A Metamodel for Modelling of Component-Based Systems with Mobile Architecture

Marek Rychlý

Abstract Current information systems tend to be distributed into networks of quite autonomous, but cooperative, components communicating asynchronously via messages of appropriate formats. Loose binding between those components allows to establish and destroy their interconnections dynamically at runtime, on demand, and according to various aspects; to clone the components and to move them into different contexts; to create, destroy and update the components dynamically at runtime; etc. Modelling of the dynamics and mobility of components brings many issues that can not be addressed by means of conventional architecture description languages. In this paper, a metamodel for modelling of component-based systems with mobile architecture is proposed.

Key words: Metamodel, Component-based system, Mobile architecture

1 Introduction

Globalisation of information society and its progression create needs for extensive and reliable information technology solutions. Several new requirements on information systems have emerged and significantly affected software architectures of these systems. The current information systems can not be realised as monoliths, but tend to be distributed into networks of quite autonomous, but cooperative, components communicating asynchronously via messages of appropriate formats. Loose binding between those components allows to establish and destroy their interconnections dynamically at

Marek Rychlý
Department of Information Systems, Faculty of Information Technology, Brno University of Technology, Božetěchova 2, 612 66 Brno, Czech Republic, e-mail: `rychly@fit.vutbr.cz`

runtime, on demand, and according to various aspects (e.g. quality and cost of services provided or required by the components); to clone the components and to move them into different contexts; to create, destroy and update the components dynamically at runtime; etc.

Modelling of the dynamics and mobility of components (i.e. the systems' dynamic or mobile architecture [11], respectively) brings issues that can not be addressed by conventional architecture description languages. Models have to provide description of hierarchical composition of components, description of binding of neighbouring components' interfaces as well as interfaces received via mobility, description of interfaces providing and requiring a component's functional operations, and description of interfaces with operations controlling the component's life-cycle, binding of its interfaces and its mobility.

In this paper, a metamodel for modelling of component-based systems with mobile architecture is proposed. Section 2 outlines the current state-of-the-art in modelling of dynamic and mobile architectures. Section 3 describes the proposed metamodel. An application of the metamodel is demonstrated in Section 4 as a model of a component-based system implementing a service of a railway interlocking control system. To conclude, in Section 5, we summarise the contribution of this paper and outline the future work.

2 State of the Art and Motivation

There are several architecture description languages that support modelling of component-based systems (e.g. component diagrams in UML [1], language ACME [8], etc.). They allow to describe a logical (structural) view of a component-based system, i.e. basic entities, their relations and features. However, the most of them do not support mobile or even dynamic architecture.

The *component model Fractal* [5], which is a general component composition framework, provides a notation for description of dynamic architecture of hierarchically nested components [3], but without component mobility. The *component model SOFA 2.0* [6] introduces a MOF-based metamodel for dynamic architecture, which is able to describe passing references of "utility" interfaces, i.e. limited mobility of the specific type of interfaces. The *architecture description language ArchWare* [2] provides constructs to describe dynamic software architectures by means of a specific UML 2 profile [12] and their behaviour by means of π-calculus [11]. Although the ArchWare is more advanced than the previously mentioned component models, it also does not directly address component mobility.

The motivation of the approach proposed in this paper is to address component mobility and design a MOF-based metamodel for component-based systems with mobile architecture. The metamodel will support subsequent description of the systems' behaviour by means of π-calculus.

3 Metamodel

The component model for mobile architectures is described as a metamodel in the context of a four-layer modelling architecture. The metamodel is implemented in OMG's *Meta Object Facility* (MOF, [9]), which is used as a meta-metamodel. The modelling architecture comprises the four layers:

M0: An information layer, which is comprised of the actual data objects. This layer contains particular instances of component-based systems, their runtime configurations, specific deployments of their components and connectors, etc.

M1: A model layer, which contains models of the M0 data. The models include structure and behaviour models that describe different perspectives of component-based systems such as, for example, UML component models or communication diagrams.

M2: A metamodel layer provides a language for building M1 models. Component models fall in this layer, as well as models of the UML language.

M3: A meta-metamodel layer, which is used to define modelling languages. It holds a model of the information from M2, e.g. MOF.

In the context of component-based development, a specific component-based system (in layer M0) contains instances of elements from its model (from layer M1). The model contains instances from a specific component model (a metamodel in layer M2), which is described by a given meta-metamodel (in layer M3).

The proposed metamodel is defined by means of *Essential MOF* (EMOF), which is a part of MOF in layer M3. The EMOF contains packages `Basic`, `Reflection`, `Identifiers`, and `Extension`, which form a minimal set of modelling elements to define simple metamodels. *Complete MOF* (CMOF), which is second and the last part of MOF, extends EMOF by `Constructs` package from UML 2 Core [10].

The component model, as a model of layer M2, can be described by means of UML 2 diagrams in two contexts:

1. as an object diagram of instances of EMOF classes from layer M3 (entities in layer M2 are instances of classes in M3), i.e. it is described as "a model",
2. as a class diagram from layer M1 (entities in layer M1 are instances of classes in layer M2), i.e. it is described as "a metamodel".

For better clearness, the component model will be described as an UML 2 class diagram from layer M1. To reuse well-established concepts of MOF, the component model's metamodel extends EMOF classes `EMOF::NamedElement`, `EMOF::TypedElement`, and `EMOF::Operation`, which are outlined in Figure 1. A complete and detailed definition of the EMOF classes can be found in [9].

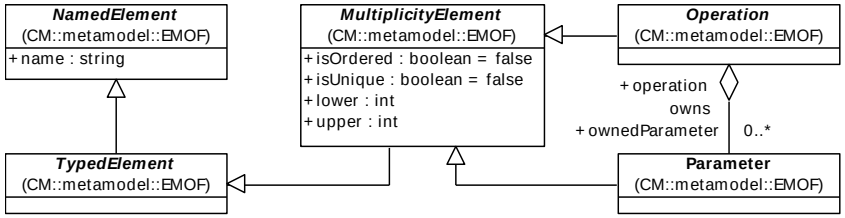

Fig. 1 A simplified part of the EMOF metamodel [9] with classes that will be extended by the component model.

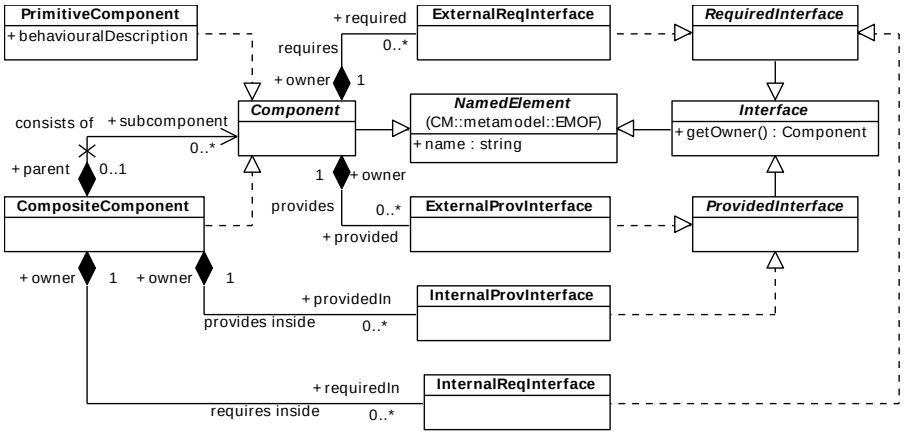

Fig. 2 Abstract component, realisations, and interfaces, extending `EMOF::NamedElement` in the metamodel of the component model.

3.1 Components and Interfaces

Figure 2 describes the first part of the component model as an extension of EMOF. The metamodel defines an abstract component, its realisations as a primitive component and a composite component, and their interfaces. All classes of the metamodel inherits (directly or indirectly) from class `EMOF::NamedElement` in package `Basic` of EMOF.

In our approach, a *component*, which is an active communicating entity of a component-based software system, can be described form two sides: as an abstract component without considering its internal structure ("black-box" view) and as a component realisation in the form of a primitive component or a composite component ("grey-box" view). The *abstract component* (class `Component` in the metamodel) can communicate with neighbouring components via its interfaces (class `Interface`). The interfaces can be provided (class `ExternalProvInterface`) or required (class `ExternalReqInterface`) by the component.

The component realisation can be primitive or composite. The *primitive component realisation* (class `PrimitiveComponent`) is implemented directly, be-

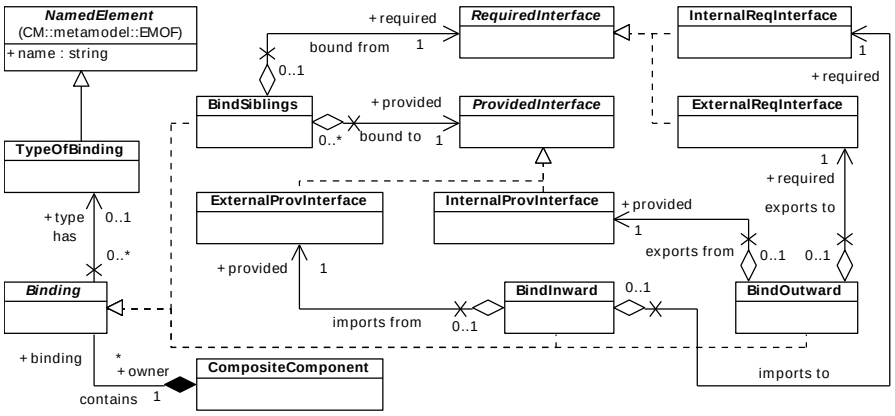

Fig. 3 Binding and its different realisations between interfaces of a composite component realisation in the metamodel of the component model. Classes `CompositeComponent` and `...Interface` are identical to the classes in Figure 2.

yond the scope of architecture description. It is a "black-box" with described observable behaviour (attribute `behaviouralDescription`). The *composite component realisation* (class `CompositeComponent`) is decomposable on a system of subcomponents at the lower level of architecture description (it is a "grey-box"). Those subcomponents are represented by abstract components (class `Component` and relation "consists of"). Moreover, every composite component realisation can communicate with its subcomponents via its provided (class `InternalProvInterface`) and required (class `InternalReqInterface`) internal interfaces (relations "provides inside" and "requires inside", respectively).

The specific interfaces have to implement methods `getOwner()`, which return their owners, i.e. objects that act as the abstract components in a case of the abstract component interfaces or as instances of the composite component realisations in a case of their internal interfaces (in accordance with `owner` roles of components in the relations with their interfaces).

3.2 Composite Components and Binding of Interfaces

Binding is a connection of required and provided interfaces of the identical types into a reliable communication link. It is described in Figure 3. Interfaces of a component (classes `ExternalProvInterface` and `ExternalReqInterface`) can be provided to and required from its neighbouring components, while interfaces of a composite component realisation (classes `InternalProvInterface` and `InternalReqInterface`) can be provided to and required from its subcomponents only. Therefore, we distinguish three types of the binding (the realisations of class `Binding`):

1. Binding *of provided interfaces to required interfaces in the same composite component realisation* is represented by class `BindSiblings`. The interfaces have to be internal interfaces of the composite component realisation or external interfaces of subcomponents in the same composite component realisation[1]. The binding interconnects required interfaces (class `RequiredInterface`) via relations "bound from" to provided interfaces (class `ProvidedInterfaces`) via relations "bound to".
2. Binding *of external provided interfaces* of a composite component realisation *to its internal required interfaces* is represented by class `BindInward`. The external interfaces are provided to neighbouring components of the composite component acting as an abstract component (relation "imports from" an instance of class `ExternalProvInterface`), while the internal interfaces are required from the composite component's subcomponents (relation "exports to" an instance of class `ExternalReqInterfaces`).
3. Binding *of internal provided interfaces* of a composite component realisation *to its external required interfaces* is represented by class `BindOutward`. The internal interfaces are provided to the composite component's subcomponents (relation "exports from" an instance of class `InternalProvInterface`), while the external interfaces are required from neighbouring components of the composite component acting as an abstract component (relation "exports to" an instance of class `ExternalReqInterfaces`).

The bindings (i.e. instances of the realisations of class `Binding`) are owned by the composite component realisations. Each binding can have a type (class `TypeOfBinding`), a specialisation of `EMOF::TypedElement`, which can describe a communication style (buffered and unbuffered connection), a type of synchronisation (blocking and output non-blocking), etc.

3.3 Types of the Interfaces

To ensure type compatibility of interfaces in a binding, each interfaces has a type (class `TypeOfInterface`, which is a specialisation of class `EMOF::NamedElement` in package `Basic` of EMOF). Hierarchy of the types of interfaces is described in Figure 4.

According to a scope of visibility of the interfaces in a composite component realisation, we can distinguish public interfaces, private interfaces, and protected interfaces. The *public interfaces* (classes realising `PublicIntType`) of a component can be accessed by its neighbouring components (via binding `BindSiblings`). If the component is a composite component realisation, its external public interfaces can be also accessed by its subcomponents and its

[1] The diagram in Figure 3 does not restrict relations of `BindSiblings` to the interfaces of the same composite component realisations; this will be defined later by means of additional constraints in Section 3.4.

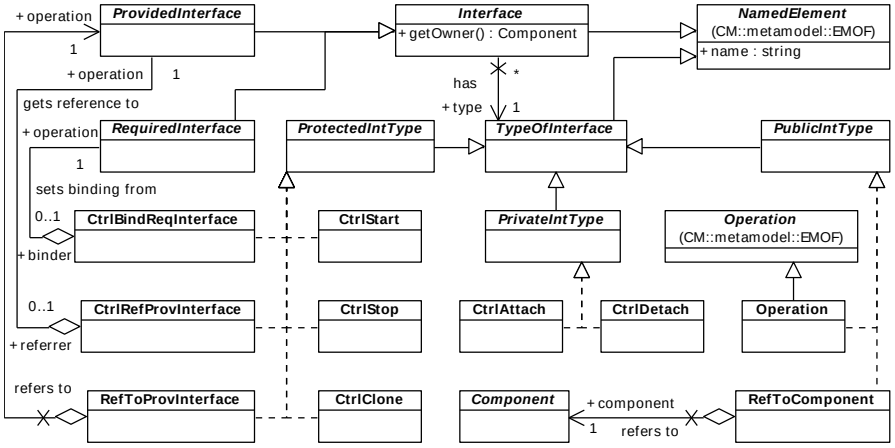

Fig. 4 Types of interfaces with class `Operation` extending `EMOF::Operation` in the metamodel of the component model. Classes `Interface`, `ProvidedInterface`, `RequiredInterface`, and `Component` are identical to the classes in Figure 2.

internal public interfaces can be accessed by its neighbouring components (i.e. the interfaces can pass the component's border via binding `BindInward` and `BindOutward` owned by the component). They can be interconnected by means of all kinds of bindings.

Contrary to the public interfaces, the *private interfaces* (classes realising `PrivateIntType`) are specific types of interfaces, which can be provided only by a composite component realisation and only to its subcomponents as the component's internal interfaces[2]. They can be interconnected only by means of binding `BindSiblings`.

Finally, the *protected interfaces* (classes realising `ProtectedIntType`) of a component can be accessed by its neighbouring components as the component's external interfaces, but if the component is a composite component realisation, they are not reachable by its subcomponents. They can be interconnected only by means of binding `BindSiblings`.

We distinguish the following types of interfaces[3] by their functionality:

- Public interface `Operation`, which extends class `EMOF::Operation` from package `Basic` of EMOF and represents a business oriented service with typed input and output parameters.
- Protected interface `CtrlRefProvInterface` provides references to given provided interface `ProvidedInterface` of type `Operation`[4], while protected in-

[2] The private interfaces can be required by the subcomponents as their external interfaces, but they can not pass borders of the subcomponents (nor any other component). It means that the subcomponents have to be primitive components.

[3] `Operation` denotes *functional interfaces*, while the others denote *control interfaces*.

[4] The restriction to the interface of type `Operation` will be defined explicitly by additional constraints in Section 3.4.

terface `CtrlBindReqInterface` allows to establish a new binding of specific required interface `RequiredInterface` of type `Operation`[4] to a provided interface of another component formerly referred by `CtrlRefProvInterface`.
- Protected interfaces `CtrlStart` and `CtrlStop` allow to control behaviour of a component (i.e. to start and to stop the component, respectively).
- Private interfaces `CtrlAttach` and `CtrlDetach` provided by a composite component realisation allow to attach a new component as a subcomponent of the realisation ("nesting" of the component) and detach an old subcomponent from the realisation, respectively.
- Protected interface `CtrlClone` provides references to fresh copies of a component.
- Protected interface `RefToInterface` is able to pass references of provided interfaces `ProvidedInterface` of type `Operation`[4], while public interface `RefToComponent` allows to pass references of a whole component `Component`, which is required to support component mobility.

3.4 Additional Constraints

We need to define additional constraints to ensure type compatibility of interfaces in bindings, i.e. instances of realisations of class `Binding` in Figure 3. Types of the interfaces are given by relation to specific instances of realisations of class `TypeOfInterface` and according to the hierarchy of the types of interfaces in Figure 4. The following formulae use a first-order logic with extra predicate symbols "$o : \mathrm{T}$" and "o is T" for restriction of o to type `T`, predicate symbol "$i \in \mathrm{L}$" for restriction of l to list `L`, predicate symbol "$x = y$" to check equality of x and y, and function symbol "$\mathrm{i.getOwner}()$" to get an owner of interface `i` (see method `getOwner()` of `Interface` in Section 3.1).

1. Bindings `BindInward` and `BindOutward` in a composite component realisation can interconnect only interfaces of the same realisation.

$$\begin{aligned}&(\forall c : \mathrm{CompositeComponent})\ (\\ &\ ((\forall b : \mathrm{BindInward} \in c.\mathrm{binding})(b.\mathrm{provided.getOwner}() = c\ \wedge\ b.\mathrm{required.getOwner}() = c)) \wedge\\ &\ ((\forall b : \mathrm{BindOutward} \in c.\mathrm{binding})(b.\mathrm{provided.getOwner}() = c\ \wedge\ b.\mathrm{required.getOwner}() = c))\ )\end{aligned}$$

2. Binding `BindSiblings` in a composite component realisation can interconnect only internal interfaces of the same composite component realisation or external interfaces of its subcomponents.

$$\begin{aligned}&(\forall c : \mathrm{CompositeComponent})\ (\forall b : \mathrm{BindSiblings} \in c.\mathrm{binding})\ (\\ &\quad(\forall i : \mathrm{InternalProvInt} \in b.\mathrm{provided})\ (i.\mathrm{getOwner}() = c)\\ &\quad\wedge\ (\forall i : \mathrm{InternalReqInt} \in b.\mathrm{required})\ (i.\mathrm{getOwner}() = c)\\ &\quad\wedge\ (\forall i : \mathrm{ExternalProvInt} \in b.\mathrm{provided})\ (i.\mathrm{getOwner}() \in c.\mathrm{subcomponent})\\ &\quad\wedge\ (\forall i : \mathrm{ExternalReqInt} \in b.\mathrm{required})\ (i.\mathrm{getOwner}() \in c.\mathrm{subcomponent})\ )\end{aligned}$$

3. Bindings `Binding` in a composite component realisation can interconnect only provided interfaces with required interfaces of compatible types.

$$(\forall c : \text{CompositeComponent})\ (\forall b : \text{Binding} \in c.\text{binding})(b.\text{provided.type} = b.\text{required.type})$$

4. Bindings `BindInward` and `BindOutward` can interconnect only public interfaces, i.e. instances of class `PublicIntType`.

$$\begin{array}{l}(\forall c : \text{CompositeComponent})\ (\\ \quad ((\forall b : \text{BindInward} \in c.\text{binding}) \\ \qquad (b.\text{provided.type is PublicIntType} \wedge b.\text{required.type is PublicIntType})\ ) \wedge \\ \quad ((\forall b : \text{BindOutward} \in c.\text{binding}) \\ \qquad (b.\text{provided.type is PublicIntType} \wedge b.\text{required.type is PublicIntType})\ )\ )\end{array}$$

5. Bindings `BindSiblings` that are inside a composite component realisation can be connected to private interfaces, only if the interfaces are internal interfaces of the composite component realisation.

$$\begin{array}{l}(\forall c : \text{CompositeComponent})\ (\forall b : \text{BindSiblings} \in c.\text{binding}) \\ \quad (b.\text{provided.type is PrivateIntType} \Rightarrow b.\text{provided} \in c.\text{providedIn})\end{array}$$

6. Instances of classes `CtrlBindReqInterface`, `CtrlRefProvInterface`, and `RefToProvInterface`, and their relations to interfaces via "sets binding from", "gets reference to" and "refers to", respectively, have to be connected with the interfaces of type `Operation` only.

$$\begin{array}{l}(\forall t : \text{CtrlBindReqInterface})\ (t.\text{operation.type is Operation}) \\ \wedge\ (\forall t : \text{CtrlRefProvInterface})\ (t.\text{operation.type is Operation}) \\ \wedge\ (\forall t : \text{RefToProvInterface})\ (t.\text{operation.type is Operation})\end{array}$$

4 A Case Study of a Component-Based System

As a case study, we create a model of a component-based system (CBS) that implements a specific service of a service oriented architecture (SOA). We adopt a specification of the SOA for functional testing of complex safety-critical systems, more specifically *a testing environment of a railway interlocking control system*, which has been described in [7]. The environment allows to distribute and run specific tests over a wide range of different testing environments, varying in their logical position in the system's architecture. The CBS implements service `TestEnvironment`, which executes a test script received from service `TestManager` via its interface `ExecuteTest` and forwards its results back to `TestManager` via interface `asyncReplyET` when the test script is finished. During its execution, the test script is interacting with a tested environment, which is specific to each instance of service `TestEnvironment`[5]. For a detailed description of the SOA and all its services, see [13].

Railway interlocking control systems are safety-critical systems and can be described as component-based systems [4]. A testing environment of such systems has to interact with the systems' components. For that reason, a

[5] Each rail yard has its own instance of the tested environment with specific sensors and actuators where assigned tests are automatically executed.

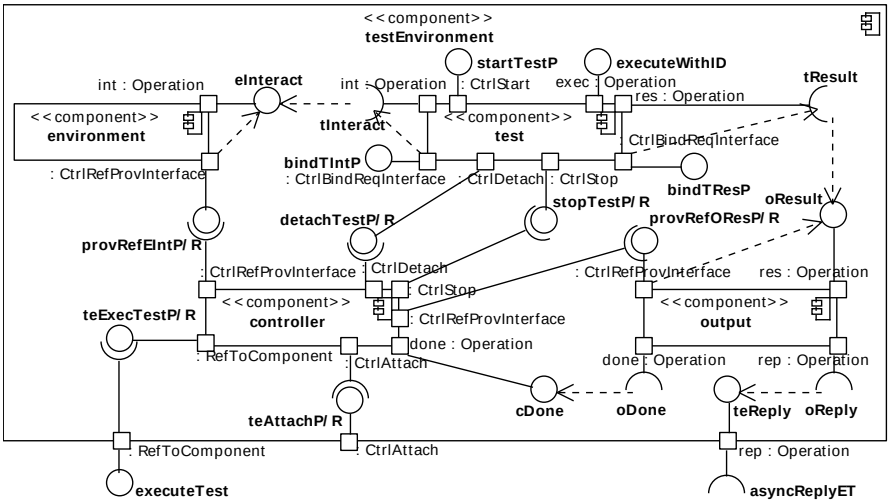

Fig. 5 Composite component TestEnvironment (a specific UML-like notation).

part of the testing environment, which is directly connected to a system under testing (i.e. the tested environment), has character of a component neighbouring to the system and can be described as the CBS. Moreover, the test scripts are distributed to different instance of service `TestEnvironment`, i.e. different parts of the system's architecture, where they interact with local testing environments. The test scripts act as mobile components in the system's architecture, i.e. in mobile architecture.

Figure 5 describes composite component testEnvironment, which represents service `TestEnvironment`. The component model (from layer M1) is described as a specific class diagram where entities of the CBS (from layer M0) will be instances of the depicted classes[6]. The class diagram is figured in a specific modification of UML 2 component diagrams' notation[7].

In Figure 5, components of the CBS are denoted by UML components, i.e. classes stereotyped as «`component`». Provided interfaces are denoted by UML interfaces, i.e. classes stereotyped as «`interface`» realised by related components that own the interfaces, while required interfaces are denoted by UML interfaces used by related components that own the interfaces. A type of a component's interface (see Section 3.3) is denoted by an UML port where the port's (optional) name and (mandatory) type are identical to a name and a class of the type of the component's interface. Bindings of

[6] Another, but less comprehensible, method is to describe the component model as an object diagram where objects in layer M1 are instances of classes from the metamodel in layer M2 and represent abstractions of entities of the CBS from layer M0.

[7] The aim is to simplify the diagram and reuse the well-established UML notation, although it is not formally defined as an UML profile.

functional required and provided interfaces are denoted by UML relations of dependency stereotyped as «use». Each binding can have its (optional) name and its type, if needed. In a case of a nameless binding of interfaces, which is common for control interfaces, it is possible to interconnect the interfaces directly. Relations of UML ports of types `CtrlBindReqInterface` or `CtrlRefProvInterface`, which represent control provided interfaces for binding of required functional interfaces or referencing functional provided interfaces, respectively, are denoted by UML relations stereotyped as «use».

Component testEnvironment receives a test script via provided interface `executeTest`, which is internally processed by component controller. The script is represented by a fresh component, which does required testing after binding of its interfaces to component environment.

At first, component controller attaches the new component as a subcomponent test of component testEnvironment via its control interface `teAttachP`. Then, it binds interfaces `tInteract` and `tResult` of the new component to interface `eInteract` of component environment and interface `oResult` of component output, respectively. Finally, component test is activated via interface `startTestP` and executed with a new identifier via interface `executeWithID`. The identifier is also returned by component testEnvironment as a reply of the test script's submission.

Component test performs the test script by interacting with component environment via its interface `eInteract`. When the test script is finished, component test sends the test's results and its identifier to component output via its interface `oResult`. Then, component output notifies component controller via its interface `cDone` and forwards the results and the identifier out of the component testEnvironment via its external interface `asyncReplyET`.

After component controller is notified about the finished test script, it is able to receive and execute another test script, i.e. to attach a new component in the place of component test. Before that, component test with the old script is stopped via interface `stopTestP` and detached via control interface `detachTestP`[8].

5 Conclusion and Future Work

In this paper, we have presented a MOF-based metamodel for modelling of component-based systems with mobile architecture. The proposed metamodel has defined classes for primitive and composite components, the components' functional and control interfaces, specific bindings of the interfaces, and types of their operations. An application of the metamodel has been demonstrated

[8] In the diagram in Figure 5, only these two interfaces of test are connected with controller, because the rest of the test's interfaces are used only during its nesting and their connections do not exist outside of controller component.

on the case study of the component-based system implementing a specific service for a testing environment of a railway interlocking control system.

We are currently working on an integration of the metamodel into modelling tools based on Eclipse Modeling Framework and on developing a graphical editor in Eclipse Graphical Modeling Framework for structural and behavioural modelling of service-oriented architectures and underlying component-based systems.

Acknowledgements This research was partially supported by the BUT FIT grant FIT-10-S-2 and the Research Plan No. MSM 0021630528 "Security-Oriented Research in Information Technology".

References

1. Avgeriou, P., Guelfi, N., Medvidovic, N.: Software architecture description and UML. In: UML Satellite Activities, *Lecture Notes in Computer Science*, vol. 3297, pp. 23–32. Springer (2004)
2. Balasubramaniam, D., Morrison, R., Oquendo, F., Robertson, I., Warboys, B.: Second release of ArchWare ADL. Tech. Rep. D1.7b (and D1.1b), ArchWare Project IST-2001-32360 (2005)
3. Barros, T.: Formal specification and verification of distributed component systems. Ph.D. thesis, Université de Nice – INRIA Sophia Antipolis (2005)
4. Bowen, J.P., Stavridou, V.: Safety-critical systems, formal methods and standards. IEE/BCS Software Engineering Journal **8**(4), 189–209 (1993)
5. Bruneton, E., Coupaye, T., Stefani, J.B.: The Fractal component model. Draft of specification, version 2.0-3, The ObjectWeb Consortium (2004)
6. Bureš, T., Hnětynka, P., Plášil, F.: SOFA 2.0: Balancing advanced features in a hierarchical component model. In: Proceedings of SERA 2006, pp. 40–48. IEEE Computer Society, Seattle, USA (2006)
7. Donini, R., Marrone, S., Mazzocca, N., Orazzo, A., Papa, D., Venticinque, S.: Testing complex safety-critical systems in SOA context. In: CISIS, pp. 87–93. IEEE Computer Society, Los Alamitos, CA, USA (2008)
8. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering **26**(1), 70–93 (2000)
9. Meta object facility (MOF) core specification, version 2.0. Document formal/06-01-01, The Object Management Group (2006)
10. UML infrastructure, version 2.1.2. Document formal/2007-11-04, The Object Management Group (2007)
11. Oquendo, F.: π-ADL: an architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. ACM SIGSOFT Software Engineering Notes **29**, 1–14 (2004)
12. Oquendo, F.: UML 2.0 profile for ArchWare ADL. Tech. Rep. D1.8, ArchWare Project IST-2001-32360 (2005)
13. Rychlý, M.: A case study on behavioural modelling of service-oriented architectures. e-Informatica Software Engineering Journal **4**(1), 71–87 (2010)