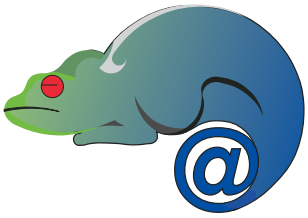
ADAPT
IST-2001-37126

Middleware Technologies for Adaptive and Composable Distributed Components

CS Adaptability Container



Deliverable Identifier: D11
Delivery Date: August, 19, 2004
Classification: Public Circulation
Authors: Gustavo Alonso, Cesare Pautasso, Biörn Biörnstad
Document version: FINAL – AUGUST 2004

Contract Start Date: 1 September 2002
Duration: 36 months
Project coordinator: Universidad Politécnica de Madrid (Spain)
Partners: Universitá di Bologna (Italy), ETH Zürich (Switzerland), McGill University (Canada), Universitá degli Studi di Trieste (Italy), University of Newcastle (UK), Hewlett-Packard (UK)

Project funded by the European Commission under the Information Society Technology Programme of the 5th Framework (1998-2002)

Dependencies with other deliverables

This deliverable is a description and evaluation of parts of the ADAPT composition engine (deliverable D14, CS Middleware) which is currently at the prototype stage (deliverable due on month 29). The deliverable builds upon deliverables D9 (CS Middleware Architecture, due on month 5) and D7 (Composition Language, due on month 11). The aspects of the evaluation plan that pertain to this module are covered by this deliverable in the form of performance measurements.

As indicated by the reviewers on the first review of the project, we have opted for a format in the report that matches scientific papers. Part of this report has been submitted for publication in: C. Pautasso, G. Alonso, Visual Adaptation of Mismatching Web Services, Second International Workshop on Semantic Web and Databases.

1 Motivation for the deliverable

According to many recent sources [10, 21, 24, 26], Web services are a step in the right direction towards fulfilling the very old idea of building software systems out of reusable, off-the-shelf, components [23, 38]. The key to achieve this is interoperability, the declared objective of most efforts around Web services [1]. One typical scenario which defines Web service composition is when value added services are built out of existing basic ones, without the component services even being aware that they are used as components [8, 22]. Today, Web services already support this kind of scenario quite well, as they greatly enhance the interoperability of software components distributed across the Web [34, 35]. However, it can still be difficult to build applications out of existing, independent Web services, as they spontaneously appear and disappear from the Internet and as they may not have been designed with compatible interfaces to begin with.
More specifically, one of the problems we are concerned with is that building applications out of composite Web services can lead to disastrous results as soon as one of the services becomes unavailable at run-time. Another problem is that independent services published at different times tend to have mismatching interfaces. At design-time, this makes their integration into a coherent application difficult, since the information to be exchanged is represented in different ways.

In this report we show how to address both of these important problems in the context of the ADAPT adaptability container. First we observe that the reliability of the composite service can be increased through redundancy. Assuming that a set of equivalent, alternative service providers are available, with ADAPT it is possible to use various exception handling and synchronization techniques to model the invocation of a service chosen from alternative providers and control precisely what happens in case of failures. Second, when building a composite service out of heterogeneous ones, ADAPT not only uses a visual language to program the data and control flow dependencies between the services, but also supports the visual definition of the necessary glue code. Using a very simple visual syntax it is possible to model the transformations of the data exchanged between mismatching service interfaces. Existing solutions to this data integration problem are usually based on style sheet transformations (XSL [32]) and tend to be quite cumbersome to program and maintain. Although it is still possible to use such techniques with ADAPT, we also offer an alternative way based on the visual definition of mappings and transformations between different XML data types. According to our experiments, not only is the ADAPT visual notation easier to program but thanks to ADAPT's compiler, it can also be executed efficiently.
Given that it is not so common to find alternative providers of perfectly substitutable services, adapting a service to a given interface can both enable its interoperability with other services which already fit

with this interface as well as extend the set of services we can consider as alternatives when handling the failure in invoking one of them.

The outline of this report is as follows. In Section 2 we give some background on the ADAPT system describing its basic Web service composition features. Then, we move to presenting its more advanced features concerning reliabile service invocation (Section 3) and heterogeneous data integration (Section 4). In Section 6 we show a performance comparison between the execution of mappings specified visually and the equivalent mappings written using XSL. In Section 7 we present some related work on Web service composition, focusing on contributions related to adaptability, before we conclude in Section 8.

2 Visual Service Composition

The main idea behind ADAPT is that a visual language, as opposed to an XML based representation, can greatly improve the understanding of composite Web services [29]. In the context of this report, no matter whether a conflict between mismatching interfaces has been resolved automatically using additional semantics [4] or a human programmer has been providing the necessary input, using a visual surface language to give a representation of the result is of fundamental importance to enable its understanding. Based on this idea we have built a true visual environment for rapid Web service composition, which requires very little knowledge of the underlying XML-based technologies and has the potential to increase developer's productivity when composing Web services. The latest ADAPT version can be downloaded from [27]. In this section we present how some of the basic features of ADAPT can contribute to addressing the problem of service interface adaptation.

2.1 Process Based Composition

A process is described using a simple visual language, which specifies its structure in terms of the data and control flow relationships between its component tasks [29]. The data flow graph of a process defines how data is exchanged between the tasks composing the process. Using a data flow based language to model the composition of Web services is also quite useful to specify data transformation using the same paradigm. In addition to modeling service composition at a large scale, a data flow representation can also be used to specify the necessary interface adaptations on a finer grain. As opposed to existing declarative approaches based on the visual specification of pattern matching rules [30, 40], the data flow graph of a ADAPT mapping defines operationally how to compose a set of operators in order to define how the data exchanged by the services should be transformed between two different representations. From the data flow graph, ADAPT automatically derives the control flow graph of the process, based on the assumption that a task cannot be invoked before its input data is available. The user can edit the control flow graph, for example, to specify additional constraints, to add exception handling tasks, or to model branches by annotating its edges with conditions.

2.2 Rapid composition of service interfaces

A process in ADAPT defines the composition of services at the level of their interfaces. The interface of a service is defined in terms of the information required to invoke it (the input parameters) and the information returned after its invocation has completed (the output parameters). These interface parameters are typed. Using typed parameters is a very simple way of annotating the interface with additional semantics, which gives the following practical benefits. First of all, the values of the parameters are constrained within a given type system, which is part of the service's ontology. At design-time, this allows ADAPT to detect and mark data flow connections between mismatching parameters. The developer receives immediate visual feedback of the problems and can add the necessary adapters. At run-time, the actual values of the parameters can be checked against the appropriate assertions and constraints, which can be derived from the type definitions.

2.3 Heterogeneous services

Another important feature of ADAPT is that the tasks composing a process can represent several heterogeneous types of software components. In addition to remote Web service invocations using the SOAP [34] protocol, ADAPT currently supports many other service *access mechanisms*. These include, for example, the execution of external UNIX/Windows programs, blocking or non-blocking calls to other ADAPT processes and the execution of X-Path [33] queries and XSL data transformations. For tasks representing fine-grained operations, it is also possible to efficiently embed small scripts written in Java within a process.
Opening up the set of service types that can be composed increases the need for including mediation as part of the composition language. The heterogeneity of the types of services that can be integrated into a process make it more difficult to adapt such services to one another. However, such heterogeneity also helps to give an elegant solution to the problem of including mediation constructs in a service composition language. In our approach, the role of adapter (or mediator) can be taken by any of the services included as part of a process. Thus, such mediation services are seen as a particular kind of service that can be included in the composition following the same visual syntax used to compose any other service.
More precisely, an interesting consequence of supporting different types of services lies in the additional flexibility provided when programming the adaptation logic between two mismatching service interfaces. Depending on the complexity of the actual transformation and on the developer's familiarity with XML-related technologies (e.g., XSL), the most appropriate kind of mediation service can be chosen to build the transformation. For example, small data conversions can be implemented efficiently with scripts written in Java, or as we will show in Section 5, XML manipulations can be both specified visually as well as with XSL transformations.

2.4 Efficient and Scalable Process Execution

Although visual process-based modeling has a significant advantage over traditional programming languages, due to its high level of abstraction which fits quite well with the service composition paradigm, it will not be successfully adopted if available tools cannot deliver a comparable level of performance when executing the composite services. Typically, once Web services and other components have been composed into a process, the result is also published as (another) Web service. To do so, the process engine is typically run embedded into an application server to drive the published composition in response to client requests [11, 17].
Also ADAPT's availability container can be deployed in this configuration, in order to manage production level processes. In addition, it has the ability to scale out over a cluster to keep a certain level of performance when facing a high workload due to a surge of client requests [28]. To further reduce the process execution overhead, in ADAPT, before they can be executed, the processes are compiled from their visual representation to executable Java code. The resulting process template plug-ins are then dynamically loaded into the container to efficiently control the enactment of many process instances in parallel.

3 Reliable Service Invocation

A Web service can become unavailable for many different reasons. The connection across the Internet to the Web service provider can fail during a SOAP message round. The Web service may have been taken offline temporarily for maintenance. The Web service might have been renamed or moved to a different server and, as a consequence, the binding information in its WSDL description may be out-of-date. Whatever the reason, from the point of view of the composite application, a failure to contact a Web service can be very similar to dereferencing an invalid pointer in traditional applications. If no corrective action is taken, it may lead to the failure of the whole business process. To ensure a successful invocation, failure handling actions can be taken at different levels in the communication stack as well as at the process level itself.

3.1 Dealing with communication failures

To deal with problems at the communication layer, the SOAP message can be transferred with an asynchronous messaging or queuing system, instead of using a synchronous HTTP connection [39]. Although its latency may increase, the communication becomes resilient to temporary communication problems, as the messaging system can store and forward the message when an appropriate path to the service provider can be found. A similar approach can also help with temporary outages of the server itself.
If the location of a WSDL interface description is not changed, and the interface of the service is not modified, changes to the binding (for example, the server's location has been moved to a different host), should remain transparent to the process because the latest version of the WSDL can be fetched before reattempting the call. If a WSDL contains multiple bindings, the SOAP communication library may attempt to contact all of them before reporting a failure of the invocation back to the application [18].

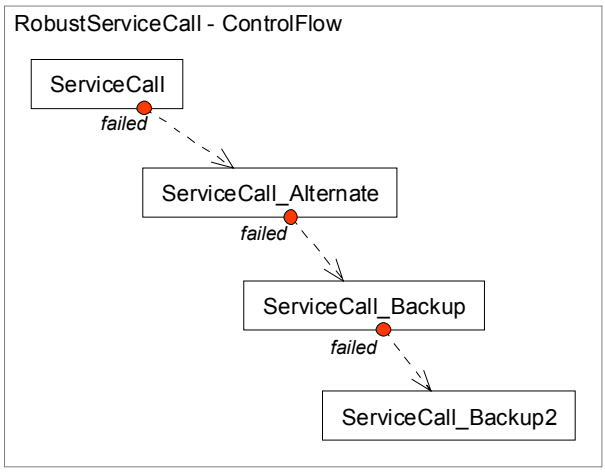


Figure 1: This chain of four Service invocations is traversed as each call fails. ADAPT attempts to contact the four alternative, equivalent services in the order specified by the control flow graph drawn by the developer.

3.2 Handling failures at the process level

All of the previous steps are usually taken inside a SOAP communication library and can be controlled by specifying the appropriate bindings in the WSDL description of the service. Although these mechanisms can improve reliability in case of short, temporary outages, the service invocation will still fail if the service cannot be contacted after a certain time. Therefore it is also important to deal with such failures at the application level, assuming that a suitable equivalent service implementation can be invoked at many alternative service providers.
In ADAPT we offer a set of language constructs to deal appropriately with irrecoverable failures of a certain Web service. This means that, when everything else fails, it is still possible to model explicitly, at a high level, what to do, using both a form of *retry on exception* and, with some restrictions, an advanced synchronization technique applied to a multi-instance pattern, both of which we will illustrate in the rest of this section.

Exception Handling The first approach is based on the basic exception handling construct of ADAPT. As shown in Figure 1, two service invocations can be connected by a `failed` control flow dependency, which will be triggered only if the first invocation fails, so that the second one can be tried in its place. The example shows this pattern applied to four services, which are connected in a *failure triggered chain*. They are to be invoked sequentially, but only if the previous service in the sequence fails.

This approach has the advantage that it is rather easy to program: in ADAPT all that is required is to make a duplicate of a service invocation (including existing data flow connections), substitute the target service to be invoked with an equivalent one and connect the two invocations with the exception handling control flow dependency. However, in our experience, this is an inflexible solution, because both the set of services to be tried and the order of the attempts is fixed and cannot be changed once the process is deployed. These limitations are overcome by the next construct, which allows the usage of a dynamic set of alternative services and does not constrain the order in which they are contacted.

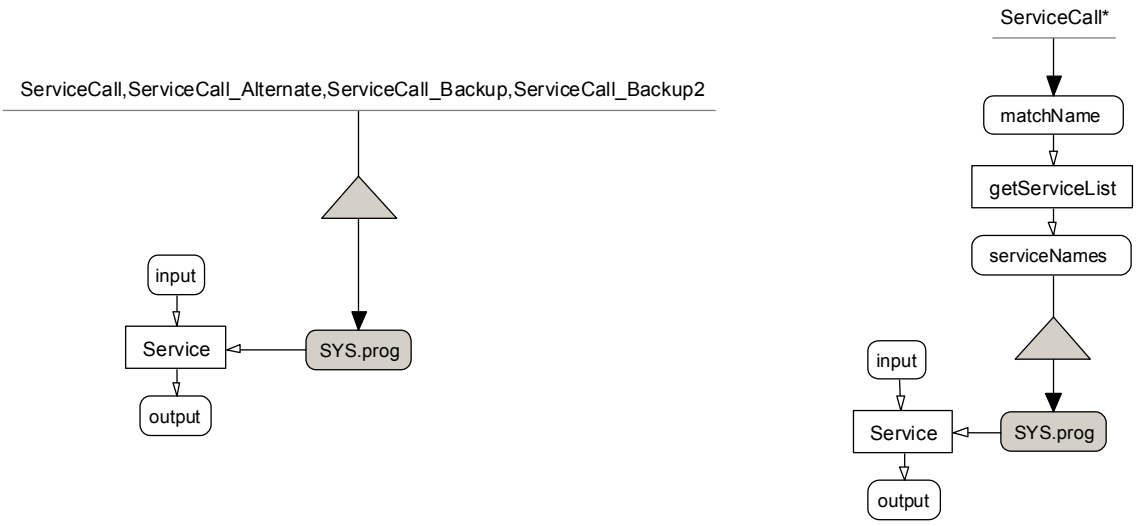


Figure 2: This data flow diagram represents the parallel invocation of the same four services of Figure 1. In the static implementation (left) the list of services is hard coded, while in the dynamic implementation (right) the list of services is retrieved at runtime. The split operator (triangle) can be configured so that the invocation will complete as soon as one of the services will respond successfully (Figure 3).

Parallel Invocation Not only the availability of the various services may change at runtime, but also their response time may depend on the current load at the different service providers and on the network congestion. The previous model captures the cascading retries that are performed if a service in the sequence of alternative invocations doesn't respond or fails. Now, although the services are alternative and equivalent, their order in the sequence is fixed at design-time and it is always the same for all executions of the process. In some cases, it may be useful to use a different pattern. Instead of modeling this behavior as a sequence of invocations, we use a set of invocations which are all started at the same time. The result of the first one which successfully completes is taken and all others (even if they don't fail) are simply ignored (or aborted). This way, the process is again resilient to failures of all but one of the Web service involved. Additionally, its performance is potentially better, as the call completes as soon as one Web service responds. In case of failures, the execution is not delayed by the calls that cannot be completed.
We can model this behavior in ADAPT by using the *parallel split operator* applied to a list of service names (Figure 2). This way, the invocation of each alternative service in the list will be initiated in parallel. The synchronization condition associated with the split operator can be set to make the parallel invocation terminate as soon as one service returns without a failure (Figure 3). With the appropriate settings, we can achieve both optimal response time (as the result from the most responsive service is taken, while all others are ignored) and we can choose to ignore the service invocations that failed.
In Figure 2 (left) the list of services is still hard coded in the process, therefore the example does not yet solve the problem of modeling a dynamic environment, where the set of Web services which can be used changes after the process which composes them has been written. To support this scenario we model the runtime discovery of the available services matching a certain criteria (for example, the name or the required input/output interface).

As shown in Figure 2 (right) the list of services to be invoked in parallel doesn't need to be hard coded, but can be parameterized based on the results of a query to a service registry.

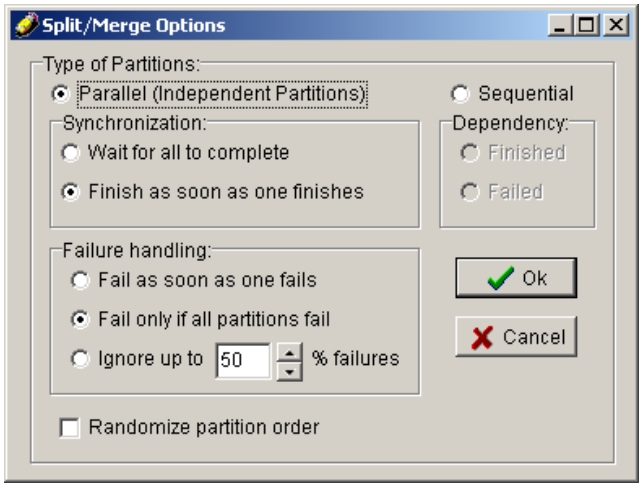


Figure 3: A screenshot of ADAPT showing the options which control the behavior of the Split/Merge operators. With them the developer can finely control if alternative services should be invoked in parallel, or sequentially. And, in the first case, if failures in invoking some of them should be ignored.

Sequential Invocation This solution, based on the parallel invocation of alternative services, is only applicable to stateless services we can afford to invoke in such manner. In some cases, for example when ordering a book, it may be necessary to try the various providers one at a time. To do so, we can select the *sequential* split operator (Figure 3) and still keep the previously described list-based patterns. The only difference being that the split operator is configured to invoke each service sequentially, and only if the previous one in the list fails.
Finally, by activating the `Randomize partition order` option, it is possible to conveniently shuffle the services in the list so that the sequence of invocation is not fixed to any particular order and the load at the various alternative providers is kept more balanced, as each execution of the process will attempt to invoke the service providers in a different order. This feature has been added for convenience only, as it is was already possible to explicitly add to the data flow a task which randomly reorders the list of services used to drive the parallel (or sequential) invocation.

4 Mapping between heterogeneous service interfaces

Both when searching for alternative implementations of a given service interface or when attempting to find a service which can directly receive an input request message produced by another service, it may be rather difficult to find exact matches. Instead, it is likely that some form of adaptation is required to make the two mismatching interfaces fit with each other.

4.1 Data representation mismatches

As summarized in [3], Web service heterogeneity can affect both the description of the service, where, for example, the name of semantically equivalent elements is different (or vice versa) as well as the data values that are exchanged: even if the interface's description matches, the actual values may differ because of the way they are encoded or because of their different semantic characterization [31].
To address this problem, a transformation of the data to be exchanged between the services is required. As the data is mostly encoded in XML format, the standard-based solution is to employ a style sheet transformation (XSL [32]). Although it is also possible to use this approach in ADAPT, some experience with this complex XML manipulation technology is required. As an alternative, the ADAPT Composition Language can also be used to solve both description and data value mismatches

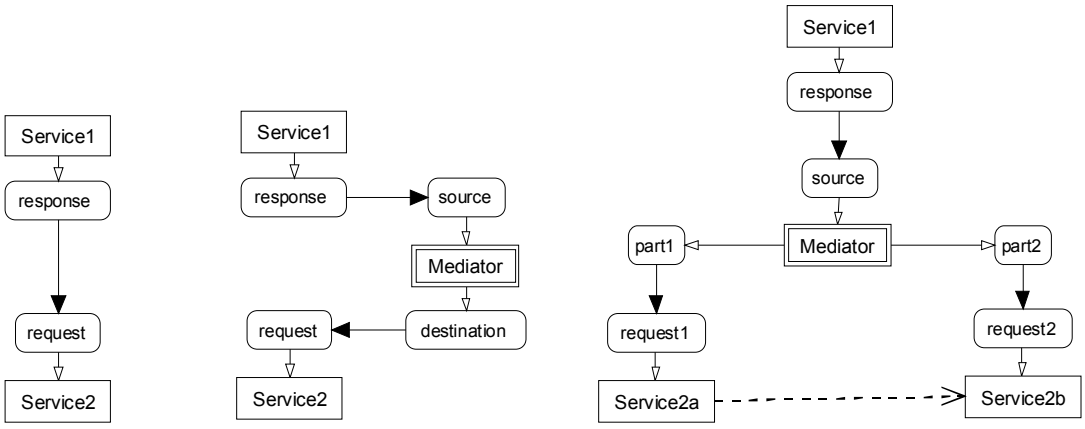
in a much more intuitive way by drawing mappings between the data models defined as part of the service interfaces.

4.2 Reusable mappings

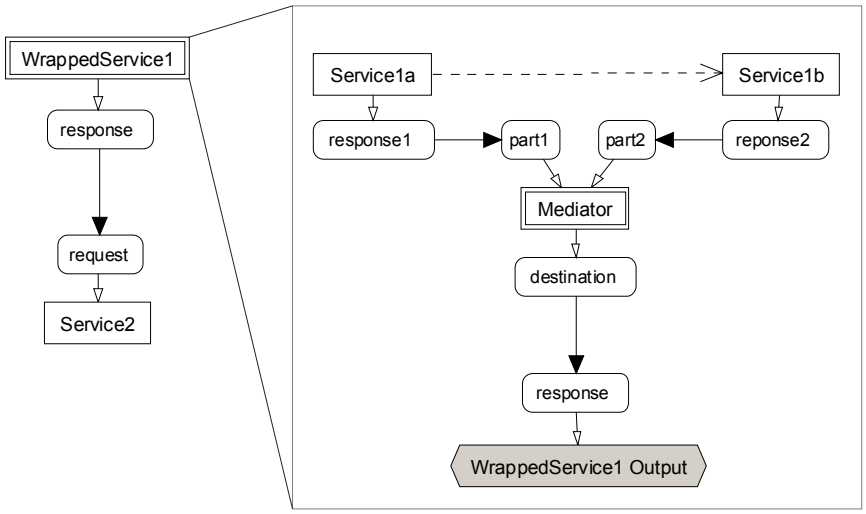
In practice, mappings between pairs of data types can be defined once and automatically reused. Whenever parameters of the corresponding types are connected the ADAPT visual development environment presents the user with a list of applicable mappings. The developer may choose to apply an existing mapping directly, or to customize one to fit with the specific pair of services. If no mapping can be found, ADAPT will let the user create a new one starting from the visual representation of the two interfaces, which is automatically generated from the service description. In the worst case, when fully interconnecting a set of *N* mismatching interfaces, the definition of $O(n^2)$ mappings may be required. However, as it is always possible to compose two or more mappings, by using a common interface based on a shared ontology, it is enough to specify *N* direct (and inverse) mappings between each service and the intermediate data representation [6].
Using processes to specify mappings can also help to solve general data integration problems, where results from many services need to be consolidated and sent to a single one, or data from a single service needs to be partitioned to be consumed by many separate services. With ADAPT's notation, it is possible to support general *N to M* mappings, where the integration of data coming from a set of *N* services to be aggregated and repartitioned to a set of *M* services can be designed visually.

4.3 Interaction-style mismatches

Mismatches in the service interfaces can also affect the interaction between the services. In this case, the granularity of the invocation may differ. For example, a large message returned by a service cannot be directly forwarded to another one. Instead, it has to be broken down into smaller messages, in order to fit with the interface of the latter. Similarly, a service may produce data to be retrieved in several stages, while such data is consumed by another one all at once. In general, although two services may feature a compatible semantic, their public process may differ, leading to an interaction-style mismatch [14].



(a) Perfect Fit (b) Data Mismatch (c) Interaction Style Mismatch



(d) A wrapper process is used to hide the interaction style mismatch

Figure 4: Example service interaction patterns showing how to use a mediator service to solve different types of interface mismatches

4.4 Mediation patterns

In the examples shown in Figure 4 different service interconnection patterns are represented to visualize how mediation services can be integrated into a process. In the examples the actual implementation of the mediator service is left unspecified. Depending on the required transformation, the most appropriate technique can be chosen. We will present a detailed example about various alternatives that can be employed to define such transformation in Section 5.
(a) No mediator is required, as the `response` of `Service1` can be directly forwarded to `Service2` as a `request`.
(b) A direct interconnection is not possible due to a mismatching data representation. Therefore a `Mediator` service is interposed between the two so that the data can be transformed from the `source` to the `destination` format.
(c) There is an interaction style mismatch because the interface of the two services have a different granularity. Thus, it is necessary to partition the `source` data originated from the `response` of `Service1` into different parts to be sent at different times to `Service2`. In addition to a data transformation, it is also necessary to correctly route the data produced by the mediator to the corresponding invocations of the second service. By including additional control flow dependencies (represented by a dashed arrow) it is possible to control the order of the interaction with the second service.
(d) A similar interaction style mismatch occurs. However, the data transformation and the interaction with the service having the smallest granularity are hidden inside a wrapper process, which produces a `response` which is compatible with the `request` expected by `Service2`. With this example, we applied ADAPT's nesting constructs to hide the adaptation inside a wrapper process, which can be reused in different contexts.

4.5 Operators

In the simplest case, for resolving conflicts in the description of equivalent data elements, mappings can be specified by directly drawing data flow connections between pairs of parameters which identify in a different way the semantically equivalent information. However, in some cases it may also be necessary to modify the values of the parameters. To do so, a mapping can contain different types of operators that are used to filter and transform the data in transit. ADAPT provides the Web service composer with a library of predefined XML manipulation, string handling, data conversion and constraint checking operators, which can be applied in any order between the source and destination

data elements of a visual mapping. In practice, the set of conversion operators can be easily extended with any user provided component which can be executed within a ADAPT process.

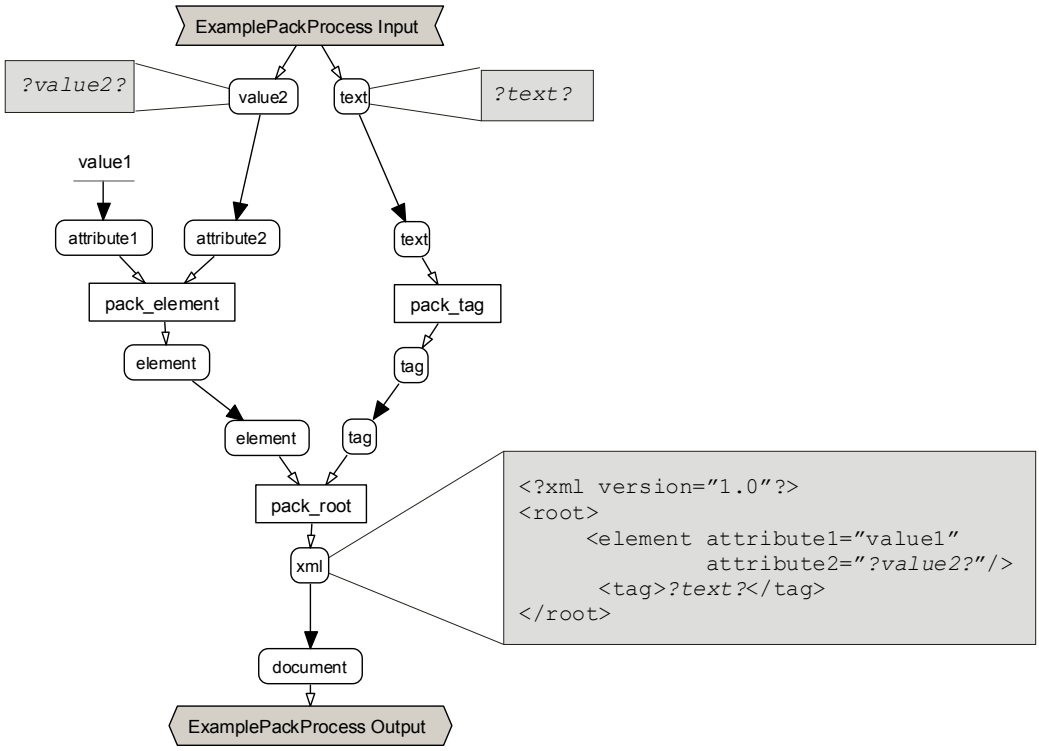


Figure 5: Example of using the XML manipulation operators to build an XML document.

XML manipulation First of all, for each XML data type found in a service's data model a pair of `Unpack`/`Pack` operators is generated. Unpacking operators take the XML serialization of a data type, i.e. an XML node in a document, and extract the values of its children elements into the operator's output parameters. Symmetrically, packing operators take multiple input parameters and encode their values into an XML document element. Considering that such operators are generated automatically when importing the definition of a service's data model, the input and output parameters of the operators contain the type information corresponding to the original XML schema [36] elements of the service's data model. Such type information can be used both to statically check that these operators are connected correctly and to guide the user in selecting the appropriate one.
When working with more complex XML data structures having nested XML data types, in order to access the innermost data elements, it is possible to compose several pairs of pack/unpack operators in a tree corresponding to the document's structure (Figure 5).

String handling The `Concatenate` operator joins two or more strings together. The `Split` operator cuts a string in two or more parts according to a given delimiter or character count. The `Map` operator translates a string into another one as specified by a user-provided lookup table. The `Replace` operator substitutes parts of a string with another, which can be identified using a regular expression.

Data conversion The `Scale` operator converts a numeric value to a different scale by multiplying or dividing it by a given factor. This can be useful for converting between different units of measure. The `Transform` operator applies any algebraic expression to its input values and returns the result. This gives a lot of flexibility in specifing a mapping, where arbitrary conversions of numeric data can be modeled.

Constraint checking These operators do not change the value of the data they receive, instead they can be added to the mapping in order to verify that the data fits to certain criterias, some of which,

e.g.,data types, restrictions, enumerations and minimum and maximum cardinality, can be extracted from the XML Schema contained in the interfaces, but others can be more semantic dependenent and may need to be manually included in the model. If the data fed into a checking operator does not fit with the specified criteria, then the execution of the whole mapping will fail with an exception.
If necessary, such operators can be tagged as *assertions*. Like in most programming languages, assertions can be used to verify the validity of some assumptions at runtime and, as opposed to normal conditional expressions, have the advantage that it is possible to instruct the compiler either to include or to ignore them. In ADAPT, they can be used to ensure that the data prepared to be sent to a certain service fits with what the service expects. In most scenarios, it can be an advantage to detect bad data early, rather than waiting for a service to reject it later, or worse, to produce inconsistent results. Similarly, it is often useful to ensure that the results returned by a service are compatible with the semantics of the services consuming them.

5 Example

In order to illustrate how to use the ADAPT Composition Language to represent data conversion operations between different service interfaces, we show a simple example about how to convert postal addresses between a service which returns them using a Swiss format to the US format understood by another service. The formats are defined by the XML Schema [36] snippets shown in Figures 6 and 7.

```
<xsd:complexType name="Adresse">
 <xsd:sequence>
  <xsd:element name="name" type="xsd:string" />
  <xsd:element name="vorname" type="xsd:string" />
  <xsd:element name="strasse" type="xsd:string" />
  <xsd:element name="plz" type="xsd:int" />
  <xsd:element name="ort" type="xsd:string" />
  <xsd:element name="kanton" type="xsd:string" />
 </xsd:sequence>
</xsd:complexType>
```

Figure 6: XML Schema definition for the `Adresse` type.

```
<xsd:complexType name="Address">
 <xsd:sequence>
  <xsd:element name="name" type="xsd:string" />
  <xsd:element name="street" type="xsd:string" />
  <xsd:element name="number" type="xsd:int" />
  <xsd:element name="town" type="xsd:string" />
  <xsd:element name="state" type="xsd:string" />
  <xsd:element name="zip" type="xsd:string" />
  <xsd:element name="country" type="xsd:string" />
 </xsd:sequence>
</xsd:complexType>
```

Figure 7: XML Schema definition for the `Address` type.

Both services have been built to manipulate postal addresses, both services use an XML Schema to define their data model. The information is even encoded in XML and transferred between the two services using the same SOAP protocol. Unfortunately the two services cannot be interconnected directly, because their data models are different. Unless a mapping between the two interface type definitions is designed and applied to the data in transit, the second service will reject the addresses received from the first one.

As we can observe from the two XML type definitions in Figures 6 and 7, the `Adresse` complex type does not match the `Address` type. Although some of its fields (elements) store equivalent information, e.g., the postal code, the fields are named differently (`plz` vs. `zip`). We also have a data

aggregation conflict in the way the person's name is stored: using two fields (`vorname`, `name`) to differentiate between first name and last name, and only one field (`name`).
In this case, the same field name (`name`) is used to tag incompatible information, if the fields with the same name would be considered to be matching, some information (the first name) would be lost. Furthermore, the first service returns Swiss addresses only, therefore there is no equivalent field to represent the `country` information, required by the second service. Finally, the information about the street is also represented differently, using one field (`strasse`) in the `Adresse` type and two fields (`street`, `number`).

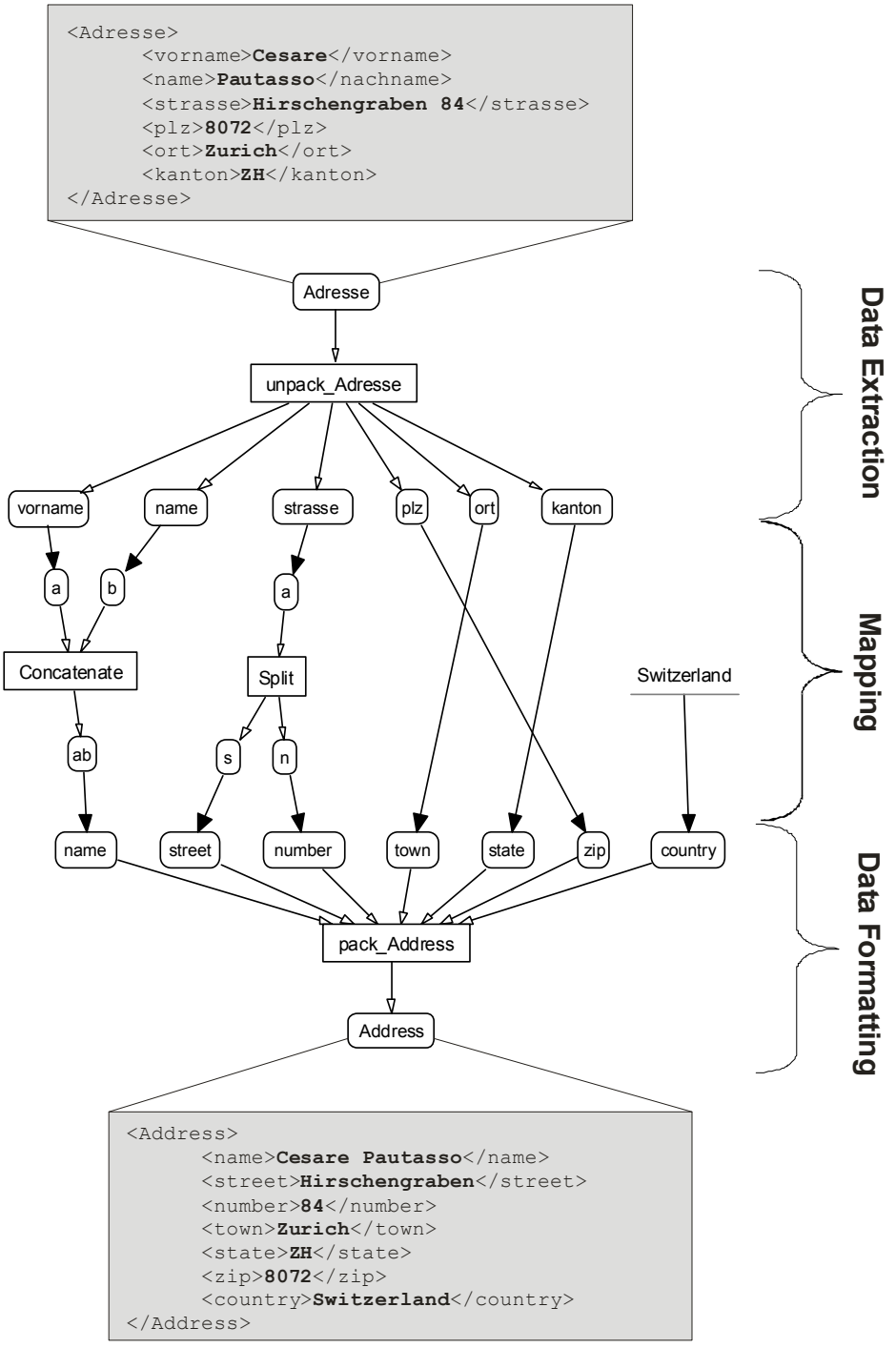


Figure 8: Example of a visual mapping between two XML complex types representing postal addresses: from a Swiss (top) to an American syntax (bottom).

The data flow diagram with ADAPT's solution is not too complicated (Figure 8) and all but the last incompatibility, concerning the `street` field, can be solved in an intuitive manner. The diagram can be followed from top to bottom. For some pairs of fields (e.g., `plz` to `zip`), where we have a mismatch only at the description level, we can directly link the equivalent fields with a data flow connection. The `country` field, for which an equivalent field is missing, can be bound to a constant value. The other fields have incompatible values, therefore it is not enough to redirect the values to the appropriate field. Instead, the values need to be manipulated using ADAPT's string handling operators, which can be used to concatenate the values of two fields (`name`, `vorname`) into one (`name`), as well as to split the value of one field (`strasse`) into two (`street` and `number`). Now, in general, determining which part of a string value corresponds to a street name and which part corresponds to a street number may be rather difficult. For this example, and for the corresponding XSL transformation (Figure 9) we assume that a street's number is always stored at the end of the string, following the Swiss convention, and that it is separated by the street name by a blank character.

The example of ADAPT's visual mapping of Figure 8 can be compared to the equivalent XSL transformation of Figure 9. In principle, it is possible to take ADAPT's visual representation and use it to generate the corresponding XSL code. For performance reasons we chose an alternative approach, which should maximize both the users' productivity (drawing a mapping can be faster than debugging XSL code) and the runtime execution's performance (ADAPT's compiler generates Java executable code from the visual notation), as we will discuss in the following section.

```
<xsl:template match="/Adresse">
 <Address>
  <name>
     <xsl:value-of select="vorname"/>
     <xsl:text> </xsl:text>
     <xsl:value-of select="name"/>
  </name>
  <street>
     <xsl:value-of select="substring-before(strasse,' ')"/>
  </street>
  <number>
     <xsl:value-of select="substring-after(strasse,' ')"/>
  </number>
  <town>
     <xsl:value-of select="ort"/>
  </town>
  <state>
     <xsl:value-of select="kanton"/>
  </state>
  <zip>
     <xsl:value-of select="plz"/>
  </zip>
  <country>
     <xsl:text>Switzerland</xsl:text>
  </country>
 </Address>
</xsl:template>
```

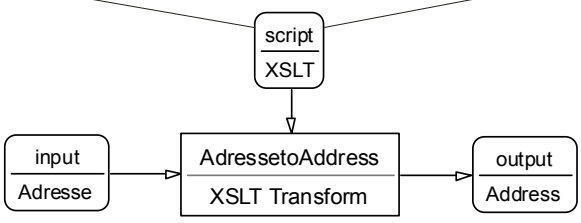


Figure 9: An alternative, equivalent representation based on XSL of the visual mapping in Figure 8 (top). The style sheet can also be invoked from an ADAPT process (bottom).

Performance, however, should not be the only criteria for comparing the two solutions. As we have previously discussed, in some cases, a graph-based notation can be automatically inverted. Furthermore, as indicated in Figure 8, it is possible to recognize in the visual mapping the following three stages:

1) The `unpack Adresse` operator *extracts* information from the original syntax of the source service.
2) A visual *mapping* based on data flow arrows and operators is used to define the transformation appied to such information.
3) The `pack Address` operator *formats* the results according to the syntax of the destination service's representation.

As discussed in [25], it is important to keep these aspects separated. This way, the semantic-level mapping can be specified in terms which are independent of the actual syntax-level formatting of the information. Since the XSL transformation of Figure 9 does not support such clear distinction, it remains more difficult to understand and maintain.

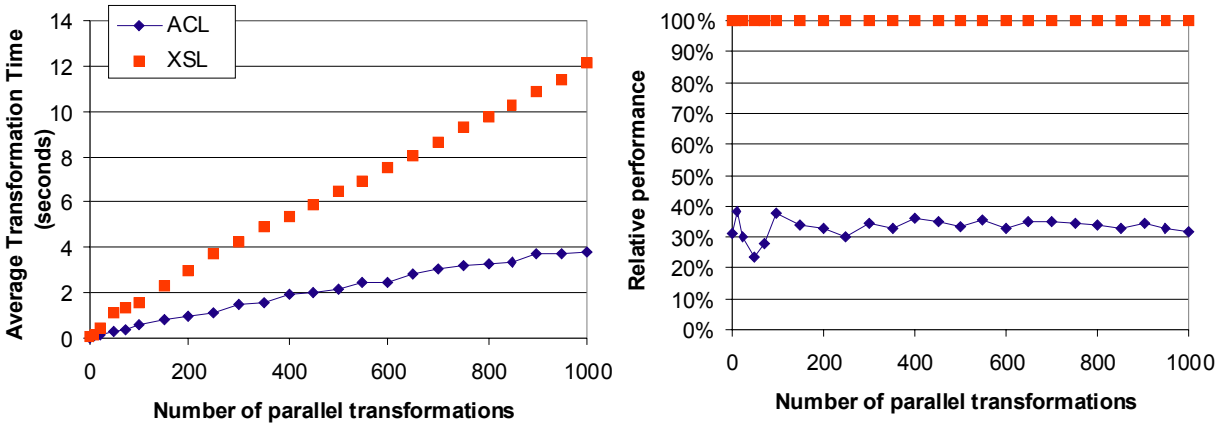


Figure 10: Absolute (left) and relative (right) performance of the ADAPT Composition Language (ACL) compared with an equivalent XSL transformation, for a variable number of transformations running in parallel.

6 Overhead

When measuring the performance of composite Web services, typically the results are bound by the time it takes to exchange SOAP messages across the Web. However, it is important to check that the transformations applied to these messages do not significantly increase this overhead. In this section we briefly present the results of a small experiment, whose goal was to compare the performance of the example mapping discussed in Section 5 implemented using the ADAPT Composition Language (Figure 8) with the performance of the equivalent XSL transformation (Figure 9) ran by the default XSL processor bundled with Java 1.4.1 Standard Edition. In both cases the Java virtual machine was running on a Linux v2.4.20 server with 4 XEON 1.5Ghz CPUs with hyperthreading enabled and 3.5GB of RAM. We chose this type of hardware environment in order to exploit ADAPT's multithreaded execution capabilities, as it was configured to use a pool of up to 64 parallel task execution threads. In order to minimize the effect of external factors, such as the time required to start a Java Virtual Machine, or the time necessary to load external files, all data was kept in main memory and the XSL processor was embedded inside ADAPT, as one of its execution subsystems, so that the

overhead of invoking it would be kept to a minimum and ADAPT's multithreaded execution would not represent an unfair advantage. All transformations were applied to the same input dataset (Figure 8 (top)) and produced the same output dataset (Figure 8 (bottom))
The results in Figure 10 (left) show the average transformation time as a function of the number of parallel transformations run concurrently by ADAPT (going from 1 to 1000). For the transformation written in XSL (Figure 9) the average time grows from 50 milliseconds up to 12.1 seconds per transformation. The equivalent transformation specified visually with the process of Figure 8 and compiled by ADAPT into Java scales better, since its average execution time grows from 15 milliseconds up to 3.8 seconds per transformation. As it can be seen from Figure 10 (right), the visual mapping outperforms the XSL transformation by about 35%.

6.1 Discussion

Our expectation was that the visual mapping, programmed in ADAPT using a process with four different tasks, each of which needs to be scheduled, dispatched and executed by the system, would incur in a much higher overhead when compared with the XSL transformation, which has also been implemented within ADAPT, but its process has only one task which directly calls the XSL processor as depicted in Figure 9 (right). Nevertheless, splitting up the transformation in three execution stages can better exploit ADAPT's multithreaded execution capabilities, which come into play as the execution of each mapping is pipelined through the system. The fact that the visual mapping is compiled into Java also helps to beat the performance of the XSL processor which, instead, interprets the transformation read from the style sheet.
Although these are just preliminary results, with a simple mapping applied to a small quantity of data, the results indicate that using a visual language in order to make it easier to program such transformations does not necessarily mean that the corresponding execution must be inefficient.

7 Related Work

Web service composition is a very active area, where many different projects and many systems are currently under development (e.g., [5, 8, 12, 15, 19, 37]). However, the problem of mediating between mismatching services has received very little attention [7].
More specifically, several process modeling languages tailored for Web service composition are competing to gain widespread acceptance: e.g., BPEL4WS and BPML. Concerning conflicting service interfaces, in BPEL4WS [19] no explicit provisions for the necessary data conversion are made [20]. However, it is still possible to apply X-Path queries [33] to the source and destination variables of an *assign* activity. Albeit with different syntax, also BPML [5] supports assignment activities which can be used in a similar way. Similar to the example of Figure 9, an X-Path processor is also part of the library of ADAPT's XML manipulation operators. In comparison with ADAPT's visual approach, the syntax of both of these process modeling languages is XML-based. Available "visual" tools [11] can be seen more like enhanced XML editors, as they closely follow the underlying XML representation. Furthermore, since these process modeling languages assume components based on Web services only, it is unclear how to explicitly model and apply data transformations (for example using a mapping written in XSL), unless, of course, a Web service implementing an XSL processor is employed, incurring in unnecessary overhead.
In order to solve conflicts between mismatching services, in [3] an approach based on database federation is proposed. In such a scenario, adapters based on Event-Condition-Action (ECA) rules triggering XSL transformations are inserted between the collaborating services at the communication layer. This solution has the advantage that it is inobtrusive, as the adaptation happens inside the SOAP engine. However, it requires appropriate personalization rules and XSL transformations to be developed and deployed inside all partners involved in the collaboration. The effort to do this should not be underestimated.
Many different domain-specific visual tools and languages applied to transformations of XML documents have been proposed. Mapforce [2] is a commercial data mapping tool for visual data

integration between heterogeneous XML and database sources, which can also generate Java, C#, C++ executable code and XSL transformations. In [30], the Visual XML Transformer (VXT) language has been introduced, advocating the suitability of visual programming techniques to simplify the specification of XML data transformations. In it, a set of Visual Pattern Matching Expressions are used to generate the corresponding XSL transformation. Likewise, in [40], a visual formalism has been applied to the definition of the structure of an XML document and, augmented with a graph rewriting mechanism, used to specify document transformations. Another form of data transformation is provided by visual query languages, which have also been applied to XML documents (e.g.,Xing [13], XML-GL [9]). Unlike ADAPT, where the same visual language is used both for modeling the composition of services as well as for specifying the necessary adaptations, these languages and tools are focused only on describing XML data transformations. Thus, they could be applied to resolving data representation mismatches, but it would be more difficult to use them to solve interactionstyle mismatches.
Mismatching service interfaces were already a problem in the pre-Web services era. In the context of Electronic Data Interchange (EDI) systems, a visual language and environment for EDI message translation has been presented in [16]. Similar to ADAPT, the mappings, which are compiled to a different representation for execution, can still be interactively debugged using the same visual notation.

8 Conclusion

Processes built by connecting heterogeneous Web services suffer from the independent evolution of their component services, whose availability, location and interface definition may change over time. Therefore, some form of adaptation is usually needed both when connecting together services which were not designed with compatible interfaces in the first place, as well as when fitting a set of services to a common interface definition, based on a shared ontology.
To address this problem, the usual solution is to define and apply style sheet transformations (XSL) to the data being exchanged. However, developing and maintaining code written in this XML-based language is difficult, error-prone, and time-consuming. As an alternative, ADAPT also supports the visual definition of process-based transformations, which can be stored in a library of reusable and composable mappings. Thanks to ADAPT's compiler, which produces Java executable code from the visual notation, our performance measurements indicate that these visual mappings can be applied to the data in transit between the mismatching services with minimal overhead.
All in all, the ADAPT system provides the user with the ability to easily compose processes out of mismatching services by resolving the conflicts between them with the same visual syntax used to compose them. This allows such service interfaces to keep evolving independently of one another, so that the processes that bind them do not impose an excessively tight coupling between them, invalidating the original Web services vision of building integrated systems out of loosely coupled service components.

References

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer, November 2003.
[2] Altova. *Mapforce 2004*. `http://www.altova.com/products/mapforce.html`.
[3] V. R. Aragao and A. A. Fernandes. Conflict Resolution in Web Service Federations. In *Proceedings of the International Conference on Web Services (ICWS-Europe 2003)*, pages 109– 122, Erfurt, Germany, 2003.
[4] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, pages 7–15, May 2001.
[5] BPMI. *BPML: Business Process Modeling Language 1.0*. Business Process Management Initiative, Match 2001. `http://www.bpmi.org`.
[6] C. Bussler. *B2B Integration. Concepts and Architecture*. Springer, 2002.

[7] C. Bussler. Semantic Web Services: Reflections on Web Service Mediation and Composition. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering (WISE 2003)*, pages 253–260, Roma, Italy, December 2003.
[8] F. Casati and M.-C. Shan. Dynamic and Adaptive composition of e-services. *Information Systems*, 26:143–163, 2001.
[9] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tanca. XML-GL: a Graphical Language for Querying and Restructuring XML Documents. In *Proceedings of the 8th International World Wide Web Conference*, Toronto, Canada, 1999.
[10] J.-Y. Chung, K.-J. Lin, and R. G. Mathieu. Web Services Computing–Advancing Software Interoperability. *COMPUTER*, 36(10):35–37, October 2003.
[11] Collaxa. BPEL Server and Designer. `http://www.collaxa.com`.
[12] ebXML. *ebXML Business Process Specification Schema (BPSS) 1.01*, 2001. `http://www.ebxml.org/specs/ebBPSS.pdf`.
[13] M. Erwig. Xing: a visual XML query language. *Journal of Visual Languages and Computing*, 14(1):5–45, February 2003.
[14] D. Fensel and C. Bussler. The Web Service Modeling Framework WSMF. *Electronic Commerce Research and Applications*, 1(2):113–137, Summer 2002.
[15] D. Florescu, A. Gruenhagen, and D. Kossmann. XL: A Platform for Web Services. In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR 2003)*, Asilomar, CA, USA, 2003.
[16] J. C. Grundy, R. Mugridge, J. G. Hosking, and P. Kendall. A Visual Language and Environment for EDI Message Translation. In *Proceedings of the 2001 IEEE International Symposium on Human-Centric Computing Languages and Environments (HCC 2001)*, pages 330–331, Stresa, Italy, 2001.
[17] IBM. BPEL4WS Java Runtime. `http://www.alphaworks.ibm.com/tech/bpws4j`.
[18] IBM and Apache Foundation. *Web Service Invocation Framework (WSIF)*, 2003. `http://ws.apache.org/wsif/`.
[19] IBM, Microsoft, and BEA Systems. *Business Process Execution Language for Web Services (BPEL4WS) 1.0*, August 2002. `http://www.ibm.com/developerworks/library/wsbpel`.
[20] S. Iyengar. Business process integration using UML and BPEL4WS. In M. Glinz and H.- P. Hoidn, editors, *Components: the Future of Software Engineering? (SI-SE 2004)*, Zurich, Switzerland, March 2004.
[21] F. Leymann. Web Services: Distributed Applications without Limits. In *Proceedings of the International Conference on Business Process Management (BPM 2003)*, pages 123–145, Eindhoven, The Netherlands, 2003.
[22] F. Leymann, D. Roller, and M.-T. Schmidt. Web services and business process management. *IBM Systems Journal*, 41(2):198–211, 2002.
[23] M. D. McIlroy. Mass-produced Software Components. In *Proceedings of the Working Conference on Software Engineering*, pages 138–150, Garmisch-Partenkirchen, Germany, 1968.
[24] C. Mohan. Dynamic E-business: Trends in Web Services. In *Proceedings of the Third International Workshop on Technologies for E-Services (TES 2002)*, pages 1–5, Hong Kong, China, 2002.
[25] B. Omelayenko and D. Fensel. A Two-Layered Integration Approach for Product Information in B2B E-commerce, . In K. Bauknecht, S.-K. Madria, and G. P. (eds.), editors, *Proceedings of the Second International Conference on Electronic Commerce and Web Technologies (EC WEB-2001)*, volume 2115 of *LNCS*, pages 226–239, Munich, Germany, September 2001.
[26] M. P. Papazoglou and D. Georgakopoulos. Service-Oriented Computing. *Communications of the ACM*, 46(10):25–28, October 2003.
[27] C. Pautasso. JOpera: Process Support for Web Services. `http://www.iks.ethz.ch/jopera/download`.
[28] C. Pautasso and G. Alonso. JOpera: a Toolkit for Efficient Visual Composition of Web Services . Technical Report TR 432, ETH Zurich - Department of Computer Science, December 2003.
[29] C. Pautasso and G. Alonso. Visual Composition of Web Services. In *Proceedings of the 2003 IEEE International Symposium on Human-Centric Computing Languages and Environments (HCC 2003)*, pages 92–99, Auckland, New Zealand, 2003.
[30] E. Pietriga and J.-Y. Vion-Dury. VXT: Visual XML Transformer. In *Proceedings of the 2001 IEEE International Symposium on Human-Centric Computing Languages and Environments (HCC 2001)*, pages 404–405, Stresa, Italy, 2001.
[31] A. P. Sheth. Changing Focus on Interoperability in Information Systems: From System, Syntax, Structure to Semantics. In M. F. Goodchild, M. J. Egenhofer, R. Fegeas, and C. A. Kottman, editors, *Interoperating Geographic Information Systems*, pages 5–30. Kluwer, Academic Publishers, 1998.
[32] W3C. *Extensible Stylesheet Language Transformations (XSLT) 1.0*, 1999. `http://www.w3.org/TR/xslt`.
[33] W3C. *XML Path Language (XPath) 1.0*, 1999. `http://www.w3.org/TR/xpath`.

[34] W3C. *Simple Object Access Protocol (SOAP) 1.1*, 2000. `http://www.w3.org/TR/SOAP`.
[35] W3C. *Web Services Definition Language (WSDL) 1.1*, 2001. `http://www.w3.org/TR/wsdl`.
[36] W3C. XML Schema, 2001. `http://www.w3.org/TR/xmlschema-0/`.
[37] R. Weber, C. Schuler, H. Schuldt, H.-J. Schek, and P. Neukomm. Web Service Composition with O'GRAPE and OSIRIS. In *Proceedings of 29th International Conference on Very Large Data Bases (VLDB 2003)*, pages 1081–1084, Berlin, Germany, 2003.
[38] G. Wiederhold, P. Wegner, and S. Ceri. Towards Megaprogramming: A Paradigm for Component-Based Programming. *Communications of the ACM*, 35(11):89–99, 1992.
[39] U. Zdun, M. Voelter, and M. Kircher. Design and Implementation of an Asynchronous Invocation Framework for Web Services. In *Proceedings of the International Conference on Web Services (ICWS-Europe 2003)*, pages 64–78, Erfurt, Germany, 2003.
[40] K. Zhang, D.-Q. Zhang, and Y. Deng. A Visual Approach to XML Document Design and Transformation. In *Proceedings of the 2001 IEEE International Symposium on Human-Centric Computing Languages and Environments (HCC 2001)*, pages 312–319, Stresa, Italy, 2001.