Same but Different: Consistently Developing and Evolving Software Architecture Models and Their Implementation

Marco Konersmann and Michael Goedicke

Abstract As software architecture is a main driver for the software quality, source code is often accompanied by software architecture specifications. When the implementation is changed, the architecture specification is often not updated along with the code, which introduces inconsistencies between these artifacts. Such inconsistencies imply a risk of misunderstandings and errors during the development, maintenance, and evolution, causing serious degradation over the lifetime of the system. In this chapter we present the Explicitly Integrated Architecture approach and its tool *Codeling*, which remove the necessity for a separate representation of software architecture by integrating software architecture information with the program code. By using our approach, the specification can be extracted from the source code and changes in the specification can be propagated to the code. The integration of architecture information with the code leaves no room for inconsistencies between the artifacts and creates links between artifacts. We evaluate the approach and tool in a use case with real software in development and with a benchmark software, accompanied by a performance evaluation.

1 Introduction

In the development of software systems, the software architecture [32] describes the software's general building blocks, including structural and behavioral aspects. Software architecture is one of the main drivers for software quality. Due to its importance, software architecture is often modeled in some way, for communicating and analyzing the architecture. This includes a brief sketch on a sheet of paper and goes up to extensive specifications in formal languages.

M. Konersmann (✉)
University of Koblenz-Landau, Mainz, Germany
e-mail: konersmann@uni-koblenz.de

M. Goedicke
University of Duisburg-Essen, Duisburg, Germany
e-mail: michael.goedicke@paluno.uni-due.de

M. Felderer et al. (eds.), *Ernst Denert Award for Software Engineering 2019*,
https://doi.org/10.1007/978-3-030-58617-1_6

While the software architecture can be modeled, it certainly will be implemented using programming languages to build an executable system. The current major programming languages, such as Java, C, Python, and so on [1], do not provide abstractions for components and connectors. A translation between the concepts of architecture specification languages and programming languages is required for a project, to understand how these concepts are implemented. Developers either create project-specific conventions to implement architectural features or use one of many component implementation frameworks. Components are then, e.g., implemented by a class definition with a name ending with `Component` within a package. The component provides interfaces by implementing a corresponding interface, that is defined in the same package. Conventions like these can be found in many software systems, often not even documented. Implementation frameworks specify recurring patterns for implementing architecture information such as component-based structures. For example, in Enterprise Java Beans (EJB) [23] Java classes with specific annotations can be used as components.

Both, conventions and frameworks, can be seen as architecture implementation languages. They consider program code to describe architectural information. But a program is not complete with an architecture alone. Further program code is necessary to implement the detailed behavior or fine-grained structures, which is required to build an executable and functional system. Architecture implementation languages also need to provide room for such program code.

Languages for specifying and for implementing software architectures share a common core: components and their interconnection [22]. As they describe overlapping information, they should be consistent. Undetected inconsistencies may be the source for misunderstandings and errors in the implementation. Automated consistency checks can identify whether an architecture specification and its implementation are in a consistent state. Existing approaches for consistency of architecture models and implementations usually focus on a pair of specification and implementation languages (see related work in Sect. 5). New approaches need to be developed for emerging or evolving languages, which take the specific features of the languages into account and respect existing, non-architectural program code.

A variety of software architecture specification languages exist, such as dedicated architecture description languages (ADLs) [18] or general-purpose modeling languages, that are feasible to describe software architectures, such as the Unified Modeling Language (UML). The different languages provide different features, including component hierarchies, different types of connectors, or different concepts of abstract behavior. The implementation of software architecture is similarly diverse. This diversity is a challenge for maintaining the consistency of architecture implementations and specifications.

This chapter presents the following main contributions:

1. The *Explicitly Integrated Architecture* (EIA) approach automatically maintains the consistency of implemented and specified architectures. It translates between program code and architecture models, by extracting architecture models from code and propagating model changes back to the code. It takes the difference of

architecture languages into account and reduces the effort for integrating new or evolved languages into the architecture model/code translation.
2. The *Model Integration Concept* is an approach for translating between program code and software design models using translation patterns.
3. The *Intermediate Architecture Description Language* is an intermediate language to translate between different software architecture languages.
4. *Architecture Model Transformations* based on the Intermediate Architecture Description Language translate between software architecture languages.
5. The tool *Codeling*[1] implements the EIA approach and has been used for evaluation in synthetic and real environments.
6. A *code generation tool* generates concrete model/code translations from translation patterns.

We will present our approach as follows: In Sect. 2 we present our approach. Codeling is presented in Sect. 3. We describe the evaluation and discuss its results in Sect. 4. Related work is presented in Sect. 5, before we conclude in Sect. 6 and present the future work.

2 The Explicitly Integrated Architecture Approach

The *Explicitly Integrated Architecture (EIA)* approach extracts architecture models from code and propagates model changes back to the code. Using the proposed approach, architecture model information is integrated with program code. The code will be structured in a way that architecture meta model and model elements and their properties can reliably be extracted and changed. Non-architectural code is not overwritten when model changes are propagated.

Figure 1 sketches an overview of the approach. Artifacts of the approach are represented with rounded boxes and translations between these artifacts with arrows. The parts of the approach are used to bidirectionally translate between program code and a specification model expressed in an architecture specification language. They are underlined in Fig. 1.

1. *Program Code:* the implementation of a software following the standards of an architecture implementation language
2. *Implementation Model:* an abstract model view upon the program code, which complies to an architecture implementation language
3. *Translation Model:* an intermediate model view for translating between an implementation model and a specification model
4. *Specification Model:* a specification of architectural concerns using an architecture specification language

[1]The tool *Codeling* and its accompanying code generator are available at https://codeling.de as open source, licensed EPL 1.0.

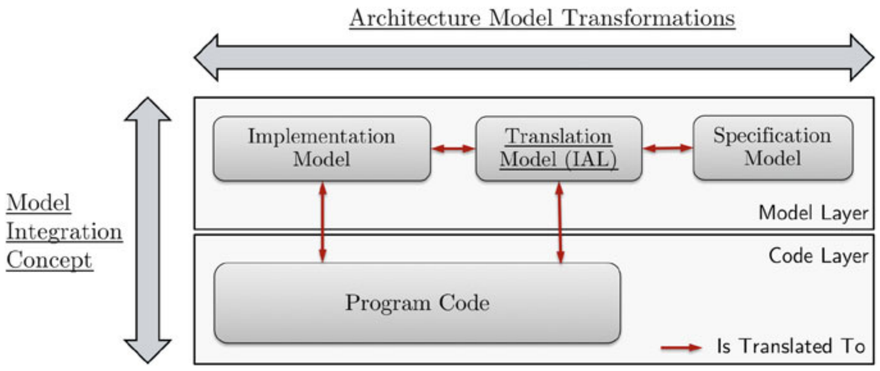


Fig. 1 The view types of the EIA approach and the means to translate between them (underlined)

The EIA is comprised of four parts:

1. The **Model Integration Concept** integrates models and meta models with program code. It is used to create well-defined translations between program code structures, model elements, and meta model elements. It extracts arbitrary design models from code and propagates changes in the models to the code again.
2. The translation model reduces the number of translation definitions required between architecture implementation and specification model languages. We define the **Intermediate Architecture Description Language (IAL)** to express translation models.
3. **Architecture Model Transformations** are used for the translation between models of different languages and for transformations within models of the IAL.
4. The **Explicitly Integrated Architecture Process** describes how these areas are used to achieve the overall objective.

While the Model Integration Concept would suffice for extracting architecture models from code, it does not provide the necessary flexibility to handle new and evolving languages with different features. The IAL and the transformations are required to fulfill these objectives.

2.1 Explicitly Integrated Architecture Process

The Explicitly Integrated Architecture Process [11] is visualized in Fig. 2. It starts from program code that complies to an implementation model, including code that has not been developed using the process yet. An empty code base can also be a start for green field development. The process defines three main steps for each direction. For extracting a specification model, the following steps are executed:

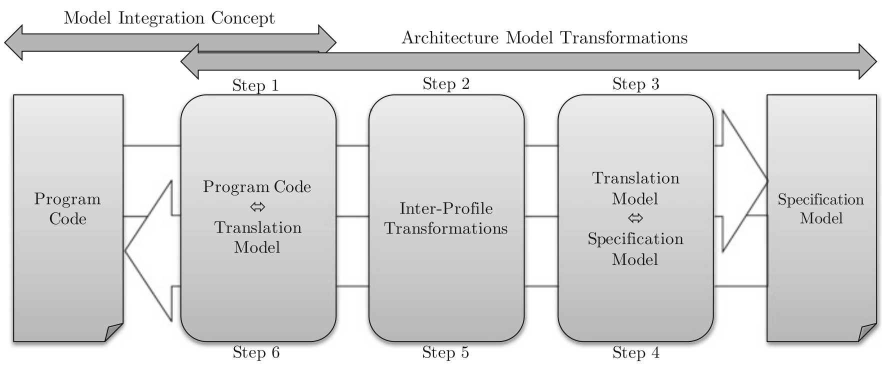

Model Integration Concept
Architecture Model Transformations
Step 1
Step 2
Step 3
Program Code
Program Code ⇔ Translation Model
Inter-Profile Transformations
Translation Model ⇔ Specification Model
Specification Model
Step 6
Step 5
Step 4

Fig. 2 Overview of the Explicitly Integrated Architecture Process

Step 1 extraction of a translation model from the program code via an implementation model;
Step 2 preparation of the translation model according to the features of the involved languages;
Step 3 translation of the translation model into a specification model.

For propagating the model changes to the code, reverse steps are executed:

Step 4 translation of the specification model into a translation model;
Step 5 preparation of the translation model according to the features of the involved languages;
Step 6 integration of the translation models with program code via an implementation model.

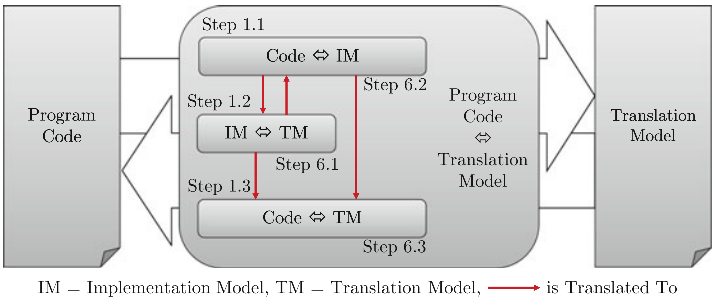


Fig. 3 Details of the steps 1 and 6 of the Explicitly Integrated Architecture Process

The steps 1 and 6 for model/code transformations are detailed in Fig. 3. Step 1 creates a translation model based on the program code. First, step 1.1 translates the code into an implementation model using the Model Integration Concept. Then, step 1.2 creates a translation model from the implementation model using architecture model transformations. Step 1.3 adds architecture information in the program code that cannot be expressed with the implementation model language to the translation model using the Model Integration Concept. To propagate changes in the translation model to the code, step 6.1 first translates the transformation model into an implementation model, before step 6.2 changes the code according to the changed implementation model. Step 6.3 then changes the code according to the translation model, with the architecture information that the implementation model language cannot express.

During the following steps, trace links [35] are created or used: Trace links between code and model elements are created in steps 1.1 and 1.3, and between model elements of different modeling languages in the steps 1.2 and 3. That is, in all steps from the code to the architecture specification model, trace links are created. Step 4 propagates the changes in the specification model to the translation model. The transformations require the trace links of step 3 to identify which model elements were created in step 3 and to identify changes. Step 6.1 uses the trace links of step 1.2 to propagate model changes in the translation model to the implementation model. The steps 6.2 and 6.3 use the model/code trace links of the steps 1.1 and 1.3, respectively, to identify the affected code elements. That is, during each stop during the change propagation, the corresponding traces from the model extraction are used.

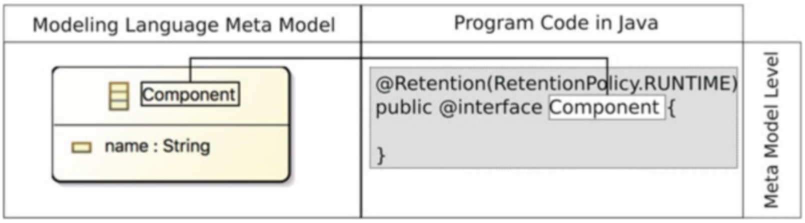
2.2 Model Integration Concept

The *Model Integration Concept (MIC)*[2] describes how model information is integrated with program code. Models in the term of this approach are always based on meta models. Other models, such as mathematical functions, are not meant here. In Fig. 1 the concept provides vertical integration. It is used to integrate and extract architecture model information from an implementation model and the translation model with/from program code. For doing so, the MIC defines bidirectional formal mappings between program code structures and an implementation model expressed in a meta model of an architecture implementation language. As an example, a Java class that implements a specific marker interface might represent a component, and static final fields within this class definition represent attributes of this component. With the MIC, the code is statically analyzed for program code structures that identify implementation model elements. Changes in the model are propagated to the code, following the mapping definitions.

Figures 4 and 5 show example mappings. Figure 4 shows a meta model/code mapping for the class *Component* with the attribute *name* of the type *String*. Meta model elements are shown on the left side. The right side shows Java program code, which represents this meta model element. The meta model class is represented

[2]Some of the ideas behind the Model Integration Concept were first described in [14].

with the declaration of an annotation with the name `Component`. The attribute *name* is not declared in the meta model/code translation. This attribute is subject to the model/code translation in Fig. 5. The declaration of the annotation has a meta annotation `Retention`, declared `RUNTIME`, which means that the declared annotation will be part of the compiled byte code and is processable in the running system.

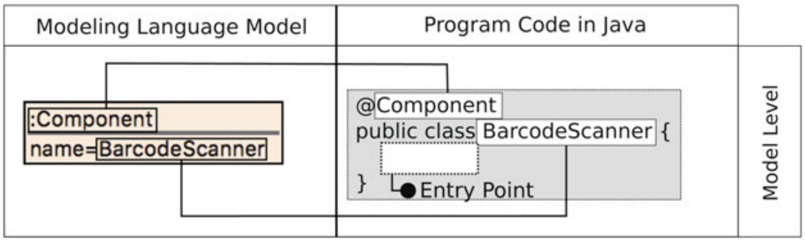


```
@Retention(RetentionPolicy.RUNTIME)
public @interface Component {

}
```

Fig. 4 An example of a meta model/code mapping

Figure 5 shows a model/code mapping for a model that instantiates the given meta model. The left side shows an instance of that meta model, a single object of the *Component* class, with the name `BarcodeScanner`. The right side shows their Java program code representation. The program code declares a type `BarcodeScanner`. The annotation `Component` is attached to the type. The type's body is a so-called *entry point*. That is, arbitrary code, such as attributes and operations, can be added here without breaking the model/code relationship.



```
@Component
public class BarcodeScanner {
}
```

Fig. 5 An example of a model/code mapping

Such (meta) model/code mappings in the MIC are the basis for generating three types of artifacts:

1. *Bidirectional transformations* between model elements and program code build a model representation based on a given program code, so that developers can *extract* an integrated model from program code. They also have to *propagate* model changes to the code, i.e., create, delete, and change program code based on model changes. It has to be ensured that these translations are unambiguous.

2. *Meta model/code translation libraries* are program code structures, which represent meta model elements. In the example, this is the annotation declaration. This program code can be generated once. The results can be used as libraries, as long as the meta model does not change.
3. *Execution runtimes* can be generated for model elements. These runtimes manage the operational semantics of integrated models.

2.2.1 Integration Mechanisms

Integration mechanisms are templates for model/code mappings. They describe a mapping between program code structures and symbolic meta model elements or symbolic model elements. Each comprises a meta model/code mapping for translating a meta model element type and a corresponding model/code mapping for translating instances of that element. Integration mechanisms can be instantiated by applying them to a specific meta model or model, i.e., by replacing the symbolic elements with specific elements.

We identified 20 integration mechanisms by building upon existing Java-based architecture implementation languages such as JEE [23] or OSGi [33]. Some are based on the work of Moritz Balz [4]. Figures 4 and 5 show mappings for the integration mechanism *Annotation Type* for translating objects in the model using the Component as an example. Further mechanisms exist for objects (e.g., marker interfaces), attributes (e.g., constant static attributes), and references (e.g., annotated references to other objects). Konersmann's PhD thesis [12] formally describes the current set of integration mechanisms for Java with formal definitions, examples, and a discussion of the effects and limitations of each mechanism.

The integration mechanisms are an important part of the MIC to reduce the effort for developing bidirectional model/code transformations and execution engines. For integration mechanisms, reusable generic code generators for transformations, meta model code libraries, and execution engines have been developed. When they are instantiated with meta model elements, the respective code can be generated (see the tool support described in Sect. 3) for the concrete mappings.

2.2.2 Operational Semantics

Two types of operational semantics exist for model elements in the context of the MIC [13]:

1. Language semantics can be implemented in the runtime. Here each instance of a model element has equal semantics, e.g., when a component is instantiated, it is registered at a registry. These semantics apply to each component instance.
2. Model semantics can be implemented within the individual model/code structures. Here each instance of a model element has individual semantics, e.g., when

a component is instantiated, individual initialization code should be called, which can be defined by a developer.

In the latter, for executing such implementation details, translations within the MIC may declare *entry points*. Entry points may contain arbitrary code, which is not considered a part of the bidirectional model/code translation. The initialization code stated above can be implemented within operations provided by the model/code structures. An execution runtime will then execute these operations.

2.3 Intermediate Architecture Description Language

The Intermediate Architecture Description Language (IAL) mediates between architecture implementation models and architecture specification models. It has the role to increase the interoperability of the approach with different specification and implementation languages. The IAL has a small core with the common elements of architecture languages [21]. The core is extended with a variety of stereotypes to represent, e.g., different kinds of interfaces, component hierarchies, or quality attributes. Models expressed in the IAL are called *translation models*.

The core comprises the following elements: The *Architecture* is the root node that represents a software architecture comprising interconnected components. The class *ComponentType* represents a named component type. *Interfaces* can be used as an abstract definition of named interfaces for component types. Component types can provide and require interfaces. *Component Instances* represent the runtime view on components' types. The provision and requirement of interfaces is instantiated, respectively.

Profiles add further concerns to the architecture language. Such concerns include, e.g., different types of connectors, component hierarchies, types of interfaces, or quality aspects. Profiles can be categorized regarding their abstract concern, e.g., the profiles *Flat Component Hierarchy* and *Scoped Component Hierarchy* both handle the abstract concern of the component hierarchy, or *Time Resource Demand* and *Security Levels* both handle software quality concerns. Some categories are mandatory, meaning that at least one profile has to be used when an architecture is described. One kind of component-type hierarchy must be chosen. Some categories contain only optional profiles, e.g., no software quality profile is necessary to be used.

Figure 6 shows the profiles of the IAL and their interrelationships regarding their interpretation. The rectangles are categories of profiles, which share an abstract concern. The rectangles with rounded corners represent profiles. Mandatory categories (which have a solid border in Fig. 6) require at least one profile to be used. The profiles and their application are described in detail in Konersmann's PhD thesis [12]. Considering the objective of the language, in the future more profiles and categories can be added.

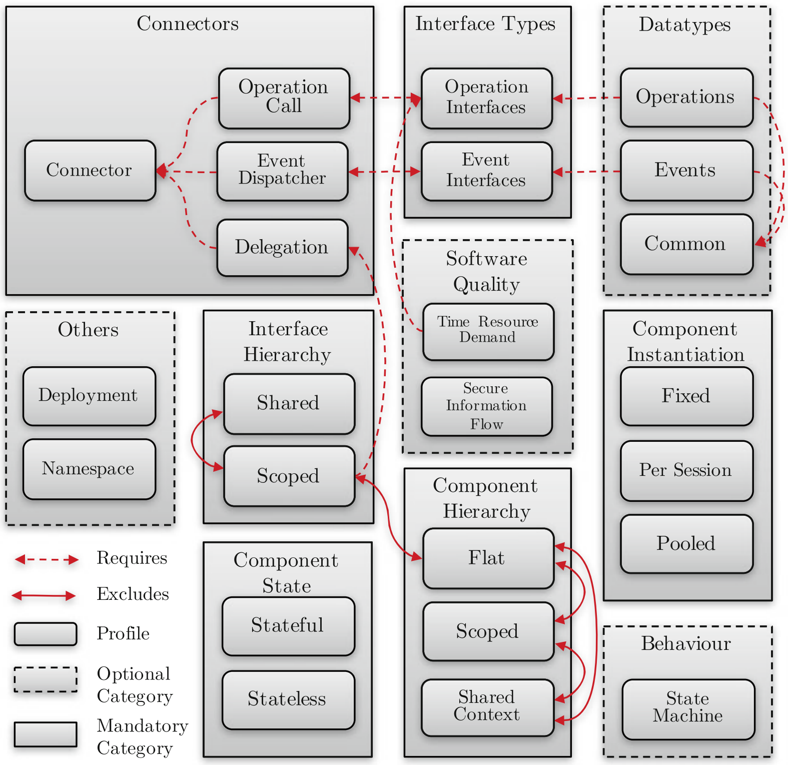


Fig. 6 An overview of profiles of the Intermediate Architecture Description Language and their interrelationships

2.4 Architecture Model Transformations

Two types of architecture model transformations are part of the EIA approach. First, transformations between architecture specification languages and the IAL as well as transformations between architecture implementation languages and the IAL are used to create a mapping between architecture specifications and implementations on a model level. Second, transformations within the IAL allow for translating between different related profiles of the IAL. In Fig. 1 the architecture model transformations provide the horizontal integration.

2.4.1 Transformations Between Architecture Languages

Transformations between architecture languages and the IAL can be defined in any model transformation technique that allows for exogenous transformations, i.e., transformations from one language to another [19].

In the implementation of the tool Codeling, we use triple graph grammars (TGGs) [26] based on attributed, typed graphs [6]. In these typed graphs, the graphs are considered models and the type graphs are meta models. A TGG describes triple rules, which declare how two graphs can be produced in alignment. They comprise a source graph, a target graph, and a correspondence graph. In our approach, one of these graphs is always a model expressed in the IAL. The triple rules are used to derive production rules and propagation rules in the context of our approach. Production rules describe how to construct a target graph from a given source graph. Propagation rules describe how to propagate changes in a target model back to the source model.

In our approach, TGGs are used:

1. to produce a translation model from an implementation model,
2. to produce an architecture specification model from the translation model,
3. to propagate changes in the specification model to the translation model, and
4. to propagate changes in the translation model to the implementation model.

We developed TGG rules for the Palladio Component Model (PCM) [5], a subset of the UML, JEE [23], and a project-specific architecture implementation language. Details on the specific rules are given in Konersmann's PhD thesis [12].

2.4.2 Transformations Within the IAL

The IAL comprises several profiles that are mutually exclusive (see Sect. 2.3). As an example, when an architecture is modeled with hierarchical component types and, at the same time, as flat component-type hierarchy, this information would be inconsistent. Nevertheless, an architecture can be expressed in an architecture implementation language that defines component-type hierarchies and should be viewed in an architecture language that can only model flat hierarchies. To respect these situations, the approach defines transformations between mutually exclusive IAL profiles, which are called *inter-profile transformations*. In the EIA process, both profiles are used in the IAL at the same time, leaving inconsistent information in the translation model. The architecture model transformations toward the target specification language only use the information they can handle, leaving trace links in the process. When changes in the specification model are translated into the translation model, the missing information is restored by exploiting the trace links. Details on the inter-profile transformations are given in Konersmann's PhD thesis [12].

3 Tool Support

We developed the following tools for the approach. Figure 7 gives an overview of the tools and their input and output. *Codeling* is the tool for executing the Explicitly

Integrated Architecture Process. It creates architecture specification model views upon program code, propagates changes in the model to the code representation, and can migrate program code from one architecture implementation language to another. Libraries in the context of Codeling support the development and execution of model/code transformations and architecture model transformations, including an automated selection and execution of inter-profile transformations.

The *code generation tool* exploits the definition of integration mechanisms for the Model Integration Concept. The tool's user maps integration mechanisms to elements of meta model elements. We developed a library of abstract transformations and execution runtimes for integration mechanisms, to decrease the effort for creating specific transformations and execution runtimes, where integration mechanisms can be applied. Based on the library of these abstract transformations and execution runtimes, the code generation tool then generates a meta model code library, model/code transformations, and execution runtime stubs.

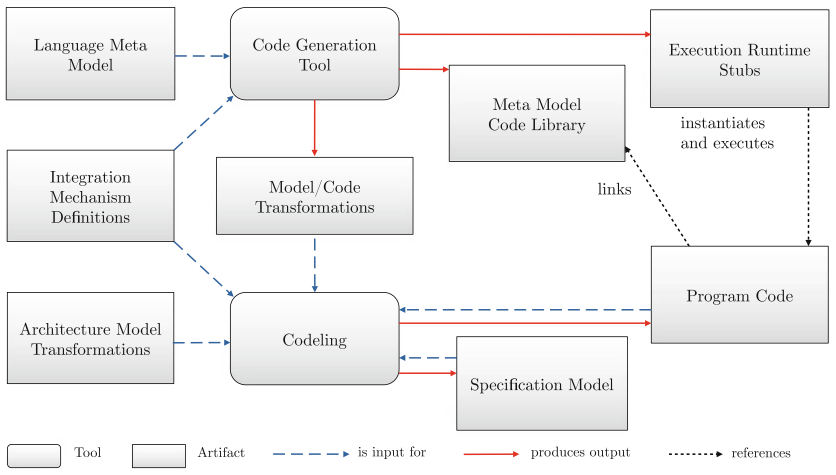


Fig. 7 An overview of the tools and the artifacts they use as input and output

3.1 Codeling

Codeling is a development and execution platform for extracting architecture models from code and for propagating changes in the models back to the code. It implements the process presented in Sect. 2.1 and provides a set of libraries to support the development of concrete transformations between program code and models.

Codeling is implemented with a modular architecture, which allows integrating further architecture implementation or specification languages, using different types of model/code or model-to-model transformation technologies.

Three types of use cases can be executed with, or are supported by Codeling:

1. For a code base, architecture specification models of different languages can be **extracted**, and **changes in these models can be propagated to the code**.
2. When a set of rules or a normative architecture exists, Codeling can extract the actual architecture from the code to **check the architectural compliance of the source code**.
3. By translating program code into the IAL and back to *another* architecture implementation language, Codeling can be a part of **software migration**. Only architecturally relevant code can be migrated. Further program code needs to be migrated by other means.

3.1.1 Model/Code Transformations

Libraries for the steps 1.1 (Code to Implementation Model), 1.3 (Code to Translation Model), 6.2 (Implementation Model to Code), and 6.3 (Translation Model to Code) of the process comprise a hierarchy of abstract classes to assist the development of concrete model/code transformations. The class hierarchy supports the transformation of Java code into Ecore [30, Chapter 5] based models and to propagate changes in the Ecore models to the Java code using the Eclipse Java Development Tools.[3]

A transformation of code into a model representation is executed in a tree structure, following containment references in the meta model: A root transformation object first translates the root code element—usually the projects at the given paths—into a model representation, the root node of the targeted model. The transformation objects store references to the code and the corresponding model element, effectively creating a trace link between the code and the model.

After the translation, the transformation object is added to a transformation object registry. This registry can be used later to retrieve model elements, which represent specific code elements or vice versa. At last, the transformation creates child transformation objects for its attributes and containment references and adds them to a pool of tasks.

Transformations for classes have transformations for their attributes and containment references as child transformations. Transformations for attributes have no child transformations. Reference transformations, including containment reference transformations, have transformations for their target objects as child references. If a containment reference is translated from a code to a model representation, the targets of the reference do not exist, because they have not been translated yet.

[3]Eclipse JDT—https://www.eclipse.org/jdt/.

Transformations for non-containment references first wait for the transformation of the target objects.

3.1.2 Model-to-Model Transformations

To execute model-to-model transformations between an architecture implementation or specification language and the IAL (the exogenous transformations in steps 1.2, 3, 5, and 6.1), Codeling provides helper classes to execute HenshinTGG [15] rules to create the target model or to propagate changes from a changed target model to a source model. Other technologies for defining exogenous model-transformations can also be applied, as long as change propagation is possible.

Codeling uses Henshin [6] rules for inter-profile transformations in step 2 and step 4. It derives the information about which inter-profile transformations have to be executed during the process execution. The information is based on the IAL profiles that are used in the HenshinTGG rule definitions between the IAL and the architecture implementation and specification languages.

3.1.3 Process Execution

Figure 8 shows a simple example of the Explicitly Integrated Architecture Process in action. In this example an EJB Session Bean `CashDesk` is added to an existing bean `BarcodeScanner`. The `CashDesk` is declared to be the parent of the `BarcodeScanner`.

(1) shows the program code for the bean `BarcodeScanner`.

(2) The implementation model is built by scanning the program code for mapped structures based on the Model Integration Concept. In this example a type declaration with an attached annotation `Stateless` is identified. The name of the declared type is identified as the name of the bean.

(3) The implementation model is translated into a translation model, an instance of the IAL.

(4) The translation model is translated into a specification model. The specification model in the example is represented using a UML component diagram. In an evolutionary step, a parent component named *CashDesk* is added.

The changes are propagated to the code as follows:

At **(5)** the architecture specification model is translated into the translation model. A new *ComponentType* with the name *CashDesk* is created, with a stereotype that allows to add children to a component type.

(6) The translation model is translated into an implementation model. In this model the hierarchy cannot be represented, because the EJB specification does not define component hierarchies.

At **(7)** the program code is adapted corresponding to the changes in the implementation model. That is, the type `CashDesk` is created.

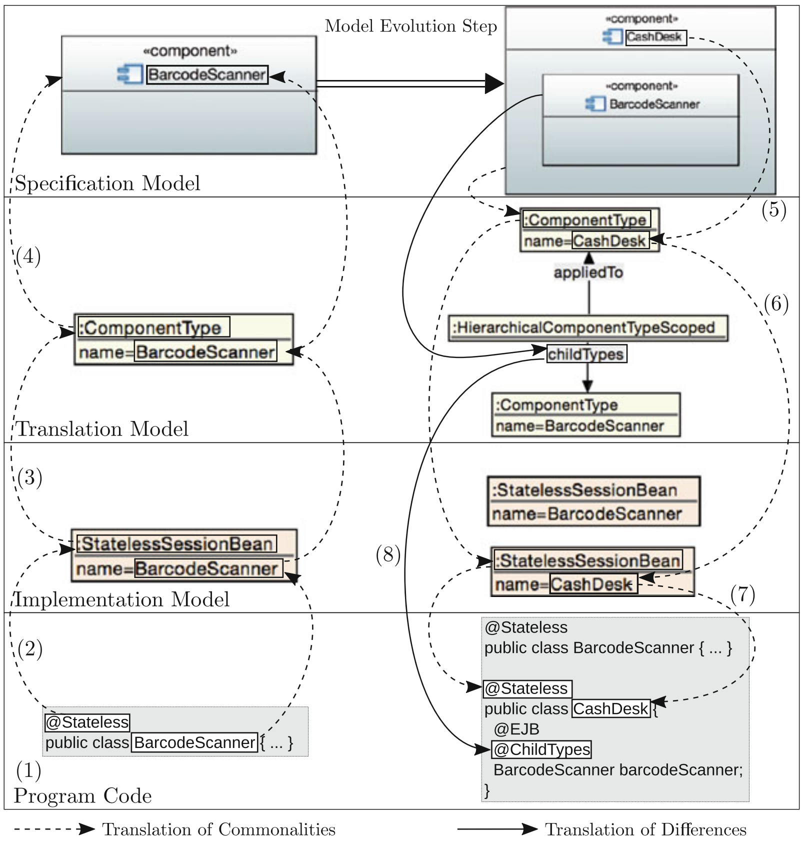


Fig. 8 An example of the Explicitly Integrated Architecture Process

(8) The architecture information that has no representation in the implementation model is translated into the code using the Model Integration Concept. In this example, the hierarchy is translated as a field in the Java-type declaration `BarcodeScanner` with the annotation `EJB`. This is an annotation of the EJB framework, which specifies that an instance of the bean `BarcodeScanner` has to be injected. Additionally, this field has the annotation `ChildTypes`, which marks the reference an instance of the *childTypes* reference. To remove the hierarchy, the code could be translated into a model using the process. As an alternative, the respective code element could be removed.

It should be noted that the hierarchy could also have been created in the terms of the approach by simply adapting the code accordingly, because the models can be derived automatically.

3.2 Code Generation Tool

Codeling provides libraries and components for defining and executing translations as defined by the Explicitly Integrated Architecture approach. This includes the definition of bidirectional model/code transformations and meta model code libraries, which, e.g., include annotations or interface definitions used by the transformations. Developing such transformations and meta model code libraries can be cumbersome and error-prone. The integration mechanisms can be used as templates for generating model/code transformations and program code libraries for meta model code. The *Code Generation Tool* generates the following artifacts (see Fig. 7):

1. A meta model code library with code that represents architectural meta model elements.
2. A set of model/code transformations for extracting a model from code and propagating changes from the model to the code, which follow the integration mechanisms.
3. Execution runtime stubs for the program code that is generated by the aforementioned transformations.

To generate these artifacts, the generator requires two items as input:

1. A *language meta model*, which describes the architecture implementation language concepts in Ecore.
2. A *mapping between meta model elements and integration mechanisms*, which describes which integration mechanism is to be instantiated for each meta model element.

As the generated transformations follow the integration mechanisms, they reliably extract model elements from the code and propagate changes from the model back to the code. A hierarchy of abstract classes have been prepared to add new translations for further mechanisms. However, especially for existing component frameworks, model/code mappings might not follow the existing integration mechanisms or require slight deviations from existing mechanisms. For these mappings, translations have to be programmed manually, using the aforementioned hierarchy of abstract classes. If a mapping only deviates slightly from an existing mechanism, the generated translation can be manually adapted. In each of the use cases shown in Sect. 4, an adaptation of generated mechanisms was necessary. For both use cases, some translations had to be defined manually.

3.3 Execution Runtimes

Within Codeling we developed a framework for implementing execution runtimes for models that are integrated with program code [13]. These runtimes are based on Java's reflection mechanism to analyze the code, inject instances, and invoke operations at run-time. The framework comprises a set of abstract classes, which have to be extended for implementing specific runtime classes. For each implemented integration mechanism, an abstract runtime class exists. For implementing an execution runtime for a meta model, we map integration mechanisms to each class, reference, and attribute declared in the meta model. The code generation tool then generates an execution runtime class for each of these pairs. This creates a tree of execution runtimes, one for each meta model element. These generated execution runtime classes contain functionality to instantiate objects, set reference targets, and attribute values based on the underlying integration mechanism. The operational semantics of the modeling language can be implemented in these classes. The runtimes can effectively be seen as interpreters for the integrated models. They can also trigger operations to invoke code, which is declared in an entry point, and therefore trigger the execution of model semantics, which are expressed in program code (see Sect. 2.2.2).

4 Evaluation

For evaluation purposes, the approach has been applied in four use cases: The first use case translates the JEE program code of an e-assessment tool into a UML model with components, interfaces, operations, and their interconnection and propagates changes in the model back to the code. The second use case translates the program code of the Common Component Modeling Example (CoCoME) [10] into a subset of the Palladio Component Model (PCM) [5]. The third use case translates the CoCoME system into the UML as another architecture specification language. The fourth use case translates the CoCoME system into JEE as another architecture implementation language. In the following sections we will elaborate on the first and second use cases. All use cases are described in detail in Konersmann's PhD thesis [12].

4.1 Use Case JACK 3

In the first use case, the development of the e-assessment tool *JACK 3* is supported by generating an architectural view in the UML specification language. JACK 3 is the designated successor of the e-assessment tool JACK 2 [31], developed at the working group "Specification of Software Systems" (S3) of the institute paluno of the University of Duisburg-Essen, Germany. Its predecessor is used in the teaching and assessment of various disciplines, including programming, mathematics, and micro-economics. JACK 3 comprises two parts: a back end written in Java using the Eclipse platform as architecture implementation language, and a front end written in Java, based on the Java Enterprise Edition 7. The front end defines a user interface, data definitions, and business logic for e-assessments. The back end evaluates solutions against the defined standard solutions. It is not changed during the development of JACK 3. Therefore, this use case focuses on the front end for supporting the development of JACK 3.

This use case translates a subset of JEE technologies (EJB 3.2, CDI 1.2, and JSF 2.2) into a UML component diagram and back. JEE is a standardized set of frameworks and APIs for enterprise systems. Application servers act as execution runtimes, which analyze the code before execution, e.g., to instantiate and interconnect JEE components (beans) or to provide web-based interfaces (e.g., REST interfaces or web services). The use case also adds time resource demand information to operations in the specification model. JEE as architecture implementation language cannot express this information. This use case shows that such differences are taken into account by the approach.

For this use case, we generated and adapted model/code transformations for JEE code. The architecture implementation language JEE defines source code structures that represent architectural concepts, and platforms that execute the code. As no Ecore meta model for JEE was publicly available, we created a meta model of JEE with 11 classes, 16 references, and 22 attributes, which represent structural elements, i.e., interrelated beans and data entities in namespaces and archives, and their operations and attributes, as part of the use case.

We assigned integration mechanisms to 6 classes, 11 references, and 12 attributes. Some mechanisms had to be adapted to match the requirements of JEE-compliant code. For the other elements, we developed individual transformations based on the abstract transformations in Codeling. The project has 12307 lines of Java code (NCLOC). The resulting model has 2446 objects with 45,354 reference instances. On a computer with an Intel i5-8250U CPU and 16 GB memory on Ubuntu Linux 18.04.2 and OpenJDK version “11.0.4” 2019-07-16, the median model extraction time is about 5 s.

Furthermore, model-to-model transformations between JEE and the IAL were defined using HenshinTGG. Our TGG for JEE and the IAL comprises 35 rules. Another TGG was developed to translate between the IAL and a subset of the UML. That TGG comprises nine rules.

The extracted architecture of JACK comprises 36 UML components that are interconnected via 33 interfaces in a correctly layered architecture. It helped the developers to understand that they implemented their architecture correctly and to see how the components in the layers are interconnected. The resulting UML model is changed in the use case by adding, changing, and deleting elements. Codeling automatically changes the code according to the changes in the UML model.

Propagating a single renaming of a component, which is translated with the Type Annotation mechanism, from the UML model to the code takes about 5 s on the computer described above. The tool uses code refactoring operations to propagate this code change. For the model extraction, the main drivers for execution time are the size of the code to be analyzed and the size of the resulting model. For propagating model changes, the main drivers are the size of the model and the number of changes.

4.2 Use Case CoCoME in PCM

In the second use case, the program code of the Common Component Modeling Example (CoCoME) [10] is translated into a subset of the Palladio Component Model (PCM) [5]. CoCoME has been developed as a benchmark for comparing software architecture languages. The original CoCoME benchmark artifacts provide the context, the requirements, the design, and the implementation of a system. The system drives the business for an enterprise that runs multiple stores. Each store contains multiple cash desks in a cash desk line [10].

CoCoME does not follow a standardized implementation framework like JEE but comes with a custom style of implementing components and their interaction. This use case shows how Codeling can be applied to software that does not follow the coding conventions of industry standard platforms but follow project-specific coding conventions. In CoCoME components are implemented using Java classes with coding conventions and a set of methods to implement, alongside with a set of adjacent classes to refine the component. The code structures are not systematically implemented, so that minor changes had to be made to the code base, to define working mappings.

A meta model for the implementation structure had to be created first, before mappings could be implemented. Our meta model of CoCoME's structure contains 12 classes, 13 references, and 1 attribute. 4 classes and 4 references could be translated using integration mechanisms or variations thereof. For the other elements, we developed individual mappings based on the abstract transformations in Codeling. The project has 9425 lines of Java code (NCLOC). The resulting model has 121 objects with 1357 reference instances. On the computer described above, the median model extraction time is about 0.6 s. The first extraction takes longer (about 3 s), because the Ecore environment and the meta model need to be loaded.

4.3 Further Use Cases

The third use case translates the CoCoME code into a UML model as another architecture specification language. For this use case, we reused transformation rules between the IAL and the UML. A UML architecture description is part of the CoCoME artifacts. We extracted the UML architecture from CoCoME using our approach and compared it to the UML architecture model in the CoCoME artifacts. As the code conventions in CoCOME are not systematically implemented, the extracted architecture initially did not match with the normative architecture. Minor changes had to be made to the code base, to extract a matching architecture. This use case shows not only that architecture specifications in different languages can be extracted from the code, it also shows that, with a normative architecture specification at hand, Codeling can be used to validate the implemented architecture against an architecture specification.

The fourth use case translates the CoCoME system into JEE as another architecture implementation language. The use case shows how Codeling can be used as a part of software migrations between different architecture implementation languages. In this use case, only the steps 1 (Program Code to Translation Model), 2 (Inter-Profile Transformations), and 6 (Translation Model to Program Code) are executed. Step 1 of this use case is equal to step 1 of the CoCoME use case. In step 2 instead of targeting the UML for translation, the architecture implementation language JEE is chosen. JEE provides only a flat component hierarchy, while the CoCoME architecture implementation language uses a scoped component hierarchy. Therefore, other inter-profile transformations are executed. Steps 3 to 5 are omitted in this use case, because no architecture specification language is involved. In step 6 a TGG is applied that is close to the one that has been used in the JACK 3 use case. The JACK 3 team had special conventions how to handle Java EE code, which did not apply in the CoCoME use case. Furthermore, the same architecture implementation language meta model and model/code transformations were used as in the JACK 3 use case.

The translation results in a new project within the Eclipse IDE, with an architecture skeleton of CoCoME in JEE 7. The parent–child relationship between components in the CoCoME architecture cannot be implemented in JEE, because JEE uses flat component hierarchies. In step 6, therefore a new model/code transformation has been added, to integrate the parent–child relationship between components. A Java annotation `@Child` now indicates whether a referenced bean is the child of another bean.

4.4 Discussion

The execution of the case studies suggested that the transformations can require considerable execution times. Translating from the code to a UML representation in the JACK use case required about 225 s on the computer stated above. The backwards translation of the changed model to changes in the code required about 330 s. No information about operation parameters was included in the code-to-model translations, because the translation required multiple hours when parameters were also translated. The CoCoME case studies required 170 s for the translation to the PCM, and 140 s to the UML. The migration of CoCoME to JEE required about 210 s on the same machine. Table 1 gives an overview of the use case sizes and their performance. The number of objects and references show the IAL model size for comparison. This does not include attribute values.

The tool Codeling uses a series of code-to-model, model-to-code, and model transformations, including triple graph grammars (TGGs), to achieve its goals. TGGs can be performance intensive. Forward and backward translation rules from TGGs have a polynomial space and time complexity $O(m \times n^k)$, where m is the number of rules, n is the size of the input graph, and k is the maximum number of

Table 1 An overview of the size and performance of evaluation use cases

Use Case	No. of (Objects IAL)	No. of References (IAL)	Runtime Code to Model [s]	Runtime Model to Code [s]	Runtime Code to Code [s]
JACK 3 to UML	1644	2141	225	330	555
CoCoME to PCM	361	507	170	–	–
CoCoME to UML	361	507	140	–	–
CoCoME to JEE	361	507	–	–	210

nodes in a rule [27]. Therefore, it is expected that an increasing model size implies a higher resource demand.

A resource demand test was executed during the development of Codeling, to find the limitations of the implementation. The detailed results of the resource demand test can be found in Konersmann's PhD thesis [12]. The translation between code and specification models can be separated into the process steps for performance analysis. The main driver for time-resource demand are the TGG rules, which took about 87% of the translation time during the extraction of a synthetic model with 300 model objects. A major observation was that the TGG rules were executed on a single core. We see a high potential for reducing this time by using other model transformation techniques, especially those using parallel execution (e.g., ATL [34]).

The definition of model/code transformations for Codeling requires programming in Java or Xtend with JDT and Ecore. The code generator supports the development of model/code transformations by automatically creating transformations for meta model elements that can be translated with integration mechanisms. In our use cases some transformations have to be developed manually. Although we implemented a supporting framework (see Sect. 3), the development of transformations is not trivial. To tackle this problem, for future work we develop a language for describing mappings between model elements and source code for the Model Integration Concept, especially considering integration mechanisms. The objective is to make the development of new integration mechanisms and individual model/code transformations easier.

5 Related Work

An extensive study of literature is part of the PhD thesis. In this section we give an excerpt of the related work presented there, enriched with some more recent findings. The approach at hand is a method for extracting architecture models from code and to propagate changes in the model to the code. Pham et al. [24] describe an approach to synchronize architecture models and code. They focus on UML components and state machines as behavior models. *ReflexML* of Adersberger and Philippsen [2] is a mapping of UML component diagrams to program code artifacts, enriched with a set of consistency checks between the model and the code. Already in 1995 Murphy et al. [20] presented an approach that creates links between higher-level model elements and program code files. The general ideas behind this approach are close to the Explicitly Integrated Architecture approach. In both approaches a semantic gap between program code elements and higher-level models has been identified and should be bridged with a mapping. Codeling maps the elements on a much more fine-grained level. While these approaches allow for mapping architectural models to code elements, they are limited to a pair of languages.

The goal of *model extraction* methods is to create a model of a software design based on code or execution traces. In general, models can be extracted statically from source code and/or dynamically from execution traces [25], e.g., sequence diagrams [29], state machines [28], or software architecture from code [8]. Model extraction approaches do not aim to propagate changes to the code, but they are used for model-based analysis and communication. The MIC can be seen as a model extraction approach, extended with delta-based code generation and adaptation means.

MoDisco [7] is a tool for model-driven reverse engineering. It allows for extracting domain-specific models from code and transforming them to other code. It also comprises a Java meta model as a basis for model transformations and model-to-code transformations. MoDisco does not target architecture descriptions directly and therefore has no mechanisms for handling the differences between languages. MoDisco could be used as a basis for bidirectional model/code transformations in Codeling, but it would be necessary to add concepts for handling integration mechanisms.

Co-evolving models and code means to propagate deltas between models and code in both directions. Existing approaches use, e.g., guidelines for manual implementations [9]. Langhammer and Krogmann [17] describe an approach for the co-evolution of models of the Palladio Component Model (PCM) and Java program code, including architectural structure and abstract behavior. Langhammer [16] describes rules for correspondence relationships between the architecture model and the program code during changes on either side. Arbitrary code within methods is preserved during model-to-code change propagation. The approach is semi-automated, meaning that in cases where full automation is not possible, a developer is asked to describe how consistency can be preserved. This approach creates a specific mapping between a subset of the PCM and Java code. The MIC can conceptually be used for arbitrary meta models and programming languages, although the tools (see Sect. 3) currently support Java only.

A special case of model/code co-evolution is roundtrip engineering (RTE) [3], a method where two representations of program code are maintained together: in a textual syntax and in a—usually graphical—model syntax. RTE offers a bijective projection between the textual and the model syntax. The models used in roundtrip engineering are close to the code structures, e.g., UML class diagrams or data models. While the MIC can handle such models and their code representation, the MIC is designed to handle design models of higher abstraction levels. The MIC does not propagate code deltas to the model and is therefore not a co-evolution approach.

It is a hybrid approach of model extraction and model-code co-evolution, which extracts models and propagates model changes to code.

Balz [4] describes an approach for representing models with well-defined code structures. He defines *embedded models* as a mapping between formal models and program code patterns in a general-purpose programming language. A major contribution of Balz' work is the formal mapping between state machines and process models, and program code. It provides explicit interfaces for implanted code. The MIC is conceptually based on the essential ideas of the *embedded models* approach. Balz defines these two specific types of models for which embedded models can be used. The MIC generalizes this approach, to be usable with arbitrary meta models. With the definition of meta models as a basis, the MIC can declare integration mechanisms as templates for program code structures for reusing mappings and generate model/code transformations and execution runtime stubs.

6 Conclusions and Future Work

In this chapter we describe the Explicitly Integrated Architecture approach and its tool support. The approach extracts software architecture specifications, e.g., UML models, from source code, and propagates changes in the model to the code. It therefore creates a volatile architectural view upon the code, reducing the need to maintain two representations of the software architecture. The approach comprises four parts: (1) The Model Integration Concept is a method to extract design models from source code and propagate model changes to the code. (2) The Intermediate Architecture Description Language (IAL) is an intermediate language for translating between architecture implementations and specifications. These different views upon software architecture usually have different features for software architectures. The IAL is prepared to handle these features by using a small core and feature modules. (3) The approach translates between languages using a set of architecture model transformations, including translations between different language features, such as hierarchical and non-hierarchical architectures. (4) The Explicitly Integrated Architecture process uses the aforementioned parts to translate between source code and architecture models.

We evaluated the approach by implementing the tool Codeling accompanied by a code generator and used it in a real software project and on a benchmark software for component modeling. The evaluation has shown that the approach is usable to extract architecture specifications in different languages from a code base and to migrate an architecture implementation to another implementation language.

For future work we will develop a language to make the development of model/code transformations easier, and we evaluate other model-transformation technologies to enhance the overall performance. We are planning to evaluate the approach further on real software systems, including software with different programming languages and of different domains.

References

1. index | TIOBE - The Software Quality Company. http://web.archive.org/web/20200218103554/https://www.tiobe.com/tiobe-index/. Accessed: 2020-02-28.
2. Josef Adersberger and Michael Philippsen. ReflexML: UML-Based Architecture-to-Code Traceability and Consistency Checking. In Ivica Crnkovic, Volker Gruhn, and Matthias Book, editors, *Software Architecture - 5th European Conference, ECSA 2011, Essen, Germany, September 13–16, 2011. Proceedings*, volume 6903 of *Lecture Notes in Computer Science*, pages 344–359. Springer, 2011.
3. Uwe Aßmann. Automatic Roundtrip Engineering. *Electronic Notes in Theoretical Computer Science*, 82(5):33–41, April 2003.
4. Moritz Balz. *Embedding Model Specifications in Object-Oriented Program Code: A Bottom-up Approach for Model-based Software Development*. PhD thesis, Universität Duisburg-Essen, May 2011.
5. Steffen Becker, Heiko Koziolek, and Ralf Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82:3–22, 2009.
6. Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Formal Foundation of Consistent EMF Model Transformations by Algebraic Graph Transformation. *Software & Systems Modeling*, 11(2):227–250, 2012.
7. Hugo Bruneliere, Jordi Cabot, Grégoire Dupé, and Frédéric Madiot. MoDisco: a Model Driven Reverse Engineering Framework. *Information and Software Technology*, 56(8):1012–1032, August 2014.
8. S. Ducasse and D. Pollet. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, July 2009.
9. Thomas Haitzer, Elena Navarro, and Uwe Zdun. Reconciling software architecture and source code in support of software evolution. *Journal of Systems and Software*, 123:119–144, 2017.
10. Sebastian Herold, Holger Klus, Yannick Welsch, Constanze Deiters, Andreas Rausch, Ralf Reussner, Klaus Krogmann, Heiko Koziolek, Raffaela Mirandola, Benjamin Hummel, Michael Meisinger, and Christian Pfaller. CoCoME - The Common Component Modeling Example. chapter 3, pages 16–60. Springer-Verlag, 2008.
11. Marco Konersmann. A Process for Explicitly Integrated Software Architecture. *Softwaretechnik-Trends*, 36(2), 2016.
12. Marco Konersmann. *Explicitly Integrated Architecture - An Approach for Integrating Software Architecture Model Information with Program Code*. phdthesis, University of Duisburg-Essen, March 2018.
13. Marco Konersmann. On executable models that are integrated with program code. In *Proceedings of the 4th International Workshop on Executable Modeling co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), Copenhagen, Denmark, October 14, 2018.*, 2018.
14. Marco Konersmann and Michael Goedicke. A Conceptual Framework and Experimental Workbench for Architectures. In Maritta Heisel, editor, *Software Service and Application Engineering*, volume 7365 of *Lecture Notes in Computer Science*, pages 36–52. Springer Berlin Heidelberg, 2012.
15. Huu Loi Lai. Entwicklung einer Werkzeugumgebung zur Visualisierung und Analyse komplexer EMF- Modelltransformationssysteme in Henshin. Master's thesis, Tecnical University Berlin, May 2013.
16. Michael Langhammer. *Automated Coevolution of Source Code and Software Architecture Models*. Phd thesis, Karlsruhe Institute of Technology, February 2017.
17. Michael Langhammer and Klaus Krogmann. A Co-evolution Approach for Source Code and Component-based Architecture Models. *Softwaretechnik-Trends*, 35(2), 2015. ISSN 0720-8928.

18. Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.
19. T.a Mens and P.b Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152(1-2):125–142, 2006.
20. Gail C. Murphy, David Notkin, and Kevin Sullivan. Software Reflexion Models: Bridging the Gap between Source and High-Level Models. In *IEEE Transactions on Software Engineering*, pages 18–28, 1995.
21. Marco Müller. Applying Formal Component Specifications to Module Systems in Java. Master's thesis, Universität Duisburg-Essen, March 2010.
22. Marco Müller, Moritz Balz, and Michael, Goedicke. Representing Formal Component Models in OSGi. In Gregor Engels, Markus Luckey, and Wilhelm Schäfer, editors, *Software Engineering*, volume 159 of *LNI*, pages 45–56. GI, 2010.
23. Oracle America, Inc. JavaTM Platform, Enterprise Edition (Java EE) Specification, v7, June 2015. http://jcp.org/en/jsr/detail?id=342.
24. Van Cam Pham, Ansgar Radermacher, and Sébastien Gérard. A New Approach for Reflection of Code Modifications to Model in Synchronization of Architecture Design Model and Code. In Slimane Hammoudi, Luís Ferreira Pires, and Bran Selic, editors, *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018, Funchal, Madeira - Portugal, January 22–24, 2018*, pages 496–503. SciTePress, 2018.
25. C. Raibulet, F. Arcelli Fontana, and M. Zanoni. Model-Driven Reverse Engineering Approaches: A Systematic Literature Review. *IEEE Access*, 5:14516–14542, 2017.
26. Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In *Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG '94, Herrsching, Germany, June 16–18, 1994, Proceedings*, pages 151–163, 1994.
27. Andy Schürr and Felix Klar. 15 Years of Triple Graph Grammars. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Proceedings of the 4th International Conference on Graph Transformations*, volume 5214 of *Lecture Notes in Computer Science*, pages 411–425. Springer, September 2008.
28. Tamal Sen and Rajib Mall. Extracting finite state representation of Java programs. *Software and System Modeling*, 15(2):497–511, 2016.
29. Madhusudan Srinivasan, Young Lee, and Jeong Yang. Enhancing Object-Oriented Programming Comprehension Using Optimized Sequence Diagram. In *29th IEEE International Conference on Software Engineering Education and Training, CSEET 2016, Dallas, TX, USA, April 5-6, 2016*, pages 81–85. IEEE, 2016.
30. David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.
31. Michael Striewe. An architecture for modular grading and feedback generation for complex exercises. *Science of Computer Programming*, 129:35–47, 2016. Special issue on eLearning Software Architectures.
32. R. N. Taylor, Nenad Medvidovic, and Irvine E. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 1 edition, January 2009.
33. The OSGi Alliance. OSGi Core. https://osgi.org/download/r6/osgi.core-7.0.0.pdf, April 2018.
34. Massimo Tisi, Salvador Martínez Perez, and Hassene Choura. Parallel execution of ATL transformation rules. In Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter J. Clarke, editors, *Model-Driven Engineering Languages and Systems - 16th International Conference, MODELS 2013, Miami, FL, USA, September 29 - October 4, 2013. Proceedings*, volume 8107 of *Lecture Notes in Computer Science*, pages 656–672. Springer, 2013.
35. Stefan Winkler and Jens von Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Software & Systems Modeling*, 9(4):529–565, September 2010.