Towards a Fault-Tolerant Architecture for Enterprise Application Integration Solutions

Rafael Z. Frantz[1], Rafael Corchuelo[2], and Carlos Molina-Jimenez[3]

[1] UNIJUÍ, Departamento de Tecnologia - Ijuí, RS, Brazil
rzfrantz@unijui.edu.br
[2] Universidad de Sevilla, ETSI Informática - Avda. Reina Mercedes, s/n. Sevilla 41012, Spain
corchu@us.es
[3] School of Computing Science, University of Newcastle, UK
carlos.molina@ncl.ac.uk

Abstract. Enterprise Application Integration (EAI) solutions rely on process support systems to implement exogenous message workflows whereby one can devise and deploy a process that helps keep a number of applications' data in synchrony or develop new functionality on top of them. EAI solutions are prone to failures due to the fact that they are highly distributed and combine stand-alone applications with specific-purpose integration processes. The literature provides two execution models for workflows, namely, synchronous and asynchronous. In this paper, we report on an architecture that addresses the problem of endowing the asynchronous model with fault-tolerance capabilities, which is a problem for which the literature does not provide a conclusion.

Keywords: Enterprise Application Integration, Fault-Tolerance.

1 Introduction

The computer infrastructure of a typical today's enterprise can be conceived as an heterogenous set of applications (termed the software ecosystem [15]) that includes tens of applications purchased from different providers or built at home in the last 20 years or even earlier. Examples of typical applications are payroll and sales systems. A recurrent challenge that appears in these scenarios is to make the existing application interoperate with each other to keep the data used by them synchronised or to create new funcionality[9]. This problem is known as Enterprise Application Integration (EAI). In either case, the challenge is about devising and deploying a number of wrapping processes responsible for interacting with the individual applications and a number of integration processes responsible for managing the flow of messages among the applications. A good alternative to support the design of the integration process is the use of Process Support Systems (PSS): a piece of middleware which, among other functionalities, provides means for specifying distributed processes (e.g. EAI solutions) and for monitoring their executions [8]. Good examples of PSSs are conventional workflow systems [8]. PSSs based on BPEL [18] are discussed in [21]. Examples of PSSs with focus on EAI solutions are BizTalk [4], Tibco [20], Camel [6] and Mule [17].

R. Meersman, P. Herrero, and T. Dillon (Eds.): OTM 2009 Workshops, LNCS 5872, pp. 294–303, 2009.

A typical approach in the design of EAI is the use of a database to store inbound messages (messages coming from the individual applications to the integration process) until all the messages needed to start the integration process arrive. A distinctive feature of this approach is that it involves only a simple message workflow composed by several tasks that receive, transform and send outbound messages (messages out of the integration process to the applications); tasks might also request a wrapping process to interact with an application, fork the message workflow, and so on. A notorious limitation of this approach is that it uses memory inefficiently in applications where requests issued by tasks to individual applications take long (hours or days) to fulfil. In these situations the integration process instance remains blocked until the response arrives. Though some techniques have been proposed to ease this limitation (notably, dehydration and rehydration [4,21]), we consider the synchronous approach unsatisfactory; thus in this article we explore the suitability of asynchronous integration processes.

The distinctive feature of the asynchronous approach is that it uses a single multi-threaded process instance per wrapping or integration process to handle all the messages. Consequently, processes that require multiple inbound messages need to include tasks to correlate messages internally. The appealing side of this approach is its efficiency in resource consumption[7,11].

EAI solutions are inherently distributed: they involve several applications that might fail and communicate over networks that might unexpectedly delay, corrupt or even loose messages. Thus they are susceptible to a wide variety of failures [8,19]. To be of practical use, EAI solutions need to include fault–tolerance mechanisms [1].

Fault–tolerance in the context of PSS is not a new topic; several authors have studied the problem before but only within the context of the synchronous approach discussed above and with a single process in mind, i.e., overlooking typical EAI solutions that involve several wrapping and integration processes. To help cover, this gap in this paper we propose a solution based on an asynchronous approach with fault-tolerance features. The only assumption we make is that messages produced by integration processes are enriched with information about the original messages used to produce them.

The remainder of this paper is structured as follows: Section 2 discusses related literature; Section 3, introduces relevant concepts. The architecture that we propose is presented in Section 4; Section 5, presents a case study; finally, we draw conclusions in Section 6.

2 Related Work

Our discussion on fault-tolerance in EAI is closely related to the issue of distributed long-running transactions introduced in[16]. This seminal work inspired research aimed at the design of process support systems with fault–tolerance features. The effort resulted in several proposals that can be roughly categorised into two classes: algorithm–based where algorithms are used to handle faults automatically and fault–handling–based where faults are handled by rules defined by the designer. An abstract model for workflows with fault–tolerance features is discussed in [3]; this paper also provides a good survey on fault-tolerance approaches. An algorithm for implementing transactions in distributed applications where the participating applications co–operate

to implement a distributed protocol is presented in [1]. An algorithm for the execution of BPEL processes with relaxed transactions where the *all or nothing property is relaxed* is presented in [12]. In [13] the authors enhance their approach with a rule–based language for defining specific-purpose recovery rules.

An algorithm for handling faults automatically in applications integrated by means of workflow management systems and amenable to compensation actions or to the two–phase commit protocol, is suggested in [8]. Exception handling in applications where compensation actions are difficult or infeasible to implement is discussed in [14].

[2] proposed an architecture to implement fault-tolerance based on ad–hoc workflows. For instance using a web server as a front-end, an application server, a database server and a logging system. They assume that a workflow is always activated on arrival of one request message which flows through the components of the workflow; thus, their recovery mechanism relies on the trace of every message through the system.

An architecture for fault–tolerant workflows, based finite state machines (message sequence charts) that recognise valid sequence of messages of the workflow is discussed in [5]. Recovery actions are triggered when a message is found to be invalid, or the execution time of the state machine goes beyond the expected time.

An approach to provide fault–tolerance to already implemented Petri net controllers is presented in [11] and [7]. The original controller is embedded unaltered into a new controller with the original functionality but enlarged with additional places, connections, and tokens, aimed at detecting failures. A discussion on how to provide Petri net–modelled discrete event systems is also presented.

From the analysis of the previous proposals, we conclude that they share a number of common issues. They all deal with a single process, except for [2], which can neither deal with multiple inbound messages. This is a shortcoming because a typical EAI solution involves several wrapping and integration processes; note, too, that the inability to deal with multiple inbound messages is problematical insofar an integration process can be activated by a single application, but there are many applications where an integration process is started by the occurrence of multiple inbound messages arriving from different applications. Another common limitation among the works mentioned above is that, with the exception of [7,11], they support only the synchronous execution model. In addition, the proposals that deal with the asynchronous model focus on Petri Net controllers, i.e., they neglect the problems of distributedness, software ecosystems, and so on.

3 Definitions

3.1 EAI Solutions

As shwon in Fig. 1, a typical EAI solution has several wrapping processes used to communicate the solution with applications and several integration processes that implement the integration business logic. Processes use ports to communicate with each other or with applications over communication channels. Ports encapsulate tasks functionalities like receive, request and send; and help abstract away from the details of the communication mechanism, which may range from an RPC-based protocol over HTTP to a document-based protocol implemented on a database management system.

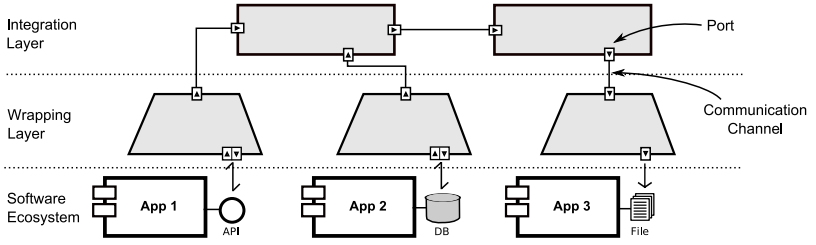


Fig. 1. Layers of a typical EAI solution

3.2 Failure Semantics

A dependable system is one on which reliance can be placed on the service that it delivers. Fault-tolerance is an important means to achieve dependability. Faults, errors and failures represent impairments to dependability [10]. A *fault* may be internal to the EAI solution or external (within the software ecosystem). In both cases, when they occur, they are the cause for *errors* that impact the EAI solution. Errors represent the point where the EAI solution deviates from its normal processing and if not handled lead the solution to a *failure* perceived by the user.

The general assumption we make about the reliability of the components involved in an EAI solution is that they might occasionally fail. Internal faults might occur in components of the EAI solution, such as processes, ports and communication channels; furthermore, external faults might occur in the software ecosystem. To provide EAI solutions with a mechanism to tolerate failures, we first need to identify the failure semantics that its components are likely to exhibit and stipulate what kind of errors the EAI solution should be able to tolerate: detect at runtime and execute a corresponding error recovery action to handle the specific error. Our architecture accounts for the following failures: omission, response, timing, and message processing failures.

Omission Failures (OMF): We assume that once a communication operation is started by a port, it terminates within a strictly defined time interval and reports either success or failure. OMF model situations where network, application and communication channel problems prevent ports from sending or receiving a piece of data within the time interval.

Response Failures (REF): REF are caused by responders (an application or communication channel) sending incorrect messages. Thus before being acceptable for processing, messages need to satisfy a validation test (e.g., headers and body inspected) that results in either success of failure.

Timing Failures (TMF): A message has a deadline to reach the end of the flow, which is verified by ports. Ports report success for timely messages and failure for messages with overrun deadlines. Both internal and external faults influence TMF.
Message Processing Failures (MPF): Ports and processes signal MPF when they are unable to complete the process of a message; otherwise success is signalled.

4 Architectural Proposal

The architecture we propose to provide fault-tolerance for EAI solutions is shown in Fig. 2 as a metamodel. This metamodel introduces the components involved in the construction of rules, exchange patterns, and mechanisms for error detection and recovery. As depicted in the metamodel, an EAISolution can define multiple ExchangePatterns (MEPs) and Rules. Events are notifications to the monitor, generated by Sources inside the EAI solution, that in conformance with our failure semantics have type EventTriggerMessageType to report successes or failures during the workflow execution. Source can be a Port or a Process. Each MEP defines a set of inbound and outbound source ports, from which events are reported. A rule is composed of a Condition and an Action. So, a condition can be SimpleCondition, represented by only a single event, or CompositeCondition, which contains two conditions connected by a LogicalOperator. When the condition is *true*, action executes the corresponding error recovery action defined by the rule. The Monitor observes one or more EAI solutions to detect potential failures and triggers mechanisms to handle them. As shown in Fig. 3 the monitor is composed of a Log, a SessionCorrelator, an ExchangeEngine, a RuleEngine, an EntryPort, and ExitPorts.

4.1 The Logging System

The logging systems is represented by the Log where all success and failure events reported by sources inside EAI solutions are permanently stored. The monitor receives

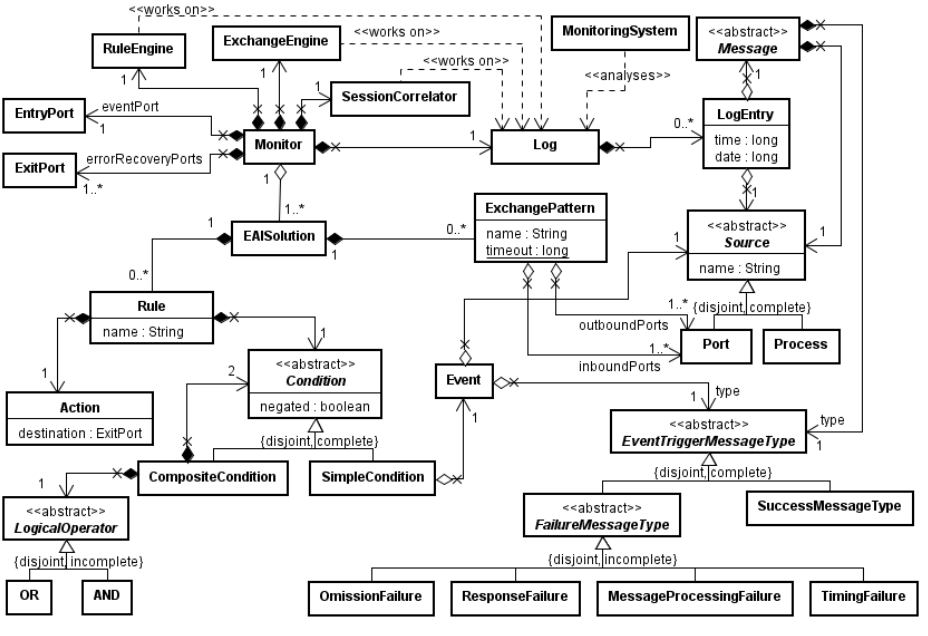


Fig. 2. Metamodel of an EAI solution with fault–tolerance

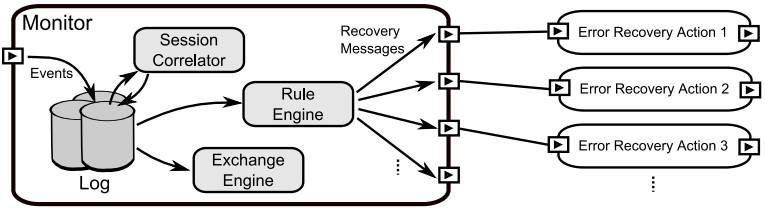


Fig. 3. Abstract view of the monitor

events through an entry port and stores them in the log from where they are available to the other components of the monitor. Roughly speaking, the log is composed of several LogEntries that record information about events, such as, the fully qualified event source name, date and time of occurrence, and a copy of the message under process at the time of occurrence of the event. By qualified, we mean the name of the EAI solution that originated the event, followed by the unique name of the source inside the solution. Name uniqueness of sources of events allows the monitor to observe one or more EAI solutions simultaneously. The log is shared by the session correlator, the exchange engine, and the rule engine. It can also provide information to a MonitoringSystem, interested in assessing the health state of an EAI solution.

4.2 The Session Correlator

The session correlator session-correlates messages inside the log. Its output is used by the exchange engine to determine the state of MEPs instances and to trigger recovery actions. Since in our architecture a task within a process can take several input messages and produce several output messages, it is not trivial to determine what messages belong to different workflow sessions. To solve the problem, we enrich composed messages with information about the original messages used to compose them; next we establish a parent–child relationship between messages to session–correlate them: two arbitrary messages m_a and m_b are session–correlated if m_a is the parent of m_b or m_b is the parent of m_a. Likewise, three messages m_a, m_b and m_c are session-correlated if m_c is an ancestor of both m_a and m_b.

4.3 The Exchange Engine

The exchange engine is responsible for managing MEPs. A MEP represents a sequence of message exchange among the participating applications. The textual notation we use to specify MEPs is shown in Fig. 4. An EAI solution can have one or more MEPs, thus different message workflows can occur inside a given EAI solution.

MEPs deal only with messages of success type from the ports listed in Inbound and Outbound sets, so there is no need to explicitly specify types. When the exchange engine finds two or more session-correlated messages in the log which came from different ports in a MEP, it creates an instance of this MEP and associates to it a max time–to–live parameter; max time–to–live is global and imposes a deadline on the instance to successfully complete. The session-correlated messages may fit into more than one

```
EAI SOLUTION name
TIMEOUT time

EXCHANGE PATTERN name
  INBOUND (inbound_ports)
  OUTBOUND (outbound_ports)
            ⋮
RULE name
  CONDITION (condition)
  ACTION destination
            ⋮
```

Fig. 4. Syntax of Exchange Patterns and Rules

MEP, so in this case an instance of each MEP will be created. Inbound contains a set of fully qualified port names, from where inbound messages come. Similarly, Outbound contains a set of port names to where outbound messages are reported. The syntax for a fully qualified name is as follows: eai_solution_name:: process_name::port_name, where eai_solution_name defaults to EAISolution.

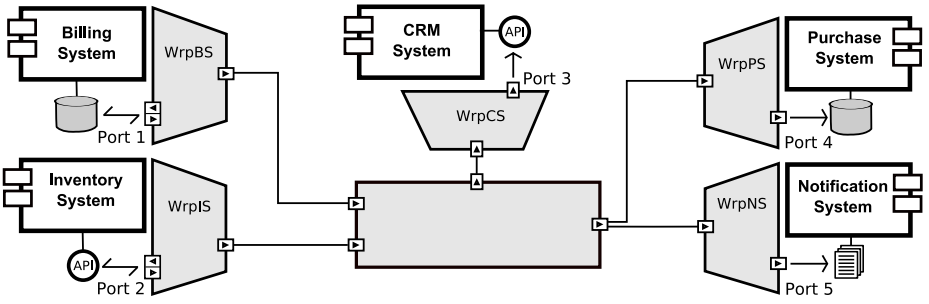
The job of the exchange engine is to detect completed, in-progress and incomplete MEPs in an EAI solution; also it detects messages that has been in the log for a long time without participating in any MEPs; we call them *odd messages*. A completed MEP instance indicates that several correlated inbound messages were processed successfully by an EAI solution's workflow within the max time-to-live deadline; the exchange engine detects them by finding all session-correlated outbound messages for this MEP instance in the log. An in-progress MEP instance contains two or more correlated messages (not necessary outbound) in the log, has not overrun its max time-to-live deadline, and is waiting for more outbound message(s) to arrive. An in-progress MEP instance is declared incomplete when its deadline expires. MEP instances fail to complete due to failures detected during their workflow execution, thus they trigger the rule engine which, if necessary, initiates the execution of error recovery actions (see Fig. 5).

It is possible that an incomplete MEP instance might be completed beyond its deadline and after the execution of its error recovery action. Situations like this are detected and signalled by the monitoring system.

4.4 The Rule Engine

The rule engine is responsible for managing the Event–Condition–Action (ECA) rules of EAI solutions. When the condition of a rule evaluates to *true*, i.e., a set of session-correlated messages that activate it is found, the rule engine creates an error recovery message and invokes an error recovery action by means of an exit port. The error recovery action contains the logic to handle the error. Error recovery actions are external to the monitor, and, although they are designed specially to be executed against an application or communication channel, they can be reused if necessary.

Rules take into account both success and failure events. Additionally, events came not only from ports but also from processes, and contain source–name and event–type, cf. Fig. 2. The general syntax is: eai_solution_name::source_name:event_type. If the source is a port, then it must also include the name of the process to which the port belongs, cf. Fig. 5.




```
EAI SOLUTION MySolution
TIMEOUT 2500

EXCHANGE PATTERN MyMEP_1
  INBOUND (WrpBS::Port1, WrpIS::Port2)
  OUTBOUND (WrpCS::Port3, WrpPS::Port4)
EXCHANGE PATTERN MyMEP_2
  INBOUND (WrpBS::Port1, WrpIS::Port2)
  OUTBOUND (WrpCS::Port3, WrpPS::Port4, WrpNS::Port5)

RULE MyRule_1
  CONDITION (!WrpPS::Port4:OK & WrpCS::Port3:OK)
  ACTION ErrorRecoveryActionForCS
RULE MyRule_2
  CONDITION (!WrpCS::Port3:OK & WrpPS::Port4:OK)
  ACTION ErrorRecoveryActionForPS
RULE MyRule_3
  CONDITION (WrpNS::Port5:OMF)
  ACTION ErrorRecoveryActionForCS
```

Fig. 5. Example of EAI solution with fault-tolerance

5 Case Study

To illustrate our approach, an EAI solution for a fictitious company is shown in Fig. 5. It integrates five applications deployed before the EAI solution which were not designed with EAI integration in mind and run independently from each other. The EAI solution has one integration process and five wrapping processes, one for each application. i The main goal of the solution is to collect bills from the Billing System (BS), merge them with their corresponding order(s) provided by the Inventory System (IS) and to produce a single merged message. A copy of the merged message is sent to the CRM System (CS) while a second copy is sent to the Notification System (NS) which is responsible for notifying the customer about his or her purchase. Finally, a third copy of the message is sent to the Purchase System (PS) which stores records of purchases. A bill may be associated with more than one order; in this case the order number is used to compute local correlation.

To better illustrate some features of our architecture we assume some constraints imposed on the EAI solution. First, the merged message must be successfully sent to the CS and PS by Port 3 and Port 4, respectively. Any failure that prevents one of the applications from receiving the session-correlated message triggers the execution of a recovery action against the application that succeeded. Second, inbound message(s) are successfully processed in two situations: when all target applications (CS, PS and NS) receive the session-correlated message, or when only the CS and the PS receive it. Failures in Port 5 do not invalidate the workflow execution, they only trigger the execution of an error recovery action that stores records stating that the customer could not be notified. The design of error recovery actions are out of the scope of this paper. The last constraint to this EAI solution is that orders and bills are delivered within 2 to 5 seconds to the target applications.

To account for these constraints, the EAI solution has two MEPs and three rules, cf. Fig. 5. The first MEP defines two inbound ports and two outbound ports. This implies that a MEP instance is completed when session-correlated messages for all these ports are found in the log. The second MEP also includes Port 5 in the outbound list; this represents another alternative for the EAI solution to complete successfully. In cases when an incomplete MEP is detected by the exchange engine, the rules are evaluated by the rule engine and the corresponding recovery actions are executed.

6 Conclusion

We have proposed an architecture for EAI solutions enhanced with fault–tolerant features, in the context of process support systems. We argued that existing proposals that deal with fault-tolerance in the context of process support systems are based on synchronous execution models and consequently, are memory consumption inefficient. In response, we explored an asynchronous execution model. We introduced our architecture proposal from the perspective of its metamodel; it includes a monitor that detects failures and triggers recovery actions. We discussed and addressed different classes of failures. MEPs and rules are configured by means of an ECA-based language that is part of the proposed architecture. Incomplete MEPs cause the activation of rules to execute error recovery actions. To support our ideas we presented a case study.

Acknowledgements. The first author conducted part of this work at the University of Newcastle, UK as visiting member of staff. His work is partially funded by the Evangelischer Entwicklungsdienst e.V. (EED). The second and first authors are partially funded by the Spanish National R&D&I Plan under grant TIN2007-64119, the Andalusian Local Government under grant P07-TIC-02602 and the research programme of the University of Seville. The third author is partially funded by UK EPSRC Platform Grant No. EP/D037743/1.

References

1. Campbell, R.H., Randell, B.: Error recovery in asynchronous systems. IEEE Trans. Soft. Eng. 12(8), 811–826 (1986)
2. Chen, M.Y., Accardi, A., Kiciman, E., Lloyd, J., Patterson, D., Fox, A., Brewer, E.: Path-based faliure and evolution management. In: Proc. Int'l Symp. Netw. Syst. Des. and Impl., p. 23 (2004)
3. Chiu, D., Li, Q., Karlapalem, K.: A meta modelng approach to workflow management systems supporting exception handling. Inf. Syst. 24(2), 159–184 (1999)
4. Dunphy, G., Metwally, A.: Pro BizTalk 2006. Apress (2006)
5. Ermagan, V., Kruger, I., Menarini, M.: A fault tolerance approach for enterprise applications. In: Proc. IEEE Int'l Conf. Serv. Comput., vol. 2, pp. 63–72 (2008)
6. Apache Foundation. Apache Camel: Book In One Page (2008)
7. Hadjicostis, C.N., Verghese, G.C.: Monitoring discrete event systems using petri net embeddings. In: Proc. 20th Int'l Conf. Appl. and Theory of Petri Nets, pp. 188–207 (1999)
8. Hagen, C., Alonso, G.: Exception handling in workflow management systems. IEEE Trans. Softw. Eng. 26(10), 943–958 (2000)

9. Hohpe, G., Woolf, B.: Enterprise Integration Patterns - Designing, Building, and Deploying Messaging Solutions. Addison-Wesley, Reading (2003)
10. Laprie, J.C.: Dependability - its attributes, impairments and means. In: Predicting Dependable Computing Systems, pp. 3–24 (1995)
11. Li, L., Hadjicostis, C.N., Sreenivas, R.S.: Designs of bisimilar petri net controllers with fault tolerance capabilities. IEEE Trans. Syst. Man Cybern. Part A: Syst. Humans 38(1), 207–217 (2008)
12. Liu, A., Huang, L., Li, Q., Xiao, M.: Fault-tolerant orchestration of transactional web services. In: Proc. Int'l Conf. Web Inf. Syst. Eng., pp. 90–101 (2006)
13. Liu, A., Li, Q., Huang, L., Xiao, M.: A declarative approach to enhancing the reliability of bpel processes. In: Proc. IEEE Int'l Conf. Web Services, pp. 272–279 (2007)
14. Liu, C., Orlowska, M.E., Lin, X., Zhou, X.: Improving backward recovery in workflow systems. In: Proc. 7th Int'l Conf. Database Syst. Adv. Appl., p. 276 (2001)
15. Messerschmitt, D., Szyperski, C.: Software Ecosystem: Understanding an Indispensable Technology and Industry. MIT Press, Cambridge (2003)
16. Molina, H.G., Salem, K.: Sagas. SIGMOD Rec. 16(3), 249–259 (1987)
17. MuleSource. Mule 2.x User Guide (2008)
18. OASIS. Web Services Business Process Execution Language Version 2.0 Specification (2007)
19. Peltz, C.: Web services orchestration: a review of emerging technologies, tools, and standards. Technical report, Hewlett-Packard Company (2003)
20. TIBCO. Tibco application integration software (June 2009)
21. Wright, M., Reynolds, A.: Oracle SOA Suite Developer's Guide. Packt Publishing (2009)