Integrating Protocol Contracts with Program Code – A Leightweight Approach for Applied Behaviour Models that Respect their Execution Context

Marco Konersmann, Michael Goedicke

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

Abstract. In the domain of information systems, behaviour is typically described without a formal foundation. These systems could benefit from the use of formal behaviour modeling. However, the perceived costs for integrating a formal behaviour modeling approach seems to be higher than the expected benefits. A framework for formal behaviour modeling and execution could help bringing the benefits of formal modeling to this domain when it imposes a low barrier for integrating the approach. To achieve this, we present our approach for designing and executing behaviour models which are encoded with well-defined source code structures. In our approach the model is statically represented in the program code. Therefore the model does not exist as a first class citizen, but is extracted from the code at design time and run time. These models can be integrated within a context of arbitrary other program code, that does not follow the semantics of the model type. They therefore impose only a small barrier for their use.

1 Motivation

The domain of information systems is driven by platforms that are defined in industry standards. These platforms typically describe the structure of information systems that use the platform, including structural specifications for data models, logical components, and user interfaces. The specifications are most often accompanied by specifications for security, logging, monitoring, and other cross cutting concerns.

Formal behaviour models are not broadly used in the domain of information systems. Although the domain includes some problems that require complex behaviour descriptions, great parts of information systems are not modeled by behaviour models. The benefit of formal behaviour models – the ability to formally reason about the behaviour and the ability to model small, interacting behavioural components – is often considered not necessary, or the costs for using formal behaviour models is considered too high for their benefits. Therefore, the widely used platforms for information systems do not include formal

behaviour models, but rely on the underlying imperative programming language for behaviour specifications. Another property of information systems is, that they are often long-living systems, which are and have been subject to evolution. Therefore changing the complete behaviour towards a formal model implementation is infeasibly expensive.

For behaviour specifications to be more broadly used in information systems, we believe that a framework needs to be available, which imposes a low barrier for its integration. We identified the following criteria for a solution for formal behaviour models in the information system domain:

- **Ease of Application Integration** As it is not expected that the complete behaviour of an information system is changed at once, it is desirable that behaviour models can easily be used for increasingly many parts of the software in an evolutionary fashion. This includes that arbitrary application program code can be called from the model to get or set external information.
- **Ease of Platform Integration** It is desirable that a solution is easily integratable into the standard platforms for information systems.
- **Integrated Editor** The model specifications should be editable in the typical editors for information systems, e.g. IDEs like Eclipse[1].
- **Ease of Development Tool Integration** The tools used for developing information systems today are very much focused on program code and text based files. This includes e.g. code repositories, collaboration tools, or code metric tools. It is desirable that a solution for behaviour models in information systems integrate well with these tools. Therefore, the key model information should be readable and editable with text editors. Ideally, it should also be debuggable in an IDE.
- **Possibility to Monitor** The model should be easy to monitor.
- **Possibility to Debug** It should be possible to debug the model. Ideally this can be done within an IDE.

Our approach for a formal behaviour model framework that can be used in information systems is to integrate model specifications into the source code of programs. With this approach, the models can be reliably extracted from the code. We call this approach Architecture-Carrying Software (ACS) [5]. In ACS we integrate static structure models, including component models and behaviour models with source code that complies with component frameworks. Other models, including quality information and deployment information, are planned to be added. Currently, the behaviour model included in ACS is a batch-oriented state machine (see [1]). This batch-oriented model is, however, not well suitable for information systems, because often one wants to interactively influence the behaviour of a subsystem.

In this paper we present an alternative behaviour model integrated with source code. The integrated model type is that of Protocol Contracts [7, 6]. The model allows for interactive behaviour, which makes it better-suited for information systems. In the following, we will first explain briefly the conceptual

[1] http://www.eclipse.org

foundations of our work in section 2. Section 3 describes the integration of Protocol Contracts with Java source code at design time. The execution runtime and the corresponding run time model are described in section 4. The idea behind a model editor for integrated Protocol Contracts is introduced in section 5. We discuss our approach in section 6, before we present future work in section 7 and related work in section 8. We conclude in section 9.

2 Conceptual Foundations

In this section we introduce the conceptual foundations for this paper. First we briefly introduce our approach for integrating model information with source code. Then we describe Protocol Contracts, before we describe the integration of Protocol Contracts with source code in section 3.

2.1 Architecture-Carrying Software

The idea of Architecture-Carrying Software (ACS) [5] is to represent architectural models in source code with sophisticated code structures. *Architecture-Carrying* means that the software itself carries its architecture information, without the need for adjacent models that are separate from the code.

The code structures that represent models in ACS only define architecturally relevant code. Therefore they include interfaces to execute arbitrary other, non-architectural code.

These code structures are not meant to be directly changed with source code editors, but with model editors. These model editors allow to edit the architecture in a representation that software architects are comfortable with, e.g. UML or formal specification languages. The editor extracts the architecture from the underlying code base and presents a model to interact with. Changes to the model are reflected by changes to the underlying code base. The model view is volatile. It only exists as long as the model editor is in use. With ACS, the architecture model is available at compile time as source code structures and at run time via reflection mechanisms.

The code can still be viewed and edited with text editors. The source code representing models should be edited with respect to the ACS code structure definitions. The source structures define so-called entry points where external code is meant to be integrated. Code that does not represent the models can still be edited freely.

Currently, the only behaviour model included in ACS is a batch-oriented state machine (see [1]). This batch-oriented model is, however, not suitable for many applications, because usually one wants to interactively influence the behaviour. In this paper, we present our ongoing work how we extend ACS with Protocol Contracts.

2.2 Protocol Contracts

Protocol Contracts [7, 6] describe event-driven, state-based behaviour of object models. In a Protocol Contract, events are presented to protocol models. Protocol Models react to events by accepting them and changing their state, or by refusing them. In the following, we will briefly introduce Protocol Contracts as they are described in [7]. An example of a protocol model is shown in figure 1.

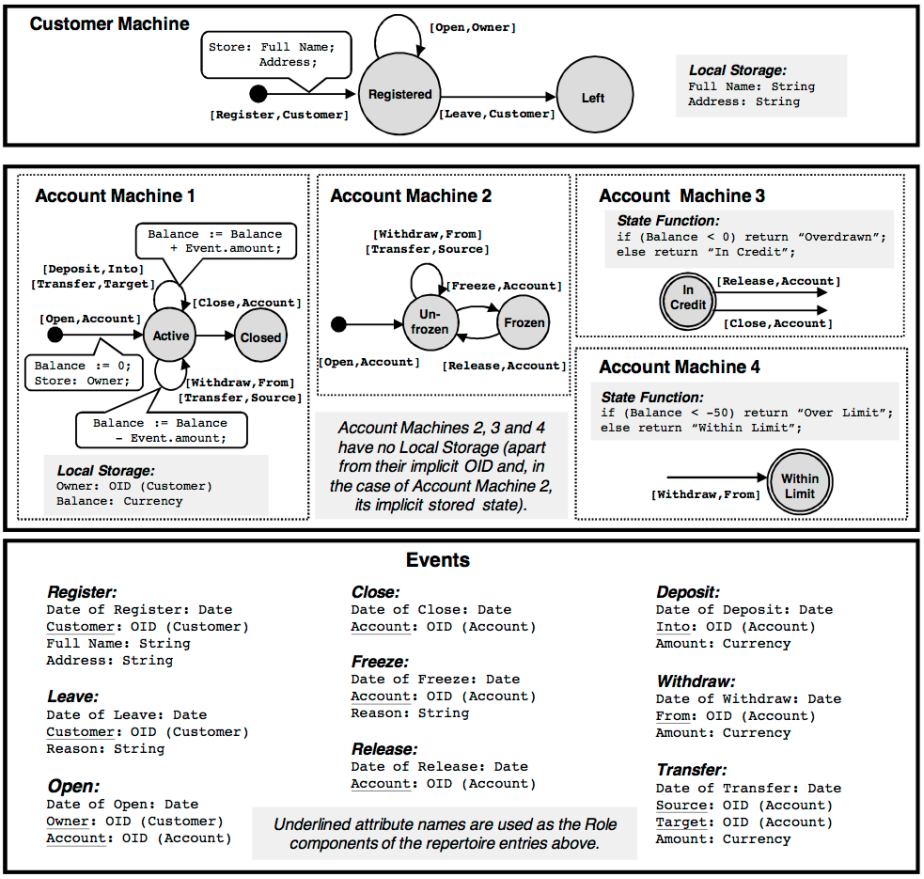

Fig. 1. An example system specified as Protocol Contract (Source: [6])

Events Events in Protocol Contracts are typed. An event type has a name and attributes. Event attributes are also typed. Attribute types are the usual primitive types of programming language. This especially includes integers, floating point numbers, booleans, and strings. In addition, attributes can have the type of an object. An example is the event of the type *Deposit* in figure 1. This event

type defines an attribute *date of deposit* of the type *Date*, an attribute *amount* of the type *Currency*, and an attribute *into*, which is a reference to an object to be affected by this event. An event type is instantiated to an event instance (also called *event* in this paper) by setting attribute values.

Protocol Machines Protocol machines describe a deterministic behaviour contract for objects. They can be built up either by states and rules how to change the states in reaction to events, or by comprising other protocol machines. The first are called *elementary machines*.

Elementary machines have a stored state. The stored state consists of typed variables, including an implicit state variable (just state variable from now on). The variables work analogously to attributes in objects of object-oriented programming languages like Java. The state variable represents a finite set of possible states, that a protocol machine can be in. The value of a state variable can be determined by the sequence of events, that the machine accepted (so called topological states), as it is known from UML state machines [9, page 535 ff.], or by a state function that is evaluated when a event is presented to the machine.

Non-elementary machines are built by nested non-elementary and eventually by elementary machines. They build their stored states based on the stored states of the nested machines. A nested machine can read and write its own stored state. It can read the stored state of all other machines in its environment. The environment of a machine is built by the stored states of its parent machines and all of their parents' nested machines.

Protocol machines have a repertoire, that describes which events are accepted or rejected. Events that are neither accepted nor rejected are ignored. Repertoire entries include:

1. an event type,
2. a reference to an object that is represented by this machine (the OID),
3. a role in which the machine accepts the event type,
4. a boolean expression based on the machine's stored state, that has to evaluate to true before the event is processed (the "test")
5. a term that expresses the update to the machine's stored state when the event is processed

The role is important when an event references several similar machines. E.g. the event *Transfer* (see figure 1) transfers money from one bank account to another. This event references one account in the role *source*, and another account in the role *target*.

An event is accepted by a machine, if (1) it has a repertoire entry for the given event type, (2) it represents exactly one object that is referenced by the event, (3) in the role stated by the object reference in the event, and (4) the test is evaluated to true. If the test evaluates to false, the event is rejected. In any other case the event is ignored.

Non-elementary machines reject an event when any nested machine rejects the event. When all nested machines ignore the event, the non-elementary ma-

chine ignores the event. When at least one nested machine accepts the event and all other nested machines accept or ignore the event, the event is accepted.

Protocol Systems Protocol systems compose protocol machines in terms of Communicating Sequential Processes (CSP) [4]. Protocol systems themselves have no stored state, event types, and referenced object. Their repertoire and their references to objects are built by their composed machines. A protocol machine that is composed by a protocol system has read access to the attributes of all other protocol machines within the system. Protocol systems can themselves be subject to composition by other protocol systems.

Protocol Models Protocol models are protocol machines that are not nested or composed by any other protocol machine or system. They describe the complete, self-enclosed behavior of the objects they represent.

3 Model Integration

For adding new behaviour models to ACS the following artefacts are necessary:

1. a meta model of the model type to integrate,
2. integration mechanisms for a specific framework or language,
3. a runtime to execute the model,
4. an editor to inspect and change the model in a model view.
5. a monitor to show the executed model

The meta model of protocol contract comprises a design time meta model and a run time meta model. This is due to the fact, that instances of protocol machine specifications are created at run time. Therefore elements exist at run time, that are not modeled explicitly at design time. In the following we present the meta model for the design time (section 3.1) and integration mechanisms for Protocol Contracts in the Java programming language (section 3.2).

The execution runtime, including the run time meta model, is described in section 4. An editor, which may also serve as a monitor and for debugging purposes, is described in section 5.

3.1 Design Time Meta Model

The ACS prototype is implemented in Java and based on Ecore models[11]. Therefore, the meta model is implemented in Ecore. In this section, we describe the design time meta model for our implementation of Protocol Contracts, which is based on the description from McNeile and Simons [7].

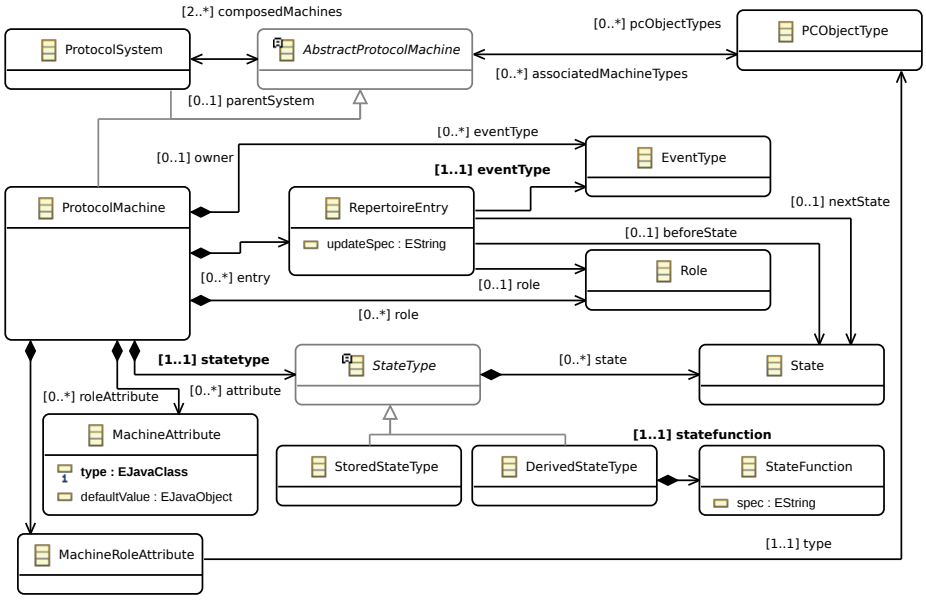

Fig. 2. Machine types in the Protocol Contracts meta model

Machine Types There are two machine types represented in our meta model: The *Protocol Machine* and the *Protocol System*. In our model (figure 2) a *ProtocolSystem* always composes at least two machines. Using this notation, a hierarchy of machines can be built. The inner nodes and the root of the tree are *ProtocolSystems*. The leafs are *ProtocolMachines*.

The *ProtocolMachine* contains *MachineAttributes*, *MachineRoleAttributes*, one *Statetype*, *Roles*, *EventTypes*, and *RepertoireEntries*. The *MachineAttributes* and *MachineRoleAttributes* specify the local storage of a machine. The first define value-based attributes. The latter define references to OIDs in Protocol Contracts. The attributes are typed, and may define a default value, which is null, when not specified. The *Statetype* defines the type of the machine, i.e. whether the machine defines topological states (*StoredStateType* in our model), or a state function (*DerivedStateType* in the model). Both types may have a number of *States*. However, the *DerivedStateType* also has a *StateFunction* which contains an attribute *spec* of the type EString (an Ecore representation of a Java String), specifying its function in terms of Java source code. The repertoire, that each machine contains, is formed by the *RepertoireEntries* in our model. The *EventType* and *Role* of the repertoire entry are represented as references of the contained elements of a machine. This allows to reuse *EventType* and *Role* elements. Repertoire entries can also reuse Event Type and Role elements of their machines. This introduces a dependency on the model level towards these other machines. The *updateSpec* defined by repertoire entries represent the update of the local storage during transitions. Each entry knows its *beforeState* and the

nextState, if applicable. This represents the precondition and the update of the state variable of stored state machines, and the pre- and postcondition for derived state machines.

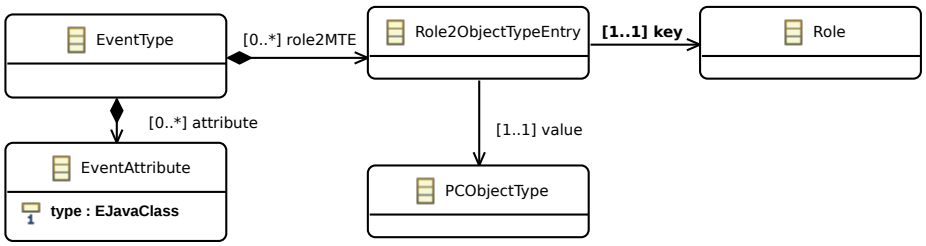

Fig. 3. Event Types in the Protocol Contracts meta model

Figure 3 shows the part of the meta model that defines event types. The class *EventType* may contain typed *EventAttributes*. The *EventType* may contain *Role2ObjectEntities*, which bind the *Role* in an event type to a *PCObjectType*. The *PCObjectType* is the type of an OID. This e.g. represents the type Customer in the example in figure 1.

3.2 Integration Mechanisms

For describing Protocol Contracts as model type for ACS, source code structures for the design time meta model elements have to be defined. In the following sections, we describe the source code structures that represent Protocol Contracts in Java code. Elements that represent the instantiation of machines and events are runtime artefacts and therefore not represented in the code.

Protocol Machine A protocol machine is represented in Java code as a Java package that includes a class, which implements a marker interface *IProtocolMachine*. We call this class the *Protocol Machine Class* (or just *Machine Class*). Listing 1.1 shows the Machine Class template. A marker interface is an interface without any operations, that only exists to mark classes. Only one Machine Class is allowed within a Java package. Other classes that define the protocol machine (as shown in the following) also reside within this package or subpackages[2]. Other classes, unrelated to the protocol machine, may also reside in that package, although we do not recommend that.

The Machine Class also includes a reference to the object type that is represented by the machine. It is an attribute in the class definition marked with a *MachineOID* annotation. The object type of the OID is a simple Java class. The type of the attribute is that class. The variable is named *oid* for convenience.

[2] This is actually a recommendation, not a requirement. For protocol machines with more than 3 or four states, we found it practical to use subpackages for structuring reasons.

```
public class $MachineName implements IProtocolMachine {

    @MachineOID
    $PCObjectTypeName oid;

    @MachineContext(
        localState = $MachineNameVariable.class,
        localStateRead = IReadableVariable.class,
        localStateWrite = IWritableVariable.class)
    IContext context;
}
```

Listing 1.1. The source code structure for a Protocol Machine Class with the reference to a Machine Attribute Class

Machine Attributes and Machine Role Attributes Machine attributes and machine role attributes are represented in a separate class, called *Variable Class*. The Variable Class contains the local storage (excluding the state variable) and the corresponding get and set methods. The methods are also represented in interfaces: one interface for get methods, the *Read Interface*, and one interface for set methods, the *Write Interface*. These interfaces are entry points for reading and changing the attributes. The Variable Class implements these interfaces. A third interface, the *Context Interface*, defines the environment of the machine. For a single protocol machine, the Context Interface extends the Read Interface and the Write Interface of the machine. The Machine Class contains a variable, with the type of the Context Interface as a reference to its environment, the machine context. Due to the interface and class structure described above, the Context Interface allows for reading and writing the local storage of the machine, and for reading the local storages of its environment. An annotation on the variable for the machine context states the Read Interface, the Write Interface, and the Variable Class. Listing 1.1 shows how the Machine Class is built with the Variable Class (here and in the following listings, a dollar sign denotes a variable in a template). The reference to the OID is a reference to the underlying object, and allows for executing operations of this object. It is therefore a reference to model-external code.

States and OIDs States are represented as a Java class that extends the abstract class *AbstractPCState* (see listing 1.2). The name of the state is represented by the class name. The class extends the abstract class *AbstractPCState*. That abstract class has a type parameter that represents the OID type of the machine. *AbstractPCState* has an *oid* reference to an arbitrary object. This is the interface of the states to model-external code. This can be used within transition code to execute arbitrary operations in the context.

```
public class $StateName extends AbstractPCState<$PCObjectTypeName> { }
```

Listing 1.2. The source code structure for a state

State Variable The state variable in Protocol Contracts can be built in two ways. In stored state protocol machines, the state variable is determined by the initial state and the updates. In derived state protocol machines, the state variable is derived using a state function.

The state variable of stored state machines does not have a static representation. Therefore no source code structure exists for that state. Machine Classes of derived state protocol machines contain a method *getCurrentState()* to evaluate the state variable. The method returns *Class<? extends AbstractPCState>*, i.e. the reference to the class that represents the current state. The method's body implements the state function.

Roles and Event Types Roles are represented as classes implementing the marker interface *IRole*. An event type is represented in the source code as *Event Type Class*. This is a class implementing the marker interface *IEventType*. Event types contain two types of meta data: (1) event attributes, and (2) roles and object references.

Event attributes are represented as object variables in the class with the corresponding type. The attribute *name* is represented by the name of the variable. The variable is complemented by a get and a set method.

PCObjectTypes are represented as Java classes. Therefore, roles and object references can be represented by variables with the corresponding class as variable type, and the role as variable name. These variables are also complemented by corresponding get and set methods. Listing 1.3 shows the source code structure for an Event Type Class.

Repertoire Entries Repertoire entries define the following data: (1) an event type, (2) a referenced object, (3) a role for which the event type is accepted, (4) a test, and (5) an update specification. In the source code these are represented as annotated methods (*Repertoire Entry Methods*), as shown in listing 1.4. The methods are contained by State Classes or a Protocol Machine Class. The test is defined by the class that implements the method. When a State Class implements the method, that state is the necessary source state. When a Protocol Machine Class of a stored state machine implements the method, the source state is the initial pseudo state. When a Protocol Machine Class of a derived state machine

```
public class $EventTypeName implements IEventType {

    $PCObjectTypeName $roleName;

    $AttributeType $attribtueName;

    // getters and setters
}
```

Listing 1.3. The source code structure for event types, including event attributes and roles

```
@RepertoireEntry(nextState = $StateName.class)
public void $eventTypeName($EventTypeName event, IContext context, $RoleName role){
    // Update Specification
}
```

Listing 1.4. The source code structure for repertoire entries

implements the method, the before state is empty. The method's parameters are a reference to an Event Type Class object (event), a reference to an object of the Context Interface (context), and a reference to the Role class. The parameter event represents the event type. The context parameter is used for the update specification and is a reference to the machine's context. The next state of the repertoire entry is given as parameter of an annotation as class reference.

Both, the Machine Class and the State Class have an attribute *oid* that is the reference to the object defined by the machine. Here the attribute acts as an interface to non-architectural code. Within the update specification, operations to the OID can be called. The semantics of the executed operations of the OID are not part of the model.

Protocol Systems The source code representation of a protocol system is a *Protocol System Class*, or shortly *System Class* (see listing 1.5). Such a class implements the marker interface *IProtocolMachine*, just as Protocol Machine Classes. In addition, Protocol System Classes are annotated with the annotation *ProtocolSystem*, which takes a list of classes as parameter, that extend the IProtocolMachine interface. One package may only contain either one Protocol System Class or one Protocol Machine Class. Subpackages may contain further Protocol Machines or Systems.

```
@ProtocolSystem({ $MachineName.class, ... })
public class $ProtocolSystemName implements IProtocolMachine {

    @SystemEnvironment
    ISystemContext context;
}
```

Listing 1.5. The source structure for Protocol Systems

Protocol Systems influence the environment of their referenced protocol machines and systems. To represent this influence, each Protocol System Class is accompanied by a *System Context Interface*. This interface extends the Read Interfaces of its composed protocol machines and the System Context Interfaces of its composed protocol systems. The System Context Interface is an attribute of the Protocol System Class, annotated with an annotation *SystemEnvironment*.

When a protocol machine is composed by a protocol system, the machine can read variables from all machines composed in the system. To reflect this, the machine's Context Interface replaces the extension of its Read Interface with the System Context Interface of the highest Protocol System in the composition hierarchy (see figure 4).

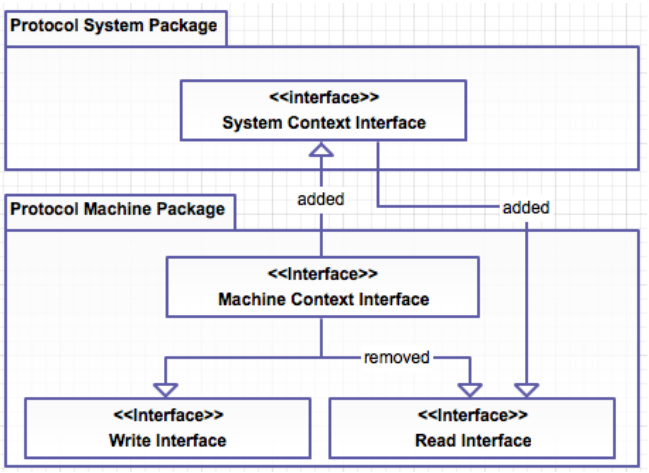

Fig. 4. The given interface and class structure ensures that the variables of all composed machine are readable by every machine in the system, and that each machine can only alter its own variables.

Protocol Models Protocol models are protocol machines that are not nested or composed by any other protocol machine or system. This can be evaluated from the machine context. Thus no explicit source code structures exist for protocol models.

3.3 Example

To show the functionality of our meta model and source code structures, we implemented a desktop example. Our example is an implementation of the Bank Model example given in [6]. The model of the example system is shown in figure 1. We will here only show parts of the example that differ enough to show the different working concepts. We therefore show here our implementation of account machine 1, a protocol machine with stored states; account machine 4, a protocol machine with derived states; and the account system, a protocol system.

Account Machine 1 Account machine 1 (AM1) is a protocol machine with stored states. All of the classes for AM1 are placed in the same Java package. Figure 5 gives an overview of the classes and interfaces in the package. The Protocol Machine Class for AM1 is depicted in listing 1.6.

The Protocol Machine Class of AM1 defines one repertoire entry from the pseudo state — here represented by the containment relationship from the Protocol Machine Class to the method — to the State *Active*. The body of the method *open* shows the update specification.

Figure 6 shows the class structure of the machine attributes for AM1 (without the influence of the system, that composes the machine). The interfaces shown in

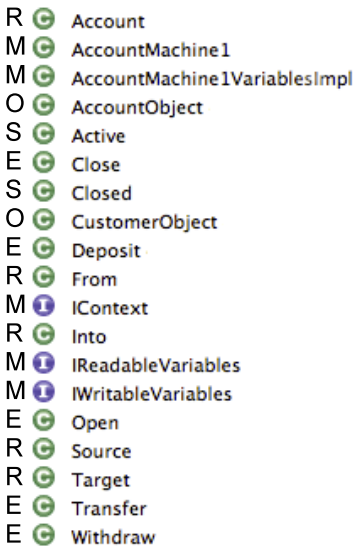

Fig. 5. The package structure of protocol machine for Account Machine 1. The annotations mean: (E) Event Type Classes; (M) Machine Class, Variable Class and interfaces; (O) Referenced Object classes; (R) Role Classes; (S) State Classes. The c in a circle denotes a class. The i in a circle denotes an interface.

```
public class AccountMachine1 implements IProtocolMachine {

    @MachineOID
    AccountObject oid;

    @MachineContext(
        localState = AccountMachine1VariablesImpl.class,
        localStateRead = IReadableVariables.class,
        localStateWrite = IWritableVariables.class)
    IContext context;

    @RepertoireEntry(nextState = Active.class)
    public void open(Open event,
            IContext context, Account role) {
        context.setBalance(0);
        context.setOwner(event.getCustomer());
    }
}
```

Listing 1.6. The implementation of the Machine Class for Account Machine 1

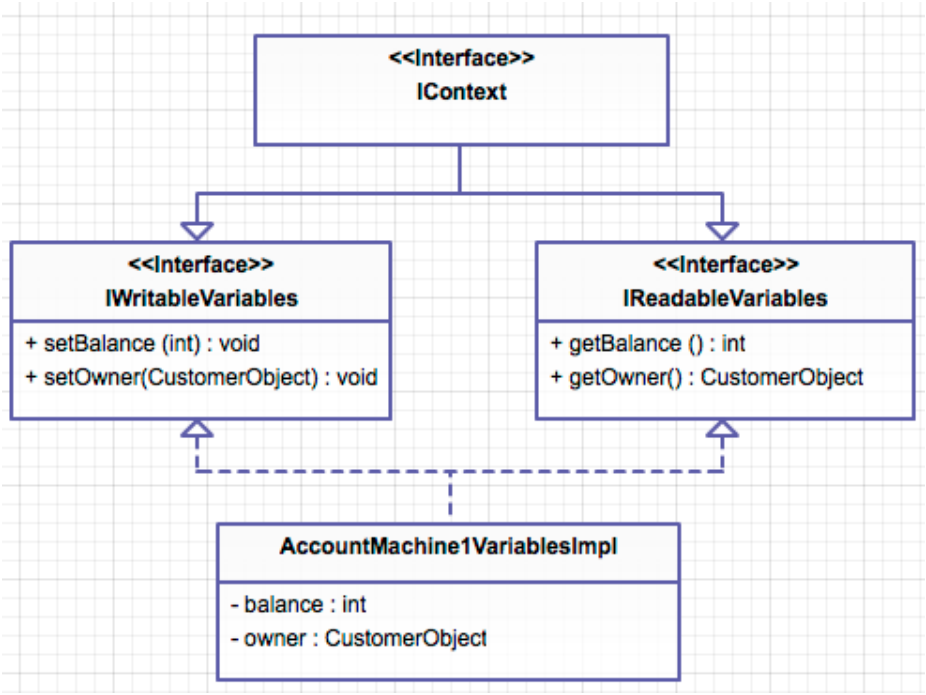

Fig. 6. The class structure for the machine attributes of Account Machine 1

this figure contain get and set methods according to their task. The implementing class is shown in listing 1.7.

The Event Type Class of Open is shown in listing 1.8. It contains the attributes and roles prescribed by the specification in in terms of attributes, get methods, and set methods.

The State Class of the state *Active* is shown in listing 1.9. It includes repertoire entry methods for all accepted events as described in figure 1. Our source code structure however does not allow to create an entry with the same update and target state, but with multiple event types and roles without copies of the update specification. We need multiple methods to represent this structure.

Some classes are not shown in detail here. The Role Classes implement the Interface IRole, but do not contain any methods or attributes. The Event Type Classes that are not shown are built accordingly to the Event Type Open in the obvious way.

```
public class AccountMachine1VariablesImpl
        implements  IReadableVariables, IWritableVariables {
    int balance;

    CustomerObject owner;

    // getters and setters
}
```

Listing 1.7. The machine attribute class of Account Machine 1

```
public class Open implements IEventType {
    Date dateOfOpen;

    AccountObject account;

    CustomerObject owner;

    // getters and setters
}
```

Listing 1.8. The Event Type Class of the event type *Open* of Account Machine 1

```
public class Active
            extends AbstractPCState<AccountObject> {

    @RepertoireEntry(nextState = Active.class)
    public void transfer(Transfer event,
            IContext context, Target role) {
        oid.notifyMoneyReceived("You␣received␣money:␣" + event.getAmount());
        context.setBalance(
            context.getBalance() + event.getAmount());
    }

    @RepertoireEntry(nextState = Active.class)
    public void deposit(Deposit event,
            IContext context, Into role) {
        oid.notifyMoneyReceived("You␣received␣money:␣" + event.getAmount());
        context.setBalance(
            context.getBalance() + event.getAmount());
    }

    @RepertoireEntry(nextState = Active.class)
    public void transfer(Transfer event,
            IContext context, Source role) {
        context.setBalance(
            context.getBalance() - event.getAmount());
    }

    @RepertoireEntry(nextState = Active.class)
    public void withdraw(Withdraw event,
            IContext context, From role) {
        context.setBalance(
            context.getBalance() - event.getAmount());
    }

    @RepertoireEntry(nextState = Closed.class)
    public void close(Close event,
            IContext context, Account role) {
    }
}
```

Listing 1.9. The *Active* state of Account Machine 1

Account Machine 4 Account Machine 4 (AM4) is a protocol machine with a derived state. Thus its structure differs slightly from AM1. Figure 7 gives an overview of the classes and interfaces in the package. The package does not contain Role Classes or Event Type Classes, because the machine relies on the classes already stated by AM1. The Protocol Machine Class for AM4 is depicted in listing 1.10. It especially contains the State Function Method *getCurrentState*, which shows the implementation of the state function in Java. All other classes are built in the ways already stated and do not include anything surprising.

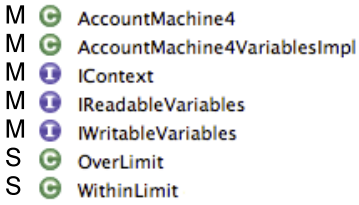

Fig. 7. The package structure of protocol machine for Account Machine 4. The annotations mean: (M) Machine Class, Variable Class and interfaces; (S) State Classes. The c in a circle denotes a class. The i in a circle denotes an interface.

```
public class AccountMachine4 implements IProtocolMachine {

    @MachineOID
    Account oid;

    @MachineContext(
        localState = AccountMachine4VariablesImpl.class,
        localStateRead  = IReadableVariables.class,
        localStateWrite = IWritableVariables.class )
    IContext context;

    public Class
            <? extends AbstractPCState<AccountObject>>
            getCurrentState() {
        if (context.getBalance() < -50)
            return OverLimit.class;
        else
            return WithinLimit.class;
    }

    @RepertoireEntry(nextState = WithinLimit.class)
    public void withdraw(Withdraw event,
            IContext context, From role) {
    }
}
```

Listing 1.10. The implementation of the Machine Class for Account Machine 4

Bank System The protocol system *Account System* (AS) composes AM1 to AM4 (AM2 and AM3 are not shown in this paper). Following the source code structures defined in section 3.2, the AS consists of one Protocol System Class

(listing 1.11) and the System Context Interface, which is an interface without any own operations. Figure 8 shows how the class structure is influenced by the system. IContext of AM1 no longer extends the Read Interface of AM1, but the ISystemContext of the AS. The ISystemContext extends the Read Interfaces of all composed machines (only AM1 and AM4 are shown in this figure). Therefore each machine has read access to all variables in the environment.

```
@ProtocolSystem({   AccountMachine1.class, AccountMachine2.class,
                    AccountMachine3.class, AccountMachine4.class })
public class BankAccountSystem implements IProtocolMachine {
    @SystemEnvironment
    ISystemContext context;
}
```

Listing 1.11. The source code of the System Class of the Bank Account System

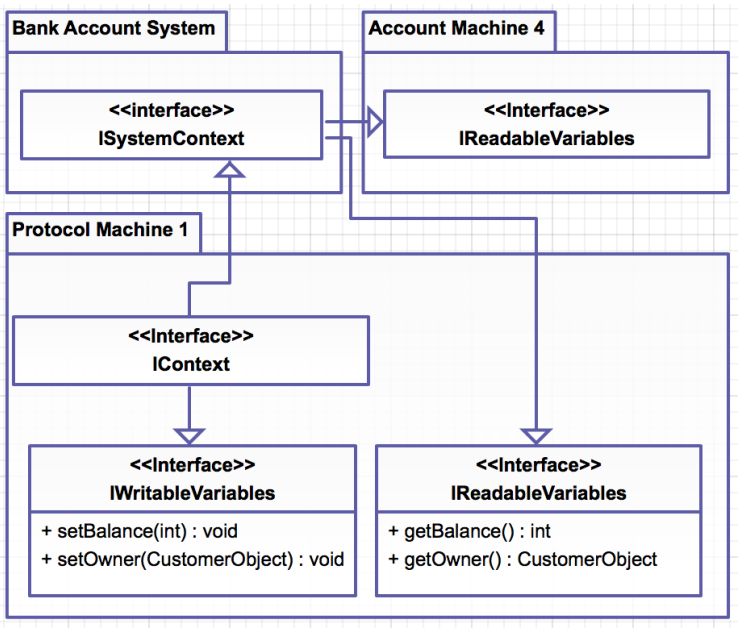

Fig. 8. The composition by the Account System has an influence on the source code structure of the Account Machine 1. IContext no longer extends IReadableInterface, but the ISystemContext. The ISystemContext extends the Read Interfaces of all composed machines (only AM1 and AM4 are shown in this figure). Therefore each machine has read access to all variables in the environment.

4 Execution Runtime

The Protocol Contracts meta model is executable. The runtime is in a prototype state. It is divided into a *bytecode to model extractor*, and a *protocol model execu-*

tor. The bytecode to model extractor parses bytecode to read the protocol model structures encoded as presented in section 3.2 using code reflection mechanisms. From these structures it builds the design time model, on which the run time model is based. The protocol model is then available as Ecore model in memory.

The protocol model executor manages the model. I.e. it provides interfaces for clients to interact with the protocol model. For executing the model, an instance of event classes can be created, filled with attribute values and presented to the execution runtime. The execution runtime uses the model information at run time, e.g. to create machine instances, switch the stored states, and uses calls to the operations of the source code structures for executing update specifications. It then reports about the acceptance of the event, which can be *accepted*, *ignored*, or *refused*.

These calls enable the source code structures to contain interfaces to model-external code. The update specification of a repertoire entry may contain calls to the underlying object using the attribute *oid*. The update specification is encoded as Java method, which is called by the runtime in an inversion-of-control pattern. When the control flow hits the underlying object, the model-external code is executed.

4.1 Run Time Meta Model

At run time, instance of machines, events, and related model elements are created. Figure 9 shows the elements for instantiating machines. Protocol systems and protocol machines each have model elements for their instantiation. They are build analogously to their type specification. *ProtocolMachineInstances* contain *MachineAttributeInstances*, which represent the actual values of the value types in the local storage. *MachineRoleAttributeInstances* represent the references to OIDs in machine instances, which are related to a role. All elements have relations to their respective type elements. This is not shown in the figure for readability reasons.

Event instances (see figure 10) are represented as *EventInstance* elements. They reference their type. They contain *EventAttributeInstances*, value-based attributes, and *Object2RoleEntries*, which map OIDs as *PCObjectInstances* to roles.

4.2 Monitoring and Debugging

The execution runtime offers a web service to register a monitor for a running protocol model. The monitor can be informed about the current system state (including the design time mode, and the runtime model), and about incoming events. A debugging interface allows to change values of machine attributes. The monitoring and debugging interfaces are currently in a prototype state.

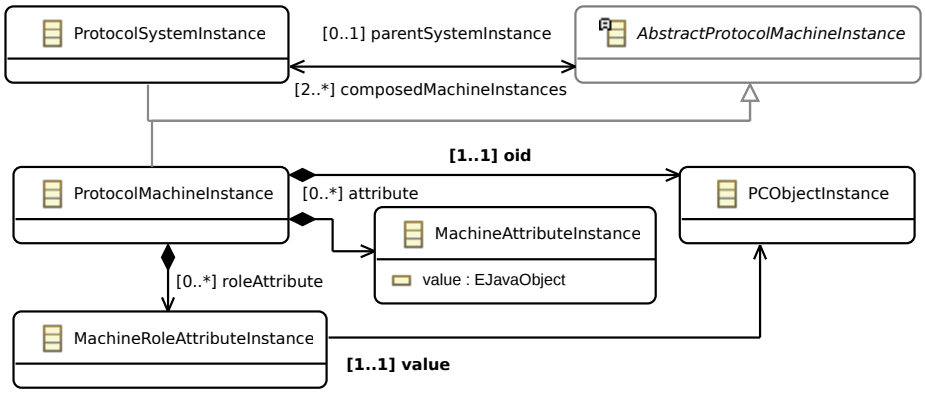

Fig. 9. Protocol machines instantiation in the Protocol Contracts meta model

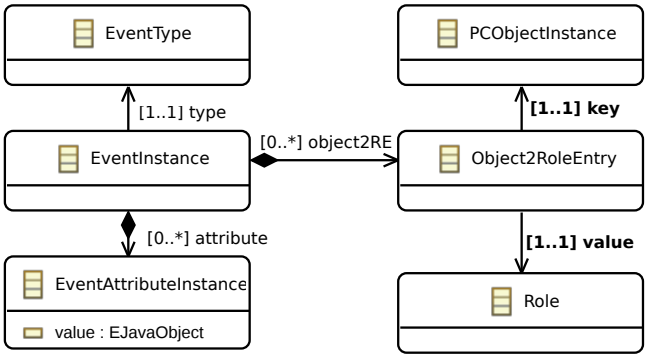

Fig. 10. Event instantiation in the Protocol Contracts meta model

5 Model Editor

The editor is divided into a *source code to model extractor/adapter* to extract the model from source code, and the model editor. The extractor/adaptor uses the Java Development Tools (JDT)[3] to parse code structures for building an Ecore model of the protocol model described in the source code, while keeping the trace links to the code. Technically, these trace links are java objects that relate source code elements to model objects, while providing methods to translate the one into the other. When the model is changed in the editor, the changes are reflected in the code, following the trace links. Therefore the source code is not overridden, but changed. The editor itself is a standard Ecore editor in Eclipse. The extractor/adaptor is not fully implemented yet.

6 Discussion

Our integration of Protocol Contracts follows several design decisions. The main variation points are the meta model and the integration mechanisms. The meta model was designed to be close at the description in [7]. As some parts of the example were not completely described, we cannot be sure that the meta model is in a final version.

Two attributes in the model are strings without semantics. The attribute *spec* in the class *StateFunction*, and the attribute *updateSpec* in the class *RepertoireEntry*. Both contain Java source code that is part of the model definition. This might seem inconsequent. These method bodies are, however, entry points for model-external code. The operations called upon the OID objects are not semantically encoded in the model.

The work presented in this paper are part of the research project ADVERT[4] that aims at using Architecture-Carrying Software for solving evolution challenges in long-living software. We plan to integrate the meta model for Protocol Contracts with the meta model for architecture descriptions from this research project. Therefore we also expect slight changes to the meta model for integration purposes.

The integration mechanisms presented in this paper are designed for Java programs. The model execution is event-based. One could possibly create other integration mechanisms that better integrates with already existing event-based communication frameworks. In the research project mentioned before, we provide integration paths to multiple runtime frameworks. Therefore we expect to create other integration mechanisms. These can, however, base largely on the mechanisms presented in this paper.

[3] https://eclipse.org/jdt/
[4] http://advert-project.org

7 Future work

As future work, we plan to further evaluate the concept and implementation, and integrate it in our framework for Architecture-Carrying Software. This includes that the Protocol Contracts are included into an existing architecture-modeling language. The architecture languages, on which ACS is based, typically have components and their interconnections as first-class entities. We intend to create a mapping from *oids* to component instances, and object types to component types. We can then define the behaviour of component types with Protocol Contracts. The interfaces to arbitrary code play an important role here, to allow for behaviour that should not be modeled on an architectural level. However, some details of the integration still have to be inspected.

Furthermore the source code to model extractor/adaptor has still to be developed. Blueprints for such components exist in the context of the ADVERT project for other model types. Therefore we expect no substantial difficulties in the development of this component. Also the execution runtime is currently in a prototype state. We need to test it with further examples to be more confident about its reliability.

The editor is currently in the work. Blueprints for this editor are also available. The editor is based on the standard reflective Ecore editor in the Eclipse IDE. Only the loading and the saving mechanisms will be overridden to extract the model from the code while keeping the trace links between model elements and the code, and to execute the changes on the code when the changes are saved.

8 Related Work

Related work to ours can be found for several aspects. Balz already created an integration for a behaviour model in his PhD thesis [1]. He integrates state machine models. His implementation of state machine models is working in a batch-like mode. I.e. a state machine is started and is executed until it terminates. The integration of Protocol Contracts is working interactively by generating events and presenting them to the protocol model.

Managing multiple representations of software design and specifically architecture has been subject to other fields of research. Related to the paper at hand is the field of Model-Driven Development (MDD) (e.g. [3], [10]) and round trip engineering (e.g. [8]).

MDD concentrates on deriving code from models. The models and the code are two representations of the program that are independently subject to evolution and maintenance. Changes in the specification can be taken over automatically in the implementation. When the program changes in the implementation, these changes cannot be automatically taken over in the specification.

Round trip engineering (RTE) describes techniques to synchronize models and code. The models used in RTE are very detailed and technical, e.g. UML class diagrams. RTE thus allows for two-way synchronization, but does not bridge the gap between abstraction levels, as our approach does.

The work presented here can be seen as part of models@runtime [2]. We have models with a high abstraction level that are not tied to the underlying technology. We have a technology specific runtime to execute the models. In addition, we have defined interfaces between the model and arbitrary source code.

A runtime for Protocol Contracts already exists (see [6]). We did not find extensive information about that runtime. For our runtime we plan to allow for inspecting and debugging of running Protocol Contracts at run time. It is not clear from [6] whether this is possible with the already existing runtime.

9 Conclusion

For behaviour specifications to be more broadly used in information systems, we believe that a framework needs to be available, which imposes a low barrier for its integration. We identified six criteria that a possible solution has to fulfill to be a candidate for a broader use in this particular domain. In this paper we presented our proposal for a framework for developing and executing Protocol Contracts in the domain of information systems. We evaluated the functionality in a small desktop example. The evaluation shows that the meta model and source code structures are suitable to model Protocol Contracts. Our implementation meets the identified criteria:

- **Ease of Application Integration** Our framework allows the Protocol Contract implementations to call external code during state transitions. This includes information interchange with program code outside of the model implementation. The oid objects represent conceptual interfaces between the model and the model-external code. This allows to integrate such a formal behaviour model incrementally into existing applications.
- **Ease of Platform Integration** The framework is implemented in Java as a library. The Protocol Contract implementation consists of code following the necessities of this framework code and a dependency to the execution runtime implementation. It is therefore easy to embed into typical information systems implemented in Java. Model instances can be created from information system platform code, and platform functions can be called from within the models. This allows to integrate the formal modeling execution framework into information system platforms.
- **Integrated Editor** We are currently developing an editor that is integrated with the Eclipse IDE, and reuses much of this IDE's concepts and code. However, the editor is not available yet.
- **Ease of Development Tool Integration** The model specification is readable and editable without an explicit model editor. A text editor suffices, but a comprehensive Java editor is recommended to read and edit the code that represents the model. As the model specification is based only on program code, it can be easily managed with source code management systems and other Java tools.

- **Possibility to Monitor** A monitoring interface for the execution runtime exists as a prototype, which allows to be informed about the current system state and about executed events.
- **Possibility to Debug** A debugging interface for the execution runtime exists as a prototype, which allows to read and change machine attributes at run time. This is only a starting point. An extension of this interface should be able to also edit e.g. running machine instances and types. However, as the model is based on Java code, especially the update specifications of transitions and the state functions can be edited at run time with the standard Java debugging mechanism.

We see that all of the criteria are fulfilled to a certain extend. The fewest developed criterion is the integrated editor, which is in the work, but not available yet.

Acknowledgements

Parts of the meta model presented in this paper are based on the work of Noyan Kurt from the institute paluno at the University of Duisburg-Essen. 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: Design For Future – Managed Software Evolution.

References

1. M. Balz. *Embedding Model Specifications in Object-Oriented Program Code: A Bottom-up Approach for Model-based Software Development.* PhD thesis, Universität Duisburg-Essen, Mai 2011.
2. G. Blair, N. Bencomo, and R. France. Models@ run.time. *Computer*, 42(10):22–27, Oct 2009.
3. A. Brown, J. Conallen, and D. Tropeano. Introduction: Models, Modeling, and Model-Driven Architecture (MDA) Model-Driven Software Development. In S. Beydeda, M. Book, and V. Gruhn, editors, *Model-Driven Software Development*, chapter 1. Springer, Berlin/Heidelberg, 2005.
4. C. A. R. Hoare. *Communicating sequential processes*, volume 178. Prentice-hall Englewood Cliffs, 1985. `http://www.usingcsp.com/`.
5. M. Konersmann and M. Goedicke. A Conceptual Framework and Experimental Workbench for Architectures. In M. Heisel, editor, *Software Service and Application Engineering*, volume 7365 of *Lecture Notes in Computer Science*, pages 36–52. Springer Berlin Heidelberg, 2012.
6. A. T. McNeile and E. E. Roubtsova. Programming in Protocols - A Paradigm of Behavioral Programming. In C. Gonzalez-Perez and S. Jablonski, editors, *ENASE*, pages 23–30. INSTICC Press, 2008.
7. A. T. McNeile and N. Simons. Protocol modelling: A modelling approach that supports reusable behavioural abstractions. *Software and System Modeling*, 5(1):91–107, 2006.

8. U. A. Nickel, J. Niere, J. P. Wadsack, and A. Zündorf. Roundtrip Engineering with FUJABA. In *Proc of 2nd Workshop on Software-Reengineering (WSR), Bad Honnef, Germany*, 2000.
9. OMG. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1, August.
10. T. Stahl, M. Voelter, and K. Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.
11. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.