UML + ROOM as a Standard ADL?

B. Rumpe, M. Schoenmakers

Dept. of Computer Science IV
Munich University of Technology
80333 Munich, Germany
[rumpe|schoenma]@in.tum.de

A. Radermacher, A. Schürr

Institute of Software Technology
University of the Federal Armed Forces, Munich
85577 Neubiberg, Germany
[ansgar|schuerr]@informatik.unibw-muenchen.de

Abstract

Designing a software system's architecture properly is one of the most important tasks of any software engineering project. Nevertheless there exists no common definition of the term "software architecture" and no standard software architecture description language (ADL). This paper discusses whether the standard OO modeling language UML is a standard ADL, explains some deficiencies if used for this purpose, and makes a proposal of how to eliminate these deficiencies. The proposal is based on the widely accepted idea that elements of the component-based OO modeling language ROOM should be integrated with UML. It explains why the idea of merging static structure diagrams of ROOM with behavior describing collaboration diagrams of UML is not sufficient and presents an additional approach for the integration problem.

Keywords

Architecture Description Language, Unified Modeling Language, Real-Time Object-Oriented Modeling, UML, ROOM, Software Architecture, Software Components

1 Introduction

Software architecture is an important field of study that is becoming more and more influential in the field of software engineering. In the last years piles of papers and books have been published about software architecture description languages, architectural styles, reference architectures, architecture frameworks, and design patterns. Furthermore, the software engineering community agrees on the fact that a software system's architecture plays a key role for planning its development process and for guaranteeing its quality concerning certain functional and nonfunctional requirements such as its "correctness", availability, performance, maintainability, portability, etc. [2]. Nevertheless, there is no common agreement on the definition of the term "software architecture". Some researchers even stress the point that a software system does not have a single architecture, but a set of architectures constructed for rather different purposes and emphasizing rather different properties of the modeled system [2].

As a consequence, most books or papers about this topic start with just another definition of the term software architecture and introduce their own *architecture description language (ADL)*. A rough classification of the ADL literature shows that there are at least two main categories of ADLs:

1. More recently published papers often favor the view that software architectures are "components + connectors + behavioral constraints" [24].
2. Elder papers usually adhere to the terminology of so-called *module interconnection languages (MIL)*, where a software architecture is a set of modules with different kinds of dependency relationships between them [14].

Nevertheless, there is some hope that the invention of new ADLs and the associated discussion "who's ADL is the best one and who's terminology is correct" may be terminated by using the *Unified Modeling Language UML* as a starting point for a standard software architecture description language [20, 3]. UML is a widely accepted OMG standard of an object-oriented modeling language, which combines a rather broad spectrum of visual modeling sublanguages:

- Class diagrams and package diagrams offer all concepts of MILs (information hiding, import relationships, inheritance, genericity, ...).
- The object constraint language OCL [25] allows for the definition of invariants as well as for pre- and postconditions and offers thereby the necessary means for "designing by contract".

[RSR+99] B. Rumpe, M. Schoenmakers, A. Radermacher, A. Schürr
UML + ROOM as a Standard ADL?
In: Engineering of Complex Computer Systems, ICECCS'99 Proceedings
F. Titsworth (eds.)
IEEE Computer Society, 1999.
www.se-rwth.de/publications

- Various types of diagrams (state transition diagrams, collaboration diagrams etc.) may be used to model the dynamic behavior of networks of related objects.
- Finally, component and deployment diagrams may be used to define a mapping of logical software objects onto available hardware components.

Despite it's richness, UML has a number of drawbacks compared with component-based ADLs like those presented in [24]. This is mainly due to the fact that UML's component diagrams are not intended to represent the *logical* decomposition of a software system into reusable and combinable subsystems:

> "*a component is a physical unit of implementation ...*" (cf. page 93 of [20]).

Furthermore, UML does not offer the concept of connectors as first-order objects, which would be a hybrid of an association (association class) and a dependency between a class and an interface of another class.

In contrast to UML another well-known OO modeling language called *ROOM* [22] — mainly used for Real-time Object-Oriented Modeling purposes — combines a variant of component (actor) diagrams with another variant of state transition diagrams. It fulfills thereby the main requirements for an ADL stated in [24], but it offers no equivalent for UML's object constraint language or its package diagrams.

Fortunately, efforts are made to merge the advantages of both approaches. In particular ROOM's actor diagrams are incorporated into UML by mapping them to UML collaboration diagrams [23]. We do not believe that it is sufficient to treat ROOM's actor diagrams as an extension of UML's collaboration diagrams, as actor diagrams deal a lot more with structure and also with *possible* paths of communication than collaboration diagrams do.

Within this paper, we will first sketch how ROOM's actor diagrams look like (cf. Section 2). Afterwards, we will discuss why these actor diagrams should be added to UML as a separate type of diagram. We argue that a mapping of ROOM's actor diagrams to UML class diagrams in addition to the existing mapping gives us a tight integration with UML. (cf. Section 3). Last but not least we will explain in more detail how ROOM actor diagrams may be translated into pure UML class diagrams (cf. Section 4).

Please note that this paper is written based on the assumption that its readers are familiar with the basic elements of UML. Nevertheless, we do hope that the explanations of all presented UML diagrams are detailed enough such that those readers, who are not familiar with UML, understand the paper's main message.

2 An Introduction to ROOM

ROOM as described in [22] mainly offers two types of diagrams. *ROOM actor diagrams* describe the hierarchical decomposition of a software system into its components as well as all possible connections (communication channels) between these components. *ROOM charts* on the other hand are a variant of hierarchical state transition diagrams derived from StateCharts [10]. The translation of any ROOM chart into an equivalent UML state transition diagram is rather straightforward and will not be regarded here.

A ROOM actor diagram defines the internal structure as well as the external interfaces of a single component (class), which is called *actor* in [22] or *capsule* in [23]. The interfaces of different components, so-called *ports*, are bound to each other via binary *connectors*. Therefore, ROOM uses a variant of the "component-port-connector" model, i.e. allows for the definition of component architectures as discussed in [24] or in [7]. The ROOM model consists of

- *Components*: A component or actor can be any element that performs some kind of computation. An actor class declaration describes the internal structure, the external interface, and the behavior of all its instances.
- *Ports*: A set of named ports defines the interface of a class of actors. Ports are the places where actors offer or require certain services. The primary communication form in ROOM is sending of asynchronous signals (messages) through these ports. Ports are bidirectional, allowing signal flow in both ways.
- *Protocols*: A protocol defines the permissible set of signals which are sent forth and back between two connected components. Any port (small black rectangle) associated with a given protocol sends the out-signals of its protocol and receives its in-signals.
- *Conjugated Ports*: Ports can be conjugated with respect to their associated protocols. A conjugated port (small white rectangle) receives the out-signals of its protocol and sends its in-signals.

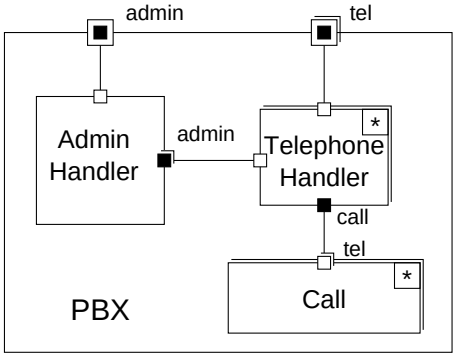

Figure 1: The PBX System (cf. page 428 in [22])

- *Connectors*: A connector establishes a connection between two ports. It binds a port of one component to a conjugated port of another component with a compatible (the same) associated protocol.

Figure 1 shows one example of a ROOM actor diagram taken from [22]. It declares the main actor class `PBX` of a Private Branch eXchange system from the private telephone network area.

Any instance of this class has one `admin` port and an unrestricted number of `tel` ports (the number of these ports varies at runtime). Furthermore, it contains a single instance of an `AdminHandler` actor as well as a dynamically changing number of `TelephoneHandler` and `Call` actor instances (shadowed boxes indicate multiple instances of components or ports).

Each `TelephoneHandler` actor has a connection to a separate `tel` port of the main `PBX` class, the `admin` port of the unique `AdminHandler` actor, and a connection to (at most) one `tel` port of a `Call` actor. The additional constraint that a single `Call` actor instance should be connected with two `TelephonHandler` actors is not modeled here.

Finally note that ROOM makes a clear separation between the *declaration* of an actor class and a *reference* to another actor class. Consider the actor diagram of Figure 1 with its actor class references `AdminHandler`, `TelephoneHandler`, and `Call`. It denotes a composition relationship between the actor class `PBX` on one hand and the actor classes `AdminHandler`, `TelephoneHandler`, and `Call` on the other hand. Furthermore, it offers additional means for information hiding, which are discussed in more details in Section 4.

3 UML + ROOM - The User's View

The integration of ROOM actor diagrams with UML has to be studied on several levels. First of all it is necessary to adapt the notation for actors, ports, and connections such that it respects the notational conventions of UML. Second, all required new language elements have to be integrated with the UML meta model, which defines the syntax of UML as well as part of its static semantics. Third, a precise definition of the extended language's dynamic semantics needs to be defined.

Within this section we will present a new proposal of how to extend the UML notation. This proposal is related to the "component-port-connector" notation of Catalysis presented in [7]. We believe that our proposal offers a clearer representation of ports and conjugated ports and a clarification of the representation of ports in UML.

In the following section we will show more precisely how the various elements of ROOM actor diagrams are related to the UML meta model. For sake of readability we will use a case by case translation of (proto-)typical ROOM actor diagrams into standard UML diagrams for this purpose. The last point mentioned above, the extended UML language's dynamic semantics definition, will not be addressed here, partly due to the fact that even standard UML does still not have a commonly accepted precisely defined semantics for the involved diagram types (cf. [4, 9, 21] for a discussion of this topic).

ROOM actor diagrams cannot be incorporated into UML without making first the decision whether they are merely added as another new diagram type or whether it is possible to regard them as an extension of an already existing diagram type. First of all UML's *component diagrams* seem to be the natural candidate for this purpose, but have to be rejected for the following reasons: the decomposition of a system into its components is a high level logical design process, whereas all publications about UML insist on the fact that component and deployment diagrams should be used for modeling a software system's physical layout.

Another obvious solution, the addition of a *new diagram type*, which plays a similar role as the good old data flow diagrams of OMT [19], is rather infeasible, too; in this case mainly due to political reasons. Almost all people involved in the development of UML agree on the fact that UML offers already too many types of diagrams and should not be extended by adding yet another diagram type. Furthermore, data flow diagrams of OMT were and are considered useless by a large fraction of the OO community. We believe

that the lack of success of data flow diagrams in OMT was mainly caused by the fact that they were not properly integrated with class diagrams. It may, therefore, be worth-while to reconsider this point, e.g. based on the experiences with object-oriented data flow diagrams in the visual programming language Prograph [6].

A third possibility, promoted by some leading developers of UML and ROOM, is presented in [23]: the extension of existing UML diagrams, namely the *collaboration diagrams*. Although, extension of existing diagrams increases the complexity of UML, it seems a politically more acceptable solution than introducing a new kind of diagrams.

Using collaboration diagrams as target for the mapping of ROOM actor diagrams was probably chosen because collaboration diagrams use *class roles* as their basic elements. From a superficial point of view class roles of UML are similar to actor class references of ROOM. They possess an optional (role) name and are associated with a class declaration. Furthermore, a single diagram may contain different class role instances which belong to the same class.

In our opinion class roles and actor class references as well as actor diagrams and collaboration diagrams are different things. An actor diagram presents a *static view* of the internal details of an actor class. More precisely, it defines structural constraints and possible communication paths for all instances of the regarded class valid during their whole lifetime. A collaboration diagram on the other hand deals with the *dynamic behavior* of a set of related class instances. It defines a possible configuration of some class instances at a certain point of time and uses this configuration to explain their runtime behavior through examples. Therefore, a collaboration diagram does not faithfully represent the static aspects of an actor diagram.

Furthermore, actor diagrams deal with the maximal configuration of the structure of a system and with the *possible* paths of communications within that structure, whereas a collaboration shows *one* possible interaction behavior. In particular one collaboration diagram describes a behavioral pattern, whereas in ROOM this is achieved in combination with the ROOM StateCharts.

As a consequence, explaining actor diagrams by defining their relationship to collaboration diagrams does not explain actor diagrams adequately.

However, this situation can be improved by treating ROOM actor diagrams as an extension of UML *class diagrams*. We believe this to be the more natural approach, which respects the static class-level properties

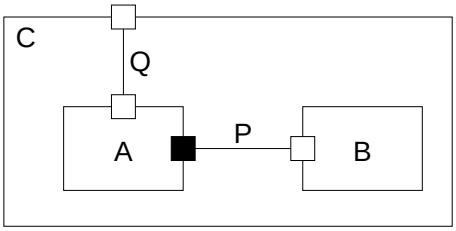

Figure 2: Simple ROOM Actor Diagram

of ROOM actor diagrams that have been neglected in the mapping provided in [23].

The following Section 4 sketches how to translate any ROOM actor diagram into an almost equivalent standard UML class diagram. As we will see, only quite a few properties, namely the binding of ports are not represented adequately. Fortunately, these binding are represented in collaboration diagrams, such that ROOM actor diagrams are fully explained using both mappings. We also demonstrate the alternative – but somewhat less readable – use of OCL constraints to denote bindings.

The purpose of the remainder of this section is to present an extended class diagram notation for "component-port-connector" architectures which is more compact and readable than the translation of ROOM actor diagrams into UML class diagrams presented in the following section. We furthermore start to discuss the mapping into standard UML notation. The proposed extension reflects the notation, the developers should use. The internal view presented in Section 4 serves as semantics definition and is mainly useful for tool implementation purposes.

Let us start with the rather straightforward translation of the actor diagram of Figure 2 into the standard UML class diagram of Figure 3. This translation works exactly if the involved protocols are unidirectional, i.e. if `P` and `Q` require that `C` propagates signals to `A` and that `A` sends signals to `B` but not the other way round. In this case any signal may be regarded as an operation of an interface declaration in UML: `C` implements the interface `Q` (denoted as a small circle — a "lollipop" — attached to the class) and uses the `Q` interface implemented by `A`, whereas `A` needs the `P` interface realized by actor `B`. The component-subcomponent relationship between `C` on one hand and `A` and `B` on the other hand is modeled as a composition association of UML, a straight line decorated with a black diamond.

This straightforward translation scheme breaks down if an actor has several ports of the same type or if a bidirectional protocol is used, where signals are

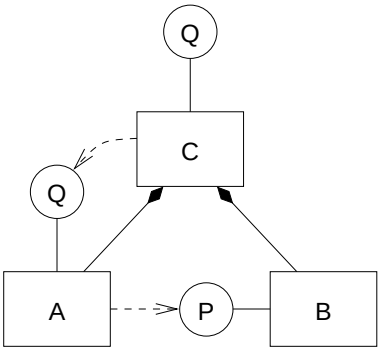

Figure 3: Simple Translation of Figure 2 into a UML Class Diagram

sent in both directions along a single connection line (instead of using two unidirectional connections for this purpose). Furthermore, ROOM already resolved the asymmetry situation resulting from the fact that export interfaces of classes have an explicit representation as small white lollipops attached to their class boxes, whereas there are no equivalent means for the definition of required but not yet resolved imports. Following the philosophy of ROOM we suggest the introduction of *import interfaces* as black lollipops, i.e. as conjugated export interfaces. Pure export interfaces always receive messages (signals) implemented by their classes, whereas pure import interfaces always send messages (signals), which are needed to implement the associated class (cf. Figure 4).

Please note that the newly introduced import interfaces are similar to so-called *component sockets* of Catalysis, whereas the original UML (export) interfaces play the role of *component plugs* [7]. Both import and export interfaces are necessary prerequisites for the definition of reusable components with "loose" ends, i.e. with unresolved import relationships. These components implement a certain set of services based on another set of services offered by a not yet determined number of lower level components. Combined with the usage of UML's object constraint

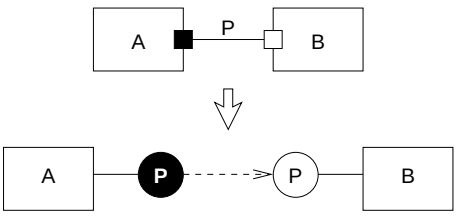

Figure 4: A new Import Interface Notation for UML

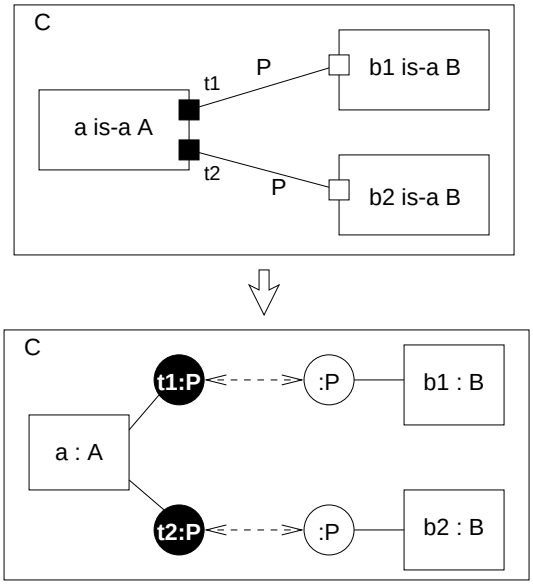

Figure 5: Interface Objects and Bidirectional Dependencies

language, import interfaces offer thereby about the same functionality as the formal import parts in the module concepts of algebraic specification languages [8, 5].

To handle more sophisticated cases, we introduce interface objects that serve as ports and bidirectional dependencies between interfaces to handle bidirectional protocols, and several ports of the same type are attached to a single actor class. In this case, component interfaces have to be modeled as objects in their own right, which are allowed to send and to receive signals (messages). Interestingly mapping ports to objects is exactly what the ROOM implementation does already.

Figure 5 sketches how an UML-like notation of these concepts might look like. It uses nesting of boxes — as suggested in [7] — for defining an aggregation relationship between the enclosing actor class `C` and the enclosed actor class references `a`, `b1`, and `b2`. The class box inscriptions of the form `b1:B` and `b2:B` have to be read as "an instance of class `C` contains one instance of class `B` with name `b1` and another instance of class `B` with name `b2` as components" (in contrast to UML collaboration diagrams, where a box inscription of the form `b1:B` expresses just the fact that an instance of class `B` plays the role `b1` at a certain point of time).

The two ports `t1` and `t2` of the actor class reference `a` have the same conjugated protocol type `P`. They cannot be translated into (conjugated) UML class

interfaces, but have to be modeled as a new kind of instantiable UML classifiers (e.g. as UML classes with the stereotype `<<port>>` or `<<conjugated-port>>` as suggested in [23]). Normal interface objects (ports) are denoted as white ellipses with an optional name followed by a ":", and a protocol type definition as inscription; conjugated interface objects (ports) are denoted as black ellipses. The following section contains a more detailed description how this extension can be mapped to standard UML constructs.

It is a matter of debate whether a dashed line with arrow heads at both ends should be used to model a bidirectional connection (as used in Figure 5) or whether a solid line would be more appropriate. In the first case the similarity between bidirectional connections and dependencies between classes and interfaces is emphasized, in the latter case the similarity between connections and regular associations between classes would be emphasized.

4 UML + ROOM - The Internal View

In this section we examine the technical integration of ROOM structure diagrams with the existing UML diagrams. There are a variety of different techniques which may be used for this purpose. The route we will follow is to provide a mapping of the new notation to the existing UML notations. As known from mathematics – and as discussed in Section 3 – a mapping can be incomplete in the sense that it does not map all aspects of the source notation properly. In this paper, we informally introduce the mapping through discussing how the concepts are mapped. A formal mapping would include the mapping of ROOM's meta model to the UML meta model. We simply provide some examples that illustrate the *result* of the mapping.

Therefore, in this section we map the ROOM structure diagram elements systematically to UML diagrams. Please note that the provided mapping can be used in two ways:

1. the mapping serves to understand the semantic integration of ROOM and UML, and

2. the mapping can be implemented in a tool to actually map the user friendly version of the ROOM structure diagrams to UML diagrams.

4.1 Actor classes and actor references

Actor classes in ROOM have a similar role as classes do in the UML. An actor class defines the structure and behavior for its instances, the actors. The structure within an actor class is defined by composing multiple named actor references and binding their ports as shown in Figure 1. It is therefore obvious to map a ROOM actor class to an UML class. To keep track of its origin and to add special properties, we decorate the generated UML class with a special stereo type `<<capsule>>` as proposed in [23].

- Actor (capsule) classes are mapped to UML classes.

- Actors are mapped to UML objects

The containment relation among actors can directly be mapped to UML aggregation and composition. ROOM defines several kinds of containment. A commonly used containment in ROOM is that of *fixed* actors. These subactors are created when the enclosing actor is created and will be deleted when the actor is deleted. This maps clearly to UML class composition.

When multiple actors of the same class are required, it is allowed to define several actors and ports. But this keeps the number of instances fixed per default and thus the multiplicity of the composition in UML is also fixed. In order to allow varying numbers of instances, ROOM provides *optional* subactors. Optional subactors reside in the scope of a actor and the replication factor `n` of an optional actor reference indicates the maximum number of subcapsules that can exist during runtime. The result is a multiplicity of `0...n`. The `TelephoneHandler` in Figure 1 shows such a situation, where even the number of instances is not limited at all.

Besides containment that maps to composition, ROOM also knows the *import* of actors, a concept which has no relationships to import dependencies of module interconnection languages. Import places of actor diagrams are placeholders (slots), where already existing actors of related actor diagrams may be inserted (and removed) at runtime. As a consequence, actor instances may be (temporarily) shared subcomponents of different diagrams. A subactor instance can be *shared* between multiple actors, when a user defines a special equivalence relation between actor references. The shared subactor is deleted when no parent actor exists any more.

- The containment of fixed and optional subactors is mapped to UML composition.

- The containment of imported actors or shared actors is mapped to UML aggregation.

- Replication factors are mapped to a fixed and maximum multiplicity.

Figure 6 shows the effect of mapping different containment types.

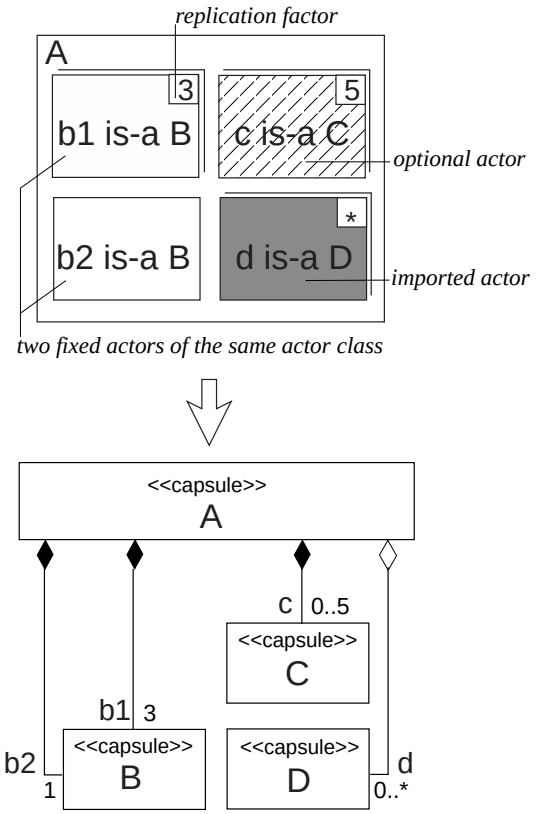

Figure 6: Mapping Different Types of Containment

4.2 Protocols and ports

Actors communicate by sending and receiving messages through *ports* to other actors. An actor may may have multiple external and internal ports, as e.g. `PBX` in Figure 1. Ports are bidirectional, allowing messages to send and to receive. An actor explicitly defines the wiring of the ports through their *bindings*. The valid interactions between two connected ports is defined in a ROOM *protocol*. A protocol defines several (the ROOM implementation actually supports only protocols with exactly two) *roles*. As the two roles are inverse to each other, they can be derived from each other by *conjugation*. Conjugating a port `P`, denoted as `P~`, is essentially to inverse its in/out directions. An actor may contain multiple ports that implement the same protocol.

Ports are similar to interface signatures in UML. When a class implements an interface we know that it understands all messages contained in the interface. But an UML interface contains only incoming messages. We therefore map one role into two interfaces, one being inward directed, one being outward directed. We map all incoming messages to one UML interface, called P_{in}, and all outgoing messages to a second interface, called P_{out}. We use a stereotype `<<role>>` to mark the purpose of these interfaces. Please note that it is possible for different protocols to accidentally share interfaces; in an optimization step sharing can be used to reduce the number of actually existing classes and interfaces.

To summarize, a port that implements a role of the protocol P is mapped to an object

- of a class that implements the P_{in} interface,
- is part of the actor using the port,
- and depends on an P_{out} interface.

Ports realizing the conjugated role can obviously be implemented using the same interfaces the other way round.

ROOM distinguishes between *relay ports* and *end ports*. A relay port connects a subactor to the environment. Messages are delegated from outside to the subactor and vice versa. End ports are linked to the internal state machine of a ROOM actor. We can regard the state machine as a special subactor, thus treating end ports like relay ports to the state machine's subactor.

To ensure proper encapsulation both kinds of ports (relay and end ports) must provide the same interface for an actor's environment. However, if a port is a relay port, then it also implements the interface that belongs to the conjugated protocol. Thus looking either from inside or outside to a port, two different protocols can be seen. If desired, the two ports of the port can be marked with the stereotypes `<<relay>>` or `<<endport>>` to keep track of the origin of the ports.

So far we have discussed the principle of the mapping. However, if actors are deeply nested, then it is very important to make optimizations versus the chain of relay ports involved. Instead of forwarding messages through chains of relay ports, the static connection of ports allows the implementation to shortcircuit directly from the sender to the receiving end port. This and other optimizations apply when cutting code from the diagrams and need therefore not be visible to the developer like compiler optimizations are not visible to the programmer.

Figure 7 shows an example for the mapping of ports.

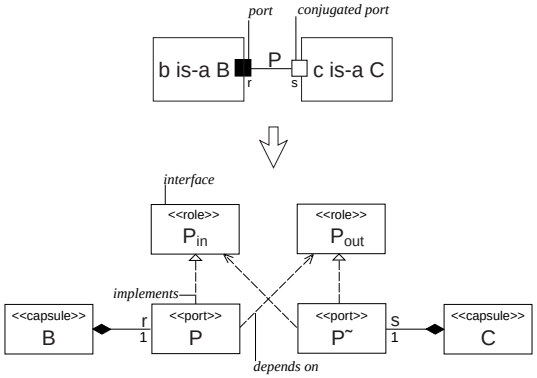

Figure 7: Mapping Ports and Protocols

Figure 8 shows a more complex example, where replicated ports and hierarchically nested actors are mapped.

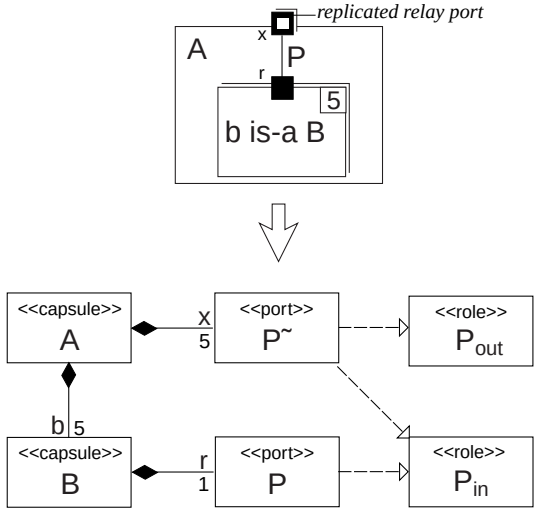

Figure 8: Mapping Replicated Relay Ports

4.3 Binding by connectors

Actor diagrams connect ports of the just declared actor class and its nested actor references. These connections are neither association instances (links) nor association declarations in the sense of UML. There is even a small difference between the links (link roles) of UML collaboration diagrams and the connections of actor diagrams. The actor diagram connections are valid for the distinguished subcomponents of all instances of the regarded actor class; in our view they are an intrinsic part of the static structure definition of the regarded actor class. Collaboration diagram links, on the other hand, usually only have to exist between

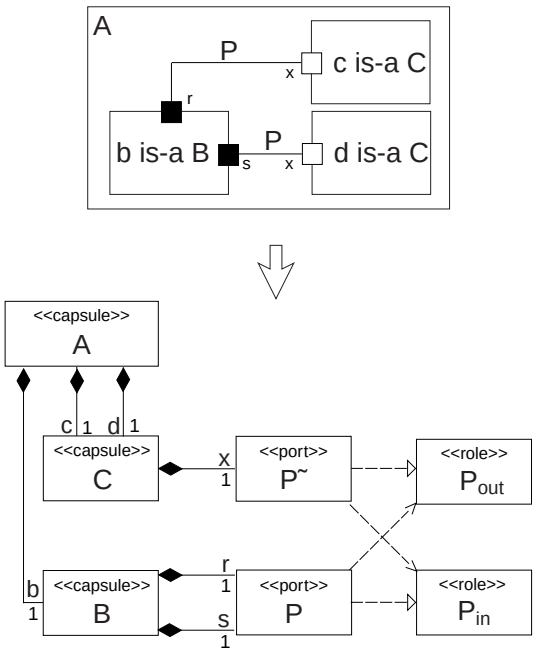

Figure 9: Mapping bindings into UML

a set of regarded class instances if a specified sequence of operations is executed.

Regarding Figure 7 we can represent the connection between both ports as an association between them, thus implementing the dependencies shown in the mapping. However, UML [20] does not clarify the meaning of an association between elements of an aggregation. Are they allowed to build links between aggregation instances or only within an aggregation instance? Several non standard UML solutions have been discussed, but all have their limitations, especially if several instances of the same port are connected to the actor.

Therefore, collaboration diagrams as discussed in [23] or even simple object diagrams are adequate for this purpose. For each actor diagram, the resulting actor class should be attached with an appropriate object or collaboration diagram.

Another feasible way is the use of the object constraint language in addition to associations, as it has a good notion of navigating through paths. OCL helps to ensure encapsulation of associations. In the example in Figure 9, the following OCL constraint prevents ports of subactors to be connected illegally to ports not in the actor, if we assume that ports establish their connection through an association (or a derived query), called `connection`:

```
self:A
   self.c != self.d
   self.b.r.connection = self.c.x
   self.b.s.connection = self.d.x
   self.c.x.connection = self.b.r
   self.d.x.connection = self.b.s
```

The first line demands that actors `c` and `d` are not the same object – something which is ensured in the actor diagram, but not in the UML diagram. The next two lines state that the port `r` from actor `b` is actually connected to port `x` from component `c`, resp. port `s` to component port `d.x`. The last two lines establish the connection from the other direction, and are according to the bidirectionality of associations in UML redundant.

5 Related Work

Wright [1], Darwin [17] and Rapide [12] are well known ADLs: they are designed to enable the construction of a software system out of basic components. Like ROOM these languages distinguish between the services a component (or capsule) *provides* and those it *requires*. All required services have to be satisfied by other components offering this service.

Wright focuses mainly on the treatment of connectors as first class citizens, which for example allows specialization between connectors. The language allows for the connection of ports (being part of component instances) via connector. The interactions possible at a connector can be defined using a formal language (CSP).

The Darwin language is developed by the distributed systems group at the Imperial College in the group of Magee and Kramer.

> "... Complex components are constructed by composing in parallel more elementary components and as a result, the overall structure of a system is described as a hierarchical composition of primitive components which at execution time may be located on distributed computers." [11]

A strength of Darwin is the ability to specify component bindings which result in an infinite number of components. This is made possible by a mechanism that instantiates a component "on demand" in the moment the signal has been sent to it. Darwin owns no abstractions for protocols and ports. It lacks also inheritance between components.

Like UML, Rapide is not a single language, but a collection of (in ideal case) complementary languages. In our context, the interesting languages are type language, describing the interface of components also architecture language which captures bindings between components. There is the possibility to distinguish synchronous and asynchronous interaction points using a component, called functions and actions in a interface definition, respectively. Behavior can be defined by means of optional statements written in an event-pattern language.

The integration of ROOM into UML presented here is not comparable with the mapping of ADLs into UML as done in [18]. In the first case the class diagrams are extended and smoothly integrated with the rest of the UML both notation and meta model level. In the later case certain aspects of the regarded ADL are translated into UML and it is assumed that the user still works with specific ADL tools. The UML then has the task of some kind of common repository.

ROOM as well as the discussed combination with UML possesses all essential features of an ADL as characterized in [13]. Actors or capsules play the role of ADL components with typed ports as interfaces. Each component belongs to a specific class which determines the components interface, its internal structure and it's behavior. For the later purpose, UML state charts combined with a standard programming language offer a partly visual, partly textual notation. These behavior descriptions are executable and possess already a (semi) formal semantic definition. Furthermore the envisaged combination of UML and ROOM provides extensive support for interface subtyping as well as implementation inheritance, a topic which is out of the scope of this paper. The connectors of ROOM actor diagrams are second-order objects without any associated type as the case with most ADLs. As a consequence a ROOM connector inherits all its properties from the ports being attached to and may not be refined.

The extensive tool support available for ROOM and UML makes the proposed combination practically useful and the proposed integration could be accomplished relatively easy by tool builders.

Since the end of April the CASE tool Rose RT has been available. Surprisingly, the implementation of UML-RT is, in our opinion, closer to the proposal made here then to the proposal of the white paper [23].

Figure 11 contains the UML-RT representation of a collaboration diagram of Figure 10 in form of a class diagram. The most interesting difference is the treatment of ports not as first-order elements, but as links to a protocol.

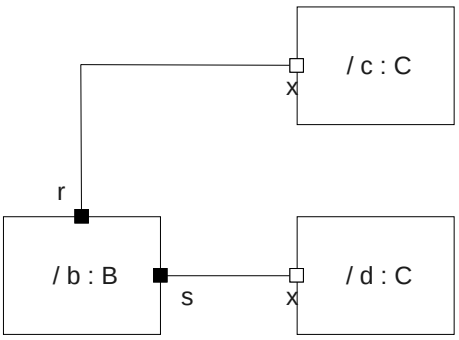

Figure 10: A UML-RT actor diagram

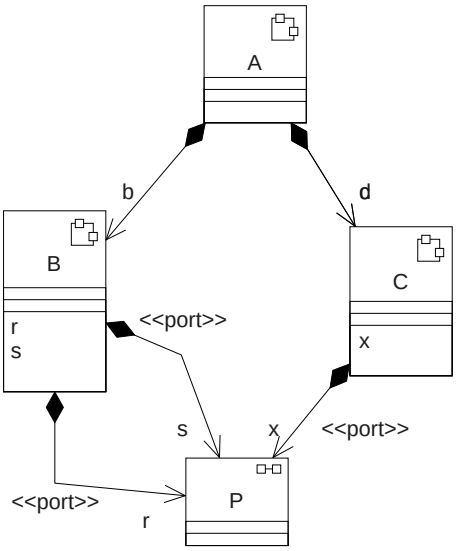

Figure 11: How UML-RT maps capsules to classes

This leads to a smaller class diagram, but removes some of the flexibility available when ports can be passed around. Furthermore, the UML-RT treatment does not distinguish a protocol and its conjugated protocol through different interfaces and therefore looses type information.

On the whole we believe our approach has some advantages: It contains a smoother transition from the interface lollipop notation to the proposed port notation. Also it assures a better integration of actor diagrams with class diagrams and separates clearly between actor diagrams and collaboration diagrams.

6 Conclusion and Future Work

Based on our evaluation of UML (version 1.4) as a software architecture description language, we found that a diagram type needed for the description of component-based architectures is missing. Luckily, proposals have already been made to integrate the actor diagrams of the component description language ROOM into a future version of UML.

The efforts documented in [23] to map ROOM actor diagrams to UML collaboration diagrams have been found insufficient to deal with the static structural properties of actor classes. We, therefore, provide in this paper a complementary mapping to UML class diagrams, which clarifies the role of ROOM actor diagrams as a combination of some characteristics of UML class and UML collaboration diagrams. Moreover, the mapping shows that ROOM actor diagrams are — in our opinion — more closely related to UML class diagrams than to UML collaboration diagrams. As a consequence, we suggested an extension of the UML class diagram notation which allows the component-based description of software architectures. This extension uses the UML lollipop notation for component interfaces (ports) and nesting of boxes (instead of diamond adornments of assocations) for the declaration of composition relationships.

Based on these efforts, refinement techniques to hierarchically decompose and restructure actor diagrams can be adapted from a similar notation in [15, 16] to the UML version of actor class diagrams introduced in this paper.

Another direction of our future work is to adapt this extension of UML with ROOM-concepts to the area of automotive systems. It is expected that in the field of embedded systems, and especially of car control systems, which are not as highly dynamic as business systems tend to be, actor diagrams are of great value and perhaps more important than class diagrams in their current form.

Furthermore, we are planning to introduce a small set of different (stereo-)types for ports and connectors. Examples for these types could be composite ports, event ports, broadcasting ports, buffered channels etc. The goal is to characterize the behavioral properties of the system and its dynamic structure in more detail in its design phase. Finally, the increasing use of spontaneous proximity networking requires new notations to capture dynamic connection structures, physical locations, and mobile actors.

Acknowledgments

The work was partially supported by the Bayerische Forschungsstiftung under the FORSOFT research consortium and the DFG under the Leibnizpreis program.

References

[1] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley, Reading, Mass., 1998.

[3] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, Reading, Mass., 1999.

[4] R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe, and V. Thurner. Towards a Formalization of the Unified Modeling Language. In *Proceedings of ECOOP'97*. Springer Verlag, LNCS, 1997.

[5] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hußmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirement and Design Specification Language SPECTRUM, An Informal Introduction, Version 1.0, Part 1. Technical Report TUM-I9312, Technische Universität München, 1993.

[6] P. T. Cox, F. R. Giles, and T. Pietrzykowski. Prograph. In *Visual Object-Oriented Programming: Concepts and Environments*, pages 45–66. 1995.

[7] D. F. D'Souza and A. C. Wills. *Objects, Components, and Frameworks with UML – The Catalysis Approach*. Addison Wesley, Reading, Mass., 1999.

[8] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications I*. Springer Verlag, Berlin, 1985.

[9] R. France, A. Evans, K. Lano, and B. Rumpe. The UML as a formal modeling notation. *Computer Standards & Interfaces*, 19:325–334, 1998.

[10] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[11] J. Kramer, J. Magee, and A. Finkelstein. A Constructive Approach to the Design of Distributed Systems. In *10th Internat. Conf. on Distributed Computing Systems, Paris*, pages 580–587, June 1990.

[12] D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, Sept. 1995.

[13] N. Medvidovic. A classification and comparison framework for software architecture description languages. Technical Report UCI-ICS-97-02, Department of Information and Computer Science, University of California, Irvine, feb 1996.

[14] M. Nagl. *Softwaretechnik: Methodisches Programmieren im Großen*. Springer-Verlag, 1990.

[15] J. Philipps and B. Rumpe. Refinement of Information Flow Architectures. In M. Hinchey, editor, *ICFEM'97*. IEEE CS Press, 1997.

[16] J. Philipps and B. Rumpe. Stepwise Refinement of Data Flow Architectures. In M. Broy, E. Denert, K. Renzel, and M. Schmidt, editors, *Software Architectures and Design Patterns in Business Applications*. Technische Universität München, TUM-I9746, 1997.

[17] M. Radestock and S. Eisenbach. Formalizing System Structure. In *Int. Workshop on Software Specification and Design*, pages 95–104. IEEE Computer Society Press, 1996.

[18] J. E. Robbins, N. Medvidovic, D. F. Redmiles, , and D. S. Rosenblum. Integrating Architecture Description Languages with a Standard Design Method. In *20th International Conference on Software Engineering(ICSE'98), Japan*, pages 209–218, Apr. 1998.

[19] J. Rumbaugh, M. Blaha, W. P. F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[20] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, Reading, Mass., 1999.

[21] B. Rumpe. A Note on Semantics (with an Emphasis on UML). In *Second ECOOP Workshop on Precise Behavioral Semantics*. Technische Universität München, TUM-I9813, 1998.

[22] B. Selic, G. Gullekson, and P. Ward. *Real-Time Object-Oriented Modeling*. John Wiley, New York, 1994.

[23] B. Selic and J. Rumbaugh. *Using UML for Modeling Complex Real-Time Systems*. ObjecTime Limited, 340 March Rd., Kanata, Ontario, Canada, 1998. http://www.objectime.com/otl/technical/umlrt.html.

[24] M. Shaw and D. Garlan. *Software Architecture - Perspectives on an Emerging Discipline*. Prentice Hall, Upper Saddle River, New Jersey, 1996.

[25] J. Warmer and A. Kleppe. *The Object Constraint Language*. Addison Wesley, Reading, Mass., 1998.