Model-Driven Generation of Dynamic Adapters for Integration Platforms

Matthias Böhm[1], Jürgen Bittner[2], Dirk Habich[3], Wolfgang Lehner[3], and Uwe Wloka[1]

[1] Dresden University of Applied Sciences, Database Group
mboehm@informatik.htw-dresden.de
wloka@informatik.htw-dresden.de
[2] SQL Gesellschaft für Datenverarbeitung mbH Dresden
juergen.bittner@sql-gmbh.de
[3] Dresden University of Technology, Database Technology Group
dirk.habich@inf.tu-dresden.de
wolfgang.lehner@inf.tu-dresden.de

Abstract. The concept of Enterprise Application Integration (EAI) is widely used for integrating heterogeneous applications and systems via message-based communication. Typically, EAI servers provide a huge set of specific inbound and outbound adapters used for interacting with the external systems and for converting proprietary message formats. However, one major shortcoming in currently available products is the monolithic design of those adapters, resulting in performance deficits caused by the need for data independence. Further, also functional restrictions must be noticed here. In this paper, we give (1) a detailed problem characterization, followed by (2) a discussion of alternative data representations and adapter architectures, and (3) we introduce our model-driven DIEFOS (data independence, efficiency and functional flexibility using feature-oriented software engineering) approach. Finally, we analyze open research challenges.

Key words: Enterprise Integration Platform, Application Integration, Adapter Architecture, Dynamic Adapters, DIEFOS Approach

1 Introduction

The trend towards heterogeneous environments comes with an increase in importance of Enterprise Application Integration (EAI). Such an integration platform consists of a set of inbound adapters, a core message broker, and a set of outbound adapters. The large number of supported external system types results in the need for data independence (independent-system-type data representations for internal processing) and, simultaneously, for efficient integration task processing (minimum overhead for data independence). These requirements—but particularly the first one—typically result in very generic inbound and outbound adapter architectures. Therefore, the architecture of such adapters is quite monolithic, resulting in low functional flexibility of such software components. This means that for each external system type, a single adapter is required, although specific functional modules could be reused. For example, a TCP connection handler that sends specific messages to physical target systems might be reused by several adapters, such as *HL7* and *B2MML* adapters.

To tackle this problem, we developed the model-driven *DIEFOS* (data independence, efficiency and functional flexibility using feature-oriented software engineering) approach. Aside from affecting the functionality, our DIEFOS approach also influences the performance of integration processes. In contrast to the short introduction of the basic idea of DIEFOS in [1], we use this paper to formally define the problem and then discuss the adapter characteristics from a pragmatic perspective in detail. Furthermore, we distinguish the main alternatives from the perspective of data representations. Moreover, we propose different alternatives from the perspective of adapter architectures. Finally, we present our model-driven DIEFOS approach and explain the core concept phases. Fundamentally, the whole concept relies on the model-driven development of adapter components for integration platforms and is influenced by the commercial integration platform *TransConnect*® of the company *SQL GmbH*.

The paper is organized as follows: Section 2 gives an overview on the adapter problem characteristic. Based on this problem description, Section 3 and Section 4 distinguish applicable approaches for data representations as well as adapter architectures. After that, Section 5 includes the core DIEFOS approach description, starting with an overview of the complete process, followed by explanations of the individual steps as well as examples and the resulting benefits of this approach. Furthermore, we highlight open problems as well as research challenges. Finally, in Section 6, we outline related work—even though the body of related work does not focus on the exact same issue addressed by our approach. We conclude our paper in Section 7.

2 Adapter Problem Characteristic

In this section, we introduce a generalized EAI server architecture and formally specify the addressed problem as a dependency problem to be solved. Subsequently, resulting problems of typical products are discussed. Finally, we classify types of adapters using the layers of transformations defined in [2].

As illustrated in Figure 1, an EAI server consists of several typical components. There is a set of `Inbound Adapters` which listen passively to incoming messages and convert those to internal representations. These internal messages are either distributed to `Message Queues` (asynchronously) or directly sent to the core message broker (synchronously), which is typically a `Process Engine`. This engine uses a set of `Outbound Adapters` to actively interact with external systems. During the whole integration process, the

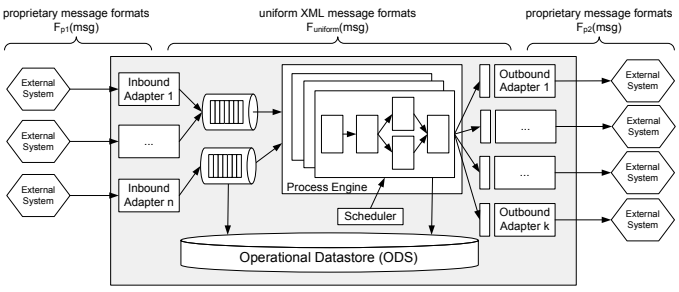

Fig. 1. Generalized EAI Server Architecture

recoverability must be ensured; thus, the internal message representations have to be stored locally using the `Operational Datastore`.

In general, the main problems result from the monolithic adapter architecture and can be stressed by the following observations. Our first observation is that an adapter interaction typically requires an input XML message and optionally produces a resulting output XML message. This architecture is used in order to provide very generic adapter interfaces. Thereby, data and meta data are mixed within one document. The lack of meta data separation results in the fact that each adapter A_x has a defined input schema $S_{in}(A_x)$ and a defined output schema $S_{out}(A_x)$. In fact, there is the dependency problem $A_x(msg) \rightarrow F_{uniform}(msg) : A_x$, where $F_{uniform}(msg)$ is the generic representation of the required input or output data following specific schemas of an adapter A_x. The second observation is that huge sets of adapters are developed independently from each other. However, this is not a suitable solution from a functional flexibility perspective. For instance, multiple adapters might use TCP as transport protocol and could thus reuse this functionality for different adapter types. Based on this description, we distinguish the following four pragmatical problems:

P1 - Poor Performance: The adapter-type-specific message format $F_{uniform}(msg) : A_x$ requires useless transformation processing within the core broker. For example, if we integrate the ERP System SAP with an RDBMS, the received SAP IDocs first have to be transformed into the $F_{uniform,out}(msg) : A_{sap}$ by the inbound adapter. Second, $F_{uniform,out}(msg) : A_{sap}$ have to be transformed into the $F_{uniform,in}(msg) : A_{jdbc}$, for example, by an XSL transformation. Finally, the $F_{uniform,in}(msg) : A_{jdbc}$ are transformed into tuple arrays and bulk-loaded into the RDBMS. Obviously, the optimal solution with regard to performance would be to transform the SAP IDocs directly into the mentioned tuple arrays. However, if this is hard-coded, this will result in the loss of data-independent processing within the core message broker.

P2 - Functional Restrictions: According to [2], all transformations of the layers *transport (L0)* and *data representation (L1)*—which are lossless transformations—should be optionally handled by all types of adapters. Imagine issues like compression and encryption. These transformations should be applicable in a uniform manner.

P3 - Development Effort: Due to the fact of functional re-usability, a layered approach seems to be advantageous. However, the characteristics of specific protocols (e.g., HL7) prevent a generic layered solution. The architecture of monolithic adapters also requires large efforts for testing because all functionalities have to be tested for all types of adapters. The high effort is further caused by redundant implementations.

P4 - Data Independence: Due to the adapters being used as logical services within the core message broker, the schemas of internal messages must be independent from the proprietary data formats (*P4: Data Independence*). This addresses the layers *schema* as well as *data representation*. Typically, EAI servers use a uniform data representation but no uniform data schema. A separation between data and meta data will bridge this gap. However, especially with efficiency in mind, the right level of data independence has to be determined (as generic as necessary, as efficient as possible).

We classify the four different problems into two specific adapter perspectives: *Data Representation* and *Adapter Architectures*. The former comprises the problems *P1* and

P4. The aim is to find the best compromise between efficiency and data independence. The latter perspective comprises *P2* and *P3*. Here, the objective is to find a solution that achieves functional flexibility with low development effort. We will present several alternative approaches for both perspectives in the following sections.

3 Perspective A - Data Representations

For this perspective, we are able to distinguish between the different approaches of internal data representations (message models) and consider the optimal data interface to the specific adapters. In general, the following three types of internal data representations can be separated.

- *Document-oriented*: Here, each message is represented by exactly one XML document. The advantage is the simple persistence as CLOB/BLOB attribute, while the disadvantage is the deep structure and the resulting high costs for single attribute accesses.
- *Attribute-oriented, coarse-grained*: Such a message consists of a list of atomic or structured attributes in the form of name/value pairs. The advantage is higher flexibility and efficiency. However, there is a high data dependence on the specific attribute types, which does not seem to be adequate.
- *Attribute-oriented, fine-grained*: Using this approach, messages are separated into their atomic attributes. Thus, the access to single values is very efficient and simple. The main problem with this approach lies in the high costs for fine-grained data persistence and in the fact that standardized XML technologies (e.g., XSL transformations) cannot be applied anymore.

Due to the high flexibility and the best overall performance, the *attribute-oriented, coarse-grained* approach is most suitable. However, in order to reach the necessary level of data independence, this approach must be modified. So, we define that all attributes are either of type BLOB (this also comprises tuple arrays) or of type XML. Based on this message model, we now consider applicable adapter interfaces, which is actually a side effect of the perspective *adapter architecture*.

- *Adapter-specific schema*: The simplest form of an adapter interface is a specific input and output schema for each adapter. Regardless of whether a message-based or parameter-based approach is used, this is not sufficient with respect to the necessary data independence. However, the advantage of this is the lightweight uniform representation of the data.
- *Universal schema*: This interface has a message-based universal schema interface. Here, the universal schema causes some overhead that influences the persistence and the internal processing. However, the universal schema (with separated data and meta data) allows for direct message exchange between adapters without the need for message transformations, and thus, it results in higher efficiency.

Finally, we propose an *attribute-oriented, coarse-grained* data representation used for *schema-independent* data transfer between adapters, where data and meta data are separated as well.

4 Perspective B - Adapter Architectures

In order to achieve functional flexibility—with the lowest possible development effort—this section presents alternative adapter architectures. Basically, the three types (1) monolithic, (2) generic and (3) generated can be distinguished. The monolithic adapter architecture is not adequate and therefore, its description is omitted here. The generic adapter (Subsection 4.1) is explained in detail because it could be applied in a suitable way. Furthermore, we outline in short how a generation approach (Subsection 4.2) might be followed as well.

4.1 Generic Adapter

The first alternative to the monolithic approach is the generic adapter architecture, where an adapter (which should be applicable for inbound and outbound purposes) allows for easy extensibility concerning different formats and protocols. Figure 2 illustrates the micro-architecture of such a generic adapter. Basically, such an adapter type is composed of a `Connector`—implementing the CCI (Common Client Interface)—which realizes the physical interoperability with external systems or APIs. Further, a specific `Protocol Handler` might be plugged into the connector but this is optional. Such a handler manages application layer protocols and is used—in a unique manner—by the `Connector`. Another extension option is to use an arbitrary number of so-called `Format Converters` in the form of a format stack. These converters are also used by the `Connector`. As a result, the adapter type components can be divided into three main parts: (1) the *connectivity*, (2) the *format transformation* (format converter and protocol handler) and (3) the *adapter-type properties*. An adapter instance of such a generic adapter type can be created providing a specification of the connector to be used, the needed protocol handler and a set of format converters. Further, the adapter type properties as well as the properties of all plugged-in function modules have to be specified. Such created instances are maintained by a specific repository manager.

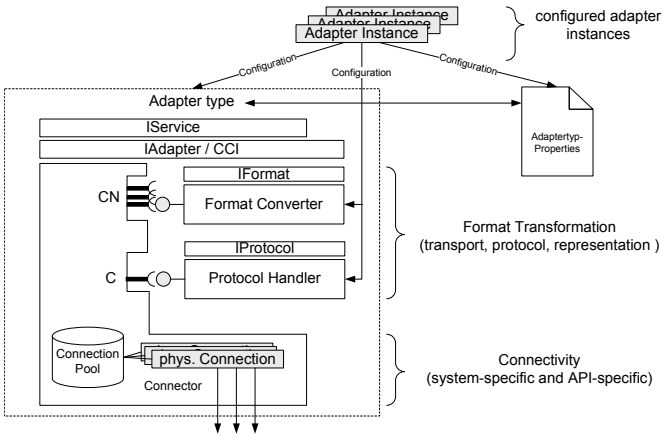

Fig. 2. Generic Adapter Micro Architecture

In order to separate the different connectors, we classify them into three layers. First, there are simple *transport connectors* like file systems, TCP and UDP. Second, there are higher-level *protocol connectors* like HTTP, SMTP, POP3, and SNMP. These

mentioned connectors encapsulate transport protocols and thus, they also must be seen as connectors. Third, there are the *system connectors*, like SAP and LotusNotes, which handle specific system libraries or standardized APIs. Based on these three connector classes, we are able to define the class of protocol handlers for FTP, HL7 MLLP, SWIFT, DICPOM, SMTP, POP3, IMAP, JMS, Elster and similar application logic protocols. Further, let us consider possible format converters. Here, conversions like Zip/Unzip, Encrypt/Decrypt, XML/HL7, XML/ASCII, XML/PDF, XML/EDI, XML/I-Doc, XML/Binary, XML/SWIFT and many others are imaginable. The defined classification lets us dynamically build stacks of functional modules. An example for that is the adapter type $A_{elster1}$, which is composed of the HTTP connector, the Elster protocol handler as well as the three format converters Encrypt/Decrypt, Zip/Unzip and Elster/XML. Note that we can easily change $A_{elster1}$ to $A_{elster2}$ by using the TCP connector. So, the functional flexibility can be achieved with low development effort.

From this perspective, the generic adapter seems to be suitable. However, there are problems with this approach. Fundamentally, this concept requires knowledge of applicable protocols and format converters. Although this is a minor problem, it stands in contrast to the tendency towards self-managing systems. In any case, the major problems are specific application protocols like HL7. So, the binary encoding of HL7 2.x requires format conversions from HL7-binary to XML. The problem is that right after the message has been converted, the connector can determine whether or not an HL7 acknowledgment has to be sent. As a result, unfortunately, a clean separation of connections, protocol handlers and format converters—used within a generic adapter architecture—does not seem to be applicable for all combinations of external systems and formats.

4.2 Adapter Generation

So, if we assume that the monolithic as well as the generic adapter approach are both not suitable to meet the requirement of full functional flexibility, we conclude that only the generative approach following the model-driven architecture (MDA) paradigm can be applied in this context. In this case, the starting point is a textual specification (informal) of the requirement for a specific adapter type—the so-called computer-independent model (CIM). This model is manually transformed into the platform-independent model (PIM). Here, a specification of adapter type functionalities (formal) is required. After that, the PIM is automatically transformed into the platform-specific model (PSM), using available platform models (PM), which are in fact kinds of meta models. Finally, the PSMs are transformed into executable code by using provided code templates.

Basically, there are two types of generation forms. First, we can use function modules (as generalized as possible) and only generate a kind of specific orchestration module that links and uses the existing (and implemented) function modules. This will result in the use of well tested components, while there is a minor performance overhead. Second, we can generate a whole specific adapter, which definitely results in much better performance. However, here, the problem of testing the generator and its output is highly significant. Within the DIEFOS approach, we use the first generation type, which will be explained in more detail in Subsection 5.2.

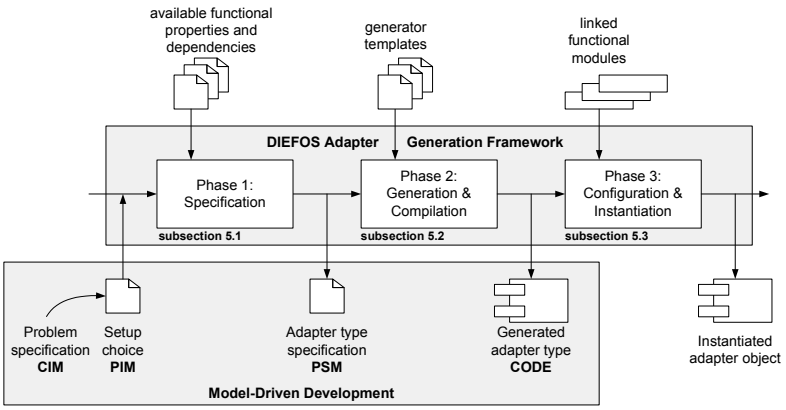

Fig. 3. DIEFOS Generator Framework - Macro-Architecture

5 DIEFOS Approach

In this section, we introduce our DIEFOS approach (**D**ata **I**ndependence, **E**fficiency and functional flexibility using a **F**eature-**O**riented **S**oftware-development approach) to solve the described problems from Section 2 using the considerations from Sections 3 and 4. Figure 3 illustrates the macro-architecture of our developed framework. However, this is a work in progress because large efforts are needed in order to provide a sophisticated and robust framework for this approach.

Basically, our framework comprises the three phases *1: Specification*, *2: Generation & Compilation* and *3: Configuration & Instantiation*. In the first phase, the CIM is manually transformed into a setup choice—the applicable alternatives are given by feature diagrams similar to Figure 3—representing the PIM. This setup choice, in conjunction with available functional properties and dependencies, is used to create the formal requirement specification (PSM). Within the generation step of the second phase, a Java class is generated from the adapter type specification input, using specified code templates. Finally, this class is compiled and loaded into the JVM. Within the third phase, the created instance of the generated adapter as well as the linked functional modules have to be configured. Obviously, we use an approach where specific functionality can be reused in function modules almost without any overhead. So, during runtime, these hard-coded modules are used as a library. In the following three subsections, we explain these three phases in very detail.

5.1 Specification

As already mentioned, the specification phase mainly has two inputs: the adapter type setup choice and a set of available functional properties and dependencies in the form of simple ECA (event, condition, action) rules. The functional properties are given by the implemented functional modules as illustrated in Figure 4, where a feature diagram

is used to illustrate the adapter specification opportunities. Imagine the setup choice for a concrete adapter type: one might think of a graphical user interface similar to a simple installation setup screen, where subcomponents can be selected or deselected when needed. The setup choice, further, is evaluated with the given dependency rules. So, if a specific component is selected (event), all related conditions are checked. If one of these evaluates to true, a specific error (action) has to be signaled. Some simple examples of such rules are the following ones (Listing 1.1):

```
Rule1 := {Connector==*; count(Connector)>1;
          MultiplicityERROR}
Rule2 := {Connector==Basic.File; AdapterType != Outbound;
          FunctionalERROR}
Rule3 := {ProtocolHandler==HL7 MLLP;
          FormatConverter !contains(XML/HL7); FormatERROR}
```

Listing 1.1. Example Dependency Rules

As we use the feature diagram to visualize the adapter type's functional properties, we want to survey its meta model in short. So, a component is represented by a hierarchy of features. If a feature is optional, this is marked with a white circle; otherwise it is marked with a black circle. Further, a white angle represents an *exclusive or* choice, while a black angle means that all sub-features are used. If a feature is represented by a gray rectangle, this means that a sub-feature diagram for this sub-hierarchy exists.

Based on this simple meta model, we define the model for integration platform adapter types. One specific adapter type has several specific process models and one connector as well as one optional protocol handler and several format converters. There, a subset of the process models `Inbound.ClientInterface`, `Inbound.Async`, `Inbound.Sync` and `Outbound` can be used. We omit more detailed explanations of

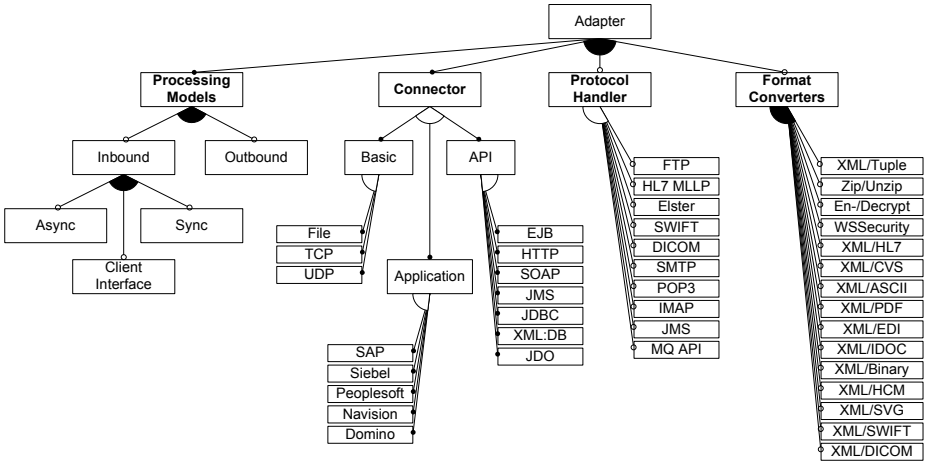

Fig. 4. Adapter Type Specification Feature Diagram

connectors, protocol handlers and format converters because they are given in Figure 4 already.

As a result of the specification phase, a formal requirement specification is created. Here, we use an XML representation of the defined integration platform adapter type model, flagged with choice parameters. Note that the produced specification is semantically correct if phase 1 has finished successfully. This is an important assumption in order to ensure robustness for the second phase of generation & compilation.

5.2 Generation and Compilation

For generation, we use the approach of function modules and only generate some kinds of specific orchestration modules that use the implemented function modules. However, even these orchestration modules have a high complexity, dealing with completely different protocols and system connectors. We can state that all `Format Converters` and specific parts of `Connectors` can be implemented as function modules. Only a marginal performance overhead has to be accepted when doing so.

The generation addresses the transformation of the produced specification of phase 1 into executable code. So, an orchestration Java class is generated, which contains the connector usage, protocol handling (if specified) and the usage of format converters, always obeying the specific configuration. Therefore, we use text templates with placeholders for atomic parameters and complex substructures. For each specific pattern, such templates are defined. Basically, we iterate through the XML specification, extracting the necessary information and replacing the defined text templates. Finally, the generated class is physically stored, compiled and loaded into the JVM. For compilation purposes, we use the Java Compiler API (JSR 199) in order to prevent the initialization of a second JVM process. Due to the fact that each generated adapter type implements the CCI (Common Client Interface), each generated adapter type can be dynamically loaded and used by the process engine in a unique manner. Restricted by the lack of space, we have to omit further details about the generation process.

5.3 Configuration and Instantiation

With the intent to use the CCI, we typically configure the generated adapter type with so-called property objects during instantiation. Thereby, a concrete adapter is created from the adapter type. Note that static configuration properties are directly used during generation, while others are dynamically provided during the phase of instantiation. In order to distinguish these two types of configuration properties, we introduce the *adapter type properties* and the *adapter properties*.

The *adapter type properties*, on the one side, specify which dynamic parameters a generated adapter should contain on the meta level. On the other side, also all configuration values that are not intended to be changed are included. Note that this assumption results in the fact that all adapter type properties are statically generated into the adapter type and cannot be changed during runtime.

In contrast to that, the *adapter properties* enable us to dynamically influence the behavior of the generated adapter. Here, two types of properties have to be distinguished as well. First, there are the properties that are specified by the adapter type properties.

Thus, a functional dependency between the *adapter type properties* and the *adapter properties* has to be noticed. Second, there are the properties given by the used function modules. These are determined by the implementation of the function modules and thus can only be affected by the setup choice regarding the used features.

All generated adapter types and configured adapters of these types are managed within a central repository. Here, the maintenance and notification of changed properties during development time and runtime is also realized. In accordance with the specific characteristics of the two types of properties, a property change request can result in two different actions. First, if the change affects an *adapter type property*, implicitly, phases 2 and 3 of the DIEFOS approach have to be reapplied in order to generate a new adapter type with the specific static property. Second, in case of an *adapter property*, only phase 3 has to be reapplied because only dynamic parameters are affected. Finally, note that the *Configuration History* might pose an open problem in this area.

5.4 Example Adapter Generation

In order to make our approach easier to understand, in this subsection, we illustrate a sample adapter type generation for a very simple adapter. Basically, we follow the three phases as described as our overall DIEFOS approach. The informal requirement is to define an outbound adapter that allows for interacting with the file system, thereby realizing zip compression as well as PGP encryption.

The result of phase 1 (specification) is an XML representation of the adapter type model. Here, Listing 1.2 shows the specification including the given adapter type properties for the format conversion step ”encrypt, decrypt”. Note that the sequence is of high importance. So, we specify to first compress and then encrypt the data before writing it to the file system. Further, for example, the format conversion step type ”zip, unzip” specifies that when writing files (push), they are zipped, while read files (pull) have to be unzipped.

```
<adaptertype name="file1">
  <type inbound="no" outbound="yes"/>
  <connector type="file"/>
  <protocol type="none"/>
  <format type="composed">
    <formatstep type="zip,unzip"/>
    <formatstep type="encrypt,decrypt"/>
      <property name="algorithm" value ="aes256"/>
      <property name="publickey" value ="~/keys/public"/>
      <property name="privatekey" value ="~/keys/private"/>
    </formatstep>
  </format>
</adaptertype>
```

Listing 1.2. Example Adapter Type Specification (XML Representation)

Based on the defined adapter type model, in phase 2 (generation & compilation), the orchestration class, illustrated in Figure 5, is generated. Here, the connector type `file` results in the two adapter properties `path` and `pickup mask`. Further, obviously this

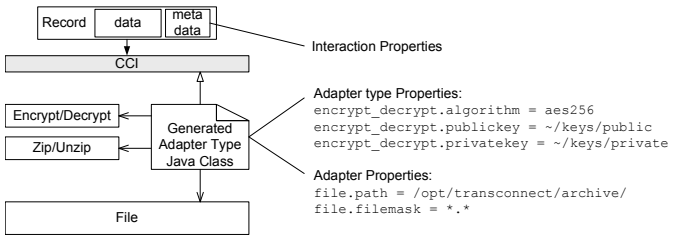

Fig. 5. Example Generated Adapter Type

generated class implements the CCI, and so, we are able to interact with this adapter using records for data and meta data separation.

Finally, the generated class has to be parameterized with concrete values for the adapter properties and then loaded into the JVM. As this simple example shows, our approach allows a very high functional flexibility, realized with marginal overhead for data independence, so that efficient processing is possible. At this place, we explicitly omit performance experiments due to the fact that they are highly dependent on the chosen integration process, the adapter constellations and further workload characteristics.

5.5 Benefits and Approach Validation

In order to validate our approach, we point out the major benefits of the DIEFOS approach. Here, we observe in detail the results according to the defined four problems (see Section 2). The first benefit reached is functional flexibility, which tackles problem *P2: Functional Restrictions*. Due to the separation into elementary format converters, protocol handlers and connectors, adapters can easily be created using a specific combination of those. While the functional restrictions were given by monolithic adapters with specific functionalities, adapters can now be flexibly composed. Thus, this combination possibility results in a wide range of concrete applicable adapters. Hence, functional restrictions are prevented.

Another benefit is the reduced development effort, addressing *P3*. On the one side (server development), the low development effort is reached by the already mentioned functional separation. Thus, the redundant maintenance of monolithic adapter code is prevented reusing clearly distinguished function modules. On the other side (integration process modeling), the low development effort is also caused by the model-driven development. Here, the dynamic adapters are simply modeled by feature selections rather than being reimplemented.

The third benefit of the DIEFOS approach is the given data independence (*P4*, if the given dynamic adapters match each other) without the need for generic data representations. The data-independence is reached by alternative applicable data formats. Hence, the adapter is mostly independent from the used data format (because one might simply model additional format converters) and the data format of the external system. Further, this is the precondition for an efficient processing of integration tasks. Hence, problem *P1: Poor Performance* can also be eliminated using this approach. As shown in the

following subsection, the optimization of dynamic adapters configurations is an open research challenge. Thus, we explicitly do not present any performance evaluations of generated dynamic adapters because the overall performance strongly depends on the cost-based optimization decisions. However, the proposed approach is the functional precondition for solving this optimization problem.

5.6 Open Issues and Challenges

Although our approach is suitable for application within enterprise integration platforms, there are open problems and research challenges ahead. We want to sketch five of them in the following:

Conceptual adapter specification model: Aside from our described specification model using feature diagrams, a conceptual model is needed specifying the vendor-independent functionalities of an adapter. This will result in a higher degree of portability across different integration platforms of different vendors. Such a conceptual model would consist of a minimum set of concepts to be supported by all integration platforms and of an appropriate extension mechanism for platform-specific functionalities. Thus, the portability of adapter specifications could be ensured (similar to an industry standard).

Debugging and testing: A major problem of the feature-oriented development of adapters is the debugging and testing of these generated software components. So, each configuration change will cause the generator to produce another software. Thus, not only software components have to be tested (which is hard enough by itself). Instead, the generator and all possible outputs have to tested. The latter has a high complexity due to the combinatorial problem of connectors, format converters, protocol handlers, configuration properties and external systems. Here, new methodologies—similar to the testing of procedural language compilers—have to be introduced. We tried to weaken the problem by using the second generation approach, where we reuse functional modules where possible. However, this is really an open research challenge.

Self-configuration of adapter specifications: Following the trend of self*-techniques, the need for protocol and formatting knowledge is an open problem. The goal is to provide only the workflow specification as well as the specifications of external systems, and then, self-configuration techniques should be applied to generate the whole inbound and outbound adapter stacks. Mainly, we see three challenges here: (1) the determination of needed connectors and protocols based on the external systems, (2) the determination of needed format conversions and protocol handling based on the specified workflow, and (3) the optimization of transformation stacks across the whole workflow description. For (3), adaptive methods have to be developed in order to reach the best possible performance, based on the specific workload characteristics. Here, the biggest challenge is the tight-coupling of adapter configurations.

Adequate separation of data and meta data: We already explained that the separation between data and meta data is necessary in order to reach the data independence. Further, this challenge focuses on the adequate management of meta data in order to realize an efficient processing on the one side and to make the generated adapter most robust on the other side. An example for sophisticated management is the determination of suboptimal transformations and the determination of non-pluggable adapter stacks.

Configuration history: Due to the need for recoverability, the change of configuration properties is an open problem. If an integration process fails with the configuration setup $c1$, it will be repeated after a specific period of latency time or upon user request. In case the configuration is changed in the meantime from $c1$ to another configuration setup $c2$, this might lead to unexpected results. In conclusion, the most robust solution is a version system for workflow and adapter configuration properties. Each recovery process can be executed with the configuration of its first failed execution.

6 Related Work

There is a lot of work concerning MDA techniques [3, 4] and MDA tools (e.g., AndoMDA, MOFLON [5] and Fujuba) as well as application integration platforms (e.g, SQL GmbH TransConnect, SAP XI, BEA Integration, MS Biztalk and IBM Message Broker). Here, we first give an overview of model-driven approaches; then, we highlight that in the context of application integration, there is very low support for model-driven development and therefore, we refer to other adapter creation approaches.

The paradigm of model-driven development following the model-driven architecture (MDA)—specified by the Object Management Group (OMG)—mainly addresses the development of software and hardware using generation techniques. The main concepts of a model-driven architecture are MOF [6], standardized meta models like UML [7], and model-model transformation techniques like QVT (Query/View/Transformations) and TGG (Triple Graph Grammars) [8]. Further, MDA is widely used for database application creation using database design tools like Sybase Power Designer or full-fledged CASE tools like Borland Together, IBM Rational Rose or Microtool Objectif. In this area, interesting approaches concerning the optimization of generated code are undergoing development, for example, within the GignoMDA project [9].

Basically, when talking about model-driven architectures within the context of application integration, process description languages like WSBPEL [10], UML [7], BPMN [11], etc. should be mentioned. However, these rather address the generation of integration processes than the generation of adapters (which are used as services in these integration processes). Further, some work on ETL process modeling like [12–14] and ETL model transformation [15–17] also exists, but it omits any details about the realization of the extraction and load components. Also, in the area of EAI, the so-called RADES approach [18] exists, which tries to give an abstract view on EAI solutions using technology-independent and multi-vendor-capable model-driven engineering methodologies. Unfortunately, this approach does not focus on the problems (data independence, efficiency and functional flexibility) considered here.

Regarding the specific problem of adapter generation, there is only little work available. First, there are approaches to automatically generate Web service adapters [19–21] and BPEL adapters [22]. These techniques are too specific to the integration technology used. Second, the semi-automated generation of adapters for legacy applications is addressed in [23]. However, such a semi-automated approach is not suitable. Also, the dynamic adapter generation approach [24] addresses the dynamic adding of new data sources and their invocation rather than the functional flexibility of adapter generation.

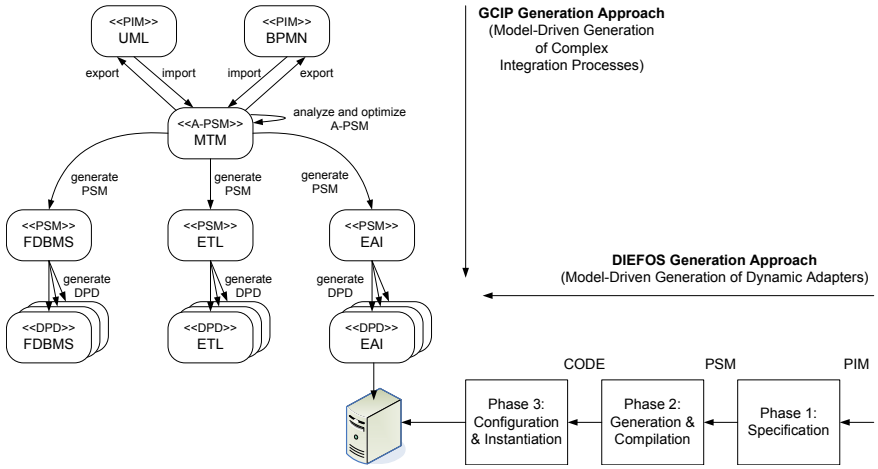

Fig. 6. Two Orthogonal Model-Driven Generation Approaches

In conclusion, we are not aware of any suitable solutions for the generic generation of adapters for integration platforms. However, we are convinced that—due to missing data independence, low performance and functional inflexibility—such a solution is certainly required, similar to our GCIP (Generation of Complex Integration Processes) generation approach [25, 26]. However, as shown in Figure 6, these two model-driven approaches are orthogonal to each other. The GCIP Framework focuses the model-driven generation and optimization of integration processes (deployed into a specific integration platform), while the DIEFOS approach deals with the model-driven generation of dynamic adapters for integration platforms (which execute the deployed integration processes). A tighter correlation of these orthogonal aspects (e.g., with different model views during generation) might be a long-term research goal.

7 Summary

The overall motivation for our model-driven DIEFOS approach was the existence of the four pragmatical problems (1) poor performance, (2) functional restrictions, (3) development effort and (4) need for data independence. The goal was to realize an adapter architecture ensuring data independence with minimal overhead concerning the processing efficiency. Furthermore, the functional flexibility should also be maximized while minimizing the development effort at the same time.

To tackle these problems, we first observed the adapter problem characteristic of real-world integration platforms. Second, we proposed different alternative approaches for the two perspectives: data representations and adapter architectures. Based on this, we introduced our model-driven DIEFOS approach generating adapter types in a feature-oriented manner and discussed several open challenges we see in this context. In conclusion, the model-driven development of integration platform components is advantageous but further research efforts are needed in order to ensure robustness.

References

1. Böhm, M., Habich, D., Lehner, W., Wloka, U.: Improving data intependence, efficiency and functional flexibility of integration platforms (poster). In: CAiSE. (2008) available at: http://www.htw-dresden.de/~mboehm/pubs/caise2008.pdf.
2. Hohpe, G., Woolf, B.: Enterprise Integration Patterns : Designing, Building, and Deploying Messaging Solutions. Addison-Wesley (2004)
3. Kleppe, A., Warmer, J., Bast, W.: MDA Explained. The Model Driven Architecture: Practice and Promise. Addison-Wesley (2003)
4. Thomas, D., Barry, B.M.: Model driven development: the case for domain oriented programming. In: OOPSLA. (2003)
5. Amelunxen, C., Königs, A., Rötschke, T., Schürr, A.: Moflon: A standard-compliant metamodeling framework with graph transformations. In Rensink, A., Warmer, J., eds.: Model Driven Architecture - Foundations and Applications. (2006)
6. OMG: Meta-Object Facility (MOF), Version 2.0. (2003)
7. OMG: Unified Modeling Language (UML), Version 2.0. (2003)
8. Königs, A.: Model transformation with triple graph grammars. In: MODELS. (2005)
9. Habich, D., Richly, S., Lehner, W.: Gignomda - exploiting cross-layer optimization for complex database applications. In: VLDB. (2006)
10. OASIS: Web Services Business Process Execution Language Version 2.0. (2006)
11. BMI: Business Process Modelling Notation, Version 1.0. (2006)
12. Simitsis, A., Vassiliadis, P.: A methodology for the conceptual modeling of ETL processes. In: CAiSE workshops. (2003)
13. Trujillo, J., Luján-Mora, S.: A UML based approach for modeling ETL processes in data warehouses. In: ER. (2003)
14. Vassiliadis, P., Simitsis, A., Skiadopoulos, S.: Conceptual modeling for ETL processes. In: DOLAP. (2002)
15. Hahn, K., Sapia, C., Blaschka, M.: Automatically generating OLAP schemata from conceptual graphical models. In: DOLAP. (2000)
16. Mazón, J.N., Trujillo, J., Serrano, M., Piattini, M.: Applying mda to the development of data warehouses. In: DOLAP. (2005)
17. Simitsis, A.: Mapping conceptual to logical models for ETL processes. In: DOLAP. (2005)
18. Dorda, C., Heinkel, U., Mitschang, B.: Improving application integration with model-driven engineering. In: ICITM. (2007)
19. Benatallah, B., Casati, F., Grigori, D., Nezhad, H.R.M., Toumani, F.: Developing adapters for web services integration. In: CAiSE. (2005) 415–429
20. Lee, K., Kim, J., Lee, W., Chong, K.: A tool to generate an adapter for the integration of web services interface. In: CBSE. (2006) 328–335
21. van den Heuvel, W.J., Weigand, H., Hiel, M.: Configurable adapters: the substrate of self-adaptive web services. In: ICEC. (2007) 127–134
22. Brogi, A., Popescu, R.: Automated generation of bpel adapters. In: ICSOC. (2006) 27–39
23. Pieczykolan, J., Kryza, B., Kitowski, J.: Semi-automatic creation of adapters for legacy application migration to integration platform using knowledge. In: International Conference on Computational Science (4). (2006) 252–259
24. Gong, P., Gorton, I., Feng, D.D.: Dynamic adapter generation for data integration middleware. In: SEM. (2005) 9–16
25. Böhm, M., Habich, D., Lehner, W., Wloka, U.: Model-driven generation and optimization of complex integration processes. In: ICEIS. (2008)
26. Böhm, M., Habich, D., Lehner, W., Wloka, U.: Model-driven development of complex and data-intensive integration processes. In: MBSDI. (2008)