, September 5-8, 2005.

[Barry 2003] D. K. Barry. Web Services and Service-oriented Architectures, Morgan Kaufmann Publishers, 2003.

[BPMN2BPEL 2008] bpmn2bpel. A tool for translating BPMN models into BPEL processes. http://code.google.com/p/bpmn2bpel/, 2008.

[Buschmann et al. 1996] F. Buschmann, R. Meunier, H. Rohnert, P.Sommerlad, M. Stal. Pattern-Oriented Software Architecture - A System of Patterns, John Wiley and Sons Ltd, Chichester, UK, 1996.

[Channabasavaiah 2003 et al.] K. Channabasavaiah, K. Holley, and E.M. Tuggle. Migrating to Service-oriented architecture – part 1, http://www-106.ibm.com/developerworks/webservices/library/ws-migratesoa/, IBM developerWorks, 2003.

[Dikmans 2008] L. Dikmans. Transforming BPMN into BPEL: Why and How. http://www.oracle.com/technology/pub/articles/dikmans-bpm.html, 2008.
[D'Souza and Wills 1999] Desmond D'Souza, Alan Wills. Objects, Components and Frameworks with UML: The Catalysis Approach. Addison-Wesley, 1999.
[Emmerich 2000] W. Emmerich. Engineering Distributed Objects. Wiley & Sons, 2000.
[Enhydra 2008] Enhydra. Enhydra Shark. http://www.enhydra.org/workflow/shark/index.html, 2008.
[Evans 2004] Eric Evans. Domain-Driven Design, Addison-Wesley, 2004.
[Fornax 2008] Fornax Project. Sculptor. http://www.fornax-platform.org/cp/display/fornax/Sculptor+(CSC), 2008.
[Gamma et al. 1994] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.
[GFT 2007] GFT. GFT Inspire Business Process Management. http://www.gft.com/gft_international/en/gft_international/Leistungen_Produkte/Software/Business_Process_Managementsoftware.html, 2007.
[Hentrich 2004] C. Hentrich. Six patterns for process-driven architectures. In Proceedings of the 9th Conference on Pattern Languages of Programs (EuroPLoP 2004), 2004.
[Hohpe et al. 2003] G. Hohpe and B. Woolf. Enterprise Integration Patterns. Addison-Wesley, 2003.
[IBM 2008] IBM, WebSphere Software. http://www-01.ibm.com/software/websphere/, 2008.
[iWay 2007a] iWay Software. iWay Adapter Technologies. http://www.iwaysoftware.jp/products/integrationsolution/adapter_manager.html, 2007.
[iWay 2007b] iWay Software. iWay Adapter Manager Technology Brief. http://
www.iwaysoftware.jp/products/integrationsolution/adapter_manager.html, 2007.
[JBoss 2007] JBoss. JBoss jBPM. http://www.jboss.com/products/jbpm, 2007.
[Josuttis 2007] Nicolai M. Josuttis. SOA in Practice - The Art of Distributed System Design, O'Reilly, 2007.
[Lonchamp 1998] Lonchamp, J. Process model patterns for collaborative work. In Proceedings of the 15th IFIP World Computer Congress. Telecooperation Conference. Telecoop. Vienna, Austria, 1998.
[Meszaros and Brown 1997] Meszaros, G. and Brown, K. A pattern language for workflow systems. In Proceedings of the 4th Pattern Languages of Programming Conference. Washington University Technical Report 97-34 (WUCS-97-34), 1997.
[Manolescu 2004] D. A. Manolescu. Patterns for Orchestration Environments. The 11th Conference on Pattern Languages of Programs (PLoP2004), September 8 - 12, 2004, Allterton Park, Monticello, Illinois, 2004.
[Manolescu 2002] Dragos A. Manolescu. Workflow enactment with continuation and future objects, ACM SIGPLAN Notices, v.37 n.11, November 2002.
[Manolescu 2000] Dragos A. Manolescu. Micro-Workflow: A Workflow Architecture Supporting Compositional Object-Oriented Software Development. Ph.D. Thesis and Computer Science Technical Report UIUCDCS-R-2000-2186, University of Illinois at Urbana-Champaign, October 2000, Urbana, Illinois.
[Mellor and Balcer 2002] Stephen J. Mellor, Marc J. Balcer. Executable UML: A Foundation for Model Driven Architecture. Addison-Wesley, 2002.
[Mittal and Kanchanavally 2008] K. Mittal and S. Kanchanavally. Introducing Java Page Flow Architecture. http://www.developer.com/open/article.php/10930_3531246_1, 2008.
[Mule 2007] Mule Project. Mule open source ESB (Enterprise Service Bus) and integration platform. http://mule.mulesource.org/, 2007.
[Novell 2008] Novell. Novell exteNd Director 5.2, http://www.novell.com/documentation/extend52/Docs/Start_Director_Help.html, 2008.
[Riehle et al. 2001] Dirk Riehle, Steven Fraleigh, Dirk Bucka-Lassen, and Nosa Omorogbe. "The Architecture of a UML Virtual Machine." In Proceedings of the 2001 Conference on

Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '01). ACM Press, 2001. Page 327-341.

[Russel et al. 2006] Russell, N. and van der Aalst, W.M.P. and ter Hofstede, A.H.M. Exception handling patterns in process-aware information systems. BPM Center Report BPM-06-04 , BPMcenter.org, 2006.

[ServiceMix 2007] Apache ServiceMix Project. Apache ServiceMix. http://www.servicemix.org/, 2007.

[Schmidt et al. 2000] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. Patterns for Concurrent and Distributed Objects. Pattern-Oriented Software Architecture. J.Wiley and Sons Ltd., 2000.

[Schümmer and Lukosch 2007] Schümmer, T. and Lukosch, S. Patterns for computer-mediated interaction. Wiley & Sons., 2007

[Stahl and Völter 2006] T. Stahl and M. V¨olter. Model-Driven Software Development. John Wiley & Sons, 2006.

[Tran et al. 2007] H. Tran, U. Zdun, and S. Dustdar. View-based and Model-driven Approach for Reducing the Development Complexity in Process-Driven SOA. In Proceedings of International Conference on Business Processes and Services Computing, Leipzig, Germany, Sep, 2007.

[Vogel 2001] O. Vogel. Service abstraction layer. In Proceedings of EuroPlop 2001, Irsee, Germany, July 2001.

[webMethods 2007] webMethods. webMethods Fabric 7. http://www.webmethods.com/products/fabric, 2007.

[Weigand et al. 2000] Weigand, H. and de Moor, A. and van den Heuvel, W.J. Supporting the evolution of workflow patterns for virtual communities. Electronic Markets, 10(4):264., 2000.

[Workflow Patterns 2008] Workflow Patterns home page, http://www.workflowpatterns.com/, 2008.

[Zdun et al. 2006] U. Zdun, C. Hentrich, and W.M.P. van der Aalst. A Survey of Patterns for Service-Oriented Architectures. International Journal of Internet Protocol Technology, vol. 1, no. 3, pages 132-143, Inderscience, 2006.