Advanced Synchronization Patterns for Process-Driven and Service-Oriented Architectures

Carsten Hentrich

CSC Deutschland Solutions GmbH
Abraham-Lincoln-Park 1
65189 Wiesbaden, Germany
chentrich@csc.com

Uwe Zdun

Distributed Systems Group
Information Systems Institute
Vienna University of Technology
Argentinierstrasse 8/184-1
A-1040 Vienna, Austria
zdun@infoysys.tuwien.ac.at

In a process-driven and service-oriented architecture, parallel and independently running business processes might need to be synchronised according to dependencies of complex business scenarios. In this paper we describe three software patterns that address such rather advanced synchronisation issues between business processes running in parallel. The patterns focus on coordinating the parallel and principally independent business processes via architectural solutions allowing architects to model the flexible synchronisation of the processes.

Introduction

Service-oriented architectures (SOA) are an architectural concept in which all functions, or services, are defined using a description language and have invokable, platform-independent interfaces [Channabasavaiah 2003 et al., Barry 2003]. In many cases services are called to perform business processes. 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. 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.

A *process* is a behavioural model expressed in a process modelling language, such as BPEL, that is instantiated and managed by a *process engine*. On a process engine multiple *process instances*

Proceedings of the 13th European Conference on Pattern Languages of Programs (EuroPLoP 2008), edited by Till Schümmer and Allan Kelly, ISSN 1613-0073 <issn-1613-0073.html>.
Copyright © 2009 for the individual papers by the papers' authors. Copying permitted for private and academic purposes. Re-publication of material from this volume requires permission by the copyright owners.

of one or more processes are typically running in parallel. Processes usually work on business data that is stored in *business objects*. Usually each process instance has its own private data space of business objects it creates, in order to limited problems of concurrent data access, such as data inconsistencies, deadlocks, or unnecessary locking overhead.

In this paper we describe three patterns that address advanced synchronization issues of parallel business processes. In this context *synchronization* means that execution in terms of the progression through the different activities of a process needs to be synchronized with other business processes running in parallel. The synchronization issues reflect requirements of complex business scenarios, and the synchronization dependencies cannot be modeled directly in the business processes via static control flow dependencies. As a result, conflicting forces arise due to the need for loosely coupling the synchronization concerns with the business process models. Besides technical forces, such as the problems of concurrent data access, supporting business agility is central. Business processes are subject to constant change. Hence, any suitable synchronization mechanism must be loosely coupled in order to support changes in the connected business processes.

The three patterns presented are:

- The REGISTER FOR ACTION pattern describes how to synchronize the execution of functionality of a target system with business processes running in parallel.
- The BUNDLE PROCESS AGENT pattern describes how business objects being generated by different parallel processes can be processed in a consolidated way.
- The PROCESS CONDUCTOR pattern addresses how to enable designers to model dependencies between parallel business processes by flexible orchestration rules.

Consider a simple example to illustrate the use of the patterns: various business processes of a company require contacting the customer via phone, but the company wants to avoid contacting the customer too often in a specific period of time. Hence, the phone calls should be consolidated. In such business scenarios that require synchronization of multiple process instances, the patterns described in this paper can be applied.

If only a specific *action*, like “put phone call into a task list” needs to be performed after synchronization has taken place, the REGISTER FOR ACTION pattern should be applied. However, the phone call might also require a business process preparing the phone call and this business process then usually needs access the private business objects of the synchronized processes. In this more complex scenario, the BUNDLE PROCESS AGENT pattern can be applied. Finally, if the need for synchronizing occurs within the processes and requires each of the processes to be stopped until the synchronizing action (which might be yet another business process) has happened, then PROCESS CONDUCTOR is applicable.

This simple scenario should illustrate: which of the patterns is chosen depends on the design of the business processes that need to be synchronized. In some scenarios, the patterns are mutual alternatives, in others combining them makes sense.

Register for Action

Business processes are executed on process engines. Sometimes the execution of an action is depending on the states of multiple business process instances running in parallel. When this is the case, the action can only be executed if those parallel business process instances have reached a certain state represented by a specific process activity.

As business processes are created and changed while the system is running, it is not advisable to define synchronization dependencies statically in the models. Instead it should be possible to define and evaluate the synchronization dependencies at runtime while still allowing business process to change independently.

How can multiple, dynamically created process instances be synchronized with minimal communication overhead and still be kept independently changeable?

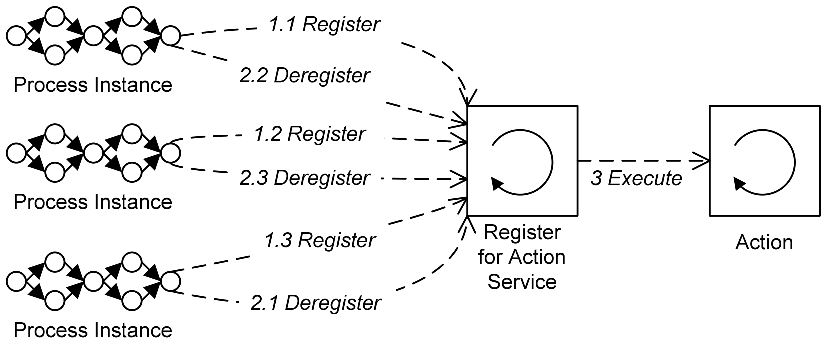
Business processes are dynamically instantiated on process engines and at different points in time. For this reason, there are usually several instances of one or more business processes running in parallel. Each of them has a different state of progression in terms of the process activity they have reached so far during execution.

When a specific *action*, such as a business function, service, or another business process, has a logical dependency to multiple business process instances, synchronization with all the process instances can be difficult. First of all, the action might not know which business process instances of the many possible parallel instances it is dependent on. But even if it knows the instances that it is dependent on, polling them for synchronization would incur a communication overhead. The same problem of a communication overhead for synchronization would also occur, if the process instances would run a synchronization protocol, for instance before triggering a callback that executes the action.

In addition, before the action can synchronize with the process instances, it needs to know that it must wait for one or more instances. That is, a mechanism is required to communicate that there is a new dependent process instance to wait for.

Business processes can change over time and new processes are constantly created, while the overall system is running. This includes that a state at which an action must be executed might change, gets added to a process, or gets removed from the process. The actions that are depending on business processes must be able to cope with such process changes. The effects of these changes should be minimized and should not impact other components in order to be manageable. A consequence is that the synchronization dependencies of the actions cannot be statically modeled in the models of the business processes or the actions, but must be defined and evaluated at runtime. In other words, a loose coupling between the action and the business processes it is dependent on is required.

Use a REGISTER FOR ACTION component that offers registration and de-registration services to be invoked from business processes. The registration informs the REGISTER FOR ACTION component to wait with the desired action to be initiated until a specific process instance has de-registered itself. When all registered processes have de-registered themselves the action will be executed.



The REGISTER FOR ACTION component offers two services:

- A registration service, where a process instance can register itself with its instance ID.
- A de-registration service that allows a process instance to de-register itself via its instance ID.

Invocation of the de-registration service means that the process has reached the state that is relevant for the action. The two services are invoked by process activities. Each registration invocation must have one corresponding de-registration invocation in a business process. This design has the consequence that the place of invocations can change as the business processes change over time. In other words, the REGISTER FOR ACTION component and the business processes are loosely coupled.

The REGISTER FOR ACTION component waits until all registered business processes have de-registered themselves. After the last de-registration, the action is executed by the REGISTER FOR ACTION component.

An important detail of a REGISTER FOR ACTION design is to determine the point in time when registration ends. As most scenarios of the pattern concern long running business processes, a registration delay is a practical solution that works in many cases. The registration delay runs a certain amount of time from the point in time when the first registration to the REGISTER FOR ACTION instance happens. For instance, if a registration delay of one day is chosen, then all registrations that accumulate throughout that day will be included. Of course, the length of the delay can be adjusted based either on previous experiences and experimentation. An alternative to a registration delay is introducing a specific registration type that ends the registration process for one REGISTER FOR ACTION instance.

Modeling the de-registration service invocation might be an issue for some business processes: de-registration should often be invoked as early as possible in order not to produce unnecessary delays for the action to be executed. If the business process contains complex decision logic there may be various paths that may lead to a de-registration service invocation at many different positions in the process. As the process execution may follow only one case specific path, de-registration must be found on all possible paths if a registration has been previously performed.

To place the de-registration service invocations at the right positions and to avoid multiple invocations of the de-registration service in case of loops in the process is sometimes not trivial, if the business process is of higher complexity. The easiest way might be to put de-registration simply at the end of the business process and thus to avoid the possible complex logic that is initiated by the different possible paths or loops. However, this is not always possible or optimal if de-registration as early as possible is required.

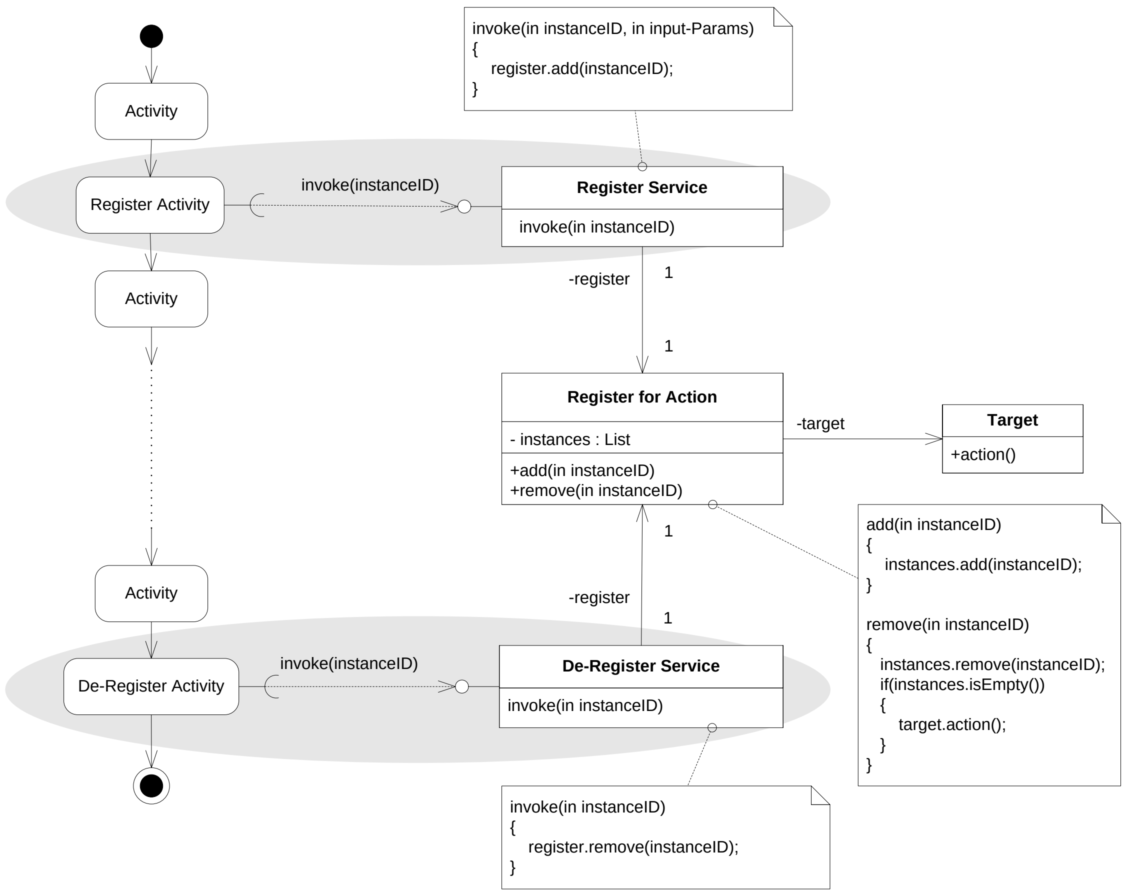
The service invocations from the business processes might be realized as SYNCHRONOUS SERVICE ACTIVITIES or FIRE AND FORGET SERVICE ACTIVITIES [Hentrich et al. 2008]. The realization using SYNCHRONOUS SERVICE ACTIVITIES is usually better suited as it is important for the business process to get informed whether the registration and de-registration was successful. If the target action is related to a more complex business process, then this consolidation can be achieved by using a BUNDLE PROCESS AGENT.

The ACTIVATOR pattern [Schmidt et al. 2000] has a similar structure as REGISTER FOR ACTION. It, however, solves a different problem, the on-demand activation and deactivation of service execution contexts to run services accessed by many clients without consuming resources unnecessarily. The patterns can be used together with REGISTER FOR ACTION using a shared structure. That is, the registration and deregistration services could be used for on-demand activation and deactivation.

Also, PUBLISH/SUBSCRIBE [Buschmann et al. 2000] has a similar structure, as it includes registration/deregistration of publishers and subscribers. This pattern can also be combined with REGISTER FOR ACTION using a shared structure. That is, the registration and deregistration services could be used to subscribing and unsubscribing to events for the time of being registered. This way, the REGISTER FOR ACTION component can communicate with its currently registered processes.

Example: REGISTER FOR ACTION example configuration

The following figure shows on the left-hand side a business process that invokes a register service. At the end of the business process, an activity invokes a deregister service. Both services are interfaces to a REGISTER FOR ACTION component, which maintains an instance list. When all instances are removed from this list, the action is invoked.



Example: Saving Costs of Postal Mail Sending

In the context of business processes that create information that must be send by mail to recipients, the pattern provides significant potential to save postal costs. If each business process produces its own letters to be sent to recipients a lot of postal costs will be created. It would be better to gather all the information created from the business processes and to wrap them in one letter. The idea of sending fewer letters will save significant postal costs. However, the problem is how to gather all the information and when is the point in time to pack all the gathered information in one letter and send it out to a recipient.

Applying the REGISTER FOR ACTION pattern it is possible to control and coordinate sending out letters to various recipients. The action associated in this context is sending a letter out to a recipient. This can be coordinated by registering all business processes that will create information to be packed in one letter for a specific recipient. Thus, the registration service can be designed to capture an additional parameter to specify the recipient. That way it is possible to pack all the information created for one recipient on one letter as the letter will be sent out when all registered business processes have de-registered themselves for a recipient.

The logic associated to the registration service and the REGISTER FOR ACTION component might be even more complex, e.g. to distinguish different priorities for the information to be sent out quickly or to send it out later. As a result there may be more complex rules to control and synchronize the business processes. However, the basic pattern represented by REGISTER FOR ACTION will always be the same.

Especially in a customer order or service management context the pattern is useful for these kinds of purposes, as to control communication towards recipients. It is suitable not only for postal communication but for various communication channels, e.g. fax, e-mail, or even telephone. The general purpose of gathering relevant information first, before initiating the communication, generally applies to all those channels.

Even as far as the telephone communication is concerned, it becomes clear that calling the recipients only once to discuss a whole bunch of open questions that stem from different parallel clearing business processes, for instance, will be better for customer satisfaction than contacting the customer several times to clarify one single issue at a time, which might even be of minor importance. The pattern provides flexible means to capture all these business scenarios and to automate significant parts of the business logic.

Known Uses

- The pattern has been used in projects to control batch processing of larger transactions. Each business process generates transactions to be made but the actual commit of the all gathered transactions needs to be done at a point in time when all related transactions to be made are identified. That way transaction costs can be saved by putting related transactions, addressed to the same account, in one larger transaction.
- The pattern has also been used to control the point in time when consolidated outbound communication to one concrete party needs to be performed in an order management context.
- The above purposes of the pattern have been used in projects in the telecoms industry in the context of order management, in the insurance industry in the context of claims handling. The pattern has also been used in banking as far as the mentioned transaction processing issues are concerned.

Bundle Process Agent

Business processes are executed on a process engine, and during their execution business objects are created and manipulated by the business process instances. Each business process instance creates and manages its own set of business objects in order to avoid data inconsistencies, locking overheads, deadlocks, and other problems created by concurrent data access. Business scenarios, such as consolidated sending of postal mail in batches, require consolidating the business objects being generated by many different process instances and then process them using a central but parallel running business process. Hence, the usual mechanisms of a process engine, in which each process instance keeps its business objects in a private data space, are not sufficient to support such scenarios.

How to gather the business objects from various business process instances and process them in a consolidated way without causing unwanted effects of concurrent data access?

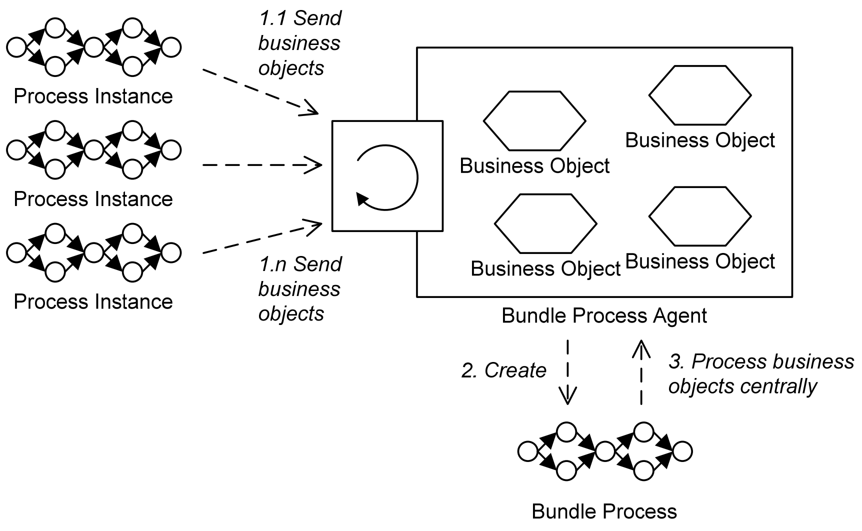
Business process instances running on a process engine have their own data space and are thus disjoint entities. When business objects are created during the execution of a business process, only the business process instances creating the objects know about their existence. That is, the business objects created by a business process instance are per default private to the business process instance. This helps to avoid unwanted effects, such as data inconsistencies, locking overheads, or deadlocks, when business process instances are running in parallel because the actions of the business process instances control all accesses to these business objects.

This technical concept can be applied to implement most business scenarios. However, there is a special case, where this technical solution does not work well alone: Consider that the business objects created by many different parallel business process instances are input objects to be processed by a central business process that logically gathers all these business objects and then processes the consolidated set of business objects. A typical example is that the business requires this business objects to be handled in a consolidated way, such as sending one letter per postal mail for a number of parallel transactions with a customer, instead of sending multiple letters. In this case, the parallel running consolidation process instance must gather the objects and process them. Unfortunately, usually process engines do not directly support such scenarios.

It is necessary to only centrally process those business objects that actually should be processed in a consolidated way. It might be that this is only a subset of business objects owned by a process instance. A process instance should still have control what business objects should be processed in a consolidated way and should thus be able to publish only those objects that it considers to be relevant.

Each of the involved business processes can potentially change over time. Hence, the consolidation architecture should not impose restrictions on the business process design that would hinder rapid changeability.

Send the business objects to be processed centrally to a BUNDLE PROCESS AGENT via a dedicated service, specified in the model of the business process. The BUNDLE PROCESS AGENT creates an instance of a bundle (consolidation) process, if no instance exists yet, and for each bundle it makes sure that only one instance of the consolidation process is running at a time. The business object bundle is gathered from different business processes invoking that dedicated service for sending the business objects. When a specified end criterion is reached, such as a deadline or a specified number of business objects in the bundle, then the bundle is centrally processed by the bundle process.



Design an architectural component that serves as a BUNDLE PROCESS AGENT, which offers a service to be invoked by business processes to send business objects that need to be processed centrally. The BUNDLE PROCESS AGENT stores the business objects being sent to it in a container that serves as a temporary repository. The container is not intended as the actual persistence mechanism of the business objects—it is rather intended to capture only what objects need to be processed centrally.

For this reason, this container might only keep BUSINESS OBJECT REFERENCES [Hentrich 2004] rather than the business objects themselves. However, it is also possible to send copies of actual business objects and not just references. Often these objects then only contain a subset of the business data of the original business objects, i.e. the subset of data that is relevant for processing the business objects centrally. In this case, it is advisable to introduce special types of business objects designed for these purposes.

The BUNDLE PROCESS AGENT waits until a specified end criterion is reached. For instance, this can be a deadline or specified maximum amount of business objects that can be bundled in one bundle. When the end of bundling is reached, the BUNDLE PROCESS AGENT instantiates a *bundle process* that processes the business objects centrally. The container with the business objects is cleared after the processing has been initiated to be ready to store new objects for the next iteration. Only one instance of the bundle process is running at a time for each bundle, i.e. the processing of a set of business objects must be finished before the next instance of a bundle process can be started. Of course, different bundles can be assembled in parallel. Consider for instance, business objects for postal mail communication with customers are bundled, to send them together. Then there is one bundle per customer.

During the execution of the bundle process new business objects are sent to the BUNDLE PROCESS AGENT by business processes running in parallel for the next iteration. These objects are again stored in the container. This way, only business objects relevant for the next iteration are kept in the container, as the container is emptied when a new iteration, i.e. a new instance of the bundle process, has been started. The BUNDLE PROCESS AGENT repeats this process in a loop.

The BUNDLE PROCESS AGENT is implemented as a COMPONENT CONFIGURATOR [Schmidt et al. 2000] to allow controlled configuration of the agent at runtime. When it is initialized it performs the following functions:

1. It is checked whether there are new business objects in the container to be processed by a bundle process
2. If there are new objects it checks whether an instance of the bundle process is still running. Only if there is no instance running, a new instance is created that processes the new objects in the container. The container is cleared to be empty for new objects after the process instance has been started. If an existing instance is still running then no action is performed, i.e. no new bundle process is created nor is the business object container being emptied.
3. The agent loops back to step 1 until the loop is aborted by some event to finalize the execution or to suspend the execution.

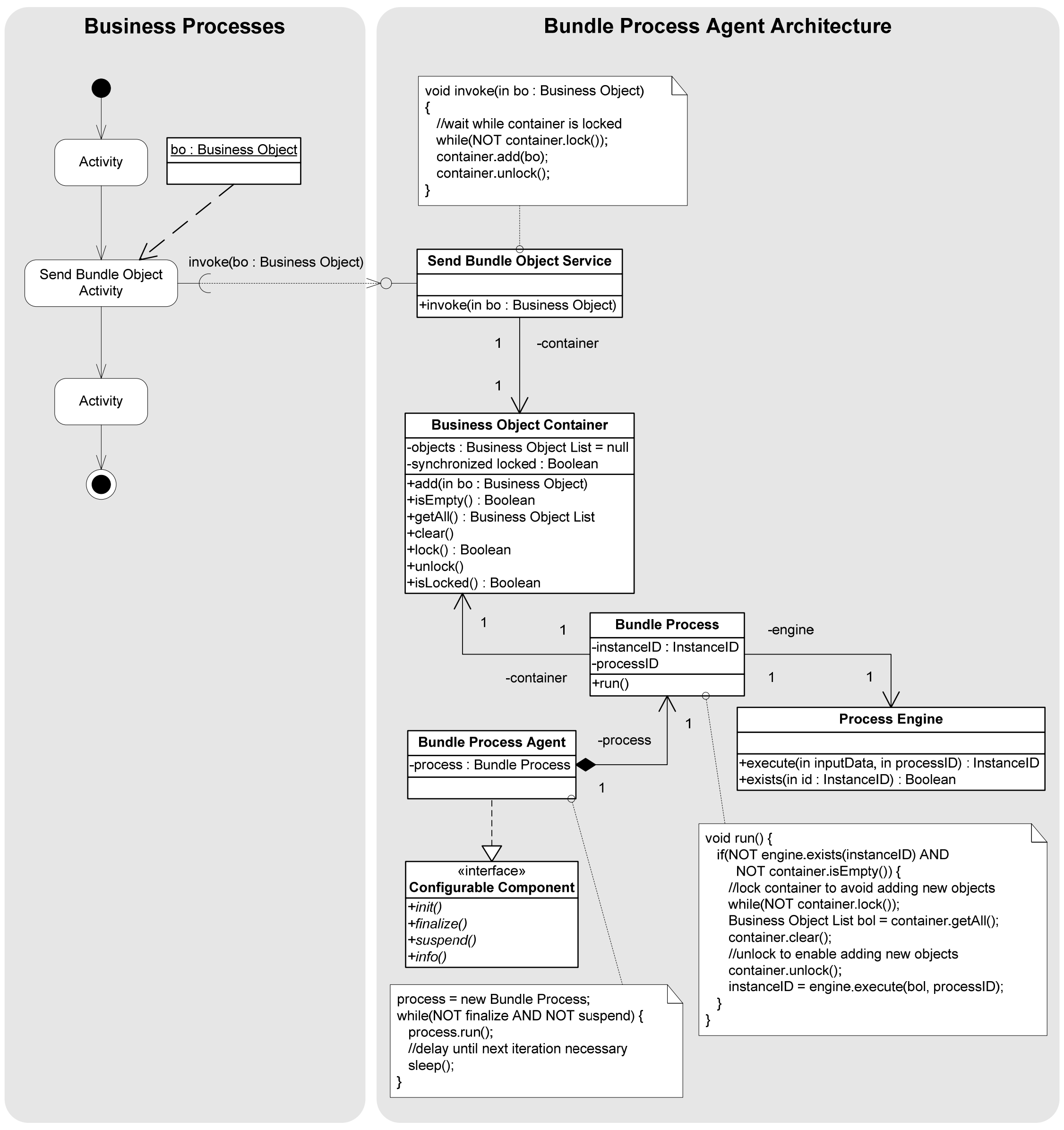
There is one concurrency issue involved in this algorithm. The service that allows business processes to send new business objects to the container might conflict with the clearing action of the container that is initiated by the algorithm described above. That means a new instance of the bundle process might be created with the given objects in the container at that point in time. After the instance is created, the algorithm prescribes to clear the container. If there are new objects added to the container while the creation of the new instance is still in progress, then these objects will be deleted from the container with the clearing action without being processed by the bundle process. In order to avoid such a situation the container must provide locking and unlocking functions that are used for the time a new instance of a bundle process is created.

The BUNDLE PROCESS AGENT pattern thus resolves issues according to complex bundling of business objects that need to be centrally processed and offers a general architectural solution that is both flexible and extensible. Different bundle processes can be used for different purposes, though this will increase the complexity of the architecture. However, there might be larger effort involved to design a BUNDLE PROCESS AGENT component. For this reason, the pattern may only be suitable in projects and programs that have a larger strategic perspective.

The pattern can be combined with the REGISTER FOR ACTION pattern in order to dynamically identify what business processes need to be considered by the bundling. As far as the service invocation from business processes is concerned the SYNCHRONOUS SERVICE ACTIVITY pattern or the variation of the FIRE AND FORGET SERVICE ACTIVITY including acknowledgement [Hentrich et al. 2008] is usually recommended to achieve some level of security that the business objects being sent have arrived at the initiated target. According to the MACRO-MICROFLOW pattern [Hentrich et al. 2007] the bundle process can be implemented as a macroflow or microflow.

Example: BUNDLE PROCESS AGENT example configuration

The following figure provides an overview of the conceptual structure how the BUNDLE PROCESS AGENT might look like including the service that provides the functionality to add business objects to the container. The structure also resolves the described concurrency issues by providing locking functionality of the container.



Business Processes
Activity
bo : Business Object
Send Bundle Object Activity
invoke(bo : Business Object)
Activity
Bundle Process Agent Architecture
Send Bundle Object Service
+invoke(in bo : Business Object)
1
-container
1
Business Object Container
-objects : Business Object List = null
-synchronized locked : Boolean
+add(in bo : Business Object)
+isEmpty() : Boolean
+getAll() : Business Object List
+clear()
+lock() : Boolean
+unlock()
+isLocked() : Boolean
1
1
-container
Bundle Process
-instanceID : InstanceID
-processID
+run()
-engine
1
1
Process Engine
+execute(in inputData, in processID) : InstanceID
+exists(in id : InstanceID) : Boolean
Bundle Process Agent
-process : Bundle Process
-process
1
1
«interface»
Configurable Component
+init()
+finalize()
+suspend()
+info()

```
void invoke(in bo : Business Object)
{
   //wait while container is locked
   while(NOT container.lock());
   container.add(bo);
   container.unlock();
}
```

```
void run() {
   if(NOT engine.exists(instanceID) AND
        NOT container.isEmpty()) {
      //lock container to avoid adding new objects
      while(NOT container.lock());
      Business Object List bol = container.getAll();
      container.clear();
      //unlock to enable adding new objects
      container.unlock();
      instanceID = engine.execute(bol, processID);
   }
}
```

```
process = new Bundle Process;
while(NOT finalize AND NOT suspend) {
   process.run();
   //delay until next iteration necessary
   sleep();
}
```

The figure shows business processes that invoke a special *Send Bundle Object Service* to send business objects. The processes may run in parallel and the services might be invoked at different points in time in the processes, i.e. the service invocation might be modeled several times in one process and might be used in various process models. The service simply adds the objects to the container. To resolve the concurrency issue, explained above, it uses locking and unlocking mechanisms. The class *Bundle Process Agent* implements the COMPONENT CONFIGURATOR [Schmidt et al. 2000] and invokes the *run* method of class *Bundle Process* in a loop. The *run* method of class *Bundle Process* retrieves the business objects from the container and creates a new bundle process if no instance is running and the container is not empty. It also uses the locking and unlocking mechanisms to prevent the concurrency issue.

The class *Process Engine* provides an interface to the API of the process engine being used to implement the business processes—in this case especially the bundle process. The *execute* method instantiates a bundle process with the given input data, which are the business objects from the container in this case. The *exists* method allows to check whether an instance of the bundle process, identified by a unique ID, is still running in the engine.

Example: Handling Complex Orders

The following example shows how two distinct strategic goals (mentioned in the known uses) have been realized using the pattern. In one larger project in the telecoms industry two important issues occurred in the context of processing complex orders from larger customers. Complex orders consist of a number of sub-orders that are processed in parallel business processes by different organizational units in the telecoms company. These sub-orders are independent to a certain degree from an internal perspective of the company. For this reason, the business processes for these sub-orders run in parallel to speed-up the completion of the overall order. However, in order to improve customer satisfaction and to reduce costs, issues that occur during the processing of sub-orders need to be clarified with the customer, whose perspective is on the overall order.

If each business process is implemented with its own issue resolution process the customer needs to be contacted for each single issue that might occur in a sub-order, or issue resolution might only be structured according to sub-orders. As a result, each process needs to implement its own issue resolution procedure in some way. To reduce the number of customer interactions and to save communication costs, the issue clarification process needs to be consolidated and treated as an own concern. That way, different processes can use the same clarification procedure and changes to these business processes associated to processing of sub-orders can remain independent.

Moreover, each customer has its own communication preferences, i.e. some want to be contacted by letter, others prefer e-mail or fax, and other customers rather prefer direct telephone communication. Additionally, some serious issues required written communication. Consequently, it was required to treat issue resolution as an own concern and to centralize the rules around the communication preferences. A concept for classification of occurring issues and a central processing of those occurring issues was required. The actual issue clarification process needed to be implemented as a rather complex business process itself that gathers all the occurring issues from those various parallel running sub-order processes. The rather complex rules for communication needed to be implemented by the issue clarification process.

The BUNDLE PROCESS AGENT pattern has been applied to deal with these requirements. A classification scheme for possible occurring issues has been designed in a business object model. The parallel running sub-order processes have just sent a type of issue that occurred during the process to the BUNDLE PROCESS AGENT. The agent created an issue clarification process that processed the issues centrally according to communication preferences. For instance, a list of issues that resulted from various sub-orders could thus be clarified in a single telephone call with the customer, or have been communication in a single letter. Direct communication via telephone of a consolidated list of issues has thus speeded-up the clarification process or has saved mailing costs, as a number of relevant issues have been gathered in a single letter.

The overall clarification process could be implemented in a controlled way considering the customer view and preferences of the overall order, while still having the ability to process the sub-orders according to different specialized internal departments. A special team to improve the clarification process as a separate concern could thus be implemented without affecting actual order processes. In that way, it has been possible to design the business process models according to different levels of expertise and to assign dedicated resources with expertise on issue resolution.

The issue classification scheme via the special business object model and the service for sending occurring issues via a service provided a clear interface that allowed new or improved sub-order processes to use it in a flexible way. The service has provided a defined interface for handing issues for clarification in a universal way. Customer satisfaction has been improved by classification of the issues and reducing the number of necessary interactions with the customers.

Known Uses

- The pattern has been used in several projects for the purpose of consolidating outbound communication to the same party in order to save costs by putting the communication content resulting from various business processes running in parallel. in one bundled communication. The bundle process controls the actual generation of the communication including format and media, i.e. letter, e-mail, fax, or even telephone, and the procedure to control the outcome of the communication. The purpose in the context was also to improve customer satisfaction via the consolidation of the communication via reducing the number of interactions and applying preferred communication mechanisms.
- The pattern has also been used to gather issues to be clarified with customers that result from various business processes running in parallel associated to a complex order. The issues are collected first and are then clarified with the customer rather than discussing each issue separately with the customer. That way issues could be clarified in relation to each other. The bundle process controls the clarification procedure in terms of a dedicated business process.
- The above two purposes of the pattern have been used in the context of order management in the telecoms industry and in the context of claims handling in the insurance industry. The pattern has served in this context as an architectural solution to the consolidation issues mentioned in larger strategic architecture projects.

Process Conductor

Interdependent processes are in execution on a process engine. The interdependency implies that execution of the processes needs to be synchronized at runtime. When business processes are executed on a process engine there are often points in these processes where dependencies to other processes need to be considered. That means a process may only move to a certain state, but can only move on if other parallel processes have also reached a certain state. Further execution of these parallel running processes need to be orchestrated at runtime, as it cannot be decided at modeling time when these states are reached or what these states are due to the separate execution of the processes and the fact that each process is a component that may change individually over time. In most cases, the rules for the orchestration need to be flexibly adaptable.

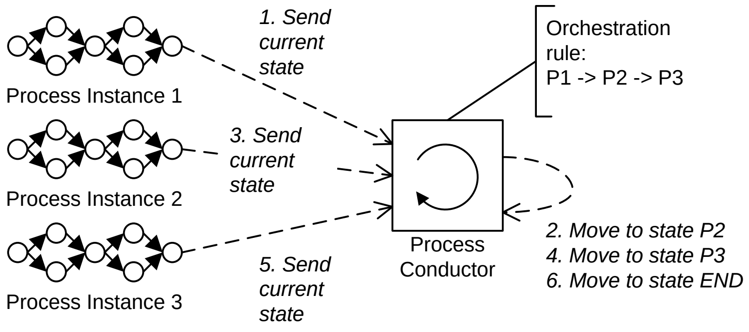
How can the interdependencies between processes be flexibly captured without tightly coupling the processes?

At some point in a process it might be necessary that the process is only allowed to move on if other processes have also reached a certain state. However, each individual process does not know what these states of other processes are and when they are actually reached, as each process runs at its own speed within its own data space. Moreover, each business process needs to be treated as a separate component and may change individually over time. Thus, processes need to be very loosely coupled as far as this aspect is concerned.

The reason for this is that it is very hard to specify in a single process model what states of other processes are relevant from an individual process's point of view and when these states are reached, nor what the relevant dependent processes are. If this is statically modeled in a process somehow the implementation will be very inflexible and changes to orchestration rules usually impact all involved processes. That means, the actual orchestration appears to be a complex concern of its own and the rules for orchestration cannot be defined attached to an individual process model. Consequently, the orchestration rules should not be captured as some types of static relationships of a process to other processes. The dependencies that will be generated if each process should know the rules for orchestration will be very hard to manage. If the rules change then each individual process needs to be changed as well. For this reason a tight coupling of the rules to each individual process is an inappropriate approach.

As a result, each process needs to be treated as an encapsulated component that does not know anything about the orchestration rules or the processes that it has dependencies to. Each process must only know its own points in the process where the dependency occurs but not what this dependency is about. New processes might be created over time, which create new dependencies and this must not affect existing process models to make the changes manageable. Each process model itself may also change, e.g. new steps are added without affecting the actual rules for orchestration as they need to be treated as a separate concern. The very problem is thus that the processes are standing in dependency but must actually not know very much about each other, as the dependency needs to be separated out of the process to treat it as a separate concern and to make the processes and the complexity generated by these dependencies manageable.

Introduce a PROCESS CONDUCTOR component that offers configurable orchestration rules to conduct a number of business process instances. The PROCESS CONDUCTOR offers a service that is only invoked synchronously by the business process instances. Each process instance provides its current state in terms of the activity it currently performs as an input parameter to this service. The service returns a result to a specific process instance only when the orchestration rules allow the process to move to the next step. This way the order of the process instances to proceed is determined via the orchestration rules of the PROCESS CONDUCTOR.



A central aspect of the PROCESS CONDUCTOR pattern is that the central conductor is only invoked synchronously. That is, when a business process reaches a critical state where it may only move on if certain other dependent processes have also reached a certain state, then a SYNCHRONOUS SERVICE ACTIVITY [Hentrich et al. 2008] is modeled at this point in the process that invokes a service. At this point, the process to be conducted blocks on the synchronous invocation until the conductor returns a result. The PROCESS CONDUCTOR service reports the state of a process instance and the ID of the instance to the PROCESS CONDUCTOR component. The states and corresponding process IDs are stored in a container. The PROCESS CONDUCTOR component applies orchestration rules which are configurable to determine the order of events that need to be fired to initiate dedicated process instances to move on.

The PROCESS CONDUCTOR applies its orchestration rules to the states and corresponding process IDs in the container. The orchestration rules simply define an order of the process states, i.e. an order of terminating the corresponding process activities. The conductor then fires events to the process instances identified by their IDs in the order that is determined by the orchestration rules. Hence, the service implementation to report the state and the process ID can be implemented as an EVENT-BASED ACTIVITY [Köllmann et al. 2007]. The process engine receives the events and terminates the activities in the order directed by the conductor. As a consequence the processes move on to the next step in the right order. The conductor repeats this process in a loop, as new processes may have registered for the next iteration.

The triggers to start one iteration of this procedure to apply the orchestration rules and to fire events to the processes can be twofold. It can happen repeatedly in defined time intervals, or it can be initiated by other dedicated event triggers, e.g. a master process has invoked the service of the PROCESS CONDUCTOR to register an initiation state that triggers the orchestration rules.

The registration of the state and process ID and the waiting position for the actual termination event to occur can also be designed as an ASYNCHRONOUS RESULT SERVICE [Hentrich et al. 2008]. In this case the business process needs to model two activities: one that places the request and a second one that gets the result, i.e. waits for the termination event.

The EVENT-BASED PROCESS INSTANCE [Hentrich 2004] pattern can also be used in conjunction with a PROCESS CONDUCTOR in case it might take a long time until the termination event occurs and it makes sense to split-up the process in two parts. That means if the termination event occurs the second part of the process will be instantiated rather than modelling a waiting position.

One must note that the pattern generally assumes that the process engine processes the terminate events in the sequence that they are fired. This implies that the activities will terminate in the intended order, i.e. the order the terminate events have been fired, and the processes will correspondingly move on in the right order. If this cannot be assumed then it might be that the activities of the processes do not terminate in the right order and consequently the processes do not move on in the right order as well. To resolve this issue the implementation can be extended by an additional service that is invoked from a business process. This additional service confirms that the activity has terminated. This is modelled as a second SYNCHRONOUS SERVICE ACTIVITY right after the first one. The next process, according to the rules, is only notified after the confirmation from the preceding business process has been received.

This may only be necessary if the order of termination is important within one iteration of notification. In many cases this is not important as it is rather the whole iteration that represents the order, i.e. all processes of one iteration may literally move on at the same time and slight differences do not matter. This also depends on the underlying rules and what these rules are further based on. According to the MACRO-MICROFLOW [Hentrich et al. 2007] pattern this may also depend on whether we are at microflow or macroflow level. Transactional microflows usually run in much shorter time (sometimes milliseconds) and even slight time differences might matter while these slight time differences might not matter at all at the macroflow level.

The pattern provides a flexible solution to determine the order of process steps that need to be synchronized by configurable rules. New processes and rules can be added without changing the architecture. Existing rules can also be modified without changing the implementation of running business processes. However, this flexible concept requires additional design and implementation effort. The design might be quite complex depending on the requirements regarding the complexity of the synchronization. For this reason, the pattern is most suitable in larger projects where architecture evolution and business agility is required.

The PROCESS CONDUCTOR pattern is a central bottleneck and it also incurs the risk of deadlocks in case the PROCESS CONDUCTOR’S orchestration rules are misconfigured or a business process fails to signal its state. In such cases, usually manual intervention is required. It makes sense to monitor the PROCESS CONDUCTOR component for such events.

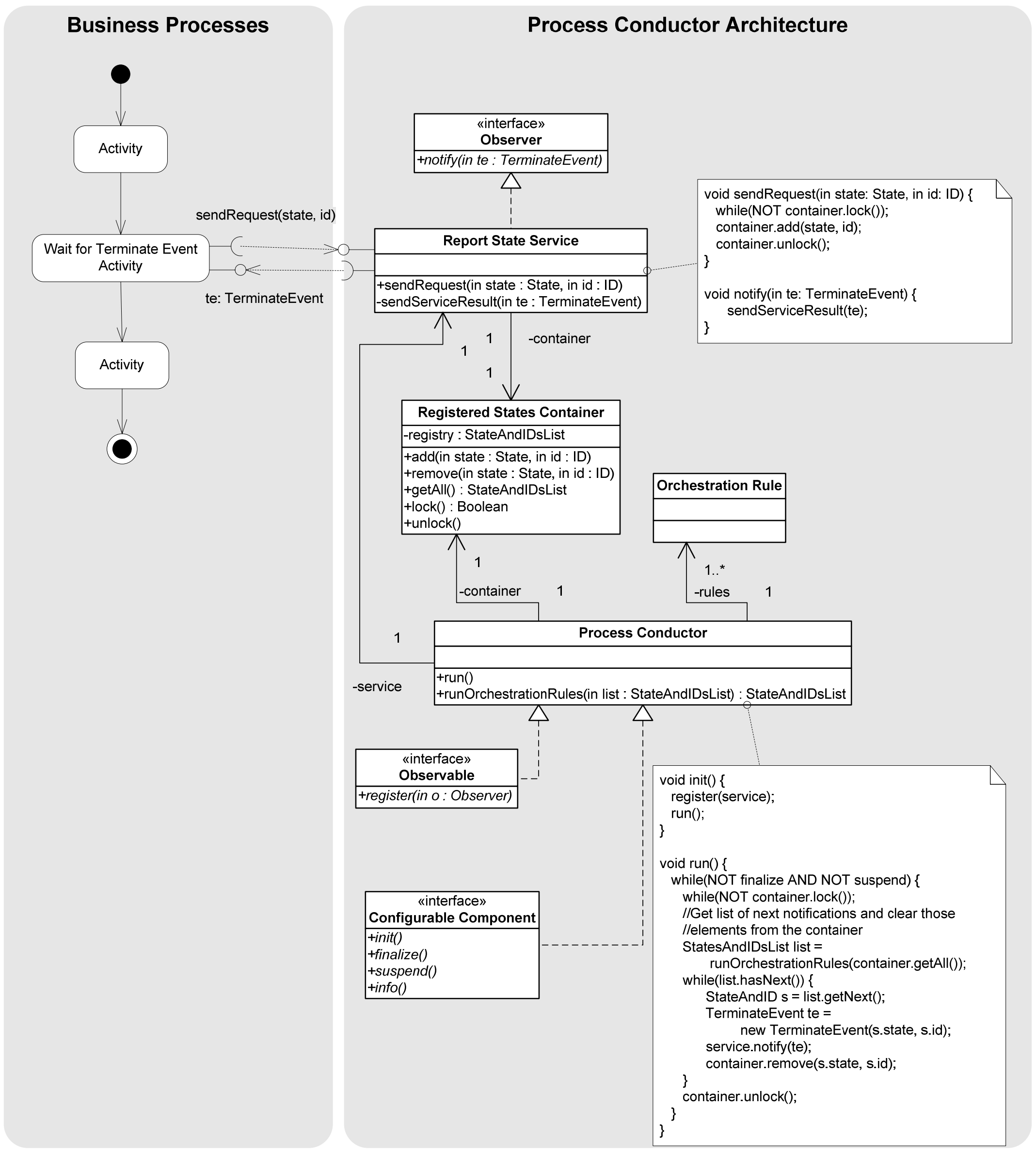
Example: PROCESS CONDUCTOR example configuration

The following figure shows an example of the general architectural concept of the solution using the OBSERVER pattern [Gamma et al. 1994] to notify the EVENT-BASED ACTIVITY of a process that waits for a terminate event to occur. The very order of sending these terminate events, i.e. the order of invoking the *notify* method of the observer class, is defined by the orchestration rules of the conductor. The orchestration rules just order a list of states associated to process instances and deliver those process instances that need to be informed in the next iteration. In the example in the figure this logic is hidden in the *runOrchestrationRules* method. That method takes a list of states associated to process instance IDs, runs the rules over them and delivers the list of process instances that apply to the rules for the next iteration. Those process instances are removed from the list given as an input parameter. The *Process Conductor* class itself is implemented using the COMPONENT CONFIGURATOR [Schmidt et al. 2000] pattern. In the figure, the class *Process Conductor* is the observable that is observed by a *Report State Service*. This service is invoked by activities from business processes as a SYNCHRONOUS SERVICE ACTIVITY [Hentrich et al. 2008]. The service notifies the process engine about terminating an activity when it receives a corresponding *TerminateEvent* from the conductor. Read and write operations on the container are synchronized using locking and unlocking mechanisms.

A variation of the pattern is that the registration of the state and process ID and the waiting position for the actual termination event to occur can also be designed as an ASYNCHRONOUS RESULT SERVICE [Hentrich et al. 2008].]. To model this variant, the *Report State Service* offers two methods: one that places the request and one that gets the result. The combination of state and ID may serve as a CORRELATION IDENTIFIER [Hohpe et al. 2003] for the second method invocation. The architectural concept then needs to change slightly according to the description of the ASYNCHRONOUS RESULT SERVICE pattern. Using this design principle the terminate event is rather captured by a pull mechanism from the perspective of the process conductor. The pull mechanism is represented by the second service invocation from the business process that actively asks for a result and the termination event might thus not be immediately reported to the process engine, i.e. in case the second method is invoked after the event has actually occurred.

On the contrary the original solution in the figure above seems rather to use a push-mechanism while following the EVENT-BASED ACTIVITY pattern, as the event is fired and reported to the process engine as soon as it occurs. However, from the viewpoint of the business process both scenarios follow a pull-mechanism, as all services are actively invoked and represent blocking calls. In the second scenario it is just two method invocations instead of just one. The second method to get the result represents the EVENT-BASED ACTIVITY in this second scenario.

The second variation of the solution can be used when sending the request needs to be decoupled from capturing the termination event. For instance, in case other process steps can be undertaken in the meantime but the conductor needs to be informed early. Doing it that way creates more time for the conductor to calculate the order of the terminate events, e.g. in case complex time consuming rules need to be applied and/or it is not necessary to report the termination event to the process engine as soon as possible.


Business Processes
Process Conductor Architecture
Activity
Wait for Terminate Event Activity
Activity
sendRequest(state, id)
te: TerminateEvent
«interface»
Observer
+notify(in te : TerminateEvent)
Report State Service
+sendRequest(in state : State, in id : ID)
-sendServiceResult(in te : TerminateEvent)
1
1
-container
1
Registered States Container
-registry : StateAndIDsList
+add(in state : State, in id : ID)
+remove(in state : State, in id : ID)
+getAll() : StateAndIDsList
+lock() : Boolean
+unlock()
Orchestration Rule
1
-container
1
1..*
-rules
1
1
-service
Process Conductor
+run()
+runOrchestrationRules(in list : StateAndIDsList) : StateAndIDsList
«interface»
Observable
+register(in o : Observer)
«interface»
Configurable Component
+init()
+finalize()
+suspend()
+info()

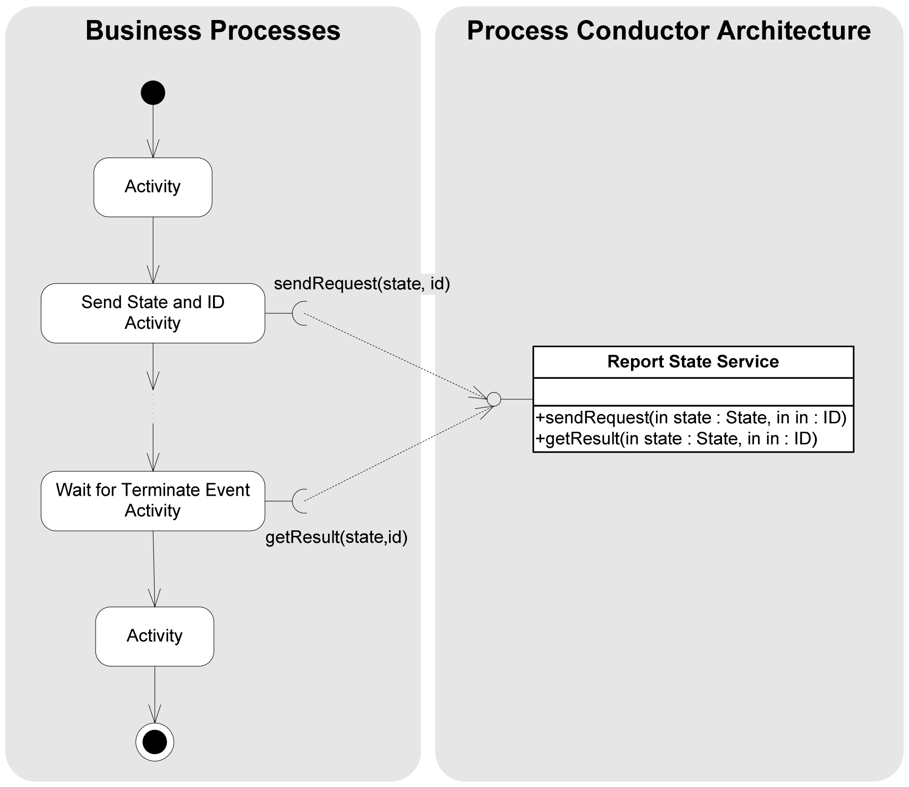
```
void sendRequest(in state: State, in id: ID) {
   while(NOT container.lock());
   container.add(state, id);
   container.unlock();
}

void notify(in te: TerminateEvent) {
      sendServiceResult(te);
}
```

```
void init() {
   register(service);
   run();
}

void run() {
   while(NOT finalize AND NOT suspend) {
      while(NOT container.lock());
      //Get list of next notifications and clear those
      //elements from the container
      StatesAndIDsList list =
             runOrchestrationRules(container.getAll());
      while(list.hasNext()) {
            StateAndID s = list.getNext();
            TerminateEvent te =
                     new TerminateEvent(s.state, s.id);
            service.notify(te);
            container.remove(s.state, s.id);
      }
      container.unlock();
   }
}
```

The following figure illustrates the variation of the pattern using ASYNCHRONOUS REPLY SERVICE.


Business Processes
Process Conductor Architecture
Activity
Send State and ID Activity
sendRequest(state, id)
Report State Service
+sendRequest(in state : State, in in : ID)
+getResult(in state : State, in in : ID)
Wait for Terminate Event Activity
getResult(state,id)
Activity

Example: Just-in-time Production Optimization

In a just-in-time (JIT) production scenario in the automotive industry the order of a car needs to be processed. The order arrives with given ordering details of the car model that needs to be manufactured in a JIT process. The parts for the car are delivered by different suppliers and the order details related to the parts delivered by a certain supplier are forwarded to each of the suppliers via a service interface. The manufacturing process in terms of the internal ordering, delivery, and assembly of the parts from those different suppliers needs to be coordinated. To coordinate the processes a MACROFLOW ENGINE [Hentrich et al. 2007] is used. The selected tool to implement the MACROFLOW ENGINE is WebSphere MQ Workflow.

The processes for ordering, delivering and assembling the parts need to be coordinated, as a parallel process instance is created for each supplier. Appropriate coordination of the process is crucial to optimize the manufacturing costs, e.g. reducing stock costs by agreeing service levels with suppliers in terms of delivery times and to place the order to the suppliers at the right point in time. In order to allow coordination of the processes and to allow optimization the PROCESS CONDUCTOR pattern has been applied. The timely coordination of orders to suppliers, parts delivery and assembly can thus be implemented using flexible orchestration rules. The rules can be modified according to improved service level agreements and to optimize the overall manufacturing process over time. The rules have been implemented and flexibly configured using the ILOG JRules [ILOG 2008] rules engine. The rules have been accessed by the PROCESS CONDUCTOR via a Java interface.

Known Uses

- The pattern has been used in projects in the telecoms industry to control technical activation of dependent products in larger orders. Each product can be processed in parallel to improve efficiency up to a certain point. Further technical activation of the products is then controlled by product rules, as certain products are dependent on each other, e.g. an internet account can only be activated if the physical DSL connection has been established.
- The pattern has been used to design synchronization points in parallel claims handling processes in the insurance industry. That is, a set of parallel sub-claims processes that belong to an overall claim is triggered off and can only move to a certain point. At this point the parallel processes need to be synchronized, i.e. the first process that reaches the point must wait until all others have reached their synchronization point too. What processes need to be synchronized is defined by configurable rules.
- In logistic processes in the transportation industry the pattern has been used to flexibly coordinate the transportation of goods delivered by different suppliers and parties. Orchestration rules have been used to allow flexible packaging and to coordinate between different types of transportation, e.g. trucks, planes, ships, and trains. That way it is possible to rather easily configure modified types of packaging and transportation due to changed conditions or different transportation criteria, e.g. security, delivery speed, and costs, which apply to different types of goods.

Conclusion

In this paper we have presented three patterns for synchronization of parallel and independently running business processes in a process-driven and service-oriented architecture. These patterns focus on coordinating the parallel and principally independent business processes via architectural solutions allowing architects to model the flexible synchronisation of the processes. The patterns are part of an ongoing effort to mine and document a pattern language for process-driven and service-oriented architecture. Previous parts of this pattern language have been published in [Hentrich 2004, Hentrich et al. 2007, Köllmann et al. 2006, Hentrich et al. 2008].

Acknowledgments

The authors would like to thank our shepherd Stephan Lukosch and the EuroPLoP 2008 writers' workshop for their valuable comments.

References

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

[Buschmann et al. 2000] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal: Pattern-Oriented Software Architecture – A System of Patterns, John Wiley & Sons, 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.

[Gamma et al. 1994] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[Hentrich 2004] C. Hentrich. Six patterns for process-driven architectures. In Proceedings of the 9th Conference on Pattern Languages of Programs (EuroPLoP 2004), 2004.

[Hentrich et al. 2007] C. Hentrich, U. Zdun. Patterns for Process-Oriented Integration in Service-Oriented Architectures, In Proceedings of the 11th Conference on Pattern Languages of Programs, (EuroPLoP 2006), 2006.

[Hentrich et al. 2008] C. Hentrich, U. Zdun. Patterns for Invoking Services from Business Processes. In Proceedings of European Conference on Pattern Languages of Programs (EuroPloP) 2007, Universiätsverlag Konstanz, 2008.

[Hohpe et al. 2003] G. Hohpe and B. Woolf. Enterprise Integration Patterns. Addison-Wesley, 2003.

[ILOG 2008] ILOG. JRules. http://www.ilog.com/products/jrules/index.cfm, 2008.

[Köllmann et al. 2007] T. Köllmann, C. Hentrich. Synchronization Patterns for Process-Driven and Service-Oriented Architectures, In Proceedings of the 11th Conference on Pattern Languages of Programs, (EuroPLoP 2006), 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.

[Völter et al. 2004] M. Voelter, M. Kircher, and U. Zdun. Remoting Patterns - Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware, Wiley Series in Software Design Patterns, J. Wiley & Sons, October, 2004.

Appendix: Overview of Referenced Related Patterns

There are several important related patterns referenced in this paper, which are described in other papers, as indicated by the corresponding references in the text. The following table gives an overview of thumbnails of these patterns in order to provide a brief introduction to them for the reader. For detailed descriptions of these patterns please refer to the referenced articles.

Pattern	*Problem*	*Solution*
ASYNCHRONOUS RESULT SERVICE [Hentrich et al. 2008]	A communication between a service and a process flow needs to be modelled that is not a synchronous communication, but rather just places the service request and picks up the service result later on in the process flow, analogous to the well-known callback principle.	Split the request for service execution and the request for the corresponding result in two SYNCHRONOUS SERVICE ACTIVITIES and relate the two activities by a CORRELATION IDENTIFIER [Hohpe et al. 2003] that is kept in a control data object.
BUSINESS OBJECT REFERENCE [Hentrich 2004]	How can 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.
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.

Pattern	*Problem*	*Solution*
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.
EVENT BASED PROCESS INSTANCE [Hentrich 2004]	How can a process instance be automatically created in case an event based activity occurs in a business process that implies automatic process instantiation, e.g. a customer placing an order?	Externalise event based activities to an external event handler component and split the process model in several parts.
EVENT-BASED ACTIVITY [Köllmann et al. 2006]	How can events that occur outside the space of a process instance be handled in the process flow?	Model an event-based activity that waits for events to occur and that terminates if they do so.
FIRE AND FORGET SERVICE ACTIVITY [Hentrich et al. 2008]	A communication between a service and a process flow needs to be modelled that is not a synchronous communication, but rather just placing the service request without waiting for any result to be returned from the service.	Model a FIRE AND FORGET SERVICE ACTIVITY that decouples the request for execution of a service from the actual execution of the service.
MACROFLOW ENGINE [Hentrich et al. 2007]	How is it possible to flexibly configure macroflows in a dynamic environment where business process changes are regular practice, in order to reduce implementation time and effort of these business process changes, as far as the related IT issues are concerned that are involved in these changes?	Delegate the macroflow aspects of the business process definition and execution to a dedicated MACROFLOW ENGINE that allows developers to configure business processes by flexibly orchestrating execution of macroflow activities and the related business functions.
MACRO-MICROFLOW [Hentrich et al. 2007]	How is it possible to conceptually structure process models in a way that makes clear which parts will be depicted on a process engine as long running business process flows and which parts of the process will be depicted inside of higher-level business activities as rather short running technical flows?	Structure a process model into macroflow and microflow.
SYNCHRONOUS SERVICE ACTIVITY [Hentrich et al. 2008]	A synchronous communication between a service and a process flow needs to be modelled such that the process is able to consider the functional interface of the service and may react on the possible results of the service.	Model a SYNCHRONOUS SERVICE ACTIVITY that depicts the functional input parameters of the associated service in its input data objects and the functional output parameters of the service in its output data objects.