Implementing the eODL graphical representation

Joachim Fischer[1], Andreas Prinz[2], Markus Scheidgen[1], and Merete S. Tveit[2]

[1] Department of Computer Science, Humboldt Universitt zu Berlin
Unter den Linden 6, 10099 Berlin, Germany
{fischer,scheidgen}@informatik.hu-berlin.de
[2] Faculty of Engineering, Agder University College
Grooseveien 36, N-4876 Grimstad, Norway
{andreas.prinz,merete.s.tveit}@hia.no

Abstract. eODL is the ITU component description language. Its current status is that it is defined textually and there are several transformation into other languages. There are also ideas about a graphical representation for eODL. In this article we present a graphical representation for some of the eODL language elements and discuss how such a graphical representation can be implemented using a high-level formal description language in comparison with a UML profile.

1 Introduction

The advantages of graphically described models of structure and behaviour opposed to textual representations are undisputed in many application domains. Nevertheless graphical modelling languages will only gain broad user acceptance if appropriate tools become available. Besides presenting the models, these tools should allow easy processing and transformation of the models. Because of the significant expenditure for the development of such editors, the search for efficient production methods is relevant in practice. Starting from a meta-model-based definition of a modelling language, support of the construction of such editors appears to be possible even if the concrete syntax form varies.

This contribution presents two possibilities for the construction of graphical editors for a special meta-model-based language, whose graphic syntax is not specified yet and must therefore be specified first. We consider the ITU-T language eODL [?], whose standard specifies a meta-model together with a textual syntax. The starting point of the graphical syntax proposed here is a set of not-standardized graphic symbols for eODL model elements, which were introduced informally in an eODL-tutorial [?]. Further suggestions have been taken from SDL [?] and UML [?,?].

The two mentioned possibilities of editor construction are mainly suitable for languages based on MOF [?] or similar meta-models (see figure ??). eODL is defined by such a meta-model. The first approach (section ??) uses an XMF case tool, for which the existing MOF based meta-model has initially to be

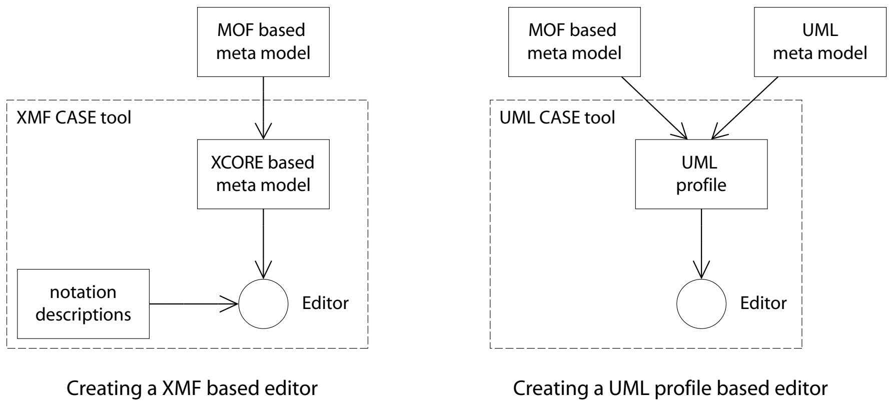


Fig. 1. Approaches of Graphical Editor Constructions

transformed into an XCORE based model. This is not a difficult task because the source and target models are almost identical. The concrete notation can be chosen freely with this approach. The procedure for doing so will be demonstrated for eODL based on some sample model elements.

The second approach (section **??**) uses an UML CASE tool with UML profile support as the editor. By definition of a UML profile for eODL, the UML CASE tool gets restricted to the syntax of a reduced UML. This approach would be general, if the case tool had permitted the definition of specific icons. Since available UML tools do not (yet) offer such a functionality, the syntax remains restricted to the utilization of build-in UML stereotypes.

Both approaches will be explained using the well known dining philosophers example. An eODL model for the dining philosophers example in a concrete textual syntax can found in the eODL tutorial [**?**].

2 The eODL langauge

2.1 Introduction

The language eODL has its origin in the TINA-C work [**?**], where a description for supporting the management of distributed objects in their whole lifecycle was required. To do so, concepts and interfaces were proposed whose operations have to be provided on each node of the respective distributed computing platform. These object lifecycle operations which have to be offered to local and remote applications are essential for compliance and interoperability.

By the standardisation work of ITU-T the TINA concepts were expanded to support the lifecycle of software components from the perspectives of four different but related views: the computational, implementation, deployment, and target environment view. Each view is connected with a specific modelling goal expressed by dedicated abstraction concepts. Computational object types with (operational, stream, signal) interfaces and ports (taken from TINA and ODP [**?**]) are the main computational view concepts used to model distributed software components abstractly in terms of their potential interfaces. Artefacts as abstractions of concrete programming language contexts and their relations to interfaces form the implementation view. The deployment view describes software entities (software components) in binary representation and the computational entities realized by them. The target environment view provides modelling concepts of a physical network onto which the deployment of the software! components shall be made. The important advantage of eODL is the technology independence of the component description from the final used component platform. Meanwhile there are public domain tools availabale for mapping eODL models (which are independend of the final used component platforms) to their corresponding technology units of CCM [**?**] and netCCM ([**?**], [**?**]).

2.2 Meta-model

It is common to say that a language have three types of features: an abstract syntax, a concrete syntax and semantics.

– The **abstract syntax** of a language describes the concepts in the language and how they are related to each other.
– The **concrete syntax** specify how the concepts in a language is actually represented. There are two different types of concrete syntax: textual and graphical.
– The **semantics** of a language says something about the meaning of the concepts in the language.

This section will give an overview of the abstraxt syntax of eODL, while section **??** and **??** will present two different ways representing the language graphically.

In the ITU-T Recommendation Z.130, which specifies eODL, the definition of abstract syntax is based on a metamodel, rather than a more traditional abstract syntax approach. The Recommendation says that "One advantage of the metamodel approach is to allow use of MOF related tools to support the automation of model transitions between the different software development phases. Another benefit is the ability to instantiate concrete models from the metamodel, which can be represented by existing languages, so an integration of different design approaches can be achieved."

The language eODL have four views, computational view, implementation view, deployment view and target environment view, and the metamodel is structured regarding to this. The computational view contains the structual aspects of functional decomposition of a system like computational objects (COs), interfaces and interaction elements. It also handles the aspects of configuration of software components, e.g. ports. The implementation view contains the aspects of the implementation fo software components. This include artifacts and implementation elements. The deployment view handles the aspects of manufacturing

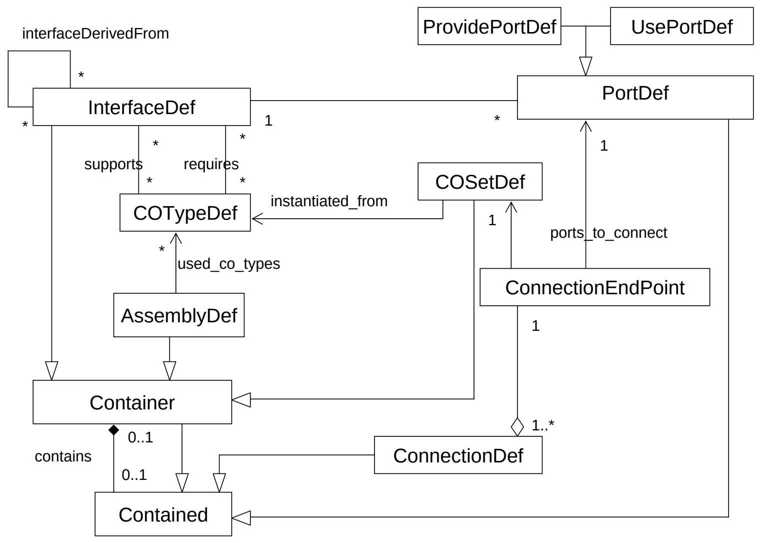


Fig. 2. The eODL metamodel

and integration of software components, e.g. assembly and initial configuration. At last we have the target environment view which focuses on target enviroment, nodes and node links. In order to introduce data types, operations, attributes, exceptions and interface types as modelling concepts in eODL, the eODL metamodel is also based on the metamodel of CORBA-IDL.

Figure **??** shows a small extraction from the eODL metamodel. Since the eODL metamodel in its whole is quite large, we will only present the concepts that will be used in the examples later in the article.

The concept of *COType* is used to specify the functional decomposition of a system. Instances of a *COType* are autonomous interacting entities, which encapsulate state and behaviour. COs interact with their environment via interfaces which are specified using the *InterfaceDef*. A *COType* may support or require an *InterfaceDef*. To support an interface type means that the COs of that *COType* provide interfaces of that interface type. To require an interface type means that COs of that *COType* use interfaces of that interface type. A CO Type is an instance of the class *COTypeDef* in the metamodel. The labels *supports* and *requires* identify the association between the *COTypeDef* and *InterfaceDef*. An *InterfaceDef* has an association to the class *PortDef*. A port is a named interaction port, where either a reference of a supported interface of a CO can be obtained or a reference of an used interface can be registered at runtime. The concepts *ProvidePortDef* and *UsePortDef* are used to model ports of a *COType* which are either used by the environment to obtain the a reference to an interface (provide port) or to store a reference to an interface based on name (use port). The class *PortDef* inherits from the class *Contained*, meaning that a *COTypeDef* instance may contain provided and used port definitions. A provided and used port definition are always associated to an interface definition. The classes *Contained*, *Container* and *InterfaceDef* are all from the IDL metamodel.

The concept of assembly is used to model software systems by sepcifying the CO Types which are involved in the system and to model the initial configuration of the system. The initial configuration is the configuration which is established at the start of the execution time of the software system and consists of initial COs and their initial connections. In the metamodel the assembly is represented by the class *AssemblyDef*. The CO Types are associated by the introduction of an association between the metaclasses *AssemblyDef* and *COTypeDef*. To model initial COs, the metamodel contains the class *COSetDef*, which defines the creation of an arbitrary number of instances of the associated *COTypeDef*. A *COSetDef* is contained in an *AssemblyDef*. To model initial connections, the metamodel contains the class *ConnectionDef*. A connection is the established between ports of of the participating COs by the exchange of the interface references. These references are obtained form a CO where the CO-Type has a provided port definition and is transferred to a CO whose type has a used port definition. In the metamodel, a *ConnectionDef* consists of a set of *ConnectionEndPoints*. A *ConnectionEndPoint* is associated with a *PortDef* of a *COTypeDef* and a *COSetDef*.

3 eODL graphics in XMF

XMF-Mosaic from Xactium is a platform for building tailored tools that should provide high level automation, modelling and programming support for specific development processes, languages and application domain. The tool is implementing the construction of a layered executable metamodelling framework called XMF that provides semantically rich metamodelling facilitied for designing languages. The Mosaic platform is also realizing the Language Driven Development (LDD) process presented by Xactium in ??. LDD is a model-driven development technology based on MDA standards, and it involves adopting a unified and semantically rich approach to describe languages. A key feature of the approach is the possibility to describe all aspects of a language in a platform-independent way, including their concrete syntax and semantics. The thought is that these language definitions should be rich enough to generate tools that can provide all the necessary support for use of the languages, such as syntax-aware ediotrs, GUI's, compilers and interpreters.

XMF provides a collection of classes that form the basis of all XMF-Mosaic defined tools. These classes form the kernel of XMF and are called XCORE. XCORE is a MOF-like like modelling language, and are self-describing. All

XCORE classes are instances of XCORE classes. XMF provides an extensive language for constructing tools called XOCL (eXtensible Object Command Language). XOCL is built form XCORE and it provides a language for manipulating XCORE objects. In addition to XCORE, XMF provides a collection of languages and tools defined in XOCL. These include the following:

– OCL used to define the rules that relate the domain concepts
– XOCL used to capture the behavoiur of the language
– XTools used to specify the concrete graphical syntax of a language and modelling user interfaces.
– XBNF used to define the concrete textual syntax of a language and to build textual parsers.
– XMAP used for model to model transformation.

The XTools are most important in this context and will be described in more details in the following sections.

3.1 Specifying the Graphical Representation

A domain model forms the fundament when specifying a language in XMF-Mosaic. The domain model defines the most fundamental part of a language, the domain concepts and their relation to each other. In this case the domain model is the abstract syntax metamodel for eODL described in Section **??**. While the abstract syntax describes the concepts in a language, the concrete syntax says something about the concrete representation of those concepts. There are two main types of concrete syntax: textual, where the intstances of the domain model are represented as text, and graphical, where the instances are represented as graphical diagram elements. The concrete graphical syntax is the one we are discussing in this article.

Figure **??** shows a metamodel of diagrams, which defines what it mean to be a diagram. This model is presented by Xactium [(??)], and is inspired of OMG's diagram interchange model [(??)].

A diagram consists of *Nodes* and *Edges*, where a *Node* again consists of a collection of *Display* elements. In this metamodel there are defined six different types of display elements: *Box*, *Ellipse*, *Line*, *Text*, *Image* and *Group*. All *Display* elements have attributes which specifies their position. Both *Ellipse* and *Box* can act as a container of other display elements. This means that a node could be represented as e.g. a *Box* with *Text* inside or as an *Ellipse* with an *Image* inside.

A *Node* has a number of *Attachment Ports* that are used to define where a *Node* could be connected to an *Edge*. A *Node* without any *Attachment Ports* cannot be connected to an *Egde*. An *Edge* has a source and a target *Node*. The model makes it possible to say something about the *line style* to an *Edge*, which defines how the *Edge* should be drawn, e.g. solid line or dashed line. If the *Edge* has any arrows, this is also specified here, by arrowhead_target and arrowhead_source. The labels of an Edge are text fields that could be attached to the start, middle or the end of the Edge.

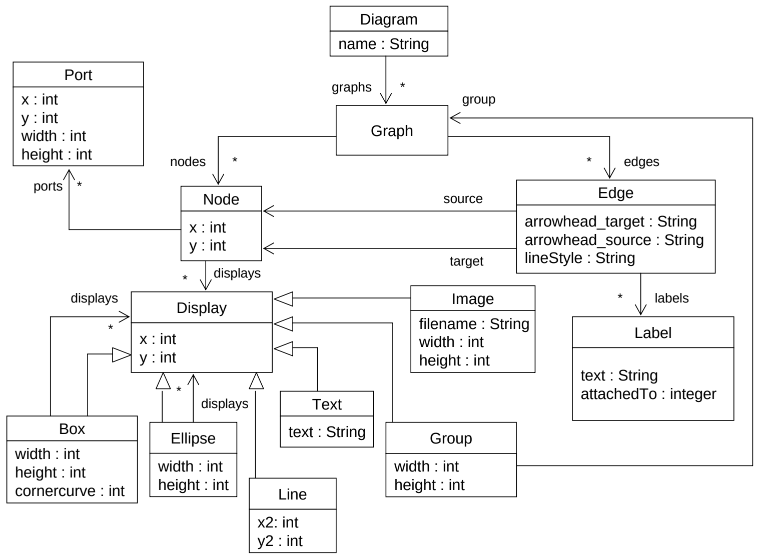


Fig. 3. A metamodel of diagrams

This metamodel of diagrams from Xactium has some weaknesses, but since our own research at the field has not come to far yet, we still want to use is as an illustration. One of the weaknesses is the abstraction level which is too low. This could be explained by the inspiration from the UML diagram interchange model. Another problem with this metamodel is that there is no possibility to describe the spatial relations between nodes. There is cases where it would be useful to explain how the display elements are arranged in relation to another in a Node that consists of more than one display element, and also to express that e.g. a Node should be placed inside another Node. This is not possible in this metamodel of diagrams.

Figure **??** is an example from the eODL editor modelled in XMF-Mosaic and shows a situation where a Node is placed inside another Node.

This diagram in contains four node types: assembly, CO set, provide port and use port, and one edge type: connection. The assembly Dinner is represented as box inside another box. The innermost box has minor height than the outermost. Inside the outermost box are there also some text telling the name of the assembly. The assembly contains a number of CO Sets with ports, and connections between the provided ports and the used ports. A CO Set is grapically represented by a box with text inside, while the used ports and the provided ports are represented by images. The provided port also has a text label. The

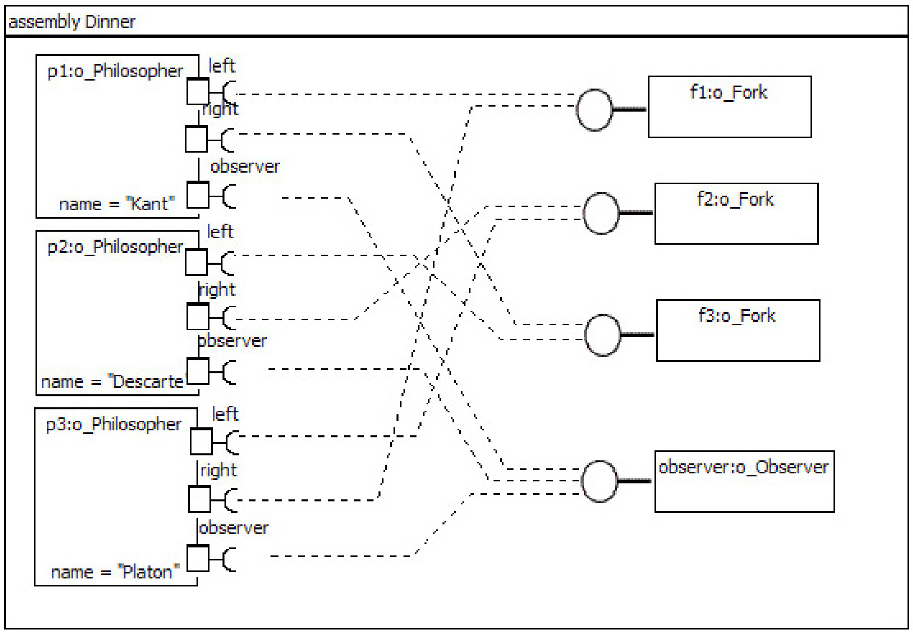


Fig. 4. An eODL model from XMF

connections are edges placed between the ports, and the linestyle of the edge are "dashed line". Neither the source or the target of the edge have arrows.

The general metamodel of diagrams in Figure **??**, is in Figure **??** adjusted to fit the eODL diagram presented above (ref. Figure **??**). This metamodel shows how the classes from the abstract syntax metamodel relevant in this example are related to classes in the general model of diagrams. For example, an *AssemblyDef* is a *Node* and represented as a number of Boxes, and a *ConnectionDef* is an *Edge*.

3.2 Some more examples from the eODL editor

When specifying the graphical syntax of eODL in XMF-Mosaic, we are at the same time building a user-interface for a model editor for eODL diagrams. Figure **??** and **??** are showing some screenshots from the eODL diagram editor. The XTools is used to specify the graphical representation of the components in the language, but provides also support for specifying how the tool bar in the editor should look like (ref. Figure **??**). Another important aspect is the possibility to say something about the events that the user raises when creating nodes and edges in a diagram, and also when editing the display elements.

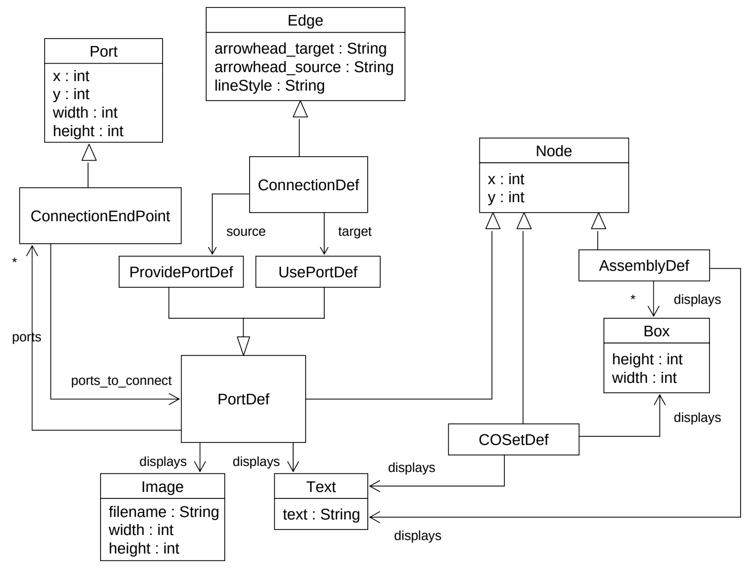


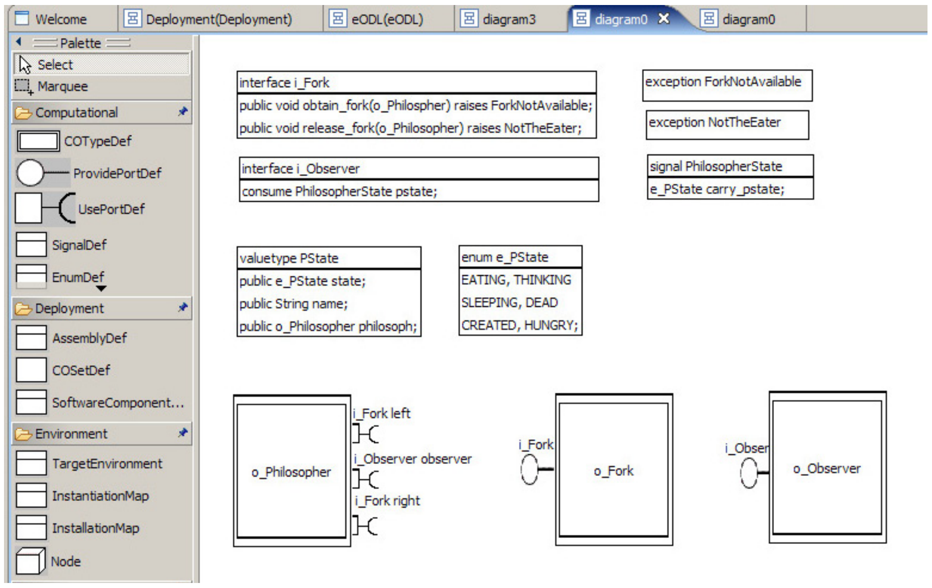
Fig. 5. The metamodel of diagram specialiced for the philosopher example

4 eODL graphics as UML profile

4.1 Profiling in UML

The *Unified Modeling Language* (UML, [**?**,**?**]) is a universal modelling language; it uses multiple modelling paradigms and several diagram types to model all aspects of a computer based system in all stages of its development. Thereby UML allows to express the system under investigation in platform independent or platform dependent models. Based on this broad conception, the UML recommendation deliberately supplies only very loose semantics and flexible notations. The UML semantics, described in English text, is in some points open for interpretation, in others it explicitly offers different semantics, defined through *semantic variation points*, and UML's graphical syntax provides many notational options.

To define a modelling language with concrete specific semantics and notations, tailored for the use in a specific domain, UML offers the extension mechanism *profiles* [**?**]. UML profiles allow language extensions that formally specialise UML's language structure (meta-model) and informally clarify its semantics and notations. But UML profiles restrict extensions of UML to specialisations of already existing UML language concepts. Thus every model developed under a



```
interface i_Fork
public void obtain_fork(o_Philospher) raises ForkNotAvailable;
public void release_fork(o_Philosopher) raises NotTheEater;

interface i_Observer
consume PhilosopherState pstate;

exception ForkNotAvailable

exception NotTheEater

signal PhilosopherState
e_PState carry_pstate;

valuetype PState
public e_PState state;
public String name;
public o_Philosopher philosoph;

enum e_PState
EATING, THINKING
SLEEPING, DEAD
CREATED, HUNGRY;
```

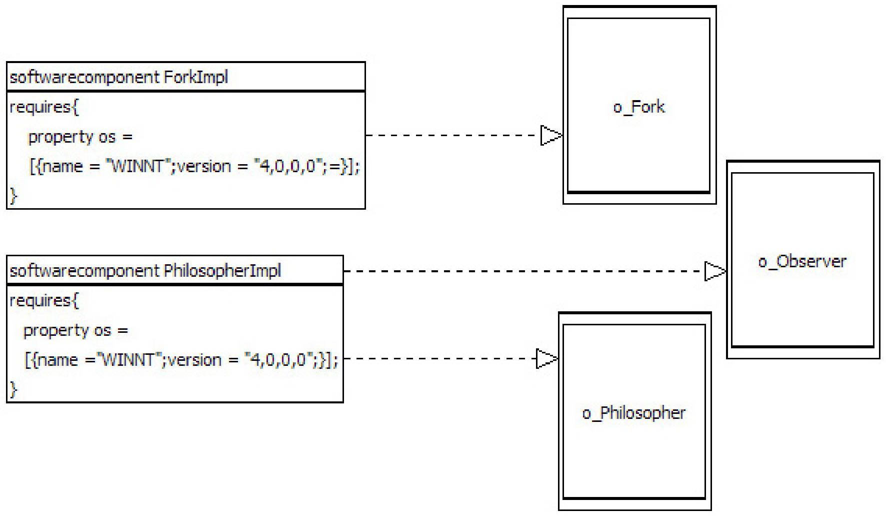
Fig. 6. Screenshot from the XMF-Mosaic

specific profile is still a UML model. This guarantees that such models can be constructed and used with existing UML tools.

A UML profile serves two general purposes: (1) it can provide a notation and tool support for this notation; (2) it can provide more precise semantics for UML by applying the specific semantics of the concrete notation as specialization to UML.

The language eODL has neither a concrete graphical notation nor tool support for it. But it has precise semantics and there are tools [?] that implement this semantics. ITU-T languages such as SDL, MSC, and eODL cover aspects or views that are also covered by UML. But the ITU-T languages provide more formal semantics and they are tailored for modelling systems in the specific domain of telecommunication.

In this section we want to present our experiences with developing a UML profile for eODL. In section **??** we describe the process of aligning the concepts of both languages. We will introduce some typical differences between the concepts of the two languages and explain how to capture those differences in a UML profile in section **??**. Finally, we will visualize a part of the philosophers example as a UML model using this profile and discuss the results according to notations and language semantics in section **??**.



```
softwarecomponent ForkImpl
requires{
   property os =
   [{name = "WINNT";version = "4,0,0,0";=}];
}
```

```
softwarecomponent PhilosopherImpl
requires{
  property os =
  [{name ="WINNT";version = "4,0,0,0";}];
}
```

Fig. 7. Screenshot from the XMF-Mosaic

4.2 Aligning the eODL Meta-model and the UML Meta-model

To develop a UML profile for an existing language, the concepts of that language have to be aligned with UML's concepts. For each language concept the most specific UML concept that still generalises the original language concept has to be identified. The specialities of the language concepts must be modelled in stereotypes, expressed using formal constraints, tagged values, as well as additional notations and semantics.

The structure of UML concepts is modelled in UML's meta-model. The concepts of eODL are also modelled in a meta-model, as presented in section **??**. The presence of the two meta-models makes it easier to find a precise profile for eODL. To find an appropriate UML concept for each eODL concept, we compared both meta-models with each other. For each eODL concept, i.e. for each class in the meta-model, we identified concepts in UML with similar semantics and compared the adjacent structure of the eODL concept class (associations, attributes, etc.) with those of the UML meta-classes. In the ideal case, we found identical structures, where all associations, attribute, etc. have the same properties, except for different names. But this ideal is rare. In the next section we will provide some examples of structural differences in the meta-models, and we will find strategies to solve those problems.

4.3 Rendering the Differences between eODL and UML in a Profile

Figure **??** depicts three examples from the meta-models of eODL and UML, showing the concepts (port, computational objects, and assemblies) on both

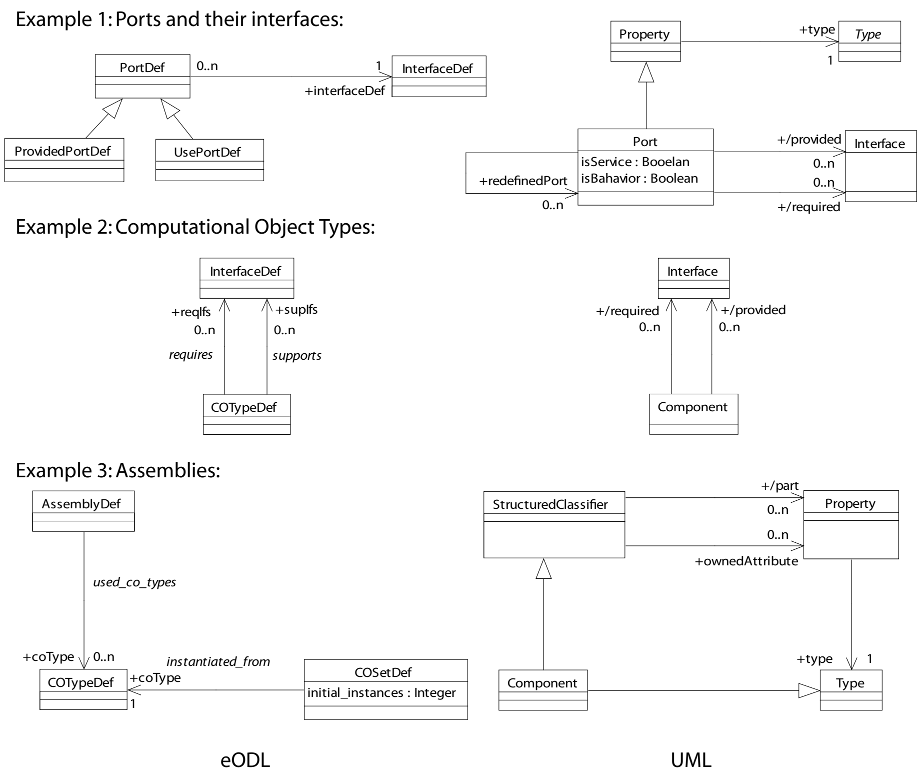


Fig. 8. Examples from the meta-models of eODL and UML

sides. The first example regards the *port* concept; in this simple case eODL and UML are almost equivalent, except that the two concrete descendants of *PortDef* are rendered in the UML attribute *isService*. A corresponding stereotype *PortDef* must only constrain *redefinedPort* to be empty (there is no explicit port redefinition in eODL) and *isBehavior* to be false (this is what matches the eODL semantics best). The properties *required* and *provided* derive from *Port*'s *type* and *isService*, therefore *InterfaceDef* must be mapped to *Type*. The derivation from *required* and *provided* can be reversed, so that for each UML model the analogous eODL model can be derived and eODL semantics can be applied to the UML model.

In the second example, showing the interfaces of computational objects, the *requires* and *supports* interfaces of computational object definitions seem to match with UML's *required* and *provided* interfaces, but they have different semantics. The set of *provided* interfaces in UML derives from all interfaces that are provided by the components ports, its realizing classifiers, and the explicitly modelled *realizedInterfaces*; where only the latter represents the obvious eODL

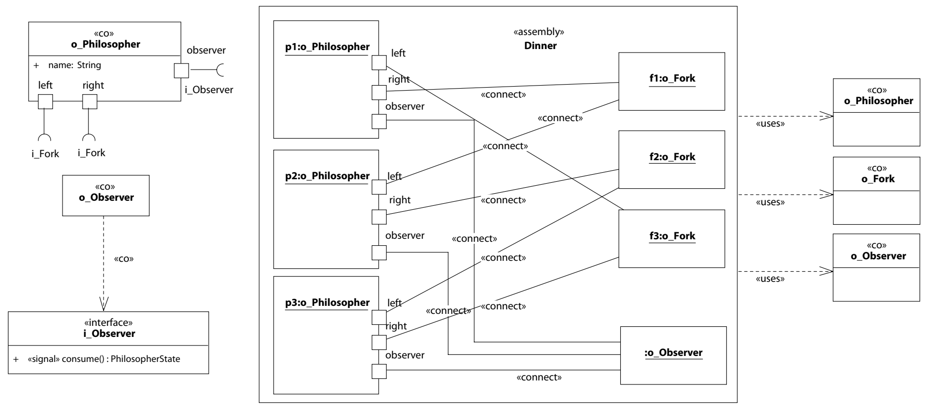


Fig. 9. An eODL model in UML using the eODL profile – Computational object types and assembly

counterpart. The same holds for *required* interfaces that should be modelled with *usedInterfaces* to match the eODL semantics.

The last example shows the composition of computational object sets in assemblies. This very specific eODL relationship has only a quite abstract equivalent in UML. Computational objects have to be modelled as properties, as structural features of a classifier (assembly); the *initial_instances* attribute can be modelled with the property's multiplicity. Only concrete concepts qualify as base classes for stereotypes, thus a concrete UML classifier has to be chosen. Both *class* and *component* are such classifiers, but only *component* is appropriate, because only it can contain connectors to connect different computational objects. The relation between *AssemplyDef* and *COTypeDef* has no concrete counterpart in UML, and is indeed not exactly necessary, because it could be derived from the computational objects contained in an *AssemplyDef*. To model the relation in UML anyway, one has to use the *usage* relationship; this is a very abstract relation between two model elements that does not really match the specific eODL semantics.

4.4 Examination of the Results of the Philosophers Example

The previous section has shown that the eODL and UML concepts sometimes match very well and sometimes require more complex stereotypes, but basically eODL can be expressed in UML by restricting UML's concept space and semantics according to the rules for defining UML profiles. Furthermore, a UML specification can easily be mapped to an eODL specification using the eODL profile, whereby this mapping reflects the eODL semantics upon UML.

The other important point, besides semantics, is notation; can eODL be satisfyingly expressed in UML diagrams? Figure **??** shows an example that covers the concepts that we introduced by the meta-model; figure **??** shows diagram

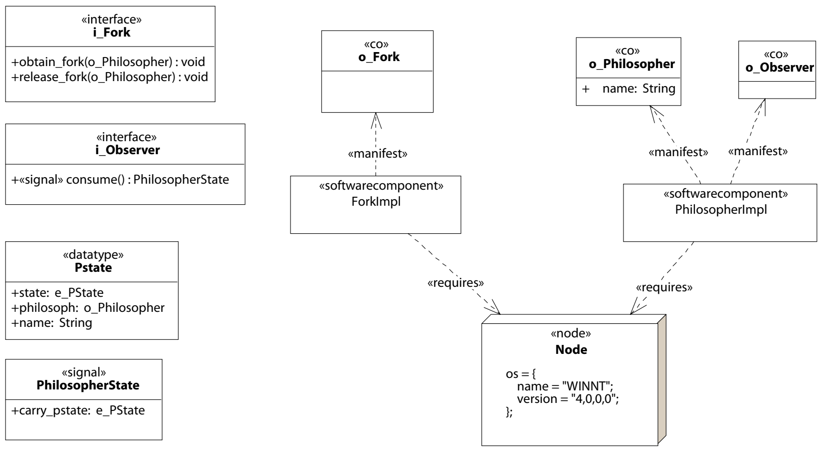


Fig. 10. More model elements in the eODL profile – Interfaces and software components.

elements for other eODL concepts. Wherever eODL and UML concepts are very similar (e.g. eODL computational object definitions and UML components), the notation is very clean, and due to eODL bonds to the UML notation is almost identical to the notation introduced in section **??**. The usage of more abstract UML concepts, on the other hand, requires a lot of stereotypes and results in diagrams that do not comply with typical UML practices. The *uses* dependency, for example, would probably never be used that way in a pure UML specification.

Finally we consider UML tool support. In the quest for a UML profile that leads to specifications that resemble the original notation, the profile engineer tends to utilize the whole spectrum of possible UML notational options and even wants to introduce own notations (theoretically allowed in UML profiles). But UML tools seldomly support every UML notation or even custom profile notations. So it is hard to write a good profile, in particular when one wants to be independent of a specific UML tool.

5 Conclusion

We have shown how two ways of representing eODL graphically: an explicit high-level description of the graphics in terms of XMF and the use of a UML profile. Both of them are connected to the eODL meta-model and both describe the same language.

As the diagrams show, there is a strong similarity of the profile diagrams with UML. This is acceptable for some parts of eODL, that are similar to corresponding UML descriptions. For parts of eODL, that do not have a UML correspondence, this similarity is annoying.

For the XMF version, there is a high-learning curve to be taken before meaningful graphical descriptions can be generated. However, the end result is very appealing.

In summary, the result with XMF is more suitable than with UML profiles. Moreover, the concept of XMF is more powerful, because it allows arbitrary structures to be displayed. Profiles on the other hand do only allow to constrain the existing UML concepts The language used by XMF for displaying the graphical structure of eODL has a semantics and could be used as a description language for languages with a graphical representation (e.g. SDL and MSC).

Overall, we can confirm from our experience that indenpendently from the chosen approach meta-model based language definition allows more efficient development of graphical editors. This is not surprising considering the success of model driven development in general.