

Integrating Protocol Contracts with Java Code

Marco Konersmann, Noyan Kurt, Michael Goedicke

plauno - The Ruhr Institute for Software Technology

University of Duisburg-Essen

Gerlingstraße 16, 45127 Essen, Germany

{marco.konersmann, noyan.kurt, michael.goedicke}@paluno.uni-due.de

ABSTRACT

Long-living software faces the developers with challenges of program understanding. This is intensified by the problem of often outdated or missing documentation. When up-to-date models are available, programs are easier to understand. To achieve this, we present our integrated Protocol Contracts. Integrated Protocol Contracts are a behaviour model type that is integrated with the source code of programs. The code is thoroughly intertwined with the code, so that the model can always be reliably extracted from the code.

Categories and Subject Descriptors

D.2.2 [Software Engineering]: Design Tools and Techniques; D.2.3 [Software Engineering]: Coding Tools and Techniques—*Program editors*

General Terms

Design

Keywords

Protocol Contracts, Behaviour Modeling, Architecture-Carrying Software, Knowledge-Carrying Code

1. MOTIVATION

When long-living software evolves, developers need to understand the software and the code structures that build the software. It seems to be easier to understand abstract models of the code, than the code itself. However, during the evolution of long-living software, models tend to become outdated or might be missing.

Our approach to this problem 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
BM-FA '14, July 22 2014, York, United Kingdom

<http://dx.doi.org/10.1145/2630768.2630771>.

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 suitable for many applications, because often one wants to interactively influence the behaviour of a subsystem. In this paper, we present our ongoing work, where we integrate Protocol Contracts with ACS.

2. CONCEPTUAL FOUNDATIONS

2.1 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.

2.1.1 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.

2.1.2 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

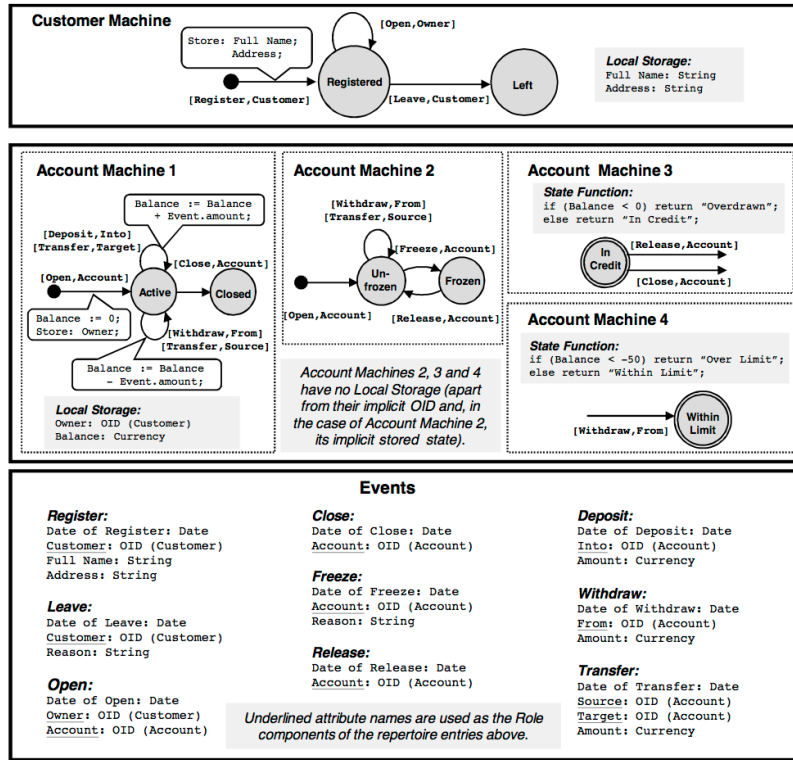


Figure 1: An example system specified as Protocol Contract (Source: [6])

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

2.1.3 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.

2.1.4 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.

2.2 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. 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 integrate Protocol Contracts with ACS.

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.

In the following we present the meta model and the integration mechanisms for Protocol Contracts in the Java programming language. We also give an outlook on towards an execution runtime for the model.

3.1 Meta Model Implementation

The ACS prototype is implemented in Java and based on Ecore models¹. Therefore, the meta model is implemented in Ecore. In this section, we describe the meta model for our implementation of Protocol Contracts, which is based on the description from McNeile and Simons [7].

3.1.1 Machine Types

There are two machine types represented in our meta model. One is *Protocol Machine*, while the other is the *Protocol System*. In our model (figure 2) a *ProtocolSystem* always composes at least two machines. If a system composes another system, the system it composes, again, has to contain at least two machines. Therefore, the class *ProtocolSystem* represent a shell for *ProtocolMachine* instances.

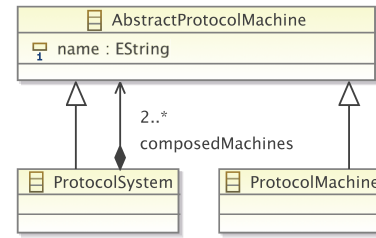


Figure 2: Machine types in the Protocol Contracts meta model

The *ProtocolMachine* contains 1 *statetype*, at least 1 *repertoire entry*, if necessary a *role*, and again at least 1 *event-Type* (see figure 3): The state type 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 repertoire entries in our model. The event type and role of

¹for information on Ecore, please see [11]

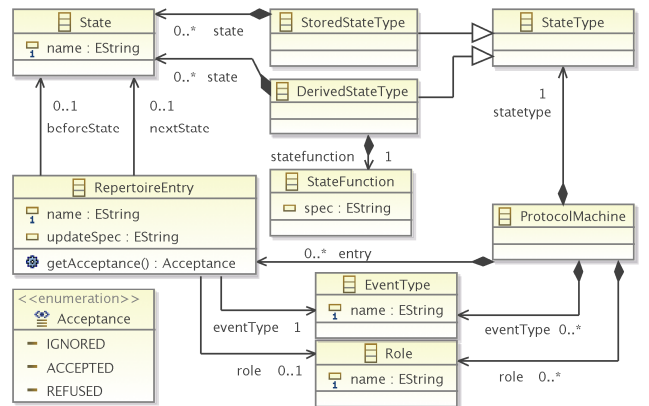


Figure 3: The repertoire of protocol machines in the Protocol Contracts meta model

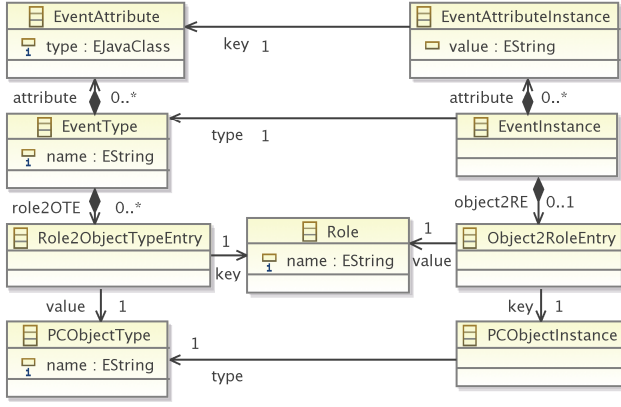


Figure 4: Event types and events in the Protocol Contracts meta model

the repertoire entry are represented as references to the respective model elements. The updates defined by repertoire entries are also determined in the entries. Each entry knows the *beforeState* and the *nextState*, if applicable. This represents the update of the state variable. Furthermore, the *RepertoireEntry* contains the operation *getAcceptance()* and an attribute *updateSpec*. The operation *getAcceptance()* returns the value *ignored*, *accepted*, or *refused*, when an event is presented to the machine. The *updateSpec* determines, similarly to *spec* in the *StateFunction*, the update operation.

The class *EventType* (see figure 4) may contain *EventAttributes* with the attribute *type*, being of the type of a Java class. In addition to that, of each *EventType*, we can create *EventInstances*, which know their type. Again, this *EventInstance* may contain *EventAttributeInstances*, which represent the *EventAttributes*. Furthermore, the *EventType* may contain a *Role2ObjectEntry*, which binds the *Role* in an event type to a *PCObjectType*.

In parallel to the class *EventType*, we also can create a *ProtocolMachineInstance* of the type *ProtocolMachine* (see figure 5). A *ProtocolMachineInstance* may contain a *MachineAttributeInstance*. This represents the instantiation of protocol machines during the run time of the system.

The *PCObjectType* in our model represents the object referenced by a machine. When a *ProtocolMachineInstance* of a *ProtocolMachine* is created, a *oid* must be assigned to the machine instance. The *ProtocolMachineInstance* also may know its *currentState*.

3.2 Integration Mechanisms

For describing Protocol Contracts as model type for ACS, source code structures for the meta model elements have to be defined. In the following sections, we describe the source code structures that represent Protocol Contracts in Java code. Here we only describe source code structures for type level elements. Elements that represent the instantiation of machines and events are only used by the model execution runtime.

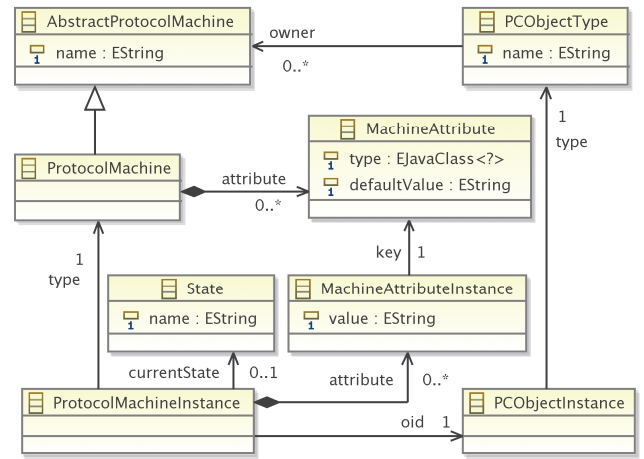


Figure 5: Protocol machine attributes and their instantiation in the Protocol Contracts meta model

3.2.1 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*). 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 subpackages². 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 called *oid*. The object type of the *oid* is a simple Java class. The type of the attribute is that class. The variable is also marked with a *MachineOID* annotation for convenience.

3.2.2 Machine Attributes

Machine attributes are represented as a variable with the respective type, a get method, and a set method. A *Variable Class* contains the stored state (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 changing and reading 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. Due to the interface and class structure described above, the Context Interface allows for reading and

²This 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 {

    Class<? extends AbstractPCState<AccountObject>>
        currentState = null;

    @MachineOID
    PCObjectTypeName oid;

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

```

Listing 1: The source code structure for a Protocol Machine Class with the reference to a Machine Attribute Class

```

public class StateName
    extends AbstractPCState<PCObjectType> { }

```

Listing 2: The source code structure for a state

writing the stored state of the machine. An annotation on the variable states the Read Interface, the Write Interface, and the Variable Class. Listing 1 shows how the Machine Class is built with the Variable Class.

3.2.3 States

States are represented as a Java class that extends the abstract class *AbstractPCState* (see listing 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 object of the type *IActor*. This is the interface to arbitrary, non-architectural code. Therefore each state also contains the information, which object the machine represents.

3.2.4 State Variable

The state variable in Protocol Contracts can be built in two ways. In topological 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.

Machine Classes of topological state protocol machines contain a variable *currentState* of the type *Class<? extends AbstractPCState>*. It thus references a class that represents a state (see listing 1).

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.

3.2.5 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

```

public class EventTypeName implements IEventType {

    ObjectClassName roleName;

    AttributeType attribtueName;

    // getters and setters
}

```

Listing 3: The source code structure for event types, including event attributes and roles

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 3 shows the source code structure for an Event Type Class.

3.2.6 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 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 implements the method, the source state is the initial pseudo state. When a Protocol System Class implements the method, there is no necessary source state. 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 Role Class (role). The parameter event represents the event type. The context parameter is used for the update specification. The role is given by the type of the parameter (role). The next state is given in the method's annotation as a 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.

3.2.7 Protocol Systems

The source code representation of a protocol system is a *Protocol System Class* (see listing 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 IProto-

```

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

```

Listing 4: The source code structure for repertoire entries

```

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

    @SystemEnvironment
    ISystemContext context;
}

```

Listing 5: The source structure for Protocol Systems

colMachine interface. One package may only contain either one Protocol System Class or one Protocol Machine Class. Subpackages may contain further Protocol Machines or 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 6).

3.2.8 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 Runtime

The Protocol Contracts meta model is executable. The runtime, however, is not finished yet. The runtime will extract model information from source code structures using Java reflection mechanisms and a pattern matching algorithm. The protocol model is then available as Ecore model. The runtime manages the model. I.e. it provides interfaces for clients to interact with the protocol model.

For executing the model, the runtime provides an interface for creating instances of the class *EventTypeInstance*. These event instances can then sent to another interface, and thus be represented to protocol models managed in the runtime. The runtime uses the model information at runtime, e.g. to switch the stored states, and uses calls to the operations of

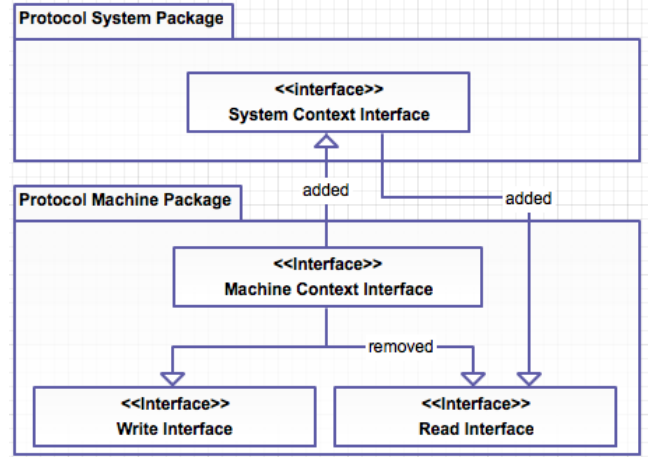


Figure 6: The given interface and class structure ensures that the variables of all composed machine are readable by every machine in the system, and that each machine can only alter its own variables.

the source code structures for executing update specifications.

These calls enable the source code structures to contain interfaces to non-model 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, arbitrary, architecturally irrelevant code can be executed.

4. 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 topological states; account machine 4, a protocol machine with derived states; and the account system, a protocol system.

4.1 Account Machine 1

Account machine 1 (AM1) is a protocol machine with topological states. All of the classes for AM1 are placed in the same Java package. Figure 7 gives an overview of the classes and interfaces in the package. The Protocol Machine Class for AM1 is depicted in listing 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 8 shows the class structure of the machine attributes for AM1 (without the influence of the system, that composes the machine). The interfaces shown in this figure contain get and set meth-




















R  Account
 M  AccountMachine1
 M  AccountMachine1VariablesImpl
 O  AccountObject
 S  Active
 E  Close
 S  Closed
 O  CustomerObject
 E  Deposit
 R  From
 M  IContext
 R  Into
 M  IReadableVariables
 M  IWritableVariables
 E  Open
 R  Source
 R  Target
 E  Transfer
 E  Withdraw

Figure 7: 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 {

    Class<? extends AbstractPCState<AccountObject>>
        currentState = null;

    @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 6: The implementation of the Protocol Machine Class for Account Machine 1

```

public class AccountMachine1VariablesImpl
    implements IReadableVariables,
        IWritableVariables {

    int balance;

    CustomerObject owner;

    // getters and setters

}

```

Listing 7: The machine attribute class of Account Machine 1

```

public class Open implements IEventType {

    Date dateOfOpen;

    AccountObject account;

    CustomerObject owner;

    // getters and setters

}

```

Listing 8: The Event Type Class of the event type Open of Account Machine 1

ods according to their task. The implementing class is shown in listing 7.

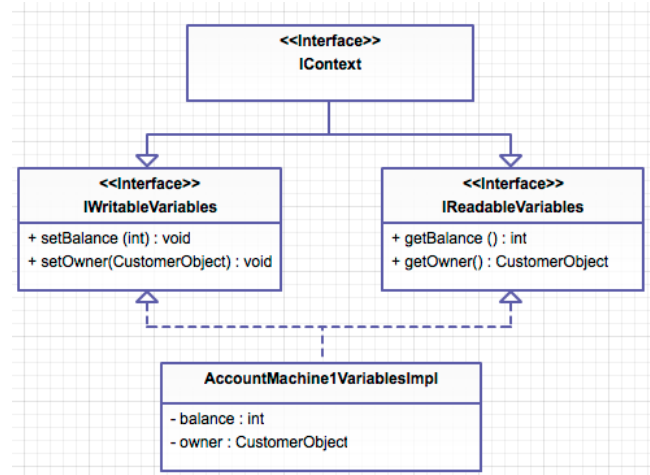


Figure 8: The class structure for the machine attributes of Account Machine 1

The Event Type Class of Open is shown in listing 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 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.

```

public class Active
    extends AbstractPCState<AccountObject> {

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

    @RepertoireEntry(nextState = Active.class)
    public void deposit(Deposit event,
        IContext context, Into role) {
        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 9: The *Active* state of Account Machine 1

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.

4.2 Account Machine 4

Account Machine 4 (AM4) is a protocol machine with a derived state. Thus its structure differs slightly from AM1. Figure 9 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 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.

4.3 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 11) and the System Context Interface, which is an interface without any own operations. Figure 10 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

```

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 10: The implementation of the Protocol Machine Class for Account Machine 4

this figure). Therefore each machine has read access to all variables in the environment.

5. 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. We also expect the meta model to slightly change during the development of the runtime, when we get new insights about the instance level.

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. We need to evaluate whether we will represent the expressions encoded there in more semantically rich forms.

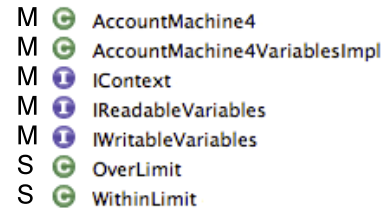


Figure 9: The package structure of protocol machine for Account Machine 4. The annotations mean: (M) Machine Class, Variable Class and interfaces; (S) State Classes. The c in a circle denotes a class. The i in a circle denotes an interface.


```

@ProtocolSystem({ AccountMachine1.class,
                  AccountMachine2.class,
                  AccountMachine3.class,
                  AccountMachine4.class })
public class BankAccountSystem
    implements IProtocolMachine {

    @SystemEnvironment
    ISystemContext context;
}

```

Listing 11: The source code of the Protocol System Class of the Bank Account System

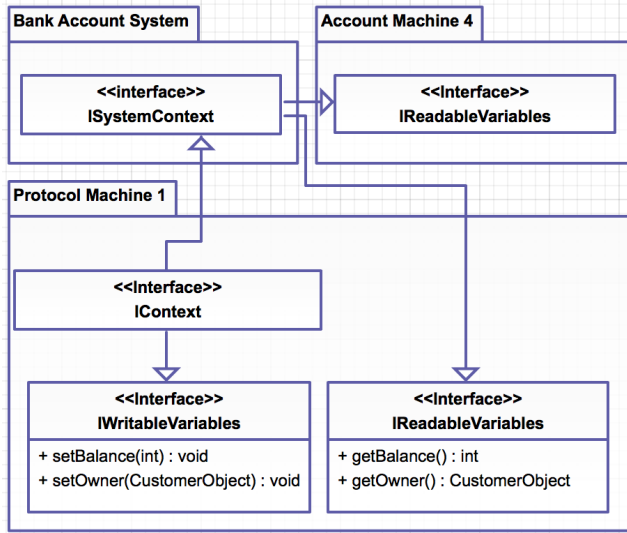


Figure 10: The composition by the Account System has an influence on the source code structure of the Account Machine 1. `IContext` no longer extends `IReadableInterface`, but the `ISystemContext`. The `ISystemContext` extends the Read Interfaces of all composed machines (only AM1 and AM4 are shown in this figure). Therefore each machine has read access to all variables in the environment.

The work presented in this paper are part of the research project ADVERT³ 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<http://advert-project.org>

6. FUTURE WORK

This paper presents our work in an early stadium. 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 an editor will be created for the Protocol Contracts based on our meta model that enables to edit the models at design time and inspect and debug them at run time. The next step however is the development of a model execution runtime.

7. EVALUATION PLAN

7.1 Functionality

Our meta model and source code fragments have been evaluated for functionality in a desktop example (see section 4). First we created a model from the meta model that represents the example system shown in [6]. Then we manually created source code following the defined structures. By construction, the example shows that the source code structures are suitable to represent the model. The completeness of the model shows that the meta model is suitable for Protocol Contracts, as they are necessary for the given example system.

The meta model is not complete yet. We did not evaluate whether our meta model and source code structures are suitable to represent nested protocol machines, since the example in [6] does not contain this structure.

We also did not evaluate our instance level meta model elements, because the model execution runtime does not exist yet.

7.2 Ease of Understanding

The work presented in this paper is embedded into a research project where we build Architecture-Carrying Software. This technique allows the software to be shown and edited at design time, and inspected and debugged at run time, based on models. We plan to develop an editor for editing Protocol Contracts at design time, and to inspect and debug them at run time.

Our hypothesis is that this editor, in combination with the integrated Protocol Contracts, allows understand the software behaviour more easily than manually written or generated code with adjacent models. To evaluate this hypothesis, we plan to execute a study with students in a controlled experiment. The students will be presented two existing programs that have to be extended due to change requests. The programs will be functionally equal. One program is developed using our Protocol Contracts runtime and editor

(T1). The other program will be developed without our runtime and editor (T2). Each student will have to edit both programs with the same change requests. One half of the students group will start with T1, the other half will start with T2. We plan to monitor the time necessary to execute the changes.

Another study in an equal setup is planned for run time debugging. This study differs in the task. Instead of changing the functionality of the program, the students will be faced with the task to identify and solve a bug that is visible due to a run time error.

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

In this paper we presented how we integrate Protocol Contract with Java source code. 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. The work presented here is in an early stadium and is thus not evaluated thoroughly yet.

Acknowledgements

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.

10. 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.