From EDOC Components to CCM Components: A Precise Mapping Specification

Mariano Belaunde and Mikael Peltier
France Telecom R&D

mariano.belaunde@francetelecom.com
mikael.peltier@francetelecom.com

Abstract. Nowadays, component-oriented approaches are being promoted by tool providers as a way to enhance the modularity and the reuse of software pieces. Moreover, there are distinct levels of abstraction where components can be defined. The EDOC specification [5] is a high-level approach that introduces composition independently of any middleware platform, while the CCM specification [3] extends the CORBA middleware to simplify the implementation of concrete software components. This paper will focus on the problem of how to specify mapping from component abstract models into more concrete component models. Moreover, we will present a model transformation language called MTrans that can be used to specify *comprehensive* mappings that can be automated in an ambiguous way.

1 Introduction

To manipulate, classify and buy "on the shelf" software components - in a similar way as hardware electronic components - is still a myth. Most software engineering tool providers refer to this technology when they present their products. Especially, many component-oriented frameworks are currently deployed in the industry (for instance the EJB framework, which is widely used). Nevertheless, there is a lot of confusion regarding what a software component is. Inter-operability between components, coming from distinct tool providers, is in fact very difficult to achieve because there is not yet a widely accepted and operational standard. Part of the complexity arises from the fact that there are already multiple middleware platforms that coexist today (such as CORBA, COM, and the EJB).

The Model Driven Architecture [1] from the OMG intend to manage this complexity by taking a model centric approach and by distinguishing models that are platform independent (PIM) from models specific to a platform (PSM). The MDA promises to standardize the mappings that will enable inter-operation and integration of components, even when these are implemented on the basis of heterogeneous platforms.

In this paper we will focus on the problem of specifying *executable* mappings between a platform independent component based specification and a platform dependent

R.-D.Kutsche and H. Weber (Eds.): FASE 2002, LNCS 2306, pp. 143-158, 2002.

component specification. An *executable* mapping is a mapping specification that can be automated by a tool. To illustrate this point we will show how CCM (CORBA Component Model [3]) component descriptions can be derived from high-level EDOC/CCA (Component Collaboration Architecture [5]) component models. The mapping specification will use the MTrans language, a generic formalism for model transformation, developed by France Telecom. This language is based on meta-modeling techniques and intends to mix the "confort" of a declarative language and the efficiency of the procedural style.

In the second section we will present the CCM and the EDOC concepts and point out what are the most relevant differences. The third section will focus on the available approaches used today to specify mappings and we will look in particular the advantages of using meta-modeling techniques. In section four we will describe the general motivation and the fundamental concepts of the MTrans language. In the next section an "EDOC to CCM" executable mapping specification will be presented, first informally and then described formally in terms of the MTrans language.

The last section will point out the future work on the topic of mapping component specifications and will assess some of the expectations in respect of the standardization.

2 CCM Components versus EDOC Components

During the last past four years the OMG has put considerable effort on standardizing the area of software components. The Corba Components [3] adopted specification reflect an attempt to integrate most of the EJB concepts in the context of the CORBA middleware. More recently, the "UML profile for EDOC" [3] specification, which is in the process to be adopted, proposed a high-level component-oriented approach for building enterprise distributed systems. Unlike CCM, an EDOC specification is "platform independent", meaning that it may be implemented in distinct platforms, such as the Microsoft COM, or CORBA. In this section we will describe both of the two frameworks. This will help to understand the mapping specification presented in section 6.

2.1 The CORBA Component Model (CCM)

The Corba Component Model is a very ambitious specification that aims at simplifying the problem of building transactional, robust and secure enterprise systems. In contrast with traditional monolithic transactional systems, it promotes modularity and the reuse of the standardized CORBA services, such as security, notification, transaction and persistence. The CCM specification includes a conceptual model of composition (see the metamodel excerpts in Figure 1), a programming model, a deployment and an execution model. By providing a well-defined environment that manages *containers* and *home* factories for the components, the programmer can focus on the re-

quired business functionality rather than dealing with all the non-functional behavior aspects (such as security).

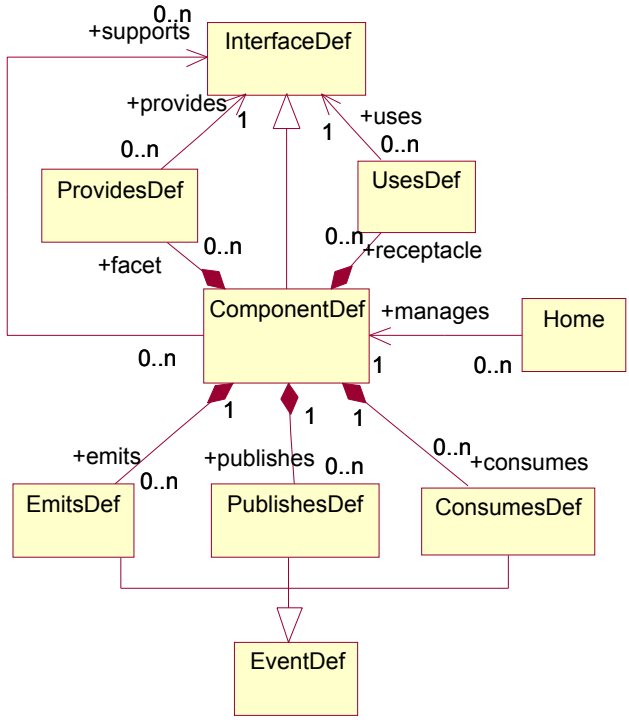

Fig. 1. The CCM metamodel

The CORBA interface definition language (CORBA-IDL) has been enhanced to address the new fundamental types needed for components. A *CCM component* declares the interfaces that are provided (facets) and the interfaces that are used (receptacles) using named ports. In addition a component declares ports for receiving asynchronous events as well as ports for emitting or publishing events. Finally, a component may define the properties that are needed for its configuration.

Surprisingly, composition in CCM is not recursive: a component cannot be defined as a composition of other "small" components. However, an assembly type can be declared to gather a set of CCM components "working together". An assembly descriptor (described using an XML vocabulary) declares the way how components instances are connected together through their respective ports. Also, it can provide collocation constraints on the component instances (either the components execute on a single node, either they are driven in a single process).

2.2 The EDOC Component Model

The RFP (Request for Proposal) "UML Profile for EDOC" [2] called for a standard way to use UML for designing distributed enterprise systems, with the assumption that the design will adopt a component-oriented approach and that it will be "mappable" to the existing software component frameworks (i.e. the CCM, the EJB, etc.).

The final submission, recommended to adoption in September 2001 is a "melting-pot" of distinct interesting works, but containing also a lot of inconsistencies. There are specific UML profiles for modeling event-driven systems, as well as for business process. Anyway, the Enterprise Collaborative Architecture (ECA) which is part of the EDOC submission, has proposed a general composition model (see Figure. 2), that summarizes well the state of the art on components: recursion in composition, port specification using protocols, configuration properties, etc. In addition, a graphical notation has been proposed to be used as an alternative to the standard UML icons. Figures 3 and 4 in this paper uses this notation which is more appropriate to figure out the component structure.

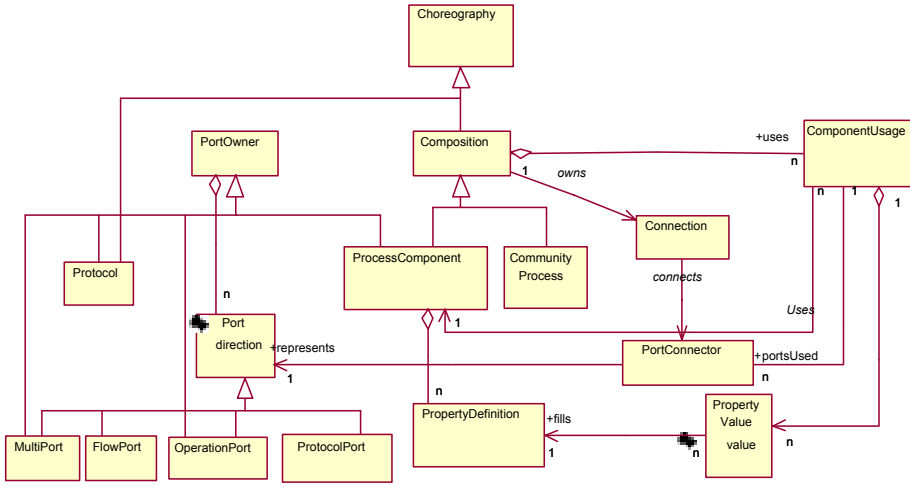

Fig. 2. The EDOC/ECA metamodel

An ECA component has an external view that includes the declaration of ports. A port is simple or composite (containing sub-ports). It plays either the role of an "initiator" or the role of a "responder". The interaction through the ports needs to conforms with a protocol specification. A protocol may imply complex interactions between the two parties. A protocol *choreography*, which is represented by an activity diagram in UML, is used to describe the ordering of the messages. Note that an interface (a collection of synchronous operations) is treated as a particular case of a protocol. A *community* is a special kind of composite that reflects a collaboration of top-level component instances. Figure 3 shows a community with three components.

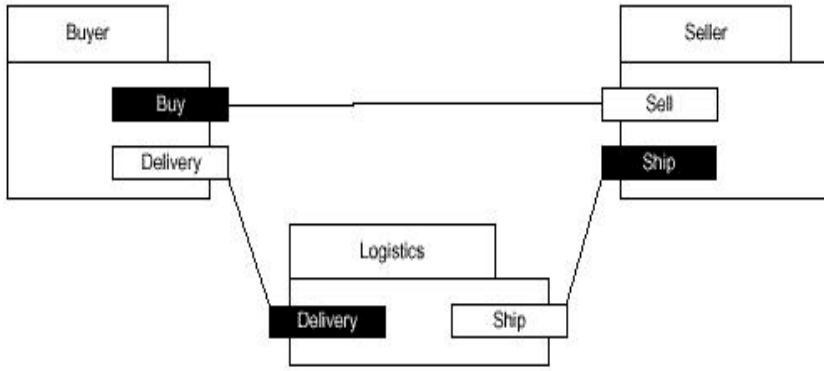

Fig. 3. The Buyer/Seller community

An ECA component may expose its internal structure (its *inside*) or it may declare a "performer" role that implements it. In the former case, the internal view of the component is described as a collaboration of other component instances (*usages* of other components). The connections between the ports of the component instances are explicitly described. A choreography may be used to describe in detail the behavior of this collaboration.

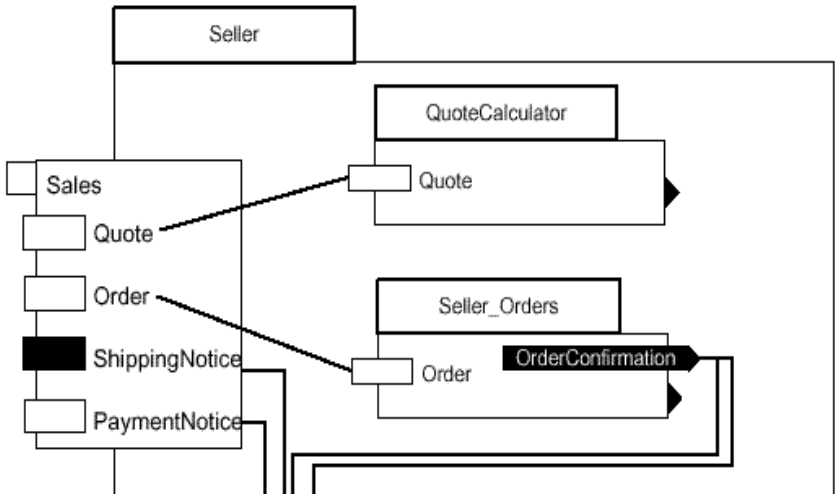

Fig. 4. A partial view of a "seller" component

3 Techniques for Specifying Mappings

A transformation is a process which is used to produce coded data in a format towards another format. This transformation can either consist of an one-to-one relation (correspondence) between elements either be more complex and involve intermediary computations to realize the transformation.

3.1 Current Approaches

The need for specifying mappings between models has increased dramatically today in the object modeling community. This is due at least to two main reasons:

- The UML graphical notation often serves as a *concrete notation* in specific domains. This is made possible because of the extensibility mechanism in UML that permits to extend the semantics while preserving the notation core. However, independently of the usage of the UML diagrams, designers have often a metamodel of the domain concepts. Thus, a mapping specification is needed to show how this domain concepts maps into the UML concepts. The Software Process Engineering Management [4], the "UML profile for EDOC" [5] and the "UML profile for EAI" [6] are examples of OMG specifications providing both a metamodel and a UML profile.
- The MDA promotes the separation between platform independent models and platform dependent. This requires to specify very precisely the mapping between models located at distinct levels. As an example, the "UML profile for EDOC" submission has proposed some non-normative technology mappings, such as "EDOC to EJB" and "EDOC to CCM"[1].

How do the mapping between a source and target models be specified today in the specifications produced by the OMG?. There are distinct approaches, but all of them are more informal than formal. The SPEM and EDOC submissions, use tables providing the correspondences between the concepts. We will typically find some explanation in natural language accompanied with a three column table in the form: *source-entity/map-comment/target-entity*.

This approach is suitable when the transformations are really simplistic, for instance, in the case of UML profiles, when each domain concept is translated using a single UML stereotype. But in general, things are more complex. In order to really take advantage of all the UML diagramming capabilities, a domain concept may be showed in different places with a distinct identity (in SPEM an activity is represented either as an operation, or as an action state, or as a use case, etc.). There are also many complex short-cuts conventions that are difficult to specify precisely using tables.

In the "UML for EAI" submission [15], in addition to this correspondence tables, the mapping rules are further refined by means of "invariant constraints". For example

[1] The mapping presented in section 6 reuses parts of the proposal included in this specification.

"The name of the terminal is the name of the target end of the association". This approach is interesting. The constraints are expressed in English but we could imagine that they can also be translated as OCL invariants (with some extensions to the OCL language). This formal specification could be used to check that a program has performed correctly the mapping. However it will be very difficult (probably impossible) to generate automatically the program itself!

To summarize, it is really difficult to specify precise mappings formally (and even harder if we want these to be readable!). In general the semi-formal mapping provided in the OMG specifications are useful to understand the fundamentals of the mapping. But in the perspective of an implementation, they offer a limited help to human programmers (and even less to an automated engine). A lot of implicit information need to be re-interpreted by the programmer. This includes for instance, the determination of the ordering to accomplish the transformation (the visiting strategy).

3.2 Towards an Executable Mapping Specification

An executable mapping specification is something that ideally should be a pure "declarative" language. However, our feeling is that a compromise is needed between "the procedural" and "declarative" paradigm in order to have a highly expressive, predictable, understandable formalism that could be automated in an efficient way.

Our focus is on executable formalisms that operate with object oriented model definitions. There are examples of other non OO approaches, like those based on XML and XSLT[2], or those based on EBNF production rules[3].

From now on, there a lot of pragmatic solutions for model transformations that mix declarative and procedural aspects. For instance, in a UML Case tool, such as the Objecteering [12], or a transformation engine such as Scriptor [13], a dedicated language is used to inspect and create the model elements (in compliance with the UML metamodel). Transformations rules are written for each UML meta-class using either pseudo-formal navigation expressions, either templates with placeholders. Furthermore, it is possible to derive automatically this transformation rules from models representing design patterns. A very important work in the topic of design pattern application has been achieved with the UMLAUT framework [9].

Anyway, in the context of our research project, we were more interested in an explicit and executable object oriented formalism, tool independent, that would not be dependent on a unique metamodel. In other words, what we were looking for was a kind of OCL extension for model transformation based on the MOF. The MOF provides a

[2] There are a lot of limitations that appear when using XSLT processors to transform models rendered as XML documents (using the XMI standard).

[3] The productions rules for XML schemas in [16] are expressed in this way. The rules are precise and unambiguous.

very simple object-oriented meta-language that is used to define other languages (metamodels). The MOF constructs are in fact those commonly used in *class-diagrams*: classes with attributes, binary associations with cardinalities in both ends, class inheritance and so on. Full object-orientation, genericity and good integration with UML graphical notation are the main advantages that emerge from this approach.

A lot of research is being carried out today to apply meta-modeling techniques to model transformation. There were also some interesting results in the last past years. In particular, we would mention the approach taken by R.Lemesle [10], which proposes a prolog-like declarative language with selection clauses and conclusions. The implementation was based on the sNets formalism [11] which was used to encode the MOF *metametamodel* concepts as well as the rules.

3.3 How to Go from an Abstract Model to a Concrete Model

When going from an abstract model of a system into another more concrete model of the same system, the transformation process requires in general additional information to be provided. This includes:

- Decisional choices to set distinct mapping alternatives. For example: what security level will be needed in the concrete system?
- Enrichment of the source model in order to take in account aspects not originally expressed in the source model but that are relevant to the target.

There are different techniques to provide this additional information. For instance global parameters may be passed to the transformation engine. In UML models, *tagged values* are often used to annotate the source model with additional data. Another approach is to use distinct inter-related models as the input for the transformation. A security model and architectural model may, for instance, be linked with the domain specific model to be transformed.

4 The Mtrans Language

MTrans is a textual formal formalism to express transformations on models. It's an executable formalism in the sense that it has unambiguous execution semantics and can be automated by a tool. Access to the source model elements and the assignment of the target model elements is expressed in the terms of the concepts and the properties existing in the source and target metamodels. MTrans reuses the navigation capabilities of OCL, a formal and pure expression language used to define constraints on OO models.

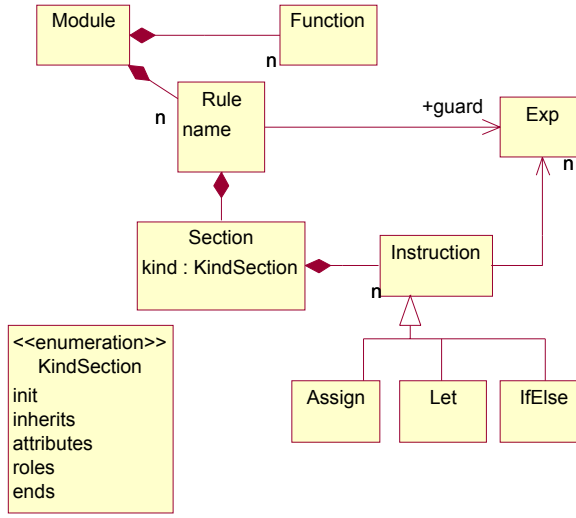

Fig. 5. Main syntax constructs in Mtrans

4.1 Foundations

A MTrans specification is made of a set of *rules* and a set of *function definitions*[4] (see fig 4). A rule indicates how an instance of the target metamodel is created and how its attributes and roles are assigned in the context of an instance of the source metamodel. A rule has an optional name and contains distinct sections. In each section there are instructions. The 'init' section contains initial computations that are performed before creating the destination instance. The 'attributes' and 'roles' sections are used to assign the properties of the destination instance (which is created at the end of the 'init' section). A rule may have additional parameters (other than the implicit 'self' contextual source instance).

A MTrans specification declares one or more rules serving as entry-points. Multiple entry points allow either to perform multiple passes, or to apply the transformations to separate parts of the model. A rule may invoke another rule. This is done using the *new* operator. *Entry-points* and *new* operations allow the MTrans programmer to have an explicit full control on the flow of execution.

A rule can inherit from one or more other rules. The destination type needs to be compliant with the destination types of the inherited rules. The inherited rules are executed

[4] OCL definitions (expressed with the 'def' keyword) are a special kind of function definition in Mtrans.

after creating the destination instance and before the execution of the property assignment.

Rule inheritance permits the reuse of code that populates the properties of the *destination* model. In contrast, a *function definition* is used to reuse the code that inspects the *source* model. A rule, for instance, cannot be invoked in a function body. As in OCL, a MTrans function has a context argument (referred as 'self'). It may be of any known type, including classes from the source model, as well as basic data-types and structured data-types (sequences, sets, etc.). All the predefined OCL definitions (such as the set of operations defined for collections like *collect* and *select*) can be used in function bodies.

4.2 Special Situations

- An *abstract* rule is a rule that can only be used by inheritance.
- A rule may not specify its destination. In such case no instance is created in the target model and only the 'init' section is available. The instructions in the 'init' section serves only to invoke other rules in inner source elements. This is often useful to entry-point rules.
- A rule may not specify its source instance. In such case the first parameter acts as the contextual argument (referred as 'self'). This is useful to create, for instance, pre-defined destination elements that are not necessarily linked with a source element.
- A function can be declared as *external*. In this case the body is not present, meaning that the implementation is provided elsewhere. This is useful for implementing predefined functions that will be linked afterwards to the transformation MTrans engine.

4.3 Advantages of an Explicit Execution Strategy

MTrans is obviously not a "pure" declarative language; instead it's a mixture of "declarative" and "procedural" formalism. A MTrans specification is in fact a *program* with a specialized structure that clearly shows what is its intent. We believe that an explicit execution strategy[5] brings many advantages in terms of the precision and the capacity to express complex situations. In non trivial transformations, it's often useful to visit a model element more than once[6]. Since we know what transformations were made before, it is possible to reference the destination instances already created. The *resolve* operation is used to obtain all the destination instances created from a source

[5] A strategy describes how we want to apply the different steps of the transformation process.

[6] Section 5.2 shows an example of this: The ECA community is visited twice, once for creating all the CCM component types and a second time for creating all the connections in the CCM assembly.

instance (this is similar to the *co-reference* mechanism found in [10]). Note also that without an explicit execution flow, the rules inheritance mechanisms, used for rule reuse, would have an ambiguous semantics.

4.4 Special Rules and Conventions in MTrans

It is not always easy for a human reader to understand OCL expressions that involves a long chain of iterative operations. In MTrans, to minimize this problem there are some special rules and some short-cut conventions (abbreviations) that make things appear simpler to the end-user of the language.

- "***Do nothing on abort***": In the assignment instruction '`myattr= self.x.y'` the assignment will take place only if the access to the x and y fields is successful;

- "***Chained assignments***": The instruction '`myrole=a;b;c;'` specifies that *myrole* will merge the results of the three expressions in a single list;

- "***Filtering notation***": The expression `x[filtering-expression]`, where x is a sequence, expresses that the resulting list will be filtered according to the filtering condition (which is expressed in terms of an implicit iterator). It I s equivalent to use in OCL a *select* operation.

- ***Implicit macros***: The filtering expression `x[M]` where M is a metaclass, expands as `x[iterator.isOclType(M)]` (only instances of type M are accepted)[7].

- "***pipeline operations***": The instruction `x->f()`, where x is a collection, results on calling once the 'f' function if f is declared as having a collection as the context argument, or results in calling n times the 'f' function if 'f' declares a non collection context argument. This convention applies also to the *'new'* operator to invoke a rule on each element of the list. Note that in OCL there is a similar convention for 'x.propertyname' where x denotes a collection.

- ***Dynamic casting***: When defining and initializing local properties definitions (using the '*let'* keyword) the type is not mandatory. However, the full signature is mandatory when defining functions signatures.

[7] In MTrans disambiguation rules are used to prevent potential conflicts that may arise. In standard OCL the [] operator is also used for accessing class-associations instances.

5 Mapping from EDOC to CCM

5.1 General Principles of the Mapping

The main issue when mapping an EDOC model into a CCM model is how to deal with the EDOC recursive composition. An EDOC component may be defined as a composition of other components, while, in CCM, an assembly cannot act as a part for another composite. When going from an EDOC model to a CCM model, we have then to decide what to do with the *inside* of a composite EDOC component. Many options are possible. For the purpose of this presentation, we assume the following mapping strategy: each EDOC component, whether it's a top-level or an inner one's, is translated as a CCM component type. Furthermore, the whole EDOC community is mapped as a single CCM assembly that declares all the needed connections between the CCM component instances. In order not to loose component encapsulation, each CCM component acts as a proxy for all the CCM components resulting from its "inside". To achieve this we need to perform a deep traversal on the ports structure of each top-level EDOC component and distinguish between ports that are used for "internal" communication from those that are used for "external" communication. Depending on the port mode (synchronous/asynchronous) and the port role (initiator/responding) we generate either sink events, source/published events, facets (provided interfaces which contain operations reflecting the synchronous flows) and receptacles (used interfaces reflecting consumed flows). [8]

5.2 A Formal Specification of the EDOC to CCM Mapping

In this section we present some excerpts of the *executable* mapping specified using MTrans. This mapping is directly expressed in terms of the EDOC and the CCM meta-models (described in Figure 1 and Figure 2)[9].

The first stage in the transformation process is to inspect in two passes the top-level Buyer/Seller community (see Figure 3). The first pass (rule R1) creates the CCM assembly structure while the second one (rule R2) creates CCM *home* instances for each EDOC component found in the community.

```
R1 entrypoint rule Assembly from CommunityProcess {
            -- We do not present this part in the paper -- }
R2 entrypoint rule undefined from CommunityProcess {
     init:
       self.uses.processComponent[unique]->new HomeDef(); }
```

[8] The principles for mapping ECA flow ports are presented in the EDOC to CCM mapping found in [3]. Although the way how to manage port composition remains very vague.

[9] Another approach could be to use the UML profile for EDOC and the UML profile for CCM. However the mapping based on the specific meta-models is easier to understand.

The rule R3, which is invoked by rule R2, performs then a deep traversal on the EDOC components. Recursion stops when no *inside* is found in the component (when 'self.uses' aborts in (1)). CCM components are instantiated and linked to the *home* that is in charge to manage it.

```
R3 rule HomeDef from ProcessComponent {
   init :
     self.uses.processComponent[unique]->new HomeDef();      (1)
   roles:
     manages = self.new ComponentDef();
   }
```

The Rule R4, invoked in rule R3, specifies how the CCM components are build from EDOC components. The logic here is a bit more complex. The various kind of CCM ports are populated according to the principles exposed in section 6.1. Ports used for external communication are generated in a different way than ports dedicated to the interaction with the internal components. Note that for an external *consuming* CCM port there is a corresponding proxy *emitting* port. For clarity, many other details of the mapping have been omitted here, such as mapping the configuration properties, the transaction parameters, and so on.

```
R4 rule ComponentDef from ProcessComponent {
  init:
   let ax_out =self.port->allAsyncInitiatorFlowPorts();
   let ax_in = self.port->allAsyncResponderFlowPorts();
   let sx_in = self.port[hasResponderSyncPorts()];
   let sx_out = self.port[hasInitiatorSyncPorts()];
   let pp = self.proxyPorts();
   let ap_out = pp.allAsyncInitiatorFlowPorts();
   let ap_in = pp.allAsyncResponderFlowPorts();
   let sp_in = pp[hasResponderSyncPorts()];                  (2)
   let sp_out = pp[hasInitiatorSyncPorts()];
   macro multiple = portUsage.outgoing->size>1;
  roles :
   emits = ax_out[!multiple]->new[External]EmitsDef();
           ap_in[!multiple]->new[Internal]EmitsDef();
   publishes = ax_out[multiple]->new[External]PublishesDef();
               ap_in[multiple]->new[Internal]EmitsDef();
   consumes = ax_in->new[External] ConsumesDef();
              ap_out-> new[Internal] ConsumesDef();
   receptacle = sp_out->new [External] UsesDef();
                sp_in->new [Internal] UsesDef();
   facet = sp_in->new[External]ProvidesDef();                (3)
           sp_out->new[Internal]ProvidesDef();
}
```

The R5 rule, described below, is invoked in rule R4 for each external and top-level port that contains at least one inner synchronous and responder sub-port (see the marks (2) and (3) in rule R4). The purpose of the R5 and R6 is to generate all the provided interfaces that are useful to a CCM component to be used from its outside. An operation is generated for each synchronous and responder sub-port belonging to the current source port. Note that the list of the sub-ports is passed as a parameter to the rule R7.

```
R5 rule [External] ProvidesDef from Port {
   roles:
     provides = self.new InterfaceDef(allSyncResponderPorts());
   }
R6 rule InterfaceDef from Port {
   parameters:
     plist : Set(OperationPort);
   roles:
     contents = plist->new OperationDef();
   }
```

All the utility functions, such as *allAsyncInitiatorFlowPorts*, used by rule *R4,* can be coded in Mtrans using the navigability facilities (in a similar way as in OCL). Alternatively, a function can be implemented directly as a native function using the target language of the compiler (such as Java or Python).

5.3 Back to the Buyer/Seller Example

The MTrans specification described in 6.2 has been applied to the Buy/Seller provisioning example (figures 3 and 4).
Below we present the structure of the resulting CCM model.

```
HomeDef Home_Seller {
            manages:
                ref ComponentDef Seller;
}
ComponentDef Seller {
            emits :
              EmitsDef QuoteRequestInternal { type=... }
              EmitsDef Quote { type= DataQuote ... }
              ...
            consumes :
              ConsumesDef QuoteRequest { type=... }
              ConsumesDef QuoteInternal { type=... }
            ...
}
HomeDef Home_QuoteCalculator {
            manages:
                ref ComponentDef QuoteCalculator;
}
Component QuoteCalculator {
            emits:
              EmitsDef Quote { type=DataQuote... }
            consumes:
              ConsumesDef QuoteRequest { type=... }
}
...
```

6 Conclusions and Future Work

Component-oriented specifications can be introduced at distinct levels of abstraction during the software development cycle. Thus a EDOC/ECA model is likely to be used at an "analysis" level while a CCM based description could be used later at "design" phase, reflecting the specific architecture decisions. Obviously, for an organization that have to maintain distinct models on their systems, it is an important issue to reduce the gap between models located at distinct level of abstractions. Moreover, in the context of the OMG Model Driven Architecture, the problem of the mapping between distinct models has been identified as being very important. We can expect that a standard for model transformation will emerge from the OMG organization in a near future.

Nowadays, mapping specifications are often defined informally. Although this kind of description is useful to understand the global meaning of the transformation process, it is not sufficient to avoid ambiguities arising from its interpretation. Moreover, an informal mapping can not be executed without an explicit programming effort.

The MTrans language has been designed to address this problem. This formalism is based on the OCL language, is tool independent and it can be used with any MOF compliant metamodel. A former partial implementation of the MTrans language has been achieved on the top of a XSLT processor. The tool was used to transform SPEM models expressed in UML into SPEM models expressed in the terms of the specific SPEM metamodel. It was also used to assist the production of deployment descriptors from CCM-like component specifications (in the context of the European EURESCOM project P924). We are currently developing a new implementation of the MTrans engine, which will be based on the *Univers@lis* model repository tool [8]. Among some of the new properties that could be addressed in a new version of the MTrans formalism we would mention an explicit support for transformation *patterns* and an enrichment of the navigational operations that could be based on techniques existing in OO databases (such as the OQL language which is the counterpart of SQL for relational databases).

References

1. OMG, *«Model-Driven Architecture»*, document ormsc/2000-11-05, November 2000.
2. OMG, *«RFP: UML Profile for Enterprise Distributed Object Computing», document ad/1999-03-10, October 1999.*
3. OMG, *«CORBA Components»,* document orbos/1999-07-02, August 1999.
4. OMG, *«The Software Process Engineering Metamodel (SPEM)», document ad/2001-06-05, June 2001.*
5. OMG, *«A UML Profile for Enterprise Distributed Object Computing»*, document ad/2001-08-19, August 2001.

6. OMG, *«UML Profile and Interchange Models for Enterprise Application Integration (EAI)»*, document ad/2001-09-17, September 2001.
7. Meta Integration Technology, *«Meta Integration Model Bridge»,* http://www.metaintegration.net/
8. M.Belaunde, "A pragmatic approach for building a user-friendly UML: Repository", UML'99 conference. Web Site: http://universalis.elibel.tm.fr/index.html
9. W.M. Ho, J-M Jézequel, A. Le Guennec and F. Pennaneac'h, *«UMLAUT: An extensible UML transformation framework»,* Technical Report 3775, INRIA, October 1999.
10. R. Lemesle, *«Transformation rules based on meta modeling»*, Proceedings of the Second International Enterprise Distributed Object Computing Workshop (San Diego), November 1998.
11. J.Bezivin, J.Lanneluc, R.Lemesle *«sNets, the Core formalism for an Object-Oriented Tool», 1995.*
12. Softeam, "Objecteering.UML Case Tool"; http://www.softeam.fr/
13. Softmaint, "The *Scriptor transformation engine"* http://www.softmaint.com/
14. OMG*, «Meta Object Facility, version 1.3»,* document ad/2000-04-03, March 2000.
15. OMG, *"UML Profil for EAI"*, OMG document ad/2001-08-02.
16. OMG, *"XML Schemas for XMI"*, OMG document ad/2000-08-14.

All the OMG documents referenced here are available from the OMG website using the url: http://www.omg.org/cgi-bin/doc?<doc-id>.