Transactional Coordination of Dynamic Processes in Service-Oriented Environments

Martin Husemann, Michael von Riegen, Norbert Ritter
Distributed Systems and Information Systems
University of Hamburg
Vogt-Kölln-Straße 30, 22527 Hamburg, Germany
{husemann,riegen,ritter}@informatik.uni-hamburg.de

Abstract

Service-oriented environments facilitate dynamic processes whose properties can be altered during runtime. The transactional support of such processes holds specific requirements that are not completely covered by existing specifications. In this paper, we introduce a life cycle model for transactional dynamic processes and analyze existing specifications with respect to their potentials to support such a model. Subsequently, we propose a framework resolving the weaknesses of existing specifications and allowing comprehensive transactional coordination of dynamic processes.

1 Introduction

Service-oriented architectures (SOA) are a popular paradigm to implement loosely-coupled distributed environments [15]. Within these environments, participants can overcome heterogeneity by communicating through standardized, implementation-independent interfaces. Application logic is typically not located in monolithic programs, but distributed between the participants of processes. This approach is referred to as *service-oriented computing* (SOC).

The fundamental concept of *Grid Computing* is to transparently provide resources to participants in a network. Recent efforts in this field have been directed at developing a technical platform for grids based on concepts of service-oriented computing and Web services [7]. Today, the *Web Services Resource Framework* [6] provides a standard reconciling the requirements of the classical Web service world with Grid-specific demands. A *Grid service* is basically a stateful Web service which provides services and can be discovered and used by service consumers within the Grid. Grids built from such Grid services are called *Service Grids*.

Although this term is deduced from the technical platform, it is also meaningful regarding the contents: Due to the high level of abstraction permitted by Grid services, Service Grids are not limited to the provision of simple resources such as computational power or storage capacity, but they can also provide complex services such as those known from Web service environments, e. g. search engine functionality or flight booking. In contrast to rather uniform *Computing Grids* or *Data Grids*, Service Grids are inherently more heterogeneous and show a greater dynamism regarding the set of participants. A visionary global Service Grid could reach the variety and complexity of today's World Wide Web.

Service Grids are an interesting implementation of service-oriented computing since they contribute Grid-specific properties such as dynamism and self-management. Collaborative processes among Grid participants can then run as *dynamic processes*. Ensuring transactional behavior under these circumstances is challenging; current Web service specifications for transactional coordination only cover some of the requirements brought along by dynamic processes.

In this paper, we present our work on transactional coordination of dynamic processes in service-oriented environments. Section 2 introduces our understanding of dynamic processes and associated aspects of interaction primitives and transaction management. Existing specifications and related work are discussed in Section 3. Section 4 presents our TracG framework aimed at supporting transactions within dynamic processes. Section 5 gives a conclusion and an outlook on future work.

2 Dynamic Processes

A process is a definition of a series of steps to reach a specified *goal*. The goal describes functional and non-functional requirements on the results and the execution of the process; it is typically specified by the *initiator* of the

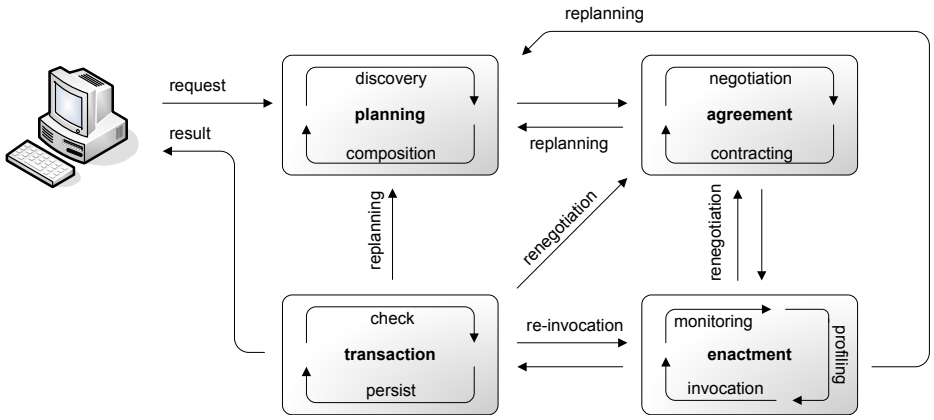


Figure 1. Life cycle of a dynamic process

process. Usually, a process is executed among a set of participants that collaborate to achieve a *consistent outcome*, even if they compete with respect to the individually desired results. The process steps are carried out in sequential or parallel *activities* on the participants.

Dynamism as a basic principle denotes the capability to adapt to changing conditions. With respect to processes, we regard dynamism as *the capability to modify the process structure during runtime*. Modifications of the process structure may arise from two cases: Firstly, as a reaction to unexpected events in the course of the process, to continue the execution towards the original process goal in consideration of the new conditions. Secondly, as a means to change the process goal and accordingly adapt the execution towards the new goal. The first case comprises exception handling, for example when the failure of a participant needs to be compensated by locating and binding an adequate replacement to avoid a failure of the whole process. However, unexpected events are not limited to errors or failures. In order to take advantage of the flexibility in agile service-oriented environments, processes should be modeled with a certain degree of freedom on the part of the participants. Consequently, the process execution cannot be specified beforehand in full detail. Instead, coarsely defined process steps need to be refined at runtime, complying with the current environmental conditions. For example, participants may wish to leave the process or they may be dismissed by superordinate participants. In such scenarios, the process execution cannot be bound to rigidly predefined structures, but it needs to be capable of short-term adaptations. Unexpected events are then not a materialization of errors, but a common phenomenon.

If such a flexible execution of processes is supported, it also allows changing the aspects of the process goal that are concerned with requirements on the execution. Changes of the aspired process *results* during runtime are rare when those can be modeled with a degree of freedom. If the premises change in such a radical way that a change of the aspired results becomes necessary, it is more convenient to abort and restart the process than to respecify all the affected aspects during runtime. We therefore concentrate on processes with stable aspired results. For such processes, we identified three degrees of adaptability:

- *static*: A process is specified in all its details before the execution is started; adaptations during runtime are not possible.
- *ad-hoc-static*: Participants and flow are determined according to the goal at the process start. After the process start, adaptations are not possible.
- *dynamic*: Participants and flow are determined at the process start, but they can be changed during runtime.

Ad-hoc-static and dynamic processes can be specified by merely noting the process goal. Participants and flow of a concrete process instance are then determined corresponding to the current circumstances during its creation.

2.1 Life Cycle of a Dynamic Process

In service-oriented environments, dynamism for processes in practice comprises short-term service composition, late binding of service instances and adapting to changes caused by errors or modified demands during the process execution. Figure 1 shows the life cycle of a dynamic process from the specification of the goal to the delivery of the results. The basic form of this life cycle was introduced with the *Adaptive Services Grid Platform* [11], however, in its original form, aspects of transaction support are not taken into consideration.

A dynamic process is initiated by a client posing a request in terms of a process goal to a *service bus* [15]. The re-

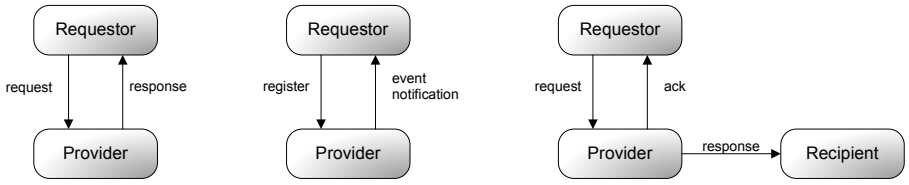


Figure 2. Interaction primitives

quest defines the process goal and the transactional requirements such as coordination types and alternative activities in case of exceptions. In the *planning* phase, an execution plan to achieve the goal is constructed. Suitable services are located and, where necessary, adequately composed to attain the required functionality. The service bus shields the client from possible service compositions through the generation of *virtual services* that are monolithic towards the client and conform to the requested functionality.

In the second phase, *agreements* for the invocation of concrete service instances are negotiated. This comprises for instance QoS requirements and transactional capabilities of services. If unrecoverable exceptions occur in this phase, the life cycle processing returns to the planning phase to determine another way to achieve the process goal by *replanning* the process with different services.

If the agreements have successfully been negotiated, the *enactment* of activities takes place in the third phase. Unrecoverable exceptions in this phase can be addressed by returns to the agreement phase (*renegotiation*) or the planning phase, depending on their severity.

If transactional behavior was specified in the request, the *transactional termination* of the process is carried out in the fourth phase through a transaction protocol. The state of the participants is verified, and if applicable, the process results are committed. Unrecoverable exceptions in this phase can lead to returns to the enactment phase (*reenactment*), the agreement phase, or the planning phase.

The life cycle of an ad-hoc-static process is largely identical to that of a dynamic process. After entering the enactment phase, however, returns to the agreement phase or the planning phase are not possible. The capabilities to react to exceptions are consequently limited, but the process can still be initiated in consideration of the current environmental conditions. The life cycle of a static process only comprises the enactment phase and optionally the transactional termination phase. Its potential for adaptability is thus mostly limited to retries of the predefined activities.

The specification and execution of dynamic processes is a complex field of its own which is beyond the scope of this paper. Here, we concentrate on transaction management in such processes.

2.2 Interaction Primitives

Analyzing service interactions in dynamic environments in special consideration of transactional support, we identified a set of *interaction primitives* from which complex interactions can be constructed. In contrast to the various *service interaction patterns* [2], the set of interaction primitives is small-sized. A separate provision for multilateral interactions is unnecessary since such scenarios can be reduced to aggregations of bilateral interactions. For similar reasons, the distinction between single-transmission and multi-transmission interactions can be omitted. We thus primarily distinguish between *immediate interactions* and *decoupled interactions*.

Immediate interactions occur in scenarios where two or more participants interact *direct*, i. e., without intermediate nodes or event-based decoupling. In the basic case, there are exactly two participants. More complex cases comprise 1:n constellations where a participant interacts with a set of other participants in a set of pairwise communication acts. In either case, the interactions may take place in a *one-way* or *two-way* manner. In one-way interactions, the sending of a message or a request is not followed by a response from the receiving participant; in two-way interactions, responses are sent. *Reliable transport* mechanisms with acknowledgments generally assure successful delivery of messages. Messaging can be performed synchronously or asynchronously. Figure 2(a) shows a direct two-way interaction between two participants.

Decoupled interactions occur in scenarios where two or more participants interact indirect or weakly coupled. Such decoupling can be conducted at the application logic level or the message flow level. Participants are decoupled at the application logic level by the *observer* interaction primitive as shown in Figure 2(b). A participant (requestor) registers with another participant (provider) to be notified in case of a certain type of event. When the event occurs, the provider sends a notification to the requestor. More than one requestor may register for a given type of event, and a provider may provide events of more than one type. Registered requestors do not know about other registered requestors, and the provider does not know the intentions of the requestors. Event notifications can therefore set off indi-

vidual activities opaque to the provider; although messages are exchanged direct between two participants respectively, the interaction is decoupled at the application logic level.

Decoupling at the message flow level is implemented by the *3rd-party* interaction primitive. A typical constellation is shown in Figure 2(c). A requestor poses a request to a provider, but the provider sends the response to another participant (recipient). The recipient can be determined by the requestor or from the results of the request processed by the provider; there may be more than one recipient. The requestor is optionally sent an acknowledgment of the successful execution. Requestor and recipient interact indirect or totally decoupled via the provider. Such decoupled interaction also allows the requestor to withdraw from the process after sending its request to the provider. If recipients are allowed to actively register with the provider for specific requests, the 3rd-party interaction primitive also enables *publish/subscribe* constellations with the provider acting as a hub between the requestor and the recipients.

2.3 Requirements on Transaction Management

Dynamic processes entail specific requirements on transactional activity control. Aborts or restarts of complex and long running processes have to be avoided. Consequently, changes of the execution structures during runtime of a process are to have minimal effects on active transactions. A framework for transactional activity control of dynamic processes should provide support of the following requirements:

- The coordination of distributed transactions with growing and shrinking sets of participants. Participants are able to join and leave a running process in a controlled manner. Superordinate participants can expel subordinate participants from the process.
- Dynamically adaptable execution structures and dynamic privilege allocation.
- Interaction scenarios containing decoupled interaction primitives.
- The resuming of interrupted (sub-)transactions with minimal loss of work after exceptions such as errors or adaptations of the execution structures.

Central aspects are the support of variable sets of participants, complex interaction scenarios, and process reconfigurations. For instance, to fully leverage the 3rd-party interaction primitive, the privilege of transaction demarcation needs to be portable among the participants of a process. The initiator of a 3rd-party interaction can usually not determine a suitable time to trigger the termination of the process and should therefore pass the demarcation privilege on to the recipient of the result.

3 Existing Specifications and Related Work

A comprehensive platform for dynamic processes has been introduced with the Adaptive Services Grid [11], but in this project, very little attention is paid to transaction management.

Several approaches for transactions in Web and Grid environments have been proposed. Jin and Goschnick introduced agent-based transactions [9], claiming that the usage of agent technology for transaction management is the right choice within a Web service environment. Their approach seems to be in a very early state. A peer-to-peer approach for Grid transactions was presented by Türker et al. [17]. It introduces a decentralized transaction model based on an optimistic variant of serialization graph testing. This approach is innovative in uniquely combining existing concepts and techniques for a new purpose. However, these concepts are more concerned with Grid-specific problems or paradigm-based suggestions instead of transaction processing in dynamic service-oriented environments.

Only few articles address annotations of Web services to support discovery and composition. The WS-AtomicTransaction [4] and WS-BusinessActivities [5] specifications mention policies expressed in WS-Policy documents to be attached to WSDL descriptions. Such concepts are also mentioned for example by Tai et al. [16] or Montagut and Molva [14]. None of these papers, however, is concerned with annotating the transactional behavior of services.

Several specifications for the management of distributed transactions in service-oriented environments have been put forward, the most prominent being *WS-Coordination* [13], *WS-CAF* [3], and *BTP* [8], which are all based on participant/coordinator patterns. BTP and WS-Tx (an earlier version of WS-Coordination) have previously been examined and compared by Little and Freund [13]. An overview and comparison of WS-Coordination, WS-CAF and BTP has been given by Kratz [10]. The *Transaction Management Research Group*[1] (TM-RG) of the *Global Grid Forum* (GGF) also examined these specifications in special consideration of their suitability for Grid environments. However, none of these investigations put emphasis on specific aspects and requirements of dynamic processes.

[1] http://forge.gridforum.org/sf/projects/tm-rg

3.1 Analysis of Prevalent Specifications

WS-Coordination (WS-C) administrates participants via a common *context* managed by the coordinator and provides two different *coordination types*. In *WS-AtomicTransaction*, the demarcation of transactions is controlled with `commit` or `rollback` messages according to the *completion protocol*; transactions are processed after the 2PC protocol. *WS-BusinessActivities* allows to coordinate participants according to the *atomic outcome* or the *mixed outcome* type. With mixed outcome, a transaction can also terminate successfully if only a subset of the participants commits. The coordinator decides to terminate or roll back the transaction after all the participants have completed their work.

The *Web Services Composite Application Framework* (WS-CAF) also employs common coordination contexts. It consists of the three frameworks *WS-Context* (WS-CTX), *WS-Coordination Framework* (WS-CF), and *WS-Transaction Management* (WS-TXM), featuring coordination protocols for distributed ACID transactions, long-running activities, and business transactions with support of non-atomic outcomes.

Business Transaction Protocol (BTP) is based on a modified 2PC protocol supporting both ACID and business transactions. With ACID transactions, the protocol proceeds as classic 2PC. Business transactions allow individual decisions about `confirm` and `cancel` messages between the two phases; a transaction can also terminate successfully if some participants are sent `cancel` messages.

All three specifications support ACID and long-running business transactions. The processing of business transactions is dependent on application logic and requires specific knowledge within the coordinator. The specifications show similar weaknesses in supporting requirements of dynamic processes such as variable sets of participants, complex interaction scenarios, and modifications of process structures. Emission of participants is not supported in the ACID protocols of WS-C and WS-CAF; it is ungoverned in the business activity protocols of WS-C and WS-CAF as well as in BTP. None of the specifications provides for an explicit management of the transaction demarcation privilege and modifications of process properties during runtime. Decoupled interaction primitives are not properly supported since none of the specifications features according control mechanisms. A specific limitation of BTP is the restriction to a two-phase protocol. WS-CAF and WS-Coordination show greater flexibility by allowing the implementation of individual protocols. All three specifications offer extension points where enhancements can be added.

4 TracG Framework

In this section, we propose our framework *Transactional activity control for the Grid*, which meets the requirements of dynamic processes on transaction management described in Section 2.3. We chose WS-Coordination as a basis of our framework since it is more lightweight than WS-CAF and more flexible than BTP. The focus in this paper is on the exit handshake protocol, which enables a process to adapt to changing conditions during runtime. Furthermore, we introduce service annotations for the discovery and binding of services depending on their transactional capabilities and explicate the propagation of the demarcation privilege.

4.1 Exit Handshake

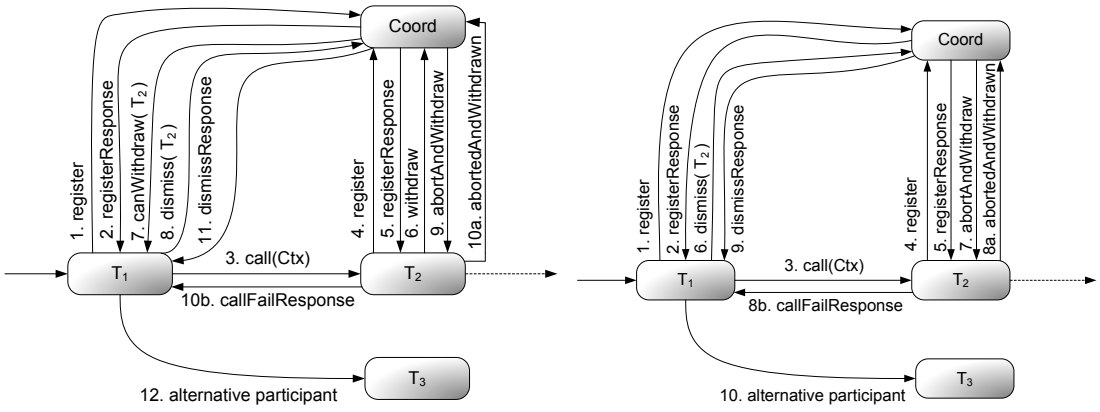
The ungoverned emission of a participant from a process may be fatal for a transaction or the whole process. A transaction management framework needs to minimize the effects of leaving participants such that the process can terminate successfully. The task of a leaving participant can for example be taken over by another, similar service. Such a takeover of tasks depends on infrastructural support. We distinguish two variants of participant emission:

- A participant wants to leave the process on its own accord, i. e., it wants to *withdraw* from the process. This may be the case in scenarios of high load on a participant or if a participant is not able to meet specified requirements on its task.

- A participant has introduced another participant into the process and decides to remove it from the process again, i. e., to *expel* it. This may be the case if the subordinate participant is not needed anymore or if a superior alternative has been detected.

In order to avoid aborts and errors, an *exit handshake* is required for the riskless emission of a participant. The coordinator acts as a mediator in such a handshake. Since the coordinator usually does not possess the necessary knowledge to decide on the vitality of a participant and consequently about the emission, the superordinate participant needs to make this decision. For reasons of clarity, we refer to a superordinate participant as *father* and to subordinate participants as *children*. In order to act as a mediator, the coordinator requires knowledge on the message flow and control hierarchy in the process. Therefore, every participant has to report its father to the coordinator during registration. The protocols for withdrawal and expulsion are shown in Figures 3(a) and 3(b), respectively.

For a participant *withdrawal*, the following cases can be distinguished:



(a) Participant withdrawal (b) Participant expulsion

Figure 3. Exit handshake

- The withdrawing participant has a father which participates in the process: The coordinator asks the father about the vitality of the withdrawing child and thus whether the withdrawal can be permitted. Participating children of the withdrawing child have to be expelled; non-participating children have to be detained from registering for the process.
- The withdrawing participant has no father which participates in the process or the father cannot judge the vitality of its child: This case can occur with decoupled interactions, for instance if the participant was introduced into the process through an observer interaction. The withdrawal is denied because there is no way to decide on the vitality of the withdrawing participant.

For a participant *expulsion*, these cases can be distinguished, depending on the state of the participant:

- The participant to be expelled is registered with the coordinator: This participant can be expelled. Participating children of the leaving participant have to be expelled as well; non-participating children have to be detained from registering for the process.
- The participant to be expelled is not registered with the coordinator: If it tries to register in the further course of the process, the registration is denied; the participant has to cancel its activities.

Emissions of participants are only possible within the limits set by the transaction protocol. After the transaction has been demarcated, the set of participants is immutable. The exit handshake protocol can seamlessly be integrated into WS-Coordination. The messages `exit` and `exited` as well as the state *exiting* of WS-BusinessActivities then become dispensable.

4.2 Annotation of Services

WS-AtomicTransaction and WS-BusinessActivities provide service annotations for transactional capabilities and requirements such as supported coordination types and protocols. In order to enable more detailed planning and adaptivity at runtime, services also have to specify their *transactional behavior* according to the following scheme:

- A call within a transactional context is *mandatory*: If the call occurs within a transactional context, the service registers with the coordinator. A call without a transactional context is denied.
- A call within a transactional context is *optional*: If the call occurs within a transactional context, the service registers with the coordinator. A call without a transactional context is also accepted, the service then proceeds without transactional control.
- A call within a transactional context is *unsupported*: If the call occurs within a transactional context, it is accepted and processed, but the service does not register with the coordinator. A call without a transactional context is accepted and processed normally.
- A call within a transactional context is *prohibited*: If the call occurs within a transactional context, it is denied. A call without a transactional context is accepted and processed.
- A call within a transactional context is *forwarded*: The service does not itself actively process calls, but delegates them to other services. If the call occurs within a transactional context, the context is forwarded with the call. The forwarding service does not register with the coordinator, but the service the call is forwarded

to may do so. This behavior supports 3rd-party interactions by allowing a transactional context to propagate from the requestor to the recipient through the provider. Transactional agreements then have to be negotiated between the requestor and the recipient.

For example, in a composition operating within a transactional context, only services with the behavioral attributes "mandatory" or "optional" can directly join the transaction. Services with the behavioral attribute "forwarded" can only be integrated if the specific children of these services have the behavioral attributes "mandatory" or "optional".

4.3 Propagation of the Demarcation Privilege

The demarcation privilege is the right to initiate the termination of a transaction. Upon demarcation, the transaction protocol coordinating the termination is started. Usually, the initiator of a process holds this privilege. In order to properly support decoupled interactions and withdrawal from processes, the propagation of this privilege to other participants must be possible.

In such cases, the demarcation privilege is passed to subsequent participants, depending on the application logic. In WS-AtomicTransaction, the new holder of the demarcation privilege has to register for the completion protocol with the coordinator. WS-BusinessActivities does not provide a completion protocol. Within the *BusinessAgreementWithParticipantCompletion* protocol, participants individually signal completion of their work to the coordinator. Within the *BusinessAgreementWithCoordinatorCompletion* protocol, the coordinator decides when participants are finished, but the protocol does not specify how the coordinator comes to this decision. It is therefore advantageous to introduce the concept of demarcation privilege to WS-BusinessActivities. The participant holding the privilege can then trigger the coordinator completion protocol without limiting the flexibility of the participant completion protocol.

4.4 Architecture

Our TracG framework for transactional activity control is based on the *WS-Architecture* specification [1] and the *Service Bus* principle [12]. Figure 4 shows its layered architecture which integrates the WS-RF specification [6] implemented in the *Globus Toolkit* [7].

An application calls services through a *service bus*. The bus is responsible for managing the life cycles of dynamic processes. It provides all necessary services regarding the planning, agreement, and enactment phase as well as the transactional termination. Specifically, the bus offers directory, composition, and transaction services. Service calls

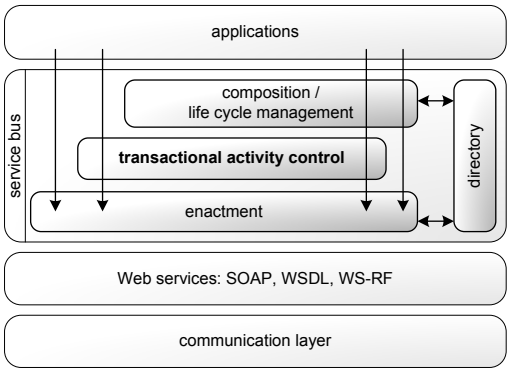


Figure 4. Layered architecture of TracG

are encapsulated by an enactment layer in order to implement, for example, transparent load balancing.

The service bus provides different types of interaction indicated by the arrows in Figure 4: If a particular service exists to handle a specific request, no dynamic service composition is conducted, but the service is called direct. This interaction is typical for static, predefined processes. If there is no service capable of directly answering the request, a dynamic service composition is performed. In both scenarios, transaction management is supported.

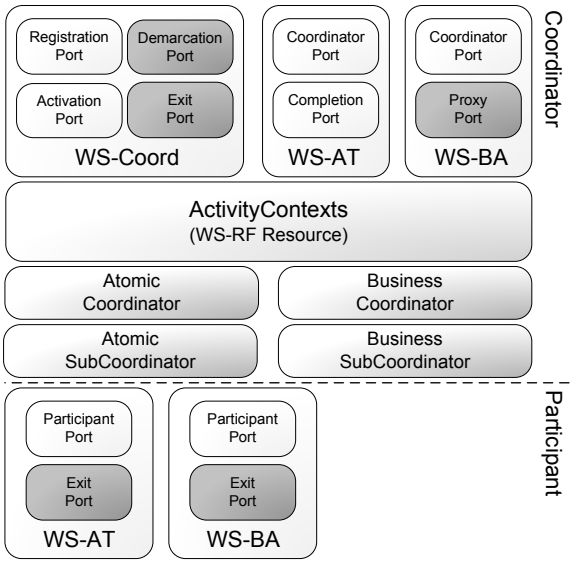


Figure 5. Coarse structure of the framework

The implementation of the framework is shown in Figure 5. The Web service port types for the protocols (WS-Coordination, WS-AtomicTransaction, and WS-BusinessActivities) have been implemented separately from each other. They invoke the corresponding coordinator, defined by the SOAP message that calls the respective port type. The coordinator classes then execute the correspond-

ing completion protocol. The status of the coordinator and the protocol itself is represented by a Web service resource [6]. The framework provides four coordinator implementations to support hierarchies of coordinators in both protocols as specified in WS-Coordination (*interposition*): two coordinator types for the roots of process trees and two coordinator types for the subordinates. The port types implementing extensions presented in this paper are indicated by a darker coloring.

5 Conclusion and Future Work

Service Grids contribute concepts such as dynamism and self-management to service-oriented computing and are therefore a promising platform for distributed processes. They allow the execution of adaptive and dynamic processes whose structure can be changed during runtime. The life cycle of such dynamic processes can be broken down into planning, negotiation, enactment, and transactional termination.

The transactional support of dynamic processes holds specific requirements. The set of participants and the structure of a process should be adaptable during runtime without needing to abort or restart the whole process. In case of decoupled interactions, some relationships between participants relevant to transaction management do not arise until runtime.

Existing standards show several weaknesses regarding the transactional coordination of activities in dynamic environments which mainly result from incomplete support of dynamically shrinking and growing sets of participants. More shortcomings can be identified with respect to complex interaction scenarios which require extended concepts such as propagation of the demarcation privilege.

Our *TracG* framework, regarding transaction management, is based on WS-Coordination. We have enhanced the specification with an exit handshake protocol. The privilege to demarcate a transaction can be passed to other participants. We proposed a concept to annotate services with their transactional behavior to incorporate transaction management in automated service composition.

An important field of our future research will be the localization of application-specific coordination logic in generic coordinator services. To enable autonomous transaction coordination, it is necessary to explicitly describe coordination logic in form of coordination rules which can then be transferred to a coordinator. To this end, we investigate the definition of coordination primitives as building blocks for coordination rules.

Furthermore, we strive to integrate the aspects of transaction management into the specification of dynamic processes. In this respect, we will examine existing process specification languages and possibly necessary extensions.

References

[1] A. Andrieux, K. Czajkowski, A. Dan, et al. *Web Services Agreement Specification (WS-Agreement)*, May 2004.

[2] A. P. Barros, M. Dumas, and A. H. M. ter Hofstede. Service Interaction Patterns. In *3rd Int. Conf. on Business Process Management, Nancy, France*, pages 302–318, 2005.

[3] D. Bunting, M. Chapman, O. Hurley, et al. *Web Services Composite Application Framework (WS-CAF) Ver 1.0*, Jul 2003.

[4] L. F. Cabrera, G. Copeland, M. Feingold, et al. *Web Services Atomic Transaction (WS-AtomicTransaction)*, Nov 2004.

[5] L. F. Cabrera, G. Copeland, T. Freund, et al. *Web Services Business Activity Framework (WS-BusinessActivity)*, Nov 2004.

[6] K. Czajkowski, D. F. Ferguson, I. Foster, et al. *The WS-Resource Framework (Version 1.0)*, May 2004.

[7] I. T. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. In *IFIP Int. Conf. on Network and Parallel Computing, Beijing, China*, pages 2–13, 2005.

[8] P. Furniss, S. Dalal, T. Fletcher, et al. *Business Transaction Protocol (BTP 1.1)*, 2004.

[9] T. Jin and S. Goschnick. Utilizing Web Services in an Agent Based Transaction Model (ABT). In *AAMAS 2003 - Workshop on Web Services and Agent-based Engineering*, July 2003.

[10] B. Kratz. Protocols For Long Running Business Transactions. Technical Report 17, Infolab Technical Report Series, Feb 2004.

[11] D. Kuropka and M. Weske. Die Adaptive Services Grid Plattform: Motivation, Potential, Funktionsweise und Anwendungsszenarien. *Emisa Forum*, 26(1):13–25, Jan 2006.

[12] F. Leymann. The (Service) Bus: Services Penetrate Everyday Life. In *Third Int. Conf. on Service-Oriented Computing, Amsterdam, The Netherlands*, volume LNCS 3826, pages 12–20, 2005.

[13] M. Little and T. Freund. A comparison of Web services transaction protocols (A comparative analysis of WS-C/WS-Tx and OASIS BTP), Oct 2003.

[14] F. Montagut and R. Molva. Augmenting Web Services Composition with Transactional Requirements. In *IEEE Int. Conf. on Web Services (ICWS'06)*, pages 91–98, 2006.

[15] M. P. Papazoglou. Service-Oriented Computing: Concepts, Characteristics and Directions. In *4th Int. Conf. on Web Information Systems Engineering, Rome, Italy*, pages 3–12, 2003.

[16] S. Tai, T. Mikalsen, E. Wohlstadter, et al. Transaction policies for service-oriented computing. *Data Knowl. Eng.*, 51(1):59–79, 2004.

[17] C. Türker, K. Haller, C. Schuler, and H.-J. Schek. How can we support Grid Transactions? Towards Peer-to-Peer Transaction Processing. In *Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA*, pages 174–185, 2005.