Towards Architectural Modeling with UML Subsystems

Harald Störrle

Ludwig-Maximilians-Universität München
stoerrle@informatik.uni-muenchen.de

Abstract. This paper presents an extension of the UML 1.4 metamodel that facilitates the description of software architectures. To this end, concepts like views, ports, connectors, and protocols are introduced, and the UML's subsystem concept is adapted. The approach is inspired by ROOM and IEEE P1471, using an explicit metamodeling approach for clarity. This paper should be seen as input for the discussion on the UML 2.0

Keywords. UML, Subsystem, Metamodeling, Software architecture, P1471, ROOM, UML/RT

1 Introduction

1.1 Motivation

Today, UML is generally acknowledged to be "*the lingua franca of the software engineering community*" (cf. [18, p. v]). As such, it is quite naturally the first choice when it comes to selecting a language for describing software architectures, for facilitating communication between the various stakeholders is the one of the prime purposes of software architectures. However, there is not much support for architectural modeling in the UML (cf. [9]). There have been several approaches to ameliorating this (see Section 1.3), none of which has gone beyond shallow extensions.

With the discussion on creating UML 2.0 gaining momentum, now is the right moment to consider bolder steps. This paper proposes significant additions and enhancements of the UML in a direct metamodelling approach. This paper is intended to stimulate the development towards UML 2.0 rather than to be directly applicable in practice, though the whole approach originated in a concrete software development effort, and is thoroughly devoted to practical purposes.

1.2 Approach

Background There two fundamentally different understandings of what a software architecture should be. On the one hand, there is the academic approach (Architecture Description Languages like Wright, cf. [2]), according to which

software architectures as coarse structures in a very general and loose sense, encompassing each and every style.

On the other hand, there is a more pragmatic approach (which I also follow) imposing some restrictions on the admissible styles in order to be able to provide better methodological and tool support. Examples of the latter approach are Real-time OO Modelling (ROOM), UML for Realtime (UML/RT) and the IEEE standard on architectural descriptions P1471 (cf. [17], [19, 13], and [11], respectively).

Starting point My approach is based on the following four observations. First of all, a software architecture is one of the first artifacts created in the course of a development project. In any case, it exists long before concrete system parts are created. In order to be able to visualize an architecture, reason about it, communicate with it and treat it in other ways, its building blocks must have an abstract, design-level representation, that is, invariably, an UML representation.

Second, with so many different stakeholders and activities based on it, an architectural design is a very rich description, and thus in need of internal structuring to separate concerns. Intuition, practical experience, and empirical surveys show that this is best achieved using multiple views.

Third, many architectures involve legacies, which are not object-oriented, and these must be dealt with adequately, too. So, there should be no technological bias towards object technology in architectural descriptions.

Fourth, "*software architecture is a framework for change*"[1], that is, one of its fundamental purposes is to foster system evolution and reuse of (its) parts. So, particular care has to be taken to control change impacts (the so called "*ripple effect*"). On the conceptual level, isolating architectural building blocks is best achieved by creating large, self-sufficient units with a strong encapsulation mechanism, that controls not only the stimuli *to* a building block, but also those *from* it. I will introduce the notion of ports for this purpose.

Goals Based on these observations, the approach taken in this paper pursues three goals. First, by being based directly on the UML metamodel, I try to achieve a maximum of expressiveness for architectural designs with a small and coherent attack. This ensures a seamless integration of new concepts, and thus retaining as much as possible of the syntax, semantics, and pragmatics of UML. The present approach tries to follow the *spirit* of UML rather than only the words of the standard. The obvious disadvantage is, that without thorough knowledge of the UML metamodel, this paper is virtually incomprehensible.

Second, there are many different approaches to architectural modeling, each with their own strengths and weaknesses. So, rather than picking a particular one I try to compile the best and most widely accepted concepts and notations from all of them, and adapt them so that they neatly fit into the UML. In particular, the extensions and modifications I shall propose are strongly influenced by ROOM and IEEE P1471.

[1] Attributed to Tom DeMarco.

Third, the kind of architectural building blocks I envision are entities of large granularity, with an identifiable, relevant and non-trivial functionality in the problem domain (i.e. similar to so called "*business objects*"). They are both conceptual and physical entities, so that they are units throughout a complete system lifecycle. In fact, they are true systems themselves. Previous approaches have used UML classes as architectural components which is not really a convincing solution. Instead, I attempt to extend UML subsystems to be usable as architectural building blocks.

1.3 Related work

In [19], some ROOM-concepts are integrated into UML as stereotypes of Class. The approach precedes UML 1.3 and P1471. It lacks a view concept and is strongly influenced by OO-notions.

Another light-weight approach is presented in [6, 14, 1]. There, some concepts of the component-and-connector style are added to UML as stereotypes of Class. This approach is strongly influenced by academic Architecture Description Languages, but fails to implant their rigor into the UML. Neither ROOM nor the P1471 are dealt with. There is no view concept, and little connection to the rest of the UML.

Strohmeier and Kandé [12] improve on this by attempting to integrate both the P1471 and ROOM into the UML metamodel. The use a mix of lightweight and heavyweight extensions, and allow an arbitrary set of viewpoints. They use stereotypes of UML classes as architectural building blocks, and, by using the pattern-notation, seem to suggest that architectural configurations should be UML collaborations (similar to what is proposed in [5]). The main concepts (Architecture, ArchitectureViewpoint, ArchitectureView and so on) are presented, but not connected to the metamodel by generalization-relationships. Thus, their roles in the UML, their semantics, syntax, and pragmatics are poorly defined, and so the integration remains shallow and informal.

Note that all of the above use stereotypes of UML class for architectural building blocks, failing to satisfy the requirements I listed above (see also the detailed discussion in Section 2.2 below). But there are also two approaches that use subsystems.

In [16], the usage of UML subsystems in behavioral designs is discussed. Though this approach does not aim explicitly at architectural modeling, clearly, subsystems are understood as architectural building blocks.

A strictly lightweight extension approach is taken in [10]: a concept of subsystem is introduced as a stereotype of package (the paper is based on UML 1.1). The classical 4+1-view model is adopted (that is, a fixed set of viewpoints), and more than a dozen stereotypes are introduced for threads, layers, and directories supporting it. Great care is taken to establish the connection between the conceptual and the implementation level. It is quite a comprehensive and coherent approach, though e.g. a view concept is missing. A predecessor to P1471, its concepts are not taken into account.

1.4 Notational conventions

In static structure diagrams referring to the UML metamodel, shaded boxes represent metaclasses of standard UML, and white boxes represent concepts introduced here. Concepts referred to, but not defined in the present diagram are shown with dashed outlines. Whenever I refer to a concept from the UML metamodel like Subsystem in the text, I put it in this typeface. Apart from this, I use all the notational conventions proposed in the UML standard document.

2 Elements of architectural modeling in UML

2.1 Overview of concepts

This approach is centered around using UML Subsystems as architectural building blocks, that is, a coherent and strongly encapsulated entity that is self-sufficient with respect to configuration, construction, reuse, and evolution. In contrast to Subsystems as they are, I propose to structure them into "*view*"s, and restrict interactions with them to exclusive points of interaction called "*ports*", which are behaviorally specified by "*protocol roles*". Subsystems may then be glued together by "*connectors*" at their ports, and the expected behavior of connectors may be specified by a "*protocol*". The overall static structure is captured by a "*configuration*", while the dynamic structure is described by a "*configuration space*". Those concepts, that exist not only at design-time have an instance-level analog.

In the following sections, each concept is presented as a new metaclass, with its *embedding* into the UML metamodel, its additional syntax (if any), and its *constraints* expressed in OCL. A few more metaclasses are added so as to facilitate the UML embedding. Following the style of the UML standard, my model is presented in a set of overlapping static structure diagrams. Also, since the model is quite closely knit, there are some forward references.

2.2 Subsystems as architectural building blocks

The following alternatives for architectural building blocks are conceivable within the conceptual framework of the UML: Class, Component, Package, and Subsystem (cf. Section 1.3 above). Consider these in turn.

Class represents classes of OO-programming languages, that is, Class has an unwanted technological bias and represents entities of too small granularity. In UML, a Component is just a piece of code. A Component has no design aspect, so it exists on the wrong level of abstraction. A Package may contain any kind ModelElement, but a package has no internal structure.

Finally, there is the concept of Subsystem, which has been introduced as a subclass of Package in UML 1.3.[2] A Subsystem has three partitions for Operations, and "specification" and "implementation elements", respectively, but no concept

[2] In the remainder, we will refer to [15], i.e. UML 1.4 rather than 1.3.

of view, or port. There are no instances of Subsystems. They may not appear in deployment diagrams, and they may be mixed with objects, classes, and packages in static structure diagrams. Finally, the semantics of Subsystems is not very well defined currently (a meagre twelve pages in the standard).

So, I propose to modify Subsystem as follows.[3] First, the partitions are replaced by set of Views. Being a Namespace, a Subsystem may own arbitrary ModelElements, and in order to eliminate hidden dependencies, no ModelElements may be imported. The package constraint 1 on p. 2-175 has to be extended to also include Ports and the other concepts introduced above. See Figure 1.

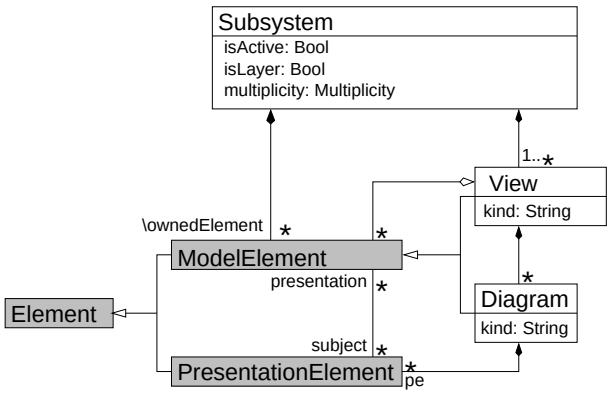

Fig. 1. Embedding of Subsystem, View and Diagram.

The last four constraints on Subsystem (cf. [15, p. 2-202]) become obsolete. As new attributes and relationships I propose to introduce the following.

isActive: Boolean Attribute with approximately the same meaning as the active class-stereotype.

isLayer: Boolean Attribute that indicates whether the respective Subsystem is a layer in a layered system.

view: View **set** The set of all Views specified for this Subsystem. A View may refer to any of the ModelElements that a Subsystem owns or imports by being a Package.

multiplicity: MultiplicityKind similar to the multiplicity of, say, ClassifierRole.

The following new constraints apply to Subsystem.

1. Views are unique, i.e., there is always at most one View of a kind.

[3] Instead of replacing Subsystems with a modified version the way I have proposed it, one may also supplement it, e.g. under the name of Unit or Capsule, as a neighbor to Subsystem.

```
context  Subsystem
inv:  self.view->forAll(v1,v2 |v1.kind=v2.kind implies v1=v2)
```

2. A Subsystem may not own public ModelElements other than Ports. All Ports must be public.

```
context  Subsystem
inv:  self.ownedElement ->
         forAll(visibilityKind="public" implies oclIsTypeOf(Port))
inv:  self.ownedElement->
         forAll(oclIsTypeOf(Port) implies visibilityKind="public")
```

3. A Subsystem must be a self contained Namespace, i.e. as a Package it must carry the top-level-stereotype.

```
context  Subsystem
inv:  self.stereotype->exists(topLevel)
```

4. In order to circumvent hidden dependencies, a Subsystem may not import ModelElements, but has to own them exclusively.

```
context  Subsystem
inv:  self.allImportedElements->isEmpty()
```

As syntax for Subsystems, I propose to use the SARA-notation (cf. [7], also used in ROOM), see Figure 2.

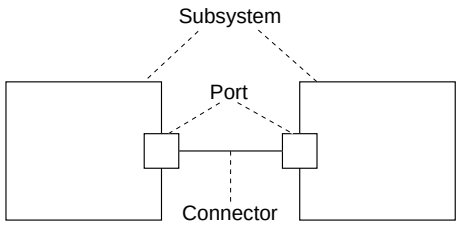

Fig. 2. New syntax for architectural modeling concepts. The folk notation for layered architectures might also be introduced, as syntactic sugar.

2.3 Viewpoint, View and Diagram to structure Subsystem

To improve handling of the design of a large Subsystem, it may be looked at from a multitude of angles, each focusing on some special aspect of the system, i.e.

its behavior, functionality, structure, performance, and so on. So, a Subsystem should be equipped with or represented by a set of views. In the UML, however, the term view is used loosely, sometimes as a synonym for (sets of) diagrams, but there are no such concepts in the UML metamodel.

The P1471 on the other hand proposes to have both viewpoints (generic "kinds" of views) and views (concrete "instances" thereof) as metaclasses—but what kind of "instance-relationship is this? Looking at the UML, there are at least two such notions: the kind of relationship between Classifiers and Instances, and the relationship between, say, metaclasses and meta-metaclasses. Now, I suggest that the instance-relationship between view and viewpoint is of the latter kind rather than of the former. So, when introducing View as a metaclass in the UML metamodel, the viewpoint-concept should correspond to a UML profile or preface. In other words, viewpoint should not be a metaclass as in the P1471.

Thus I suggest that there are two new metaclasses View and Diagram as direct children of ModelElement so as to stress the importance of these concepts. A Subsystem owns Views, which own Diagrams, which own PresentationElements. A View refers to (some of) the ModelElements owned by its Subsystem. Note that different diagrams may share the same ModelElements, but not the same PresentationElements.

Since the model shall accommodate arbitrary viewpoints, View needs a string-valued attribute kind. See Figure 1 for a synopsis of the embedding of View and Diagram. View has the following attributes and associations.

diagram: Diagram **set** The Diagrams that belong to a View. Diagrams are used to structure the PresentationElements referred to by a View. Indirectly, they also express a structure of the ModelElements represented by the PresentationElements.

me: ModelElement **set** The ModelElements belonging to a View. Since several Views may refer to the same ModelElement (e.g. Ports), a weak aggregation must be used.

kind: String The pragmatic function of a view. This attribute should not be an enumeration type (like role of Subsystem), as our approach does not in any way restrict the number or function of viewpoints, and so this attribute may take on arbitrary values.

Determining just which viewpoints are relevant for some particular project is an essential step during requirements analysis. Though I do not want to prescribe the number and kind of viewpoints one may have, there are some that are likely to occur almost always, and I shall mention them explicitly.

interfaces An interfaces view is the simplest abstraction of a system: it just presents Subsystem as a black box with Ports.

functionality A functionality view describes the functionality of a Subsystem. It can be described simply by a UseCase, and possibly a Collaboration between the Ports and the UseCases with a number of Interactions. Since a Port is an Actor, the scenarios of use cases can be

described quite precisely by the ProtocolRole of the Port. When refining the Subsystem, these specifications remain, and ensure a smooth transition between development phases.

behavior The behavior view of a Subsystem can be described by a StateMachine. Its triggers and effects may refer to the Signals and Operations sent or received by the Ports.

structure The static structure of a Subsystem may be defined by a Configuration and some Connectors. The dynamic structure may be represented as a ConfigurationSpace, that is, a set of Interactions on the Subsystems defined the Configuration—the latter are the representedClassifiers of the Collaboration.

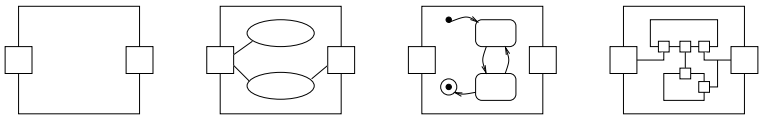

Fig. 3. Syntactic sugar for various common views (left to right): abstract examples for interfaces, functionality, behavior, and structure views.

Additional syntactical sugar may be supplied like in Figure 3. The following new constraints apply.

1. Every PresentationElement of (some Diagram of) a View must the presentation of some subject.

```
context View
inv: self.me.presentation->includesAll(self.diagram.pe)
```

2. The interface view contains only Ports.

```
context View
inv: self.kind="interface" implies self.me.oclIsTypeOf(Port)
```

3. The behavior view contains only StateMachines and Collaborations. There may only be one such StateMachine.

```
context View
inv: self.kind="behavior" implies self.me.forAll->
        (v |v.oclIsKindOf(StateMachine) or v.oclIsTypeOf(Collaboration))
   and  self.kind="behavior" ->count (oclIsTypeOf(StateMachine)) ≤ 1
```

4. The structure view contains a single Configuration and a set of Connectors that establish the binding between the Configuration and the Ports of the Subsystem. It may also contain a ConfigurationSpace.

```
context  View
inv:  self.kind="structure" implies
          self.me.oclIsTypeOf(Configuration)
          or self.me.oclIsTypeOf(Connector)
          or self.me.oclIsTypeOf(ConfigurationSpace)
   and    self.kind="structure"->count (oclIsTypeOf(Configuration))=1
   and    self.kind="structure"->count (oclIsTypeOf(ConfigurationSpace))= 1
```

2.4 Ports and ProtocolRoles to encapsulate Subsystems

In order to turn Subsystems into truly self-contained architectural units that are immune to change impact, a Subsystem must control the dependencies *on* it as well as those *of* it, that is, incoming as well as outgoing information. In UML 1.4, Subsystems have only Interfaces with Operations: this restricts only incoming information, and it exhibits a bias to OO technology. Note also, that Operations are synchronous, whereas many architectures are distributed, using an asynchronous calling mechanism (i.e. like Signal). So, I propose to define a new metaclass Port that is behaviorally specified by ProtocolRole. A ProtocolRole is specified by a StateMachine and two sets of incoming and outgoing Signals or Operations.

Again, there is the question of how and where to introduce these concepts within the UML metamodel. Obviously, a Port would be a Namespace, but not typically a Package. More precisely, since a Port specifies a system boundary, it should be as an actor to a UseCase. The simplest way to achieve this is to introduce Port as a subclass of Actor. Also, since Subsystems are to appear in deployment diagrams, Ports have to be there too, and so, Port must also be a Component. ProtocolRoles, on the other hand, have no relative in UML, so I propose to introduce them as a direct child of ModelElement. See Figure 4 for the embedding.

Port has the following attributes and associations.

role: ProtocolRole Analog to role of ConnectorEnd.
constraints: Constraint **set** A set of constraints on the role, i.e. on the (sets of) incoming and/or outgoing Signals.
characteristics: String Additional characteristics may be provided concerning e.g. performance aspects.

There are no new constraints. There may be several kinds of Ports, and different ways of implementing them, that have to be omitted due to space restrictions. See [20] for more details. ProtocolRole has two sets of Operations and Signals by the names in and out, and an facultative StateMachine.

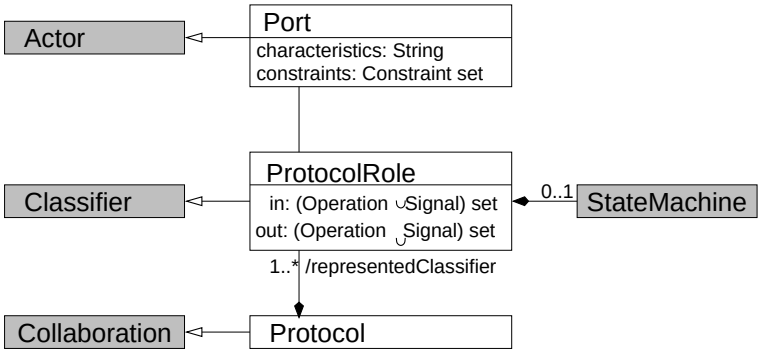

Fig. 4. Embedding of Port and ProtocolRole.

2.5 Connector, ConnectorEnd and Protocol as architectural glue

Subsystems are glued together by Connectors at their Ports. So, Connector is a kind of of Association, and, following the UML, thus requires the auxiliary construct of ConnectorEnd. Every ConnectorEnd is then attached to exactly one Port. Connector is an abstract class with two subclasses, SimpleConnectors (for ideal synchronous channels) and ComplexConnectors (for asynchronous and/or faulty channels). A Connector may be specified by a Protocol, which is basically a Collaboration of ProtocolRoles, see Figure 5. ConnectorEnd has the following new attributes.

source: Bool Indicates whether there are Signals sent or Operations called from this ConnectorEnd.
sink: Bool Converse of source.

There are the following constraints.

1. The representedClassifiers of a Protocol must be ProtocolRoles.

```
context Protocol
inv: self.representedClassifier->forAll(oclIsTypeOf(ProtocolRole))
```

2. Connectors may have only ConnectorEnds as their connection.

```
context Connector
inv: self.connection->forAll(oclIsTypeOf(ConnectorEnd))
```

Note that several ConnectorEnds may be attached to a single Port. Note also, that both Ports and ConnectorEnds play a ProtocolRole, so that the attachment may be checked for compatibility. Thus, a Port is a kind of formal contract between the Subsystems connected by that Connector.

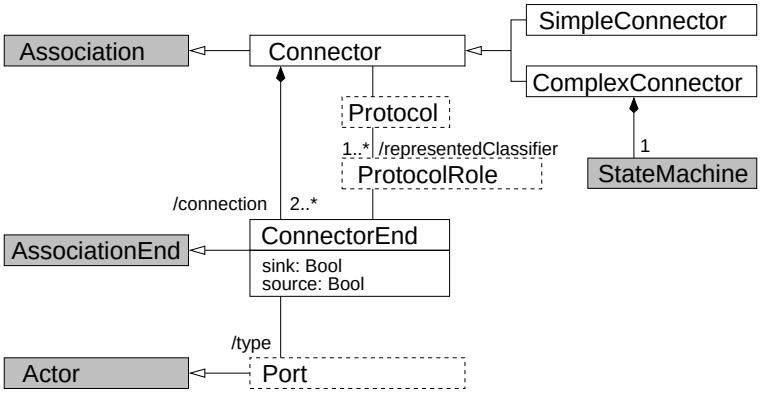

Fig. 5. Embedding of Connector, Port, ProtocolRole and Protocol.

2.6 Configuration and ConfigurationSpace as architectural structures

To wrap up, we need a further concept to represent the static structure of an architecture, that is, an ensemble of Subsystems with their Ports and Connectors between these. For this purpose, I introduce the metaclass Configuration as a child of ModelElement, see Figure 6. Configuration has the following attributes and associations.

component: Subsystem **set** The Subsystems that form the Configuration.
connector: Connector **set** The Connectors among the Subsystems of the Configuration.
binding: SimpleConnector The binding connects (some of) the Ports of the Subsystem with Ports of a sub-Configuration.

1. The ProtocolRoles of Ports connected by a Connector have non-empty incoming (outgoing) sets of Signals, if the respective ConnectorEnd is a source (sink).

```
context Configuration
inv: let con: ConnectorEnd=self.connector.connection
     in con.source implies con.port.role.in ->notEmpty
     and con.sink implies con.port.role.out->notEmpty
```

Connectors are always part of a Configuration. The Ports of Subsystems of a sub-Configuration belong to a different Namespace. Thus, Connectors can not tunnel Subsystem boundaries.

Many systems also have a dynamic structure, that is, their structure evolves over time. So, a Configuration is really only a snapshot of a systems structure. The collection of snapshots and their relationships are represented by a new metaclass called ConfigurationSpace. Intuitively, a ConfigurationSpace can easily be seen as

a role model, since both Subsystem and Port are Classifiers, ClassifierRole can be used to represent them in Interactions.[4]

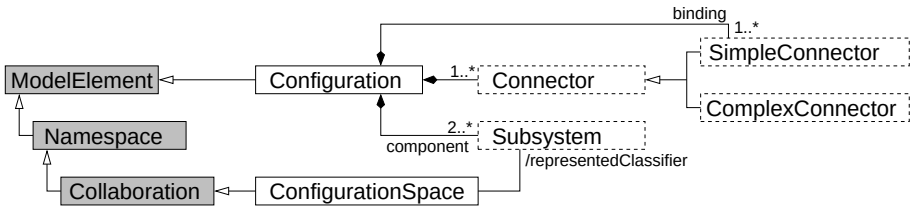

Fig. 6. Embedding of Configuration.

1. The representedClassifiers of a ConfigurationSpace may be only Ports or Subsystems.

   ```
   context ConfigurationSpace
   inv: self.representedClassifier.oclIsTypeOf(Port)
      or self.representedClassifier.oclIsTypeOf(Subsystem)
   ```

2. A ConfigurationSpace may own only ProtocolRoles, ConnectorRoles, and ConnectorEndRoles.

   ```
   context ConfigurationSpace
   inv: self.ownedElement.oclIsTypeOf(ProtocolRole)
      or self.ownedElement.oclIsTypeOf(ConnectorRole)
      or self.ownedElement.oclIsTypeOf(ConnectorEndRole)
   ```

3. The type of a ClassifierRole in a ConfigurationSpace must be a ConnectorEndRole.

   ```
   context ConfigurationSpace
   inv: self.representedClassifier.base.type.oclIsTypeOf(ConnectorEndRole)
   ```

2.7 Type and instance levels

In UML 1.3, Subsystem already had a boolean attribute isInstantiable, but there was no metaclass SubsystemInstances—instances of Subsystems existed only implicitly as the composition of the Instances of the ModelElements contained in

[4] Note that this embedding implies the existing of the further auxiliary metaclasses ConnectorRole and ConnectorEndRole. These, however, are omitted here for clarity. The avid reader is, once again, referred to [20].

the Subsystem. In UML 1.4, this has been fixed by adding SubsystemInstance as a child of Instance. In my approach, there are also instances of Configuration, Connector, ConnectorEnd, and Port with the obvious names.

However, there may not be instances of View, ConfigurationSpace, Protocol and ProtocolRole: these are only specification elements and are not reified directly, but only as a consequence of the code that ultimately implements all these concepts. The details have to be omitted here due to space restrictions; confer [20] for a complete account.

3 Pragmatics

In this section, I want to briefly discuss a pragmatic issue related to my approach, the software process. When architectural units are indeed self sufficient, coarse entities, as I have assumed as my starting point (cf. Section 1.1), the systems resulting from the composition of Subsystems have a recursive, or fractal, structure. Using a traditional waterfall or iterative process, it becomes difficult to establish a mapping between the *product* structure and the *process* structure. Using a language of process patterns instead of a monolithic process solves this problem [5, 3, 4, 20, 21].

4 Conclusions

I have proposed an extension to the UML 1.4 to support modeling of software architectures, building on ROOM, P1471 and, to a lesser degree, ADLs like Wright. Among the concepts added are those known from the Component-and-Connector architectural style, where UML subsystems take the place of architectural components. Other common architectural styles (e.g. layered) are also catered for in my approach. The extensions are presented in a direct metamodeling approach. The resulting metamodel is P1471-compliant.

I have pursued three goals (cf. Section 1.2): direct metamodeling for smooth integration, compile best practices, and reshape Subsystem as architectural units. The first goal has obviously been achieved. Concerning the second point, since this approach builds mainly on the concepts of ROOM and P1471, I think it is reasonable to consider these approaches as mainstream, and so, if they deal with the right notions, then so does my approach. The third issue will have to be resolved by more practical work with it: the experiences collected so far have not shown any major defects, or fatal restrictions or omissions. See [20] for further concepts like architectural style and product line architecture, implementation-relationships, view integration and so on. There, one may also find a worked-out example. Compared to previous work, the present approach is much more comprehensive, explicit, and tightly knit. It is also really integrated into the UML.

I now want to counter a number of possible objections. However, the approach has been developed in a CASE-tool development effort, that is, quite a practical task. And with its direct metamodeling approach, the whole point of this paper

is to keep all changes "under the hood", that is, only its effects are seen on the outside, not the technicalities. The usual diagrams remain usable without changes (recall that Subsystems may occur in static structure diagrams).

Second, one might object that I propose real changes, so that this approach could not be packaged up as a profile. Most of the extensions, however, are conservative, and could be reformulated using stereotyping, so that they could be packaged up as a UML profile. The one exception is the new notion of Subsystem I propose: either, the metamodel is actually modified, or another notion (e.g. by the name "Unit") has to be added as a neighbor of Subsystem.

The OMG is currently soliciting input for the discussion on UML 2.0. One of the shortcomings of the UML 1.4 is the lack of adequate provision for modeling software architectures, and I hope and expect that this will be addressed quite profoundly. I hope that my paper will contribute to this discussion.

Acknowledgments Thanks go to Alexander Knapp and Florian Hacklinger.

References

1. Marwan Abi-Antoun and Nenad Medvidovic. Enabling the refinement of a software architecture into a design. In France and Rumpe [8], pages 17–31.
2. Robert J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, May 1997. Issued as CMU Technical Report CMU-CS-97-144.
3. Scott W. Ambler. *More Process Patterns: Delivering Large-Scale Systems Using Object Technology*. Cambridge University Press, 1998.
4. Scott W. Ambler. *Process Patterns: Building Large-Scale Systems Using Object Technology*. Cambridge University Press, 1998.
5. Desmond Francis D'Souza and Alan Cameron Wills. *Objects, Components and Frameworks with UML. The Catalysis Approach*. Addison-Wesley, 1999.
6. Alexander Egyed and Nenad Medvidovic. Extending Architectural Representation in UML with View Integration. In France and Rumpe [8], pages 2–16.
7. Gerald Estrin, Robert S. Fenchel, and Mary K. Razouk, Rami R. Vernon. SARA (System Architect's Apprentice): Modelling, Analysis and Simulation Support for Design of Concurrent Systems. *IEEE Transactions on Software Engineering*, 12(2):293–311, 1986.
8. Robert France and Bernhard Rumpe, editors. *Proc. 2nd Intl. Conf. on the Unified Modeling Language (≪UML≫ 1999). Beyond the Standard*. Number 1723 in LNCS. Springer Verlag, 1999.
9. David Garlan and Andrew J. Kompanek. Reconciling the Needs of Architectural Description with Object-Modeling Notations. In Selic et al. [18], pages 498–512.
10. Christine Hofmeister, Robert L. Nord, and Dilip Soni. Describing Software Architecture in UML. In Patrick Donohoe, editor, *Software Architecture. TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, pages 145–159. Kluwer Academic Publishers, 1999.
11. The IEEE Architecture Working Group. Draft Recommended Practice for Architectural Description, IEEE P1471/D5.1. Technical report, IEEE, October 1999. Available at `www.pitecanthropus.com/~awg`.

12. Mohamed Mancona Kandé and Alfred Strohmeier. Towards a UML Profile for Software Architecture Descriptions. In Selic et al. [18], pages 513–527.
13. Andrew Lyons. UML for Real-Time (Overview). Technical report, ObjecTime Inc., see `www.objectime.com`, 1998.
14. Nenad Medvidovic. *Architecture-Based Specification-Time Software-Evolution*. PhD thesis, University of California, Irvine, 1999.
15. OMG Unified Modeling Language Specification (draft, version 1.4). Technical report, Object Management Group, February 2001. Available at `http://cgi.omg.org/cgi-bin/doc?ad/01-02-14`.
16. Gunnar Övergaard and Karin Palmkvist. Interacting Subsystems in UML. In Selic et al. [18], pages 359–368.
17. Bran Selic, Garth Gullekson, and Paul T. Ward. *Real Time Object Oriented Modeling*. John Wiley & Sons, 1994.
18. Bran Selic, Stuart Kent, and Andy Evans, editors. *Proc. 3rd Intl. Conf. ≪UML≫ 2000—Advancing the Standard*, number 1939 in LNCS. Springer Verlag, October 2000.
19. Bran Selic and James Rumbaugh. Using UML for Modeling Complex Real-Time Systems. Technical report, Rational Software Corp. & ObjecTime Ltd., 1998.
20. Harald Störrle. *Models of Software Architecture. Design and Analysis with UML and Petri-nets*. PhD thesis, LMU München, Institut für Informatik, December 2000. In print, ISBN 3-8311-1330-0.
21. Harald Störrle. Describing Process Patterns with UML. In Giovanni A. Cignioni, editor, *Proc. Eur. Ws. Software Process Technology*, number N.N. in LNCS. Springer Verlag, 2001. accepted for publication.