On executable models that are integrated with program code

Marco Konersmann
paluno, University of Duisburg-Essen
Essen, Germany
marco.konersmann@paluno.uni-due.de

ABSTRACT

Software development is usually supported by design models. These models and the program code represent overlapping information redundantly. When the implementation is changed, the models are often not changed 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. In this paper we present a method for creating executable models, that are integrated with the program code. The integration allows for an interplay between the model and the program code at run-time, and leaves no room for inconsistencies between the artifacts.

KEYWORDS

executable models, knowledge-carrying code, model integration, model extraction, consistency

1 INTRODUCTION

During development, models are often used to specify design aspects of software. Such design models can be close to the implementation, such as class diagrams, which have equivalent elements in the code. Other design models are more abstract, such as component models or behaviour models like state machines. A specified design is realized by implementation artefacts, e.g. the program code, configuration data, and execution platforms [11, pp. 337].

Design models and implementation concern the same subject, while focusing on different aspects. Design models focus on an abstract view for design, communication, and analysis (cf. [11, Chapter 2]). The implementation focuses on the details of an executable system, with dependencies to the execution platform. Both can be seen as different views on the design with overlapping information which they represent redundantly. But both views also have their own, original information. The implementation, e.g., contains detailed behaviour descriptions and platform dependent code. The model, e.g,. contains performance information for analysis reasons. At the end the views cannot be separated completely, and both views are appropriate for their task.

When the implementation is changed, the models are often not changed along with the code, which introduces inconsistencies between these artefacts. Such inconsistencies imply a risk of misunderstandings and errors during the development, maintenance, and evolution. We therefore propose an approach where the model and program code are tightly integrated. I.e., the model information is represented as specific program code structures, which can be dynamically interpreted, a model representation can be extracted. The design models may have operational semantics. E.g., components can be started and stopped at run-time and state machines can be in different states and execute transitions. In this paper we present a method for creating executable models, that are integrated with the program code. The integration allows for an interplay between the model and the program code at run-time, and avoids inconsistencies between the artefacts.

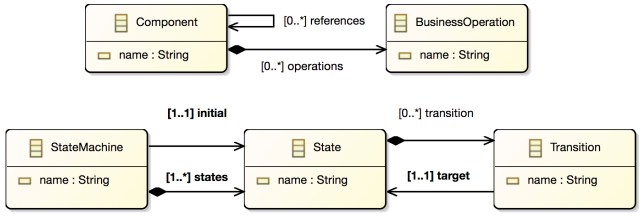

Figure 1: The structure and behaviour meta models of the running example

In the following section, we first describe an simple application which will serve as running example. In Section 3 we describe our approach for integrating models with program code as foundation. Section 4 describes how we use templates for translating model information into code structures and back. These templates are the basis for generating execution runtimes in Section 5. A discussion follows in Section 7. Related work is described in Section 6. Finally, we conclude the paper in Section 8.

2 EXAMPLE APPLICATION

The approach that we present in this paper is usable for all meta models in a subset of Ecore [10]. To illustrate it we now describe a simple software system with interconnected components and a state machine as the basis for a running example. Figure 1 shows a simple meta model for the architectural structure on the upper side, and a simple meta model for a state machine in the bottom. The structural part describes components with business operations. When a component has a reference to another component, it may invoke its operations. The state machine meta model comprises a class for a state machine, which contains a list of states and has an initial state. A state has a list of transitions that target a next state. All classes in the meta model have an attribute *name*.

The example system (see Figure 2) is an excerpt of a store system, based on the CoCoME case study [5]. It consists of two components: *CashDesk* and *BarcodeScanner*. The bar code scanner has an operation *scanItem* to scan an item. The cash desk can be used to add items to a virtual shopping cart for billing. It therefor references the bar code scanner. The cash desk provides the operations *addItemToCart* and *checkout*. The cash desk's behaviour is designed as a state machine. The initial state is *Ready*, which describes that the cash desk is ready for a new sale. When an item is scanned, it is in the state *WithinSale*. It remains in this state until a checkout is performed. Then it returns to the *Ready* state.

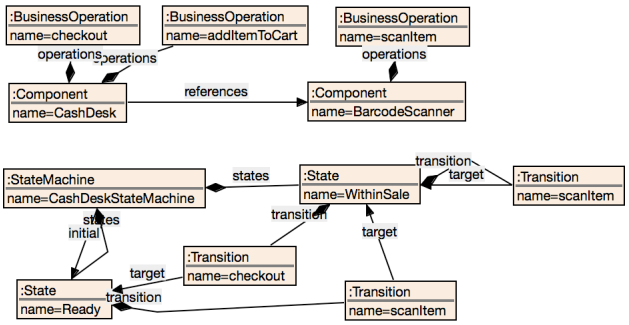

Figure 2: The model of the example application: Two interconnected components with operations and a state machine for the cash desk component

3 FOUNDATIONS: INTEGRATING MODELS WITH CODE

The Model Integration Concept (MIC)[7, Chapter 5] integrates architecture model information with program code. It is the foundation for our generation of model execution runtimes. In the MIC, modelling languages must conform to a subset of Ecore as meta meta model, which includes classes, attributes, containment and non-containment references. Program code must be written in a general purpose programming language, which allows for introspection for execution runtimes. Translations in the MIC define an equivalence relation between model elements and program code structures. There are two types of translations: *Meta model/code translations* define equivalence relationships between meta model elements and program code structures. *Model/code notations* define equivalence relationships between model elements and program code structures. The MIC uses template-based code generation, and static code analysis for extracting models from the code.

Figure 3 and 4 show example translations. Figure 3 shows a meta model/code translation for the class *Component* with the attribute *name* of the type *String*. The meta model elements are shown on the left side. The right side shows Java program code, that represents this meta model element. The meta model class is represented 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 Figure 4. 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.

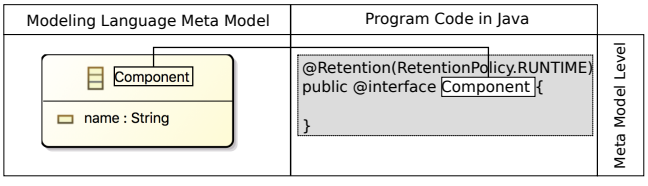

Figure 3: An example of a meta model/code translation

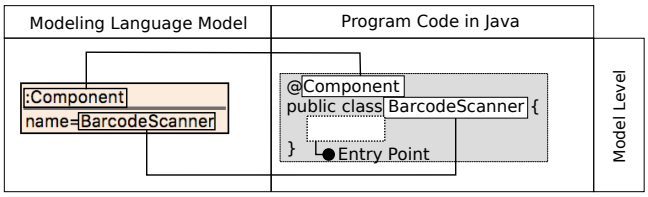

Figure 4: An example of a model/code translation

Figure 4 shows a model/code translation 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*. I.e., arbitrary code can be added here without breaking the model/code relationship. E.g. attributes and operations can be added.

Such (meta) model/code translations in the MIC are the basis for generating three types of artefacts: (1) *Bidirectional transformation* 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, that represent meta model elements. In the example, theis 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. In this paper we describe how execution runtimes in the MIC work, and how they are generated.

Two types of operational semantics exist for model elements in the context of the MIC: (a) Language semantics can be implemented in the runtime. Here each instance of a model element has equal semantics. E.g., when a transition of a state machine is triggered, the state machine switches to another state. These semantics apply to each transition. (b) 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 transition of a state machine is triggered, not only the machine's actual state should be changed. In the example application the transition *scanItem* triggers the operation `scanItem` of the bar code scanner and the operation `addItemToCart` of the cash desk. The transition *checkout* triggers the operation `checkout` of the cash desk. The state machine language definition does not contain elements for expressing such detailed behaviour. They are implementation details in the context of the MIC. 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 operation calls stated above can be implemented within operations provided by

the model/code structures. An execution runtime will then execute these operations.

4 TEMPLATES FOR MODEL-CODE TRANSLATIONS

We analysed the requirements of Java-based frameworks towards the program code for identifying patterns for how models can be integrated with program code. In related work we defined translations and model-specific execution runtimes for state machines and processes [2], interface automata [8], component specifications [6], and deployment models [9]. Based on these works, we defined a set of generic translations between models and code—called *Integration Mechanisms*—that can be used as templates. Each Integration Mechanism comprises a meta model/code translation and a corresponding model/code translation. Mechanisms can be instantiated by applying them to a specific meta model or model element, i.e., by replacing a place holder with a specific (meta) model element. Integration Mechanisms are declared for a type of meta model element. I.e. a class, an attribute, a reference, or a containment reference of the Ecore meta meta model. The complete set of currently defined Integration Mechanisms for the Java programming language can be found in [7, Chapter 5.6], with formal definitions, examples, variants, and a discussion for each mechanism.

We now describe a selection of mechanisms using the running example. For a state machine currently no first class elements exist in the Java language or its adjacent frameworks used in practice. The mechanism for the class *StateMachine* is the *Type Annotation* mechanism. This mechanism has already been shown in the example in Figure 3 and 4 for the class *Component*.

The mechanisms for the class *State* is the *Marker Interface* mechanism. Figure 5 shows the mechanism, when it is instantiated for this class. On the meta model level, the class is translated into an interface declaration with the name of the class. On the model level an object of that class is translated into a type declaration with the value of the attribute *name* as type name. The type implements the interface. The entry point in this mechanism is used analogously to the entry point in the Type Annotation mechanism. In addition, operation signatures can be added to the marker interface, which implies that they have to be implemented by all types that implement the interface. An execution run-time may invoke these operations. The Marker Interface mechanism is usable to represent classes that have behavioural execution semantics, and provide callable semantics that may differ between the single instances. An execution runtime must instantiate the type and provide the resulting object.

The mechanism for the reference *initial* from a state machine to a state is the *Annotated Member Reference* mechanism. Figure 6 shows the mechanism, when it is instantiated for this reference. On the meta model level, a reference is translated into an annotation declaration with the name of the reference, analogously to the Type Annotation mechanism. On the model level, the reference is translated into a field declaration with the name of the target's *name* attribute. The interface that represents the target meta model class is the field's type. An execution runtime must inject an instance of the target's type into this field, so that it is usable by the execution semantics of the state machine. Which instance to inject is subject

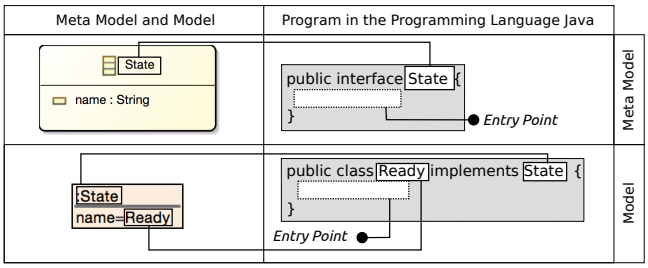

Figure 5: The mechanism *Marker Interface* instantiated for the class *State*

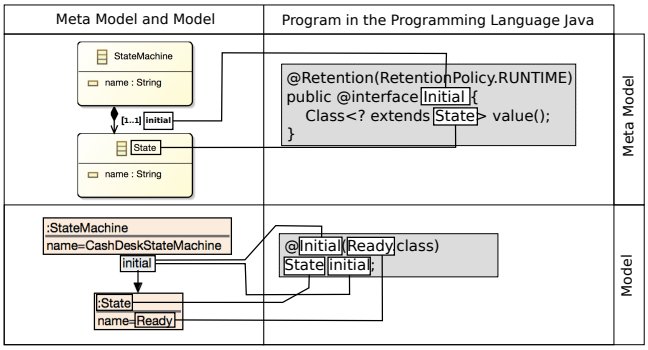

Figure 6: The mechanism *Annotated Member Reference* instantiated for the reference *initial*

to the specific runtime. With this construct, a new initial state can be declared without changing the type of the object attribute `initial`. This mechanism does not define an entry point. Variants for this mechanism exist. Here the variant is shown for reference targets which are translated with the Marker Interface mechanism, with a maximum of 1 target. When multiple targets are possible, an array is used as field type and annotation parameter. If the target translated with the mechanism Type Annotation, the target's type is used directly as field type for x..1 references. For x..* references, the field is an array of the type "Object", the root type in Java.

The mechanism for the containment reference *transition* between states is the *Containment Operation* mechanism. Figure 7 shows the mechanism, when it is instantiated for this reference. On the meta model level, the reference is translated into an annotation declaration with the name of the reference, analogously to the Type Annotation mechanism. On the model level, the reference is translated into a public method declaration with the name of the target's *name* attribute. The method's return type is `void`. The method is annotated with the annotation of the meta model/code translation. The method is both a representation of the reference between the two objects, and a representation of the transition object. An execution runtime for this model element must allow to invoke this method, and therefore the execution semantics of the transition. The entry point of this mechanism is the method body. The execution semantics can be entered within the body, and has access to other code elements, e.g. to fields declared by a Annotated Member Reference mechanism.

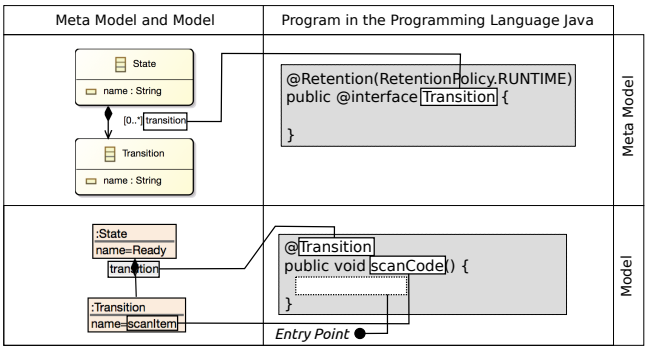

Figure 7: The mechanism *Containment Operation* instantiated for the reference *transition*

5 EXECUTION RUNTIMES FOR MODEL-CODE TRANSLATION TEMPLATES

We developed a framework for implementing execution runtimes for models that are integrated with program code.[1] These runtimes are based on Java's reflection mechanism to analyse the code, inject instances, and invoke operations at run-time. The framework comprises a set of abstract classes, that have to be extended for implementing specific runtime classes. For each Integration Mechanism an abstract runtime class exists. For implementing an execution runtime for a meta model, we map Integration Mechanisms to each class, (containment) reference, and attribute declared in the meta model. A 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 modelling language can be implemented in these classes. The runtimes can effectively seen as interpreters for the integrated models. They can also trigger operations to invoke code, that is declared in an entry point, and therefore trigger the excecution of model semantics, which are expressed in program code.

Listing 1 shows the generic runtime class for the Type Annotation mechanism as an example. It takes the implementing type and the annotation type as constructor parameter. The method *initialize* verifies that the type actually has the annotation attached, instantiates the type, and stores the instance. Runtime classes for classes in the meta model also initialize the runtimes for their attributes and references, and runtime classes for containment references initialize the runtimes for their targets.

The runtime classes for specific meta model elements extend these generic mechanism runtime classes. Listing 2 shows the runtime class for the class *StateMachine* in the meta model. It extends the `TypeAnnotationRuntime` shown in Listing 1. The state machine runtime class is constructed with the implementing type as parameter, e.g., the type `CashDeskStateMachine`. The runtime initialization is implemented in the type `TypeMechanismRuntime` in Listing 1. When the component runtime is initialized for a specific implementing type, first all runtimes for attributes, containment references, and at last for all cross references are initialized. The execution runtime class in Listing 2 has been generated automatically. Operational semantics is added by manually adding public operations that act upon the model instance. The state machine runtime implements operational semantics, that is common to all state machines: Setting an initial state, executing transitions, and storing the next state after transition execution. In the given runtime class, the current state is set after the initialisation of the runtime for the reference *initial* in line 27. It uses the runtime for the reference *initial* to store the runtime of its current target. Further operational semantics is added to the runtime class with operations. The operation `executeTransition` first validates whether the transition given by its name is executable from the current state. Then it takes the runtime for the reference *transition* from the current state. This is a containment operation, which means that it may implement individual semantics in an operation. It invokes that operation and reads the target state. The target state is declared by an annotation on the containment operation (not shown in the example). This invocation shows how the runtimes call individual operational semantics of the model elements: The operation that represents the transition may contain arbitrary code. In the running example, it logs the execution (not shown here). The operation `isExecutable` validates whether a transition can be triggered from the current state. The operation `getPossibleTransitions` uses the runtime graph to return the names of all possible transitions, i.e., the name of all transition operations.

```
1  public class TypeAnnotationRuntime<T> extends TypeMechanismRuntime
       <T> {
2
3    private final Class<T> implementingClass;
4    private final Class<? extends Annotation> typeAnnotation;
5
6    public TypeAnnotationRuntime(Class<T> implementingClass, Class<?
         extends Annotation> typeAnnotation) {
7      this.implementingClass = implementingClass;
8      this.typeAnnotation = typeAnnotation;
9    }
10
11   @Override
12   public void initialize() throws IntegratedModelException {
13     verifyTypeAnnotationMechanism();
14
15     try {
16       instance = implementingClass.getConstructor(new Class[0]).
             newInstance(new Object[0]);
17       Runtimes.getInstance().put(instance, this);
18     } catch (final Exception e) { ... }
19   }
20
21   private void verifyTypeAnnotationMechanism() throws
         IntegratedModelException {
22     if (!implementingClass.isAnnotationPresent(typeAnnotation))
23       throw new IntegratedModelException(...);
24   }
25 }
```

Listing 1: The generic Type Annotation runtime class (shortened)

To show how the program code uses the integrated model, we present the interplay between the cash desk and its state machine. The interaction is not defined in any of the modelling languages. Therefore it must be implemented in program code manually. In the running example the invocation of the cash desk's business operations trigger transitions in the state machine. Listing 3 shows

[1] A part of the tool *Codeling*, which implements the MIC – https://codeling.de

```
public class StateMachineRuntime<T> extends TypeAnnotationRuntime<
    T> {

  StateRuntime<?> currentStateRuntime; // not generated
  // Reference Runtimes
  InitialRuntime initialRuntime;
  StatesRuntime statesRuntime;

  public StateMachineRuntime(Class<T> implementingClass) throws
      IntegratedModelException {
    super(implementingClass, StateMachine.class);
  }

  @Override
  public void initializeContainments() throws
      IntegratedModelException {
    super.initializeContainments();
    // Reference Runtimes
    statesRuntime = new StatesRuntime(this);
    statesRuntime.initialize();
  }

  @Override
  public void initializeCrossReferences() throws
      IntegratedModelException {
    super.initializeCrossReferences();
    // Reference Runtimes
    initialRuntime = new InitialRuntime(this);
    initialRuntime.initialize();

    // Set current state (not generated)
    currentStateRuntime = (StateRuntime<?>) initialRuntime.
        getTargetRuntime(initialRuntime.getTargets()[0]);
  }

  public InitialRuntime getInitialRuntime() { ... }
  public StatesRuntime getStatesRuntime() { ... }
  public StateRuntime<?> getCurrentStateRuntime() { ... }

  // Operational semantics (not generated)
  public void executeTransition(String transitionName) throws
      IntegratedModelException {
    if (!isExecutable(transitionName))
      throw new IllegalStateException(...);

    TransitionRuntime transitionRuntime = currentStateRuntime.
        getTransitionRuntime();
    transitionRuntime.invoke(transitionName);
    StateRuntime<?> targetStateRuntime = (StateRuntime<?>)
        transitionRuntime.getTargetRuntime().getTargetRuntime(
        transitionName)[0];
    currentStateRuntime = targetStateRuntime;
  }

  public boolean isExecutable(String transitionName) throws
      IntegratedModelException {
    return getPossibleTransitions().contains(transitionName);
  }

  public List<String> getPossibleTransitions() throws
      IntegratedModelException {
    TransitionRuntime transitionRuntime = currentStateRuntime.
        getTransitionRuntime();
    List<Method> containmentOperations = transitionRuntime.
        getContainmentOperations();
    return containmentOperations.stream().map(Method::getName).
        collect(Collectors.toList());
  }
}
```

Listing 2: The *StateMachine* runtime class (shortended)

how the implementation of the running example makes use of the state machine's execution runtime to implement this behaviour. The cash desk is implemented using the Type Annotation mechanism. In an initialization method, it creates and initializes the runtime for its state machine. The parameter is the Java type, that implements the state machine using the Type Annotation mechanism. In the business operation `addItemToCart`, the cash desk first uses the bar code scanner's business operation `scanItem` to scan an item. Then it executes the transition `scanCode` of the state machine. In the business operation `checkout` the items are cleared and the transition `finishSale` is triggered. The cash desk uses the operational semantics of the state machine runtime to execute transitions. The initialization of the runtime graph behind the state machine runtime is automatically provided by the execution runtime framework.

```
@Component
public class CashDesk {

  final LinkedList<String> items = new LinkedList<>();
  StateMachineRuntime smr;
  @References BarcodeScanner barcodeScanner;

  public void init() throws IntegratedModelException {
    smr = new StateMachineRuntime(CashDeskStateMachine.class);
    smr.initialize();
  }

  public void addItemToCart() throws IntegratedModelException {
    items.add(barcodeScanner.scanItem());
    smr.executeTransition("scanCode");
  }

  public void checkout() throws IntegratedModelException {
    items.clear(); // Should execute a real sale
    smr.executeTransition("finishSale");
  }
}
```

Listing 3: *CashDesk* implementation that uses the State Machine runtime

6 RELATED WORK

Executable models use generic model runtimes or generated code, which implements the semantics for a modelling language. These runtimes are comparable to runtimes for program code structures in the MIC. In the approach at hand, specific program code structures represent the model elements and are executed by extensible, generic runtimes. This allows for bidirectional interaction between the behavioural semantics of the model and other program code.

ArchJava [1] adds (textual) architecture model information to Java programs. It is translated into byte code for the Java virtual machine. The execution engine in the sense of model execution is the compiled byte code. In contrast to ArchJava the approach in this paper is not specialised for a specific modelling language. Balz [2] describes *embedded models* as an approach for representing models with well-defined code structures. He developed specific model/-code translations for state machine models and process models. The MIC generalizes the embedded models approach and introduces Integration Mechanisms as translation templates. These templates are here used for developing generic execution runtimes for Ecore-based models.

Alf [4] provides a Java-like syntax for fUML [3]. This approach is specialized for a specific modelling language and a specific programming language. Our approach is more generic, as it can be used with all models based on a subset of Ecore, and all programming languages that allow for introspection and dynamic operation calls. However, our tools currently only support Java.

Yakindu[2] is an IDE for specifying state charts, that are integrated with program code. The approach presented in this paper aims at integrating arbitrary models with program code instead.

[2]https://github.com/Yakindu/statecharts

7 DISCUSSION

The goal of the MIC is to integrate models with program code, so that the models and code can be kept in an consistent state. A model can be extracted from the code, changed with its usual editors, and the changes can be propagated to the code. We integrated the generation of execution runtime classes into the code generation of the MIC tools. We added a runtime framework and code generators for each Integration Mechanism. With the presented approach we can define operational semantics for all meta models that are integrated with the program code using the MIC.

The design-time aspects of the MIC have been used in a set of case studies to evaluate its applicability in the context of information systems [7, Chapter 10]. The runtime aspect has not been evaluated in a broader context than application examples like the one presented above. At the time of writing we therefore cannot assess the effort for creating a runtime with actual operational semantics for complex systems in comparison to other approaches.

Our execution runtimes are based on a set of abstract classes, which provide common behaviour like instantiation or utility functions. Execution runtime classes can be generated, when a meta model element is mapped to an Integration Mechanism. Given an Ecore meta model and a set of mechanisms, a basic runtime can be generated, that makes available a runtime object for each model element. Operational semantics still has to be added. On the other hand the MIC—as foundation—is a requirement to use these runtime classes. For each Integration Mechanism used to represent model elements as program code structures, a generic runtime class and bidirectional model/code transformations have to be implemented. We are investigating how the effort for these developments can be reduced. Once the generic runtimes exist, generating the specific runtimes is automated. The generated and adapted execution runtimes are directly used by software developers in the context of the program. The runtime classes publish the operational semantics by their public operations. Still, the developers have to understand how the models and the integration work.

Conceptually, the approach is limited to a subset of Ecore as meta meta model, and can be used to handle models that use classes, references, containment references, and attributes. For further elements of Ecore currently no Mechanisms are defined, and the associated tools do not support them. We plan to extend the concept and tools to the complete Ecore. The programming language must support introspection for managing the run-time model. The current Integration Mechanisms heavily rely on templates/generics and object-orientation, but other Mechanisms could be developed.

At the moment, the model execution engines do not allow for replay and monitoring besides logging. We intend to integrate such means generically into the framework, so that Ecore models that are used with the MIC can be monitored and debugged. Currently there is no support for a model visualisation. We are planning to support the design time and run time visualisations of the models.

The presented approach has advantages in comparison to related work: It can be used for each modelling language that can be expressed by the used subset of Ecore, instead of being specialized for a specific model type. It is targeted at abstract design models like architectural models and abstract behaviour, for which no first class entities in programming languages exist usually, while other bidirectional model/code translation approaches focus on more low level models, such as UML class diagrams. However, it also has disadvantages: While not measured in a realistic scenario yet, it can be assumed that the generic runtimes perform worse than runtimes that are specialized for specific model types, or language and model semantics that is directly implemented in code.

8 CONCLUSION

In this paper we presented our approach for executable models, that are expressed with specific program code structures. The execution runtimes make use of templates—called *Integration Mechanisms*—that express how (meta) model elements can be expressed in program code. Based on these templates we generate translations between the program code and its model representation, and a graph of execution runtime classes, that handle the instantiation of the executable model. Operational semantics can be added to these classes. Interactions between the execution runtime and other program code is possible in both ways.

ACKNOWLEDGMENTS

The work presented in this paper is partially funded by the DFG (German Research Foundation) under the grant number GO 774/7-1 within the Priority Programme SPP1593.

REFERENCES

[1] Jonathan Aldrich, Craig Chambers, and David Notkin. 2002. ArchJava: Connecting Software Architecture to Implementation. In *Proceedings of the 24th International Conference on Software Engineering*. ACM, 187–197. https://doi.org/10.1145/581339.581365

[2] Moritz Balz. 2011. *Embedding Model Specifications in Object-Oriented Program Code: A Bottom-up Approach for Model-based Software Development.* Ph.D. Dissertation. Universität Duisburg-Essen.

[3] Object Management Group. 2016. Semantics of a Foundational Subset for Executable UML Models (fUML). http://www.omg.org/spec/FUML/1.2.1/ OMG Document Number: formal/2016-01-05, http://www.omg.org/spec/FUML/1.2.1/, accessed 2017-04-15.

[4] Object Management Group. 2017. Action Language for Foundational UML (Alf) – Concrete Syntax for a UML Action Language, Version 1.1. http://www.omg.org/spec/ALF/1.1/

[5] 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. 2008. CoCoME - The Common Component Modeling Example. In *The Common Component Modeling Example*, Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and František Plášil (Eds.). Springer-Verlag, Berlin, Heidelberg, 16–53. https://doi.org/10.1007/978-3-540-85289-6_3

[6] Marco Konersmann. 2014. Rapidly Locating and Understanding Errors Using Runtime Monitoring of Architecture-carrying Code. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering (RCoSE 2014)*. ACM, New York, NY, USA, 20–25. https://doi.org/10.1145/2593812.2593814

[7] Marco Konersmann. 2018. *Explicitly Integrated Architecture – An Approach for Integrating Software Architecture Model Information with Program Code.* Ph.D. Dissertation. University of Duisburg-Essen, Essen, Germany.

[8] Marco Konersmann and Michael Goedicke. 2015. Integrating Protocol Contracts with Program Code – A Leightweight Approach for Applied Behaviour Models that Respect Their Execution Context. In *Behavior Modeling – Foundations and Applications*, Ella Roubtsova, Ashley McNeile, Ekkart Kindler, and Christian Gerth (Eds.). Springer International Publishing, 197–219. https://doi.org/10.1007/978-3-319-21912-7_8

[9] Marco Konersmann and Jens Holschbach. 2016. Automatic Synchronization of Allocation Models with Running Software. *Softwaretechnik-Trends* 36, 4 (Nov. 2016), 28–29. http://pi.informatik.uni-siegen.de/stt/36_4/./01_Fachgruppenberichte/SSP2016/ssp-stt/24-Automatic_Synchronization_of_Allocation_Models_with_Running_Software.pdf

[10] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. *EMF: Eclipse Modeling Framework 2.0* (2nd ed.). Addison-Wesley Professional.

[11] R. N. Taylor, Nenad Medvidovic, and Irvine E. Dashofy. 2009. *Software Architecture: Foundations, Theory, and Practice* (1 ed.). Wiley.