A Pattern Taxonomy for Business Process Integration Oriented Application Integration

Helge Hofmeister* and Guido Wirtz
Otto-Friedrich-University Bamberg, Distributed and Mobile Systems Group
Feldkirchenstr. 21, 96052 Bamberg, Germany
hofmeister@ecoware.de, guido.wirtz@wiai.uni-bamberg.de

Abstract

Application Integration is a work and cost intensive part of both, software development projects and ongoing software maintenance. Thus, software vendors as well as application owners need mechanisms to cope with the complexity of this task. Due to its inherent feature of decoupling applications in a robust manner, message-oriented integration techniques have gained much attention among different integration styles for a long time. Message oriented integration patterns have been analyzed in much detail as well. Recently, higher-level styles that provide an abstraction closer to application issues are under discussion. Because of its high abstraction level providing a global view, business-process-integration based application integration is a promising candidate to start with.

This paper presents a refined approach of Business Process Integration Oriented Application Orientation (BPIOAI). It facilitates the task of application integration by means of defining standard integration processes based on a taxonomy of integration patterns and a set of integration services. The identified taxonomy takes the structure induced by the integration processes into account. It provides a standardization that is especially suitable for centralizing the application integration development in central development centers of large organizations. Besides these benefits, this work provides an approach toward a domain-specific language for the domain of application integration.

Keywords: *application integration, business processes, integration pattern, standards*

1. Introduction

As application orientation has been a crucial part of everyday work in almost all application areas of IT for some time, there is a bunch of proposed techniques and approaches to cope with its complexity in a manner that accelerates new developments and lowers maintenance costs. Whereas software development in general has seen different maturity levels and paradigms, the task of system integration has faced a lot of paradigms itself. In an overview given by Linthicum [1], different *integration styles* that describe how integration is managed are distinguished, i.e., information-oriented, business process integration-oriented, service-oriented and portal-oriented application integration. For real-life application integration, the paradigm of *message-based* integration, has been considered most useful due to it's characteristics of decoupling systems in a manner that make compound systems more tolerant against failures and, hence, more reliable. Even if message-based application integration is applied as an *information-oriented application integration* (IOAI) approach, the messaging basically provides the technical infrastructure, i.e., the means for reliable and asynchronous communication. The most promising recent paradigm of application integration, *Business Process Integration Oriented Application Integration* (BPIOAI), provides a more abstract as well as global view to integration. This is achieved by introducing a business process controlling the order of how participating application systems are invoked. The process layer of BPIOAI does not solely make direct use of application systems. Furthermore, conventional IOAI based systems can be used as long as they expose a well-defined interface to the business process. Since BPIOAI works on-top of IOAI, the basic exchange of information is handled with messages, too.

A crucial part of supporting integration in everyday work, is the identification of situations, problems and lessons learned from experience and *best-practice* knowledge that occur and apply frequently despite the different specifics of the problem at hand. *Pattern* as introduced by Alexander [2] and applied to the domain of software engineering by the Gang of Four [3], are a well-known means to capture and re-use such kind of knowledge. Because re-use requires knowledge about existing patterns, guiding effective work with patterns and their proper combination by classifying, grouping and naming patterns with the help of a pattern lan-

*Helge Hofmeister is an external PhD student with the Distributed and Mobile Systems Group at Bamberg University.

guage is even more desirable. However, a pattern language may be strictly tuned to a restricted application area. Hence, for more complete application domains, a domain specific language (DSL) should be used [4].

In order to apply patterns to the domain of message oriented application integration, Hohpe and Woolf propose 65 standard *Enterprise Integration Patterns* [5] grouped according to 6 different root pattern and describe how these patterns can be applied to build message-based integration systems in a standardized way. Additionally, guidelines in terms of descriptions that allow to compose a solution for a given integration problem by sequencing several patterns around a message-based communication channel are provided. Since most of Hohpe's message-oriented patterns provide additional functionality on-top of a messaging-system and more abstract integration styles are usually based on a messaging layer, most of these patterns should be as well useful in the context of other integration styles.

This paper presents a refined approach of Business Process Integration Oriented Application Integration (BPIOAI) that defines a set of standard integration processes and services and uses this setting to categorize the messaging patterns more strict as it is done by Hohpe and Woolf [5]. By adding two context specific modelling languages on-top of Hohpe's basic patterns, categories are built that empower the basic patterns to describe a complete Application Integration Language (AIL).

Besides discussing related work in section 2, the rest of this paper outlines our approach by defining standardized integration processes (section 3), discussing so-called *idioms* for the most important integration pattern (section 4) and illustrating the approach by means of a simple case study (section 5). Section 6 discusses issues of future work.

2. Related Work

The Enterprise Integration Patterns of Hohpe and Woolf [5] define the starting point and setting of our work. Van der Aalst discusses different patterns in the area of workflows [6] by distinguishing data [7] and resource [8] patterns. These patterns are designed independently of application domains and are not specific to application integration. Nevertheless, in the case of BPIOAI, generic workflow patterns support the analysis and even more the implementation of integration specific patterns because they provide help in implementing workflow systems that are in turn a building block of BPIOAI based integration systems [1].

Jayaweera et al. focus at the functional level of process models in order to facilitate the generation of e-commerce systems out of business models [9]. In doing so, they describe how existent business models are transformed into process models. The abstraction level of this business-centric work is being located one conceptual level above our work. This is because we aim to guide the implementation of the integration systems automating the processes as they are generated by the approach of Jayaweera or other business process methodologies.

3. Standardized Integration Processes

We add a behavioural layer on top of the pattern language of Hohpe that basically deals with data aspects. This layer can be understood as a replacement of the pipe-architecture that is used by Hohpe [5] to categorize patterns for message based application integration. Because our scope of integration toranges up to cross-system service orchestration, the behavioral layer defines generic and configurable integration processes instead of a simple pipe. These integration processes allow to describe the logic of cross-system composite applications on a business logic centric process layer on top that handles the detailed tasks due to distributed and heterogeneous systems. These single steps of the standard integration processes were surveyed among integration specialists within a large German IT services company and are understood as loosely coupled functionality that is coarse grained in terms of integration functionality. This kind of tasks is handled by so-called pre-defined *integration services* (IS). Based on the taxonomy that is provided by the disjoint functionalities of the IS (cf. section 4), Hohpe's integration patterns are used to support the implementation of these services. Hence, BPIOAI focused integration tools such as IBM Websphere [10], SeeBeyond's ICAN Suite [11] or SAP's Exchange Infrastructure [12] can easily reuse the functionality that is provided by the basic patterns and the integration services as well.

We introduce *two integration processes that orchestrate the IS*: the so-called *Integration In-Flow* (IIF) reads data from and the so-called *Integration Out-Flow* (IOF) stores data to connected legacy systems. The IIF and IOF steps are described in the activity diagrams of Fig.1 and Fig.2, respectively. Since the integration processes need to be generic, they should be capable to handle different communication semantics. For the same reason, the Integration Process needs to expose a generic interface to the business application on top of it. Thus, the IIF starts with the reception of either a synchronous or the reception of an asynchronous call or intrinsically. Subsequently, an *Event* is created by an IS. This *Event* is used as a ticket that is passed to subsequent IS as well as to a possible composite application on top of the integration processes. The data that should be transferred by the IIF is read by an IS called *Receiver Service*. This service has information about how to connect to the legacy systems as well as how data should be filtered, collected or reassembled. Optional steps of the processes are depicted as having a decision block as their predecessors. Thus, the subsequent step of acknowledging the

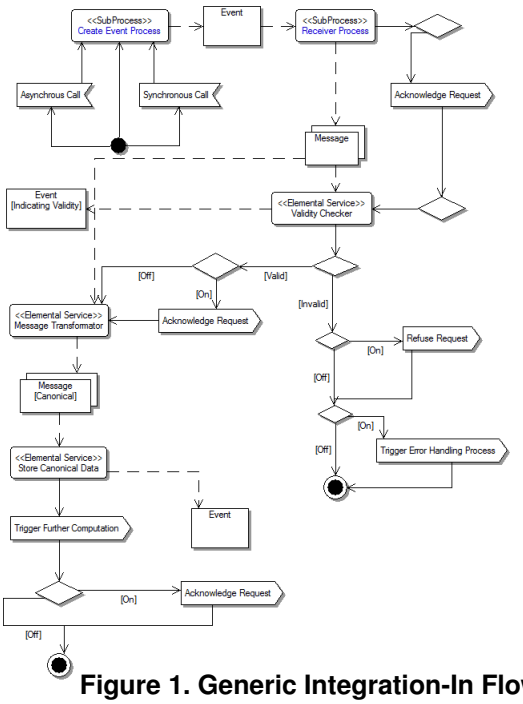

Figure 1. Generic Integration-In Flow

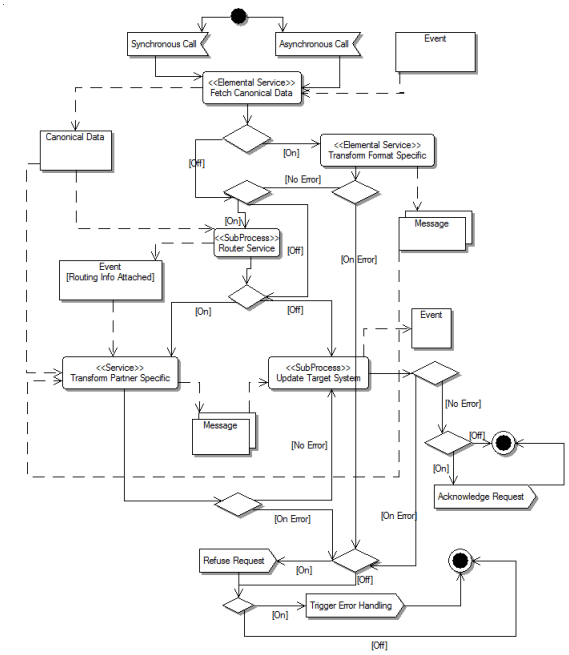

Figure 2. Generic Integration-Out Flow

request is optional. For synchronous calls, this acknowledgment would be the answer to the request. For asynchronous calls, this would be a subsequent message that is sent back to the originator of the request. If configured accordingly, the received data could be validated. Whether this solely involves syntactic schema checking or additional semantic checks as well depends on the implementation of the *Validity Checker*. Based on the validity the process may either continue, the request could be refused or a more sophisticated error handling process may be triggered. If the data is valid it is transformed into a canonical data model ([13]). This intermediary data format is used to decrease the need for different implementations of the *Transformator Service*. Finally, the canonical data is stored in a data store and the ticket identifying this data is either directly passed to the IOF or to the composite application. Subsequently, components may solely deal with this ticket. For that sake the Event Service may attach certain information to the *Event*. This is a similar mechanism to the *slip* [5]. But in contrast to the message slip, not all information has to be included because the data store is accessible with the *Event* as the key. Thus, components could look up data as needed.

The IOF process for updating application systems (cf. Fig. 2) is generic as well. Note that a synchronous call to the coupling system may still be active. Thus, the IOF either continues the execution of a synchronous or an asynchronous call. If a synchronous call is continued, the termination of the request is handled by the final acknowledgment step of the IIF. Subsequently, a service is used to fetch the actual data based on the *Event* from the data store that keeps the canonical data. Depending on the configuration, it may be required to pre-transform the canonical data. If so, a *Transformator Service* may be invoked. Subsequently, a dedicated IS attaches routing information to the *Event*. Based on this information the data could be transformed into the partner specific format by another Transformator Service. Afterwards the data is written to the target system by the Updater Service. In case of a failure that may occur while updating the target system, the request may be refused and error handling may be triggered.

4. Idioms for the Integration Services

In order to standardize the implementation of the coarse-grained integration services, we use Hohpe's [5] integration patterns to distinguish different IS by integration *idioms*. In the following, we introduce and discuss the Integration Services for the In-Flow and Out-Flow as well as the idioms that may be used to implement them if required.

The **Event Creator Service** incorporates the concept of *triggering* the coupling-system. This may happen intrinsically or extrinsically. In case of intrinsic event generation, the coupling-system itself generates an *Event* independently of the connected application system's state. Extrinsic event generation describes that the connected application system triggers the coupling-system by an internal state-transition. In either case the output of this service is always an unique event. The following integration patterns are useful to the Event Creator Service[1]:

[1] All the pattern and pattern names used here are due to Hohpe [5]

Pattern/Service	Communication Semantics	Actual Data Format	Canonical Data Format	Sender	Receiver
Selective Consumer	$\approx$	$\bowtie$	$\approx$	$\approx$	$\approx$
Polling Consumer	only applicable for async. processing	$\bowtie$	$\approx$	$\bowtie$	$\approx$
Splitter	$\bowtie$	$\bowtie$	$\approx$	$\approx$	$\approx$
Message Sequence	$\bowtie$	$\bowtie$	$\approx$	$\approx$	$\bowtie$
Aggregator	$\bowtie$	$\bowtie$	$\approx$	$\approx$	$\approx$
Resequencer	only applicable for async. processing	$\bowtie$	$\approx$	$\bowtie$	$\bowtie$
Validity Check	$\approx$	$\bowtie$	$\approx$	$\approx$	$\approx$
Message Transformator	$\approx$	$\bowtie$	$\bowtie$	$\approx$	$\approx$
Store Canonical Data Format	$\approx$	$\approx$	$\bowtie$	$\approx$	$\approx$
Fetch Canonical Data Format	$\approx$	$\approx$	$\bowtie$	$\approx$	$\approx$
Content Based Router	$\approx$	$\approx$	$\bowtie$	$\approx$	$\bowtie$
Dynamic Router	$\approx$	$\approx$	$\bowtie$	$\approx$	$\bowtie$
Recipient List	$\approx$	$\approx$	$\bowtie$	$\approx$	$\approx$
Event Message	$\bowtie$	$\bowtie$	$\approx$	$\approx$	$\bowtie$
Document Message	$\bowtie$	$\bowtie$	$\approx$	$\approx$	$\bowtie$
Command Message	$\bowtie$	$\bowtie$	$\approx$	$\approx$	$\bowtie$
Independent = $\approx$	Dependent = $\bowtie$				

Table 1. Implementation Dependencies

- In the case of extrinsic triggering, the *Selective Consumer* may act as a filter and block events that are not relevant for the coupling-system. This minimizes the complexity within the application system since it does not need to filter itself.
- The *Polling Consumer* that is able to detect state transitions by itself, is used whenever the sending application system should not be aware of being coupled with another application system, but should act as the initiator of interactions.

The **Receiver Service** de-couples the functionality of reading data from the functionality of triggering the system in order to decrease the complexity of the single IS and to increase their re-usability. The implementation of the Receiver Service heavily concerns the connectivity to the connected application systems. Anyway, in our description we rely on the integration middleware and solely focus on the functional creation of messages. The *Message Sequence* pattern may be used here as well as the *Splitter*, *Aggregator* and *Resequencer* pattern. *Message Sequence* is distinguished from the *Splitter* pattern, since the purpose of splitting a message is different. However, the real implementation may be identical.

The additional integration services to be used in our inflow process, i.e., the **Validity Service** and the **Message Transformator** cannot be supported by the integration patterns according to [5]. The same holds for the **Data Fetcher** and the **Message Transformer** services that are used in the outflow processes. Implementation support can be found here for the

- **Router Service** through the *Content Based Router*, *Dynamic Router* and *Recipient List* pattern as well as for the
- **Updater Service** that is supported by a wide range of patterns including *Message Sequence*, *Aggregator*, *Resequencer*, *Event Message*, *Document Message*, and *Command Message*.

Implementation Dependencies

Patterns and idioms describe best practices and solution descriptions in a standardized form. Nevertheless, the description is not formal and the actual implementation of the idioms is still to be performed case by case. But the implementation itself is under some circumstances reusable. In order to provide support for a central design center, we analyzed the idioms in terms of their dependence to certain factors. With this information, the implementation effort estimation can be improved since the identification of the required integration services, idioms and dependency factors are already known at design time. This information can be used to determine which elements could be re-used and which need to be re-implemented. The dependency information is shown in Table 1. The single rows show the idioms that may be required for a specific service. The columns show the different dependency factors:

- **Communication semantics**: Most IS are called by a service orchestration layer and usually the result of a service needs to be returned synchronously. In some cases, asynchronous processing may be appropriate, if the communication between the connected application systems is asynchronous as well.

- **Effective data format**: Although one of the motivations for the integration processes is the standardized data-format independent way of connecting to application systems, some integration services are nevertheless dependent to the data format that is used by a connected system. Thus, some of the idioms need to be implemented for every single data format of a connected system. Note that this dependency factor solely describes the dependency from the format.
- **Canonical data format**: Usually, a canonical data format is desirable because it reduces the amount of required data mappings. If a certain idiom does not have the purpose of data transformation, it still may be dependent on a canonical data format as well.
- **Sender** and **Receiver**: Integration processes are designed to be executed in some sort of integration server middleware that provides adapters to handle the effective communication with application systems. In other scenarios the integration server as well as the application systems may be able to deal with Web Services or other ways of communication like, e.g., CORBA [14]. Anyway, certain aspects of the implementation may be dependent on the connected system. The *Sender* (*Receiver*) factor indicate that an idiom's implementation may be dependent on the sending (receiving) application system, respectively.

5 Business Case

To demonstrate the benefit of our categorization, we give a real-life scenario that is used to integrate multiple ERP systems of an industry company *I* with a portal application as well as external suppliers of that company. In doing so we focus on the value that comes with the approach in terms of standardization and re-usability. We present our example as the governance of a company may proceed in alignment with the integration processes we defined.

Multiple divisions of company *I*, possibly working with different ERP systems, order certain goods and services within their ERP systems. These orders are sent to a composite application within a common portal. Additionally, suppliers are being notified about new orders. Afterwards, suppliers may log-on to the portal, check changes and accept/decline orders. Relevant actions of the supplier, such as accepting an order, are then posted to the according ERP system. This logic is described as an event-driven business process that controls the integration (cf. Fig.3) and is carried out by an integration middleware that is capable of executing processes. Note that the business logic is implemented within the portal application and the processes within the integration middleware are solely integration workflows.

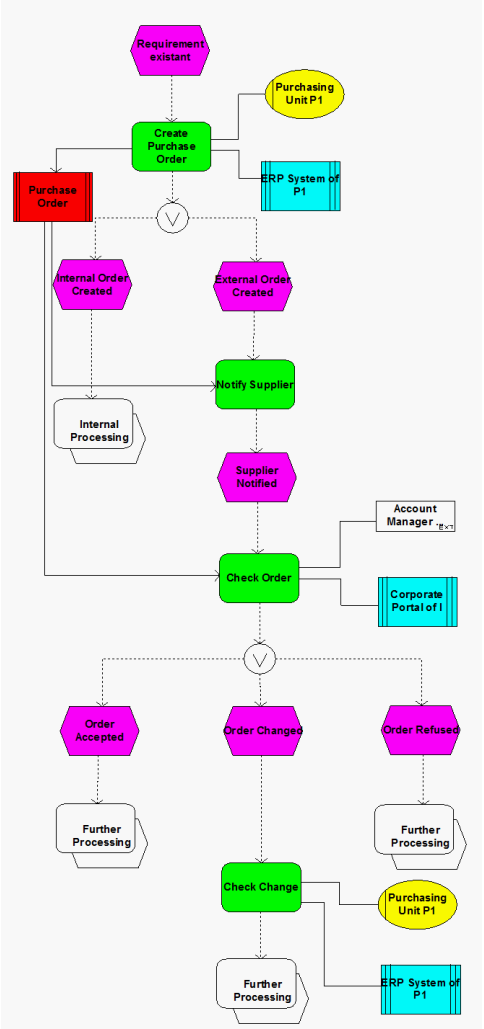

Figure 3. Business Process of the Example

Keeping the standard processes, services and patterns in mind, the central governance of *I* can easily design the coupling-system by analyzing the requirements of the business process. Afterwards, the design can easily be implemented by an IT supplier.

The ERP system of the purchasing unit and the portal application need to exchange purchase orders and order changes. In addition, the supplier has to be informed about new orders. As a result of a questionnaire presented to the process owner, the governance identifies that this is to be done via email. Additionally, only the ERP system is capable of writing orders to a relational database and for receiving order changes by retrieving them from the database. In turn, the portal computes purchase orders that are specified in the XML-schema of *I's* industry [2]. The same applies for generated order changes.

Following the IIF, it is identified first that the ERP system does not trigger the coupling-system actively and that not

[2]This format is used as the canonical data format within the company.

all orders need to be transferred to the portal since internal orders are handled differently. Thus, the *Event Creator Service* applies the *Polling Consumer* as well as the *Selective Consumer* idiom. As these idioms are solely dependent on the data format of the sender and on the type of sending system, the implementation can be re-used for similar scenarios involving the same type of ERP system. Required functionality for the *Receiver Service* is checked next. Here, only a database adapter with basic read operations is used and, hence, no idioms are applied. Due to the data-constraints applicable for the relational database used, invalid messages are not considered a problem by the governance and the *validity check* is not used. In order to avoid double computation of orders, the *acknowledge request* option is used. [3]

For transforming the in-memory representation of database entries into the canonical data format, the governance dictates the use of the *Transformator Service* for purchase orders. The according service is generic and it's reuse is recommended. Afterwards, the business process requires an email to be sent to the supplier. Therefore, the IOF is called asynchronously with the *Event* as the single parameter. The questionnaire to the process owner identifies that multiple mailboxes need to be informed via email, i.e. the customer as well as the purchasing unit that wants to monitor all emails that are sent to external suppliers. The detailed design of the IOF is as follows: The *Router Service* applies the *Recipient List* pattern. From an outside point of view, the Router Service splits the message in two and sends them to the *Transformator Service* in order to transform them into email-messages according to a given XML schema. For the update step, the design only recommends the usage of an SMTP-mail adapter that deals with the specific XML-message.

The next step is checking the order by the supplier by logging into the portal and reviewing the order. Since this is not an integration task, it is not described by the patterns and processes here.

However, it is an important point to stress that the portal application uses the data that are stored into the coupling-system's data store and, hence, has to be able to compute the data format of the coupling system [4].

As the final part, the design should include the propagation of the order change back to the ERP system. Therefore, the design is aligned with the integration process again. We ommit the details since the IOF for the update of the ERP system does not introduce the need for additional idioms.

[3] This acknowledgement is realized by writing an additional entry in the table of the ERP system using the IOF that connects to the ERP system.

[4] This is the line we draw between a legacy application and a composite application. In the case of a legacy system, the IOF would be used to write the data to the application system.

6 Conclusions

Our work standardizes the design of coupling systems by using a pattern taxonomy within the central governance of a customer for systems that are implemented using the BPIOAI middleware from SAP - the Exchange Infrastructure. This way of designing coupling systems also brings value to the implementation of composite applications.

Currently, we are collecting requirement classes from different real-life case studies. In doing so, we aim to define a formal language that transforms the requirements of a business process in conjunction with a dynamically created questionnaire into the implementation of a BPIOAI coupling system using the taxonomy presented here.

References

[1] D. S. Linthicum, *Next Generation Application Integration*. Boston, MA USA: Addison-Wesley, 2004.

[2] C. Alexander, *A Pattern Language*. Oxford University Press, 1977.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Addison Wesley, 1996.

[4] J. Greenfield and K. Short, *Software Factories*. Indianapolis, USA: Wiley Publishing, Inc., 2004.

[5] G. Hohpe and B. Woolf, *Enterprise Integration Patterns*, ser. The Addison Wesley Signature Series. Pearson Education Inc., 2004.

[6] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow patterns." *Distrib. and Parallel Databases*, vol. 14, no. 1, 2003.

[7] N. Russell, A. H. M. ter Hofstede, D. Edmond, and W. M. P. van der Aalst, "Workflow data patterns: Identification, representation and tool support." 2005.

[8] N. Russell, D. E. ter Hofstede, Arthur H.M., and W. M. van der Aalst, "Workflow resource patterns," *BETA Working Paper Series*, vol. WP 127, 2004.

[9] P. Jayaweera, P. Johannesson, and P. Wohed, "Process patterns to generate e-commerce systems." in *ER (Workshops)*, ser. LNCS, H. Arisawa et al., Eds., vol. 2465. Springer, 2001, pp. 417–431.

[10] [Online]. www-306.ibm.com/software/info1/ websphere/index.jsp?tab=solutions/process

[11] [Online]. www.seebeyond.com/software/ican.asp

[12] [Online]. "help.sap.com/saphelp_nw04/ helpdata/en/14/80243b4a66ae0ce10000000a11402f/frameset.htm"

[13] G. Kaufman, "Pragmatic ecad data integration," New York, NY, USA, Tech. Rep. 1, 1990.

[14] "Corba specification, v3.0.3." [Online]. Available: http://www.omg.org/cgi-bin/doc?formal/04-03-12