Process-Oriented, Consistent Integration of Software Components

R. Depke, G. Engels, S. Thöne  
Dept. of Computer Science  
University of Paderborn  
33095 Paderborn  
{depke,engels,seb}@upb.de

M. Langham, B. Lütkemeier  
S&N AG  
Klingenderstr. 5  
33100 Paderborn  
{mlandham,bluetkemeier}@s-und-n.de

Abstract

The integration of software components becomes a more and more important issue in software engineering. It results from the rising demand for the coupling of business applications in order to provide automated processes. Basically, software components and even whole applications have to be integrated in a consistent way, i.e., their export interfaces have to be respected by the importing components. In the same manner the demand to integrate components even at runtime requires integration specifications to be flexible. Especially, the type system of component interfaces has to support a tunable degree of freedom. This allows the insertion of components with interfaces of restricted but sufficient degree of compatibility. In this paper, we develop a concept for consistent and flexible integration of components. We present a process modeling language that combines UML and XML in order to support consistent, flexible, and executable processes. Finally, we provide a formalization of the proposed component type system and we describe the tool support for construction and execution of consistent process models.

Keywords: component-based software development, internet and web-based systems, enterprise systems, process modeling, UML, type system, XML

1. Introduction

Integration of software components, currently known as Enterprise Application Integration (EAI), is a very important problem for companies that wish to automate their business processes. As long as their software systems are running independently, most of the business processes need human interaction to overcome borders between the underlying software systems. Especially, if the companies want to offer their services online over internet or intranet (e-business), they have to integrate their systems in order to enable an efficient response to customers’ and partners’ requests.

In this typical scenario, software components or whole applications usually do not know that they are integrated. Therefore, the integration happens outside of the components. The control and data flow between the software components is handled by a dedicated integration facility. Such a broker component is responsible for the application of the business rules that determine the component integration. The business rules reflect in the descriptions of the integration processes. Such process descriptions are demanded to be machine executable in order to automate the integration of components.

An execution oriented representation of a process description is usually not easy to read and to understand for human designers. In order to abstract from the implementation details and the technical representation it is helpful to create a model of the process. Moreover, models help to overcome different forms of presentation and offer a more uniform view to the problem domain.

In contrast to common analysis models, which are filled with more details during software development phases, process models should contain all semantic details to make them executable. Despite this completeness the process models should be easy to understand, e.g., that they can be discussed with business engineers. This increased understandability can be achieved by visual models.

Software components are composed by the use of connectors. According to Garlan [4] connectors are entities that determine the interaction type of two components. The type specification of a connector determines which kind of data may be transmitted via the connector. In this way, the consistency of the data flow from one component to another is ensured. A contradicting demand consists in the flexibility of the type system. General connectors are required that can be applied for the connection of many different components. The
type of the data flow via a general connector should only be partially fixed.

Especially, this aspect becomes important if components deal with XML documents. The type of an XML document is given by an XML schema. Often, when XML documents are transferred between different components only part of the XML schema is relevant, e.g., if only a part of the XML document is processed by a component. In industry, there is a growing spread of components which support the exchange of XML documents: More and more databases offer XML interfaces which affect the encompassing applications. Also, the upcoming web service technology basically depends on the exchange of XML documents. This situation requires a particular consideration of XML-based data flow.

Altogether, there is a need for visual, complete and executable process models that handle XML documents and that manage the dichotomy between flexibility and consistency of the connectors’ type system in software component integration.

In this paper, a process modeling language is presented which meets the named requirements. The language is a profile of the already well-known and extensible visual modeling language UML [8]. The extension mechanism allows to introduce XML document types as data types into models. The semantics of the type system for XML documents in process models is formalized in order to make clear how consistent and flexible component integration is feasible.

The paper is organized as follows: In Section 2 we survey the literature for solutions to the problem of flexible software component integration. In Section 3 we give an example which illustrates the necessity of flexible and consistent software component integration and the benefit of our solution. We present our concepts and introduce a modeling language that supports a type system that is both strict and flexible. The formalization of the solution is described in Section 4. Section 5 gives an overview of a newly developed model editor and a workflow processor which executes the process models. Section 6 concludes the paper.

2. Related work

The flexible integration of software components is an important topic that is currently discussed in the literature [9, 12, 1]. A main problem of component integration is to check or to effect compatibility between different components. Besides functional requirements there also exist compatibility requirements regarding, e.g., the types of protocols used, the types of event handling or the amount and types of resources used. Yau et al. [12] describe a framework for component integration that supports compatibility checks of component interface properties and that enables the coupling of components at runtime.

Bennett et al. [1] present a software architecture that addresses the problem of application integration. Single applications are considered as service components that fulfill some dedicated requirements. Typically, services are “thin” applications. Service components that currently exist in some kind of electronic marketplace are composed to new services on demand during runtime. Service composition relies on composition templates which are populated by services from the marketplace.

Bennett, Yau et al. [12, 1] stress the importance of architectures that integrate components during runtime in a flexible way. With respect to their proposals the question arises in which way flexibility of component integration can be achieved. It remains to clarify under which conditions one component (type) can be substituted by another similar one.

The process-oriented integration of applications is also an important aspect of workflow management systems. Workflow models describe the control and data flow needed to integrate applications. The models are executed by the workflow management system and in doing so the concerned applications are accessed via adaptors. The architecture of workflow management systems is standardized in the reference model of the workflow management coalition (WFMC) [10]. The reference model is also respected by the Workflow Management Facility V1.2 standard of the OMG which provides workflow management based on CORBA [7].

Both the WFMC and the CORBA effort have drawbacks. The reference model of the WFMC does not support a standardized visual representation of workflow models, like for example one based on UML. The CORBA facility lacks a formal language for process models with well defined syntax and semantics. Both efforts do not offer a flexible concept of integration of applications by use of XML.

The process-oriented integration of software components by use of models has also been carried out in Petri net contexts. For example, FUNSOFT nets are high level Petri nets which have been introduced by Gruhn to model the dependency of activities and the flow of objects within workflows [2]. A major advantage is the formal semantics of Petri nets which makes them executable. But the type system of the approach is as rigid as in common programming languages and thus the requirement of flexible component integration is missed. Finally, different to UML, Petri nets were not able to reach similar acceptance in industry for modeling busi-
ness domains, processes, and applications.

Since none of the proposals fulfills the identified demand of flexibly typed, executable, XML- and UML-based workflow models in a comprehensive way we will now present our own proposal.

3. Flexibly typed, executable process models

This section introduces the necessary concepts for modeling static and dynamic aspects of executable integration processes. At the beginning we give an example from the banking area that is continued throughout the section. The four subsections deepen special modeling aspects: Subsection 3.1 deals with the specification of available components and their functionality. In 3.2 the port type system for interface specifications is introduced that simultaneously allows both flexible and consistent composition of software components. Subsection 3.3 shows how state-transition diagrams can be used to flexibly compose the dynamic process models in consideration of type safety. The last subsection 3.4 introduces the concept of typed transitions in combination with transformations. These transformations increase flexibility, because they allow the linking of even incompatible interface types.

The following concepts are illustrated by a simple example from the banking area (see Figure 1): A bank wants to replace hand-written forms for remittances by an online-banking facility. Therefore, they introduced a web portal where customers can fill in electronic forms. The order information is then submitted as an XML document via internet. Arrived at the bank, the document must be automatically processed to execute the remittance. Assume that there is a finance application to transfer the money from the source to the target account. If the target account belongs to another bank, the credit must be communicated to that bank. Furthermore, the bank is placed under the obligation to save backups of all incoming order documents in an XML archives database.

This simplified banking problem can be solved by modeling a suitable integration process and executing it by a process broker. The customer’s transfer order becomes the input to the process. In each step, the process broker invokes one of the existing software services (see Figure 2). After completion the results are returned to the customer again.

The output of one service usually contains intermediate results which are required by the next. As the considered components offer XML interfaces (see Subsection 3.1), the workflow processor must forward these results as XML documents which are then gradually processed by succeeding services. Thus, control and data flow mostly depend on each other. For instance, in the banking scenario the output document of the debit service contains a credit order that must be sent to one of the credit services in order to compensate the debit (see Figure 2).

Taking this requirement into account the following concepts are based on a document migration model, which is similar to known circulation folder and object migration models of workflow management systems [5]. Instead of an object that is transmitted from one worker to the next, here an XML document is transmitted from one software service to the next and processed in accordance to their functionality.

3.1. Integration of components as services

Since the interfaces for access to the software components are very different, they are connected to the workflow processor via individual adaptors. Each adaptor provides a set of services that encapsulate the native interface of the components. This way the possibly complicated native interfaces do not need to be known by the workflow processor. It can rather communicate with the adaptors via uniform XML interfaces: Each service expects an XML document as input, uses the encapsulated functionality, processes the document and returns a new or modified XML document as output.
The encapsulation also improves the maintainability, because it allows for an easy change of the integrated components: If the native interface of a component changes or a new component must be integrated, only an adapter must be created or rewritten to support the new interface. An extension of the workflow processor is not necessary.

Figure 3 shows adapters and services of the banking example. There is one adapter for the finance application of the bank that offers services for debiting and crediting sums to be transferred. The credit service is used if the target account is at the same bank. Otherwise, the remoteCredit service of the Communicator adaptor is used. The XML-Archives adaptor with the writeBackup service encapsulates a database that saves backup copies of all incoming order documents.

Adaptors and their services are modeled in a structural, static model which UML class diagrams are used for. There, adaptors can be modeled as interfaces and their services as methods of the respective interface. Figure 4 shows the static model of the banking example. The attached input and output ports specify the requirements of the services concerning the XML interface. The next section describes this concept in detail.

3.2. Interfaces based on ports

An advantage of XML is its flexibility in saving structured data. XML schemas are used to define certain XML structures and document types. A document type is identified by name and namespace of a certain XML element. If an XML document contains this element as root element, it belongs to that type.

The service interfaces usually require special document types. These type requirements must be included in the service specifications, in order to enable consistency checks concerning a right service composition and to avoid interface mismatches. For this reason, the types are specified by ports: Input ports declare which XML document types fulfill the input requirements of a service, output ports which types may be returned as output.

Exemplary, consider the debit service of the given static model (Figure 4): It understands input documents of type transferOrder and returns documents of type creditOrder. Namespaces are omitted for better readability.

There are some services, for which the document structure is not relevant, e.g. the writeBackup service that saves arbitrary XML documents in a database. The wildcard #any instead of the root element specifies the input port of those services.

For some services, the document structure is only partially relevant: They do not mind the entire structure of the document, but require a certain subelement. These partial type requirements are specified by listing the subelements after the #any wildcard.

Because there are services which can process or produce documents of different types, ports can also contain more than one document type. Then a document conforms to the port, if it conforms to at least one member of the document type set.

As UML is not intended for presenting port specifications, they are assigned to the services as UML notes.
The content of the notes must follow a strict syntax of the ports, see Figure 4. Therefore, the port notes can be interpreted by designers and also by tools.

**Open ports.** Consider services that always return the same document types as they get as input. Normally, their output port would read the same as their input port then. Since this is often the case, when the document structure is not relevant for service execution, the input port mostly allows arbitrary document types (#any). But this wide range should be avoided for output ports, because otherwise a wide-ranged input port would be required for the receiver of the output as well. However, services with such loose input requirements are rare, because they usually require special document types to successfully access the document content and to perform their individual tasks. Thus, wildcards in output ports would decrease the variety of possible service compositions.

Consider the writeBackup service in Figure 4 for instance: As it does not modify incoming documents, its output port normally would read #any as well. But then a type checking editor would not allow any of the remaining services as successor, for they all have got stricter input requirements and cannot process an arbitrary document type.

Open ports are helpful to specify those output ports without using the wildcard #any. They rather just specify the service behavior, not to modify the incoming document type. But as such generic ports should not lack type safety, they are bound to concrete document types as soon as the actual input types to the service are known from the data flow context. The number of possible output receivers is increased and a higher flexibility for the use of the service is achieved. Thus, open ports and their binding overcome the discrepancy between flexibility in using the service and type safety required to check consistency.

Open ports are specified by the placeholders #copyRoot and #copySubs. The binding replaces these placeholders by concrete XML elements, when the service is used in a dynamic model as described in the next section. If there are some root- or sub-elements that are not preserved by the service, they can be excluded by specifying a list of exceptions as additional postfix for each placeholder.

### 3.3. Process diagrams

The process diagrams used to model the integration processes extend UML activity diagrams. Activity diagrams are a well-known kind of state-transition diagrams. They are frequently applied for workflow modeling [3], which has similar requirements for dynamic models. The diagrams offer all necessary elements to model flexible control flows including concurrency and synchronization, branchings and subprocesses, definite start and end states. There are mainly two kinds of states to distinguish in activity diagrams: pseudo states to model the control flow and activity states to model operation calls. As process diagrams only allow services as activities, they are also called service states. Directed transitions connect the states with each other.

Usually, object flow states are used to model data flow in activity diagrams. Here a lot of them were necessary to model the document migration for each transition. Therefore, they are omitted to avoid a too large number of them, which would affect the understandability of the models. Instead, the control flow semantics of transitions is extended by data flow. Thus, transitions represent both control flow as well as document flow of the migrating XML document.

If a service state embeds a service whose signature contains parameters, these must be set to concrete values or to expressions which are evaluated at runtime. XML query languages like XPath [11] can be used, if the argument depends on the actual content of the migration document.

Figure 5 shows two process diagrams that model the necessary processes for the banking example. The process diagram transfer represents the core activities concerning the money transfer from the source to the target account. After the backup saving it contains the two steps of debiting the money from the source account and then crediting it to the target account. Depending on whether the receiver's account is located at the same bank, a local or remote crediting is done.

The second process diagram illustrates, how processes can be composed from subprocesses by subactivity states. In this case, the transfer process and another subprocess for customer relationship analyses are used as submodules for the remittance process. This composability increases the reusability of the models.

The two subprocesses are modeled as concurrent process sections. Those concurrent sections get individual copies of the migration document. This facilitates the concurrent access to the document data and avoids deadlocks, which might occur during synchronized data access. At the join state the different document copies have to be merged again. Therefore, a new migration document is created and all resulting copies are included under the new root element. The transition directly following the join state can then apply a suitable transformation stylesheet to restructure the merge document and to get rid of any redundant data. The use of transformations on transitions is discussed.
Thus, there is no extra-work for the designer except what can be done by a modeling tool as presented in Section 5. All this can be derived from the static model. For pseudo states exist special rules to calculate the ports. All this can be calculated by the modeling tool, which prevent incompatibilities and dramatically reduce the rate of interface mismatches.

Since it is sufficient, if the modeling tool internally manages the ports of the states, they are omitted from the diagram to reach better readability.

As shown in Figure 5, the process diagrams get input and output ports of their own. They specify which document types can be processed and produced by the process as a whole. This is necessary to check port consistency, when the diagram is embedded in subactivity states.

3.4. Bridging incompatibilities with typed transitions

Although sometimes source and target port of a transition do not conform to each other, the migration document supplied by the source port can still contain the right information, but merely in the wrong structure. Then, the transition can be made possible by a structural transformation of the document. These transformations can be specified by suitable XSLT stylesheets [11] to be performed by an XSLT processor.

The transformations can be regarded as internal services which are necessary to connect the external ones. In order to maintain the understandability of the diagrams, they are not represented as further service states, but assigned to the state transitions. For that purpose a type concept for transitions is introduced, where each transition type represents a certain transformation function. Every transition instantiates one of the types to apply the assigned transformation. This type concept improves the reusability of the transformation functions, because they must be defined only once and can then be reused for all instances of this type. This also facilitates any stylesheet modifications.

Similar to services, the transition types get input and output ports, too. They specify the document types which the stylesheet is able to transform, and those which result after the transformation. To increase the flexibility also open ports are allowed. When a transition of a certain type is created, it can be checked, if the output of the source state conforms to the input port of the transition type, and if vice versa the output port of the transition type conforms to the input of the target state (see Figure 7). This check of port compatibility supports the designer in creating consistent transitions only.

Figure 6 shows the definition of two transition types: The assigned XSLT stylesheet of insertRemote trans-
forms XML documents of type creditOrder [ ] and inserts the subelement ⟨remoteServer⟩. After the migration document has passed a transition of this type, it conforms to the port creditOrder [remoteServer]. The second transition type #idle is used for transitions that do not need any transformations, because their source port already conforms to their target port.

4. Formalization

Ports were introduced in order to support the plug-in of services into workflow models and document transformation along transitions of workflow models. The connection of an input port to an output port has to ensure that a flow of XML documents via this connection is capable. Ports must establish a type system for XML documents. Therefore, this section formalizes the concept of ports.

The set Xeles consists of all XML elements defined by any XML schema. Every element in Xeles is uniquely given by its name and the namespace defined by the XML schema. The set of all valid and well-formed XML documents is defined as Xdocs. Two functions are defined for XML documents: root : Xdocs → Xeles delivers the root element of a document and c : Xdocs → 2Xeles gives the set of all XML elements contained in a document.

An XML document type is essentially given by a root element. This root element may be replaced by the wildcard #any. Besides, additional mandatory subelements may be required by the type. In this way a XML document type invents partial constraints for type compatible XML documents. The requirements lead to the following definition of XML document types:

$$DocTypes = \{(r, s) \mid r \in Xeles \cup \{\#any\}, s \subseteq Xeles\}$$

The function \(L\) returns the set of documents conforming to a given document type: \(\forall type = (r, s) \in DocTypes:\)

$$L(type) = \begin{cases} \bigcup_{x \in Xeles} L((x, s)), & \text{if } r = \#any \\ \{doc \in Xdocs \mid \text{root}(doc) = r \land s \subseteq c(doc)\}, & \text{else} \end{cases}$$

Based on the XML document types a port \(p\) is defined as a set of XML document types: \(p \subseteq DocTypes\). The set of all ports is named Ports. Consequently, the function \(L\) is continued to the set Ports in order to get all conforming documents for a port \(p\) in Ports:

$$L(p) = \bigcup_{t \in p} L(t)$$

With the help of this definition the compatibility condition can be formulated as follows: An output port \(p_1\) may only be connected to an input port \(p_2\), if \(L(p_1) \subseteq L(p_2)\) is true, i.e. all documents possibly returned through the output port \(p_1\) must also be contained in the document set of \(p_2\). Whenever two ports are to be connected, this condition has to be checked (see section 3.3). It is difficult to determine \(L(p)\) for any given port \(p\): For example, consider a subelement whose number of occurrence is unbounded. Then there is an infinite set of documents conforming to that port. Thus, the subset relation cannot be decided by comparison of the individual elements.

As the set \(L(p)\) is not available, the compatibility condition has to be decided on the ports themselves. For this purpose, a special subport relation “\(\subseteq\)” between ports \(p_1, p_2 \in Ports\) is defined as follows:

$$p_1 \subseteq p_2 \iff \forall (r_1, s_1) \in p_1 \exists (r_2, s_2) \in p_2 :$$

$$r_2 = r_1 \lor r_2 = \#any \land (s_2 \subseteq s_1)$$

This subport relation is constructed such that each document type of \(p_1\) needs a corresponding document type in \(p_2\) which has the same root element or the wildcard #any and which does not require more subelements than the type of \(p_1\) can provide.

In contrast to the compatibility problem \(L(p_1) \subseteq L(p_2)\) the subport relation \(p_1 \subseteq p_2\) can easily be decided by simple set operations. By use of the following theorem, a tool can easily verify the type compatibility of ports by just checking the subport relation.

**Theorem 1 (Compatibility)**

\(\forall p_1, p_2 \in Ports : p_1 \subseteq p_2 \Rightarrow L(p_1) \subseteq L(p_2)\)

The theorem states that the compatibility condition directly follows from the subport relation. The proof
is omitted due to space limitations. It makes use of the construction of the subport relation “⊆”. In few steps it can be concluded that under the assumption of $p_1 \subseteq p_2$ an arbitrary document contained in $L(p_1)$ is also a member of $L(p_2)$ [6].

Figure 7 shows an example, where the subport relation is used as compatibility condition. There a transition type is to be instantiated which is only allowed if the ports are compatible to one another.

Open ports rely on open document types. These document types extend the types of DocTypes by allowing the root element of types also to consist of literals #copyRoot or #copySubs. Open ports are then defined as sets of open document types and OpenPorts is the set of all open ports. In order to use open ports they are mapped to non-open ports by a binding function. The binding of an open ports is relative to a non-open port

$$
\text{bind} : \text{Ports} \times \text{OpenPorts} \rightarrow \text{Ports}.
$$

The root and subelements’ types of the non-open argument port are returned by bind if the open argument port contains open types #copyRoot or #copySubs.

Transition types are flexible because their target port can also be open. Transition types are formalized as triples $(\text{source}, \text{target}, \text{trans})$ where $\text{source} \in \text{Ports}$, $\text{target} \in \text{OpenPorts}$, and trans is a XSLT transformation. trans converts a document doc which is compatible to a port $p \subseteq \text{source}$ into a document trans(doc) which is compatible to a port $\text{bind}(p, \text{target})$.

The application of the concept of open ports to workflow models affects the flexibility of the approach. An output port of a service or of a transition type can be propagated to the succeeding service’s or transition’s output port if that output port is open. By use of the function bind it can be proved that transitions of transition type $(\text{source}, \text{target}, \text{trans})$ from input port $\text{in}$ to output port $\text{out}$ convert documents in a type compatible way:

**Theorem 2 (Migration)**

$$
\forall (\text{in, out, (source, target, trans)}) \in \text{Transitions} : \\
\forall \text{doc} \in L(\text{in}) : \text{trans}(\text{doc}) \in L(\text{out})
$$

This theorem ensures that the application and binding of transition types respects the concerned types of the transition. A analogous theorem holds for services with open output ports.

Thus, the function $\text{bind}$ may be applied to all services and transitions from the beginning of the workflow up to the final state. Consequently, all open ports are bound to non-open ports and the types demanded by all the services and transitions are fixed [6]. Thereafter, the resulting workflow model can be executed by a workflow processor.

5. Realization

The concepts were realized in cooperation with SkN AG in Paderborn. Their e-business integration platform sunShine (www.s-und-n.de/produkte/sunshine/sunshine.htm) is based on the open source software cocoon (xml.apache.org/cocoon) and already provides an extensive set of so-called transformers which process incoming XML documents. They can easily be used as adaptors to integrate components ranging from databases to enterprise java beans.

However, as many other similar products, the platform lacks a facility to visually model flexible processes, but only allows to specify simple sequences of transformers. Therefore, the proposed concepts were implemented and added to the SkN e-business integration platform.

Figure 8 reveals the architecture of the implementation. It consists of tools for process modeling and exe-
The visual process editor for process diagrams allows the designer to define the static elements and their visual composition to process models (see Figure 9). The editor checks well-formedness of the control flow, e.g., that every state is reachable from the start state.

Based on the formalization of the port concept the editor checks type consistency wherever input and output ports are connected. Inconsistent models are prevented by preserving the compatibility conditions. Based on the port concept the editor offers only those services and transition types suited to fit to the port arrangement of the current diagram context.

The process models are encoded as XML documents using the Process Markup Language (PML). The use of this problem-oriented XML format improves the interchangeability of the models because today many applications provide an XML interface.

The PML encoded models are loaded together with the XSLT stylesheets for transitions by the process execution component. It executes the process specification like a state-machine and delivers incoming migration documents to the individual services according to the control flow. By the port specifications the executor can verify at runtime, if the actual output of the services really conforms to the given port.

6. Conclusion

In this paper a visual and executable process modeling language based on UML was presented. Its aim of flexible and consistent integration of software components that offer and accept XML documents was reached by the concept of ports. Especially, the concept of open ports made it possible to define strict but not too restricting type requirements for XML documents that flow through components and connectors. Only necessary, partial type requirements for input and output XML documents are defined by ports. Thus, the software components could be combined in far more ways but the relevant parts of the processed XML documents were always type safe. The formalization of the new port concept was a necessary prerequisite for the construction and implementation of an editor and an execution component for process models.

References