A Conceptual Framework and Experimental Workbench for Architectures

Marco Konersmann and Michael Goedicke

paluno - The Ruhr Institute for Software Technology,
University of Duisburg–Essen
Gerlingstraße 16, 45127 Essen, Germany
{marco.konersmann,michael.goedicke}@paluno.uni-due.de

Abstract. When developing the architecture of a software system, inconsistent architecture representations and missing specifications or documentations are often a problem. We present a conceptual framework for software architecture that can help to avoid inconsistencies between the specification and the implementation, and thus helps during the maintenance and evolution of software systems. For experimenting with the framework, we present an experimental workbench. Within this workbench, architecture information is described in an intermediate language in a semantic wiki. The semantic information is used as an experimental representation of the architecture and provides a basis for bidirectional transformations between implemented and specified architecture. A systematic integration of model information in the source code of component models allows for maintaining only one representation of the architecture: the source code. The workbench can be easily extended to experiment with other Architecture Description Languages, Component Models, and analysis languages.

1 Motivation

Current approaches for software architecture development propose to develop software architecture specifications using sophisticated languages, to analyze the architecture, and eventually to implement it using modern software component technologies (cf. [1]). Following these best-practices thus leads to at least two representations of the software architecture: the specification and the implementation. The specification is an abstract, precise description of the architecture, which lacks implementation details. The implementation includes the architecture and any other implementation detail of the software. Architecture analysis can introduce further representations, when architectures are transformed into analysis languages.

The simultaneous existence of more than one representation of the architecture makes it harder to maintain because all representations have to be kept consistent. Else, faults might be introduced during maintenance and evolution when developers rely on outdated or false information. To avoid such faults, we present a conceptual framework for software architecture, that ensures the consistency between architecture representations.

M. Heisel (Ed.): Krämer Festschrift, LNCS 7365, pp. 36–52, 2012.
© Springer-Verlag Berlin Heidelberg 2012

The framework is accompanied by an experimental workbench. The experimental workbench is the technical platform for the framework. It can be used for experimenting with the features of the framework, and for viewing and editing architectures in arbitrary Architecture Description Languages (ADLs) and Component Models (CMs). In this work an ADL is any language used to describe a software architecture on a higher level of abstraction than the implementation code of current programming languages.

The remainder of this paper is structured as follows: In section 2 we discuss how other approaches relate to the problem. In section 3 we present our framework. Section 4 presents the experimental workbench for software architectures, which is based on the framework. Section 5 shows the benefits of the workbench in a small example. We discuss our results and identify future work in section 6 before we conclude in section 7.

2 Related Work

Managing multiple representations of software design and specifically architecture has been subject to other fields of research. Closely related to the paper at hand is the field of Model-Driven Software Development (MDSD) (e.g. [2]), model execution (e.g. [3,4]), and round trip engineering (e.g. [5]). In this context we differentiate these approaches regarding the abstraction levels that they focus on and the possibility of bidirectional transformations between representations of the design or architecture.

In MDSD abstract domain specific models of the software to be developed are created with domain experts. These domain models are refined with detailed technical models that are not relevant to the domain, but to the platform that will run the software. Such models are often the basis for automated code generation. The generated code has to be enriched with implementation details. Model-Driven Architecture (MDA) [6] is a MDSD approach by the Object Management Group (OMG)[1]. In MDA Platform-Independent Models (PIM) are the domain models. Platform-Definition Models (PDM) are the basis for translating PIMs into Platform-Specific Models (PSM). PSM can be run on their corresponding platform.

MDSD concentrates on deriving code from models. The specification (PIM, PDM, and PSM) and the code are two representations of the architecture that are independently subject to evolution and maintenance. Changes in the specification can be taken over automatically in the implementation. When the architecture changes in the implementation, these changes cannot be automatically taken over in the specification. MDSD bridges the gap between the abstraction levels of the representations, but changes can only be carried over one way, from the abstract specification to the detailed source code.

Model execution (e.g. Executable UML [3,4]) reduces the representations to the specification. The specifying model is enriched with clear semantics. Thus the models can be executed. In these approaches, typically less abstract models

[1] `http://www.omg.org`

are used, that can be easily translated into programming language semantics, e.g Unified Modeling Language (UML) class diagrams or state charts. Model execution thus does not bridge the gap between abstraction levels.

Round trip engineering describes techniques to synchronize models and code. Changes in the models can be automatically translated into the corresponding code, and changes in the code can be automatically translated into the corresponding model. The models used in round trip engineering are very detailed, technical models, e.g. UML class diagrams [7] or state charts. Round trip engineering thus allows for two-way synchronization, but does not bridge the gap between abstraction levels.

In this section we have shown that the current approaches do not solve the problem that we have identified in section 1, because the combination (1) bridging the gap between abstraction levels, and (2) allowing bi-directional synchronization between code and model, is not addressed by these approaches.

3 A Conceptual Framework for Architectures

To address the problem, we now present a conceptual framework for architectures. An overview of the artifacts of the framework and their relationships is given in figure 1. The main goal of the framework is to support the maintenance and evolution of software architectures. The development of the framework pursues three objectives:

1. It is ensured that the architecture implementation and specification are consistent.
2. The architecture can be viewed and edited using arbitrary ADLs and their respective editors, and deployed using arbitrary CMs.
3. Analyses can be performed over the architecture to evaluate its quality and validity.

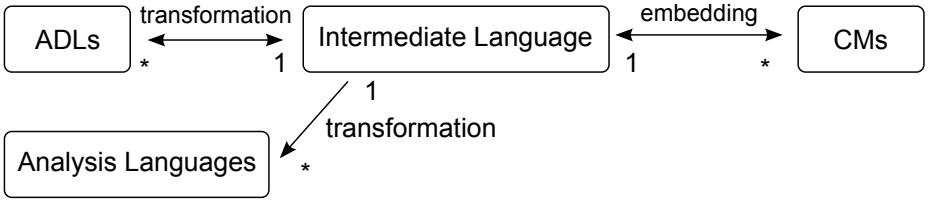


Fig. 1. Overview of the conceptual framework for architectures

The realization of objective 1 avoids that faults are introduced during maintenance and evolution. As shown in section 2, having only the specification does not solve the problem. In our framework the only persistent representation is the implementation. In our approach we assume that the architecture will be implemented using a well-defined CM, e.g. using Enterprise Java Beans [8] of the Java Enterprise Edition (JEE) [9].

CMs and ADLs differ in their features and their abstraction level (cf. [10]). CMs include implementation details, and typically only define the structure of the system formally. ADLs do not include implementation details, but typically include aspects besides the structure, e.g. behaviour or quality (cf. [10]).

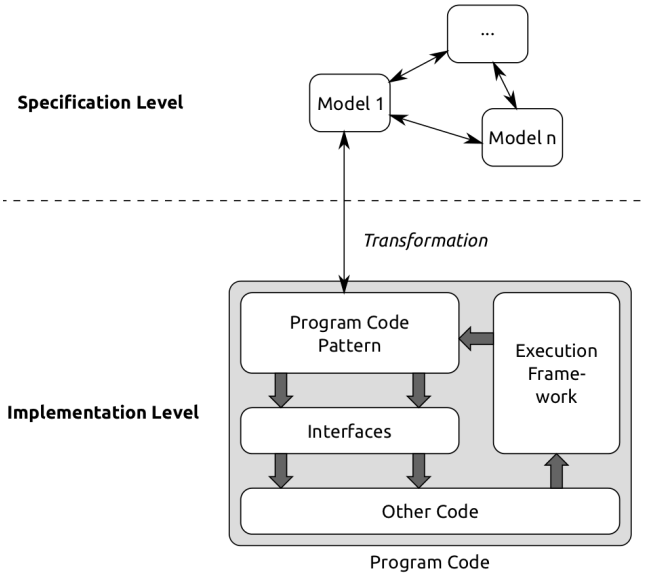


Fig. 2. An overview of the embedded models approach: Model elements are represented using program code patterns. The code patterns provide interfaces for interacting with arbitrary other code. An execution engine understands the embedded specification and executes the embedded model. (Source: [11])

To achieve objective 1, the abstract architecture information of ADLs is explicitly integrated into the source code that forms the CM. This is performed using the *embedded models* approach [11]. The architecture is thus explicitly accessible at design time and at run time and at the same time specification and implementation. In the embedded models approach rules are defined how to systematically and explicitly integrate this information into the implementation, utilizing program code patterns. Figure 2 gives an overview about the structure of the embedded models approach. Models in the specification level are transformed into program code patterns on the implementation level. In the architecture framework, the models on the specification level are the models expressed in ADLs. Program code patterns in the architecture framework are based on the patterns introduced by the CMs in use, enhanced with information that cannot be expressed in the CM, e.g. utilizing typed meta data such as annotations in Java [12]. The program code patterns provide interfaces to arbitrary program code that is not part of the architecture description, but

represents the implementation details. An execution framework is used to execute the model code. In the architecture framework, the execution framework is based on the existing execution frameworks of the CMs, e.g. application servers. These have to be enhanced to understand the semantics of the embedded architecture information. An example of an embedded model is given in section 5.

The objectives 2 and 3 are based on the observation that many different ADLs, architecture analysis languages, and CMs exist and are used in practice. The framework allows to develop software architectures using almost arbitrary ADLs and CMs. To achieve this, the features of all considered ADLs need to be embedded into each considered CM. However, many ADLs and CMs exist for many different domains, and one can imagine that more will exist in the future. Defining rules for embedding each ADL in each CM does not seem realistic, as one definition for each pair would have to be created. The same holds for analysis languages. Instead, we introduce an intermediate language that reduces the *n:m* relationship to a *n:1:m* relationship. We create bidirectional mappings between ADLs and the intermediate language, and unidirectional mappings between the intermediate language and analysis languages. The intermediate language is embedded into the CM's source code.

ADLs and CMs both describe architectures, but they have different features (cf. [10]). Within the group of ADLs and the group of CMs the features also differ. The relationship between the ADLs and CMs via the intermediate language has to reflect these differences. Features of ADLs and CMs that have a direct equivalent in another representation can be used in both representations by defining a model transformation. When no equivalent exists, a complex transformation should be defined that emulate the missing feature, if possible. An example for a complex transformation is the feature of hierarchical architectures: In such architectures *composite* components have other components as children. These children are hidden from the parent component's context. The interfaces of parent components are delegated to their children's interfaces and vice versa. The children are interconnected via their interfaces. When an ADL is chosen that allows for hierarchical architectures and a CM is chosen that only allows flat architectures, the CM does not have a direct equivalence of the parent and child relationship. It might possible to emulate such a behaviour using only the concepts available in the CM. In that case a complex transformation can be used. If the feature cannot be emulated, the feature cannot be used in the first place. If the feature is mandatory to be used (e.g. because it is a core concept of the language), but it cannot be expressed in the second representation the representations are incompatible. This means that choosing the first representation excludes the latter from being chosen.

To consider this aspect, the meta model of the intermediate language is modular. In the framework this modularity is currently described using an orthogonal variability model (OVM) [13, p. 72ff.]. When an ADL or a CM is chosen for working with the architecture, the ADL's or CM's features define the configuration of the variability model. The variability model defines which modules in

the meta model of the intermediate language are active, and which other representations are compatible. Figure 3 shows an example of this dependency: The features identified for ADLs and CMs include the variation point *Component Hierarchy*. The variation point is expressed with a triangle. The variants of this variability point are *Hierarchical* denoting a hierarchical component model, and *Flat*, denoting the absence of hierarchy. The variants are mutually exclusive, as denoted by the *[1...1]* expression at the variation point. When choosing an ADL with a hierarchical component model, the variant *Hierarchical* is activated. Consequently the reference in the meta model of the intermediate language is available.

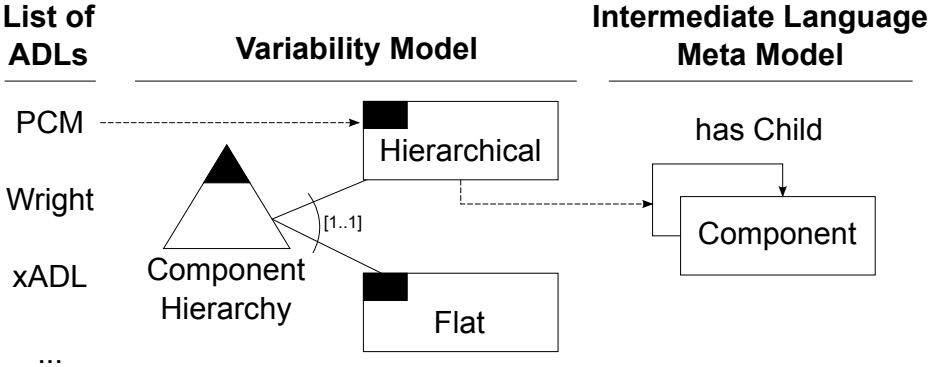


Fig. 3. The Intermediate Language has a variable meta model. Depending on the chosen ADL or CM, features of the meta model are activated or not. The dashed arrow from the list of ADLs to the variability model means that the ADL PCM has a hierarchical component model. The dashed arrow from the variability model to the meta model means that choosing the variant *hierarchical* enables the reference in the meta model.

In this section we presented our conceptual framework for ADLs. In the next section, we present an experimental workbench for architectures, which is based on this framework.

4 An Experimental Workbench for Architectures

We developed an experimental workbench for architectures based on the conceptual framework. The workbench allows to experiment with architectures, using different ADLs, CMs, and analyses. An overview of the workbench is given in figure 4. The core of the workbench is a Semantic MediaWiki (SMW)[2]. A semantic wiki contains pages with informal information enriched with typed key-value pairs of structured information (attributes). Pages can be grouped into categories. This semantic information can be subject to queries for systematically

[2] `http://semantic-mediawiki.org`

finding information in the wiki. The wiki provides flexibility in the information structure and a REST[3] [14] interface for accessing the information with arbitrary clients. This renders the SMW useful for experiments.

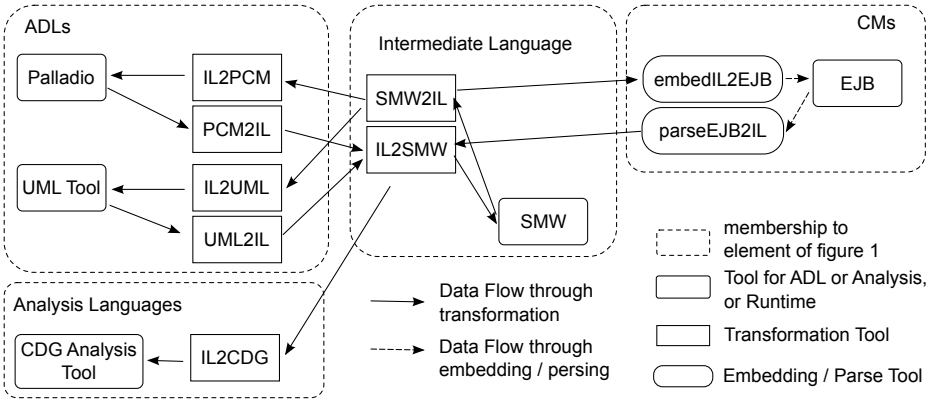


Fig. 4. Overview of the data flow in the experimental workbench for architectures

For representing architecture information in the wiki, we implemented a simple architecture language for the wiki consisting of categories for pages and attributes. The language has the role of the intermediate language in our framework. The language is presented in figure 5 and should be seen as initial approach for further experiments with the conceptual framework for architectures. The language consists of *components*, *interfaces*, and *operations*. Operations are atomic entities that have a name. Interfaces comprise a set of operations. Components have *required* and *provided* interfaces, and *common parameters*, which are required interfaces that are also provided (cf. [15]). Components instantiate child components that are identified by names. Thus a hierarchy of components is defined. Child components are connected to each other using common interfaces. This means that two components can be connected to each other when one provides the interface that the other component requires. Interfaces of children can also be delegated by the parent. A delegated provided interface is not connected to a required interface, but is provided by the parent to its context. Delegated required interfaces of children are also required by the parent. Delegated common parameters of children are common parameters of the parent. With that language, static structures of architectures can be defined. SMW allows arbitrary additional attributes to be attached to pages.

The information in the wiki can be transformed into ADLs and into analysis languages. To address ADLs we implemented bidirectional transformations between the intermediate language and a subset of the UML, and a subset of the Palladio Component Model (PCM) [16]. Despite the name "Palladio Component Model",

[3] Representational State Transfer.

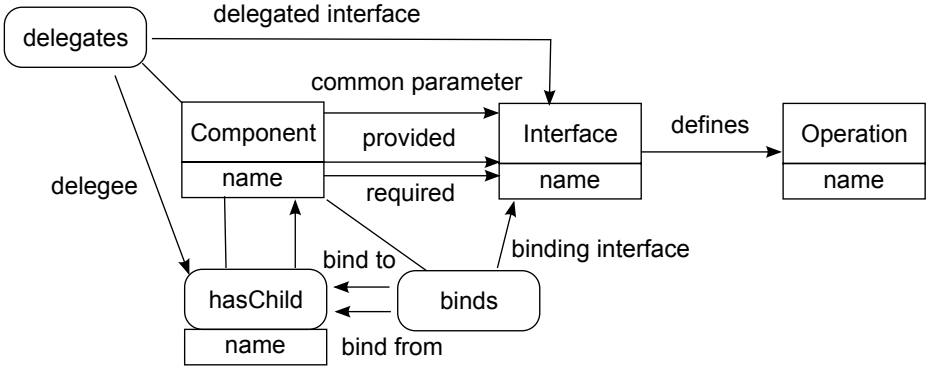


Fig. 5. The definition of the intermediate language in the semantic wiki. The rectangles represent categories of pages, with their attributes below. Rounded rectangles represent so-called internal objects. These are attributes that can take multiple values and references. The arrows between categories and between categories and internal objects are references. The role of the reference is labeled at the arrow. The line between a category and an internal object shows that the internal object is defined by the category.

the PCM is regarded as an ADL here, because it is used to describe architectures on a higher level of abstraction than the implementation code. We can thus transform architectures defined in the wiki into these languages, and use their respective editors to modify the architecture. In the case of PCM, the Palladio Simulator[4] can also be used to execute performance tests on the architecture. These tests require more information than the current intermediate language provides. When this information is given in PCM, and the architecture is transformed into the intermediate language, this additional information is added to the respective pages using attributes. Currently that information remains unused in other representations.

To realize the transformation, we parse the wiki data and store it as an Ecore model. Ecore is the meta model used in the Eclipse Modeling Framework (EMF) [17]. For UML an Ecore meta model exists in the Eclipse UML2 project[5]. The PCM meta model is also defined in Ecore using EMF (cf. [18]). The transformations between the Ecore models is realized with ATL, the model-to-model transformation tool of the Eclipse Modeling Project[6].

To address analysis languages, a unidirectional transformation has been developed from the intermediate language to a simple component dependency graph (CDG). The CDG is a directed graph that consists of *components* and *dependency* relations between the components. Figure 6 shows an example of an architecture and its CDG. The dependency relationship from component *Shop* to

[4] `http://www.palladio-simulator.com`
[5] `http://www.eclipse.org/modeling/mdt/?project=uml2#uml2`
[6] `http://www.eclipse.org/modeling/`

ShoppingCart means that *Shop* directly depends on *ShoppingCart*. A direct dependency is defined as follows: When a component delegates a provided interface to its child, then the parent directly depends on the child (e.g. *WebShop* and *Shop*). When a child delegates a required interface to its parent, then the child depends on the parent (e.g. *ShoppingCart* and *WebShop*. When a required interface of a component A is connected to the provided equivalent of another component B, then A directly depends on B (e.g. *Shop* and *ShoppingCart*. An unconnected required interface is a dependency relationship to a node named after the interface (e.g. *WebShop* and *IDatabase*. A dependency relationship is typed with the interface that the dependency is based on. The CDG allows for analyzing the architecture regarding cycles in the dependencies and unreachable components, by using standard graph analysis techniques. As the CDG does not carry all information that is relevant to the architecture, a bidirectional transformation is not possible. To realize the transformation between the intermediate language and the CDG, we use the model transformation tool ATL. The CDG is realized as an Ecore model.

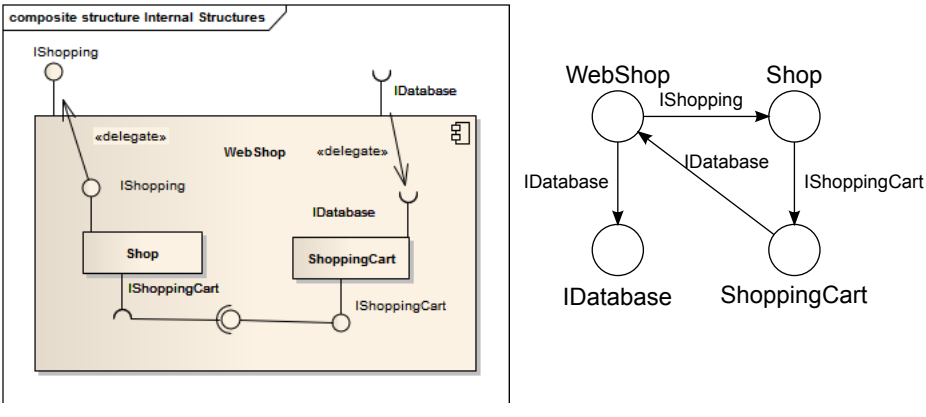


Fig. 6. The left side shows an example architecture expressed in UML. The right side shows its component dependency graph.

For executing the architectures defined in the intermediate language, we embedded the intermediate language into Enterprise Java Beans (EJB) [8]. The embedding ensures that the architecture definition is correctly and precisely implemented, and that changes to the implemented architecture can be included in the architecture definition in the wiki. In the following, the embedding of the language shown in figure 5 is described.

The component in the intermediate language is a page with a name. In EJB, we chose to use singleton beans as components. Singleton beans exist exactly once during the run time of the system. Using more than one component instance is not possible using this pattern. The pattern is shown in figure 7.

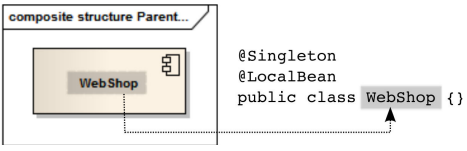


Fig. 7. The pattern for a component

Figure 8 shows the patterns for required and provided interfaces as well as common parameters of components. Provided interfaces are represented as EJB bean interfaces. Required interfaces are represented as required bean interfaces. These required bean interfaces are injected by the execution environment. Common parameters are interfaces that are required and provided. Thus the interface operations delegate the execution to the required interface. The pattern for an interface and its operation is not shown. They are represented as interfaces and their operations in Java.

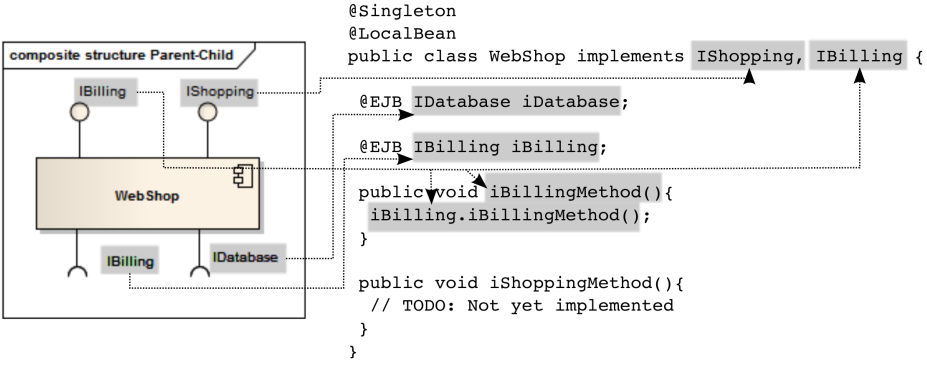


Fig. 8. The pattern for required and provided interfaces, and common parameters of components

Figure 9 shows the pattern for child components with delegated provided interfaces and delegated common parameter interfaces. Child components are are injected to the parent using the EJB injection mechanism. Children and their interfaces are in a Java package named after the parent. In addition to the pattern for provided interfaces, the pattern for delegated provided interfaces delegates the execution of the interface operations to the child component. Delegated common parameter interfaces also use this pattern, but additionally define and call a method of the child component that provides to the child the reference to the implementing bean.

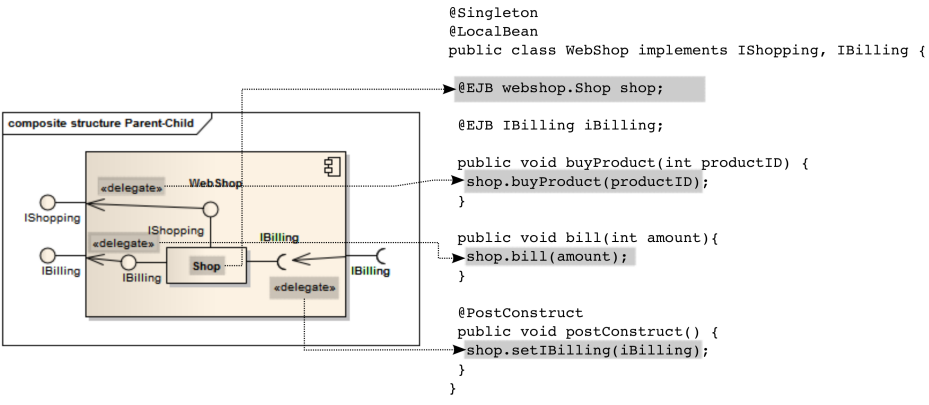


Fig. 9. The pattern for child components with delegated provided and common parameter interfaces

The pattern for a child component with delegated required interfaces is shown in figure 10. The requiring child component has a method for receiving a bean providing the required interface. The parent component uses the EJB injection mechanism to obtain a such a bean. It then calls its child's setter method.

In this section we presented a workbench for experimenting with software architectures and for exploring the possibilities of the conceptual architecture framework.

5 Working with the Workbench

In this section we present an example how to work with the experimental workbench presented above. The transformation and embedding steps are not shown in this example because they are executed by command line programs. As a first step we implemented a small example architecture in UML using the Eclipse UML tool. The example architecture is shown on the left side of figure 11. It contains the composite component *WebShop* that provides the interface *IShopping* and requires the interface *IDatabase*. The interface *IBilling* is a common parameter. The interface *IShopping* and the common parameter *IBilling* are delegated to the child component *Shop*. Another child component is *ShoppingCart*, which requires the interface *IDatabase*. This requirement is delegated to the parent component. *Shop* and *ShoppingCart* are connected by the interface *IShoppingCart*.

The architecture is then transformed into the intermediate language in the SMW. Figure 12 shows a screenshot of the *WebShop* component representation in the SMW. The presentation of the data can be arbitrarily designed. As the next step, the architecture is embedded into EJB source code. The representation of the parent component in the resulting source code is shown on the right side of figure 11.

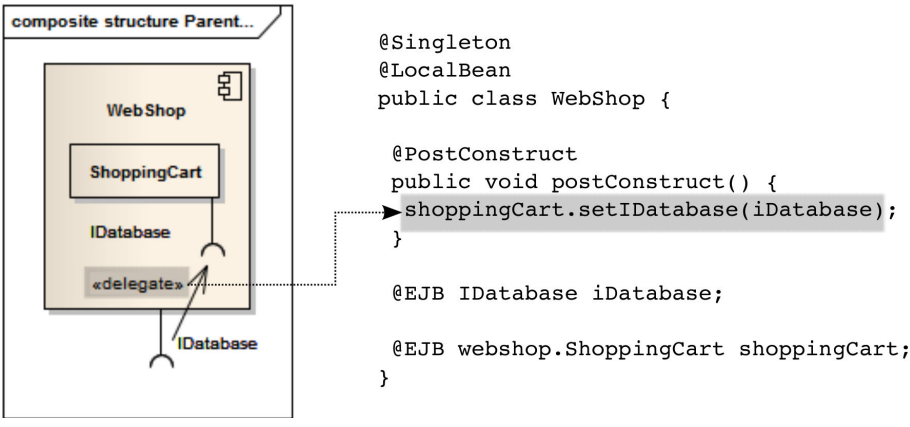


Fig. 10. The pattern for a component with a delegated required interface

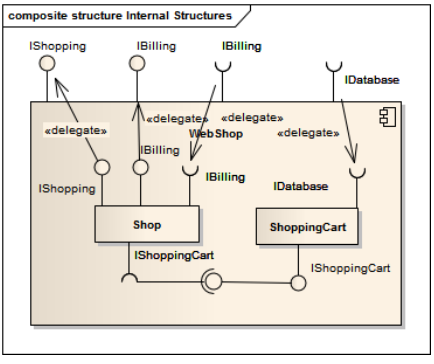
During the evolution of the system a change in the architecture takes place: The common parameter *IBilling* is now a provided interface delegated to a new child component *Billing*. The change happens in the source code of the EJB component model. The UML model and the EJB implementation of the new architecture are shown in figure 13. The right side of the figure shows the changed EJB implementation. After transforming the embedded model into UML via the intermediate language, the new architecture can be seen in UML on the left side of figure 13.

6 Discussion

In this section we discuss our ongoing work regarding the conceptual framework and the experimental workbench, and the results presented in this paper.

6.1 Conceptual Framework

With the framework we strive to achieve three objectives. The first objective is to ensure that the architecture implementation and specification are consistent. To achieve this objective we use bidirectional transformations between ADLs and the intermediate language, and the embedded models approach to integrate the architectural information in the intermediate language with the source code of CMs. Both techniques can be used to transfer information consistently. The model transformation is a deeply researched and well understood field. However, one has still to be careful when defining bidirectional transformations. Some languages might have features that are hard to integrate with the features of other languages. The embedded models approach is not that mature. Embedded models are currently only researched regarding behavioural models, specifically state machines and process models. Architecture definitions are more complex




```
@Singleton
@LocalBean
public class WebShop implements IShopping, IBilling {

 @PostConstruct
 public void postConstruct() {
  shop.setIShoppingCart(shoppingCart);
  shop.setIBilling(iBilling);
 }

 @EJB webshop.Shop shop;

 @EJB webshop.ShoppingCart shoppingCart;

 @EJB IBilling iBilling;

 @EJB IDatabase iDatabase;

 public void buyProduct(int productID) {
   shop.buyProduct(productID);
 }

 public void bill(int amount){
   shop.bill(amount);
 }
}
```

Fig. 11. The example architecture in UML on the left side, and embedded into EJB on the right side

than these models. Behavioural models are just one aspect of software architecture. Several viewpoints have also to be considered, e.g. static and dynamic structure, communication, deployment, and quality aspects. The complexity of architectural description is a challenge yet to address in the embedded models approach.

The second objective is that the architecture can be viewed and edited using arbitrary ADLs and their respective editors, and deployed using arbitrary CMs. This objective is addressed by the intermediate language and the bidirectional transformations between the ADLs and the intermediate language, as well as the embedded models approach. Instead of defining patterns for embedding each ADL in each CM, potentially in several programming languages, the intermediate language is embedded into the CMs, and transformations are defined between ADLs and the intermediate language. This reduces the effort for adding new ADLs and CMs to work with the framework. However, the variability of the intermediate language is a challenge. ADLs and CMs have different features, and even a feature such as a component hierarchy may be differently realized. E.g. in one ADL a child component is a static component that is defined in its parent. In another ADL a child component may be a named instance of a component that is defined in the same scope as the parent. This has to be addressed when the variability model of the intermediate language is elaborated.

For the framework to be useful, it should be possible to adapt the intermediate language to further progress in the area of architecture descriptions. To achieve this, the intermediate language should be modular and thus allow extensions in the future.

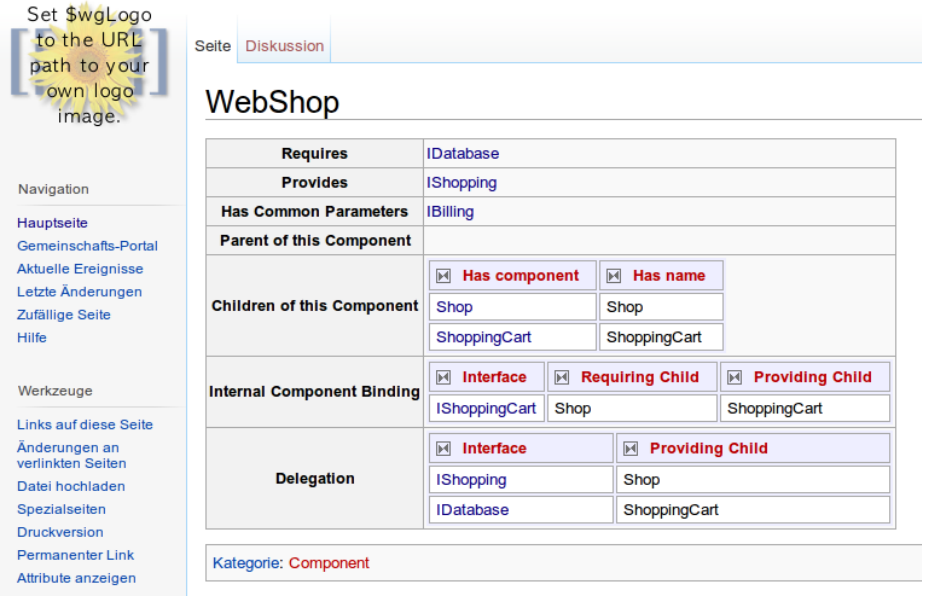


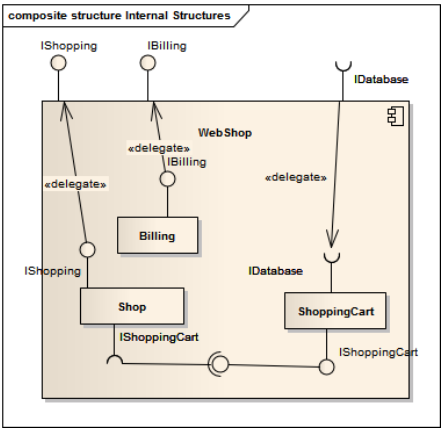
Fig. 12. The *WebShop* component of the example architecture in the SMW

The third objective is that analyses can be performed over the architecture to evaluate its quality and validity. This is addressed by defining unidirectional transformations between the intermediate language and analysis languages. Some types of analysis can also be performed using ADLs, e.g. performance analyses using the PCM. Some kinds of analysis can also be performed using the implementation. E.g. stress tests using the embedded architecture and a stress test driver. Here complex dependencies may be introduced for executing such analyses: Some analyses, such as performance tests require detailed information about the system's behaviour and e.g. its deployment. Such information is only available from some ADLs. These complex dependencies can be addressed by identifying which variants of the meta model of the intermediate language are necessary for an analysis language.

As we elaborated, the objectives stated in the beginning of this paper are addressed and it seems we can successfully achieve the objectives. However, we have found challenges to reach the objectives we have identified, that are addressed in our ongoing work.

6.2 Experimental Workbench

The experimental workbench has been developed to experiment with, evaluate, and refine the framework. The use of the SMW is helpful for testing new language constructs, because it allows arbitrary data to be added to pages. However, the language presented in this paper is far from being useful for realistic case studies




```
@Singleton
@LocalBean
public class WebShop implements IShopping, IBilling {

 @PostConstruct
 public void postConstruct() {
  shop.setIShoppingCart(shoppingCart);
 }

 @EJB Shop shop;

 @EJB ShoppingCart shoppingCart;

 @EJB Billing billing;

 @EJB IDatabase iDatabase;

 public void buyProduct(int productID) {
  shop.buyProduct(productID);
 }

 public void bill(int amount){
  billing.bill(amount);
 }
}
```

Fig. 13. The example architecture after the changes embedded into EJB on the right side, and in UML on the left side

and needs to be refined. Also, the variability aspect of the intermediate language is currently not taken into account in the SMW.

The architecture representation in the SMW contradicts the idea to have only one persistent representation, because the architecture is persistent in the code and in the SMW. In the long run, the SMW should not store persistent architecture information. Instead, a transformation (bi- or unidirectional) could be developed from the intermediate language that is embedded in a CM to the SMW in terms of an ADL or analysis language. The SMW could provide a good basis for documenting the architecture, as it allows for arbitrary information to be added to elements, including informal text and figures.

7 Conclusion

We have presented our approach that helps to avoid faults due to inconsistencies between architecture specifications and their implementation. In contrast to related work shown in section 2, our approach is based on the idea to have only the source code as persistent representation of the architecture, while still bridging the gap between the different abstraction levels of the specification and the implementation of software architecture. Using the embedded models approach, the architecture information is explicitly integrated in the source code and accessible at design and at run time. The conceptual framework allows for modifying the architecture with arbitrary Architecture Description Languages and Component Models, as long as transformations and embedding mechanisms have been defined for these languages. The modeled architecture can be analyzed

using languages that are embodied in the framework using transformations. An intermediate language has been introduced to reduce the effort of defining transformations between architecture descriptions.

We have also presented an experimental workbench for architectures that is based on the conceptual framework. The workbench allows for experimenting with architectures, and elaborate the framework. It uses a semantic wiki as a core, that contains the architectural information and allows for arbitrary extensions of the intermediate language. The workbench also includes a set of programs to transform the architecture information into ADLs and analysis languages, and to embed the architecture in CMs. Currently the workbench supports a subset of UML and the Palladio Component Model as ADLs, a component dependency graph as analysis language, and an embedding into Enterprise Java Beans as Component Model. We showed how to use the workbench in a simple example.

As future work we plan to address the challenges identified in the discussion in section 6. In addition, we plan to develop a tool suite for unifying the definition of transformations between the intermediate language and ADLs and analysis languages. We also want to research more deeply the possible patterns and mechanisms for embedding architecture information. Another challenge is the question, how to manage the architecture when in one system more than one ADL or CM is used.

References

1. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software Architecture: Foundations, Theory, and Practice. Wiley Publishing (2009)
2. Brown, A., Conallen, J., Tropeano, D.: Introduction: Models, Modeling, and Model-Driven Architecture (MDA) Model-Driven Software Development. In: Beydeda, S., Book, M., Gruhn, V. (eds.) Model-Driven Software Development, pp. 1–16. Springer, Berlin (2005)
3. Luz, M.P., da Silva, A.R.: Executing UML Models. In: 3rd Workshop in Software Model Engineering, WiSME 2004 (2004)
4. Mellor, S.J., Balcer, M.: Executable UML: A Foundation for Model-Driven Architectures. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
5. Nickel, U.A., Niere, J., Wadsack, J.P., Zündorf, A.: Roundtrip Engineering with FUJABA. In: Proc of 2nd Workshop on Software-Reengineering (WSR), Bad Honnef, Germany (2000)
6. Mukerji, J., Miller, J.: Technical Guide to Model Driven Architecture: The MDA Guide v1.0.1. Technical report (2003)
7. OMG: OMG Unified Modeling LanguageTM (OMG UML), Superstructure, Version 2.4.1, Object Management Group (August 2011)
8. Sun Microsystems, Inc.: JSR 318: Enterprise JavaBeans^{TM}3.1 (December 2009), `http://jcp.org/en/jsr/detail?id=318`
9. Sun Microsystems, Inc.: JSR 316: Java^{TM}Platform, Enterprise Edition 6 (Java EE 6) Specification (December 2009), `http://jcp.org/en/jsr/detail?id=316`
10. Müller, M., Balz, M., Goedicke, M.: Representing Formal Component Models in OSGi. In: Engels, G., Luckey, M., Schäfer, W. (eds.) Software Engineering. LNI, vol. 159, pp. 45–56. GI (2010)

11. Balz, M.: Embedding Model Specifications in Object-Oriented Program Code: A Bottom-up Approach for Model-based Software Development. PhD thesis, Universitat Duisburg-Essen (May 2011)
12. Sun Microsystems, Inc.: JSR 175: A Metadata Facility for the Java™ Programming Language (2004), http://jcp.org/en/jsr/detail?id=175
13. Pohl, K., Böckle, G., van der Linden, F.: Software product line engineering - foundations, principles, and techniques. Springer (2005)
14. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. PhD thesis, University of California, Irvine (2000), http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
15. Schumann, H., Goedicke, M.: Component-oriented software development with pi. Technical Report 1/94, Department of Mathematics and Computer Science, University of Essen (1994)
16. Reussner, R., Becker, S., Happe, J., Koziolek, H., Krogmann, K., Kuperberg, M.: The Palladio Component Model. Technical report, Chair for Software Design & Quality (SDQ), University of Karlsruhe (TH), Germany (May 2007)
17. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0, 2nd edn. Addison-Wesley Professional (2009)
18. Becker, S., Koziolek, H., Reussner, R.: Model-Based Performance Prediction with the Palladio Component Model. In: Proceedings of the 6th International Workshop on Software and Performance, WOSP 2007, pp. 54–65. ACM, New York (2007)