Extending Profiles with Stereotypes for Composite Concepts[1]

Dick Quartel, Remco Dijkman, Marten van Sinderen

Centre for Telematics and Information Technology, University of Twente,
PO Box 217, 7500 AE Enschede, The Netherlands
{D.A.C.Quartel, R.M.Dijkman, M.J.vanSinderen}@utwente.nl

Abstract. This paper proposes an extension of the UML 2.0 profiling mechanism. This extension facilitates a language designer to introduce composite concepts as separate conceptual and notational elements in a modelling language. Composite concepts are compositions of existing concepts. To facilitate the introduction of composite concepts, the notion of stereotype is extended. This extension defines how a composite concept can be specified and added to a language's metamodel, without modifying the existing metamodel. From the definition of the stereotype, rules can be derived for transforming a language element that represents a composite concept into a composition of language elements that represent the concepts that constitute the composite. Such a transformation facilitates tool developers to introduce tool support for composite concepts, e.g., by re-using existing tools that support the constituent concepts. To illustrate our ideas, example definitions of stereotypes and transformations for composite concepts are presented.

1 Introduction

The profiling mechanism, as defined in the UML 2.0 Infrastructure Specification [10], is a lightweight metamodel extension mechanism. It allows one to specialize any language, provided its metamodel is defined in the MOF, by specializing existing concepts that are represented in the metamodel of that language. By defining profiles on top of a general-purpose language one can re-use tools for the general-purpose language to support the languages that are defined by the profiles. Furthermore, one can develop dedicated languages for specific stages in the design process or specific application domains. Hence, the profiling mechanism combines the efficiency of general purpose languages with the intuitive clarity of dedicated languages.

We claim however that besides specialization, the profiling mechanism should support the extension of metamodels with composite concepts, i.e., concepts that are defined as compositions of existing concepts. In general, the introduction of composite concepts and associated language elements facilitates the task of a modeller and

[1] This work is part of the Freeband A-MUSE project (http://a-muse.freeband.nl), which is sponsored by the Dutch government under contract BSIK 03025.

L. Briand and C. Williams (Eds.): MoDELS 2005, LNCS 3713, pp. 232-247, 2005.
© Springer-Verlag Berlin Heidelberg 2005

increases the clarity of models, because frequently occurring compositions of concepts can be replaced by composite concepts. In addition, the possibility of defining composite concepts allows one to use a general-purpose language consisting of a limited number of elementary and generic concepts. More complex concepts can then be defined as compositions of those elementary and generic concepts. The benefit of such an approach is that, on the one hand, it is easy to maintain consistency and tool support for a limited set of elementary concepts, while, on the other hand, it provides clarity and ease of use, because complex concepts can be defined directly and clearly.

For example, consider the extension of the UML 2.0 action semantics with a TimedOperationCall, which represents the handling of an operation call, including the possibility to set a maximal completion time. A timed operation call involves a number of elementary actions, such as CallOperationAction, AcceptCallAction, ReplyAction and AcceptTimeEventAction (see also the elaboration of this example in section 3.3). This means one has to be able to define which elementary actions are involved and how these actions are related. This is however not possible by defining a timed operation call as a stereotype of an existing concept using the current profiling mechanism.

The contribution of this paper is twofold. First, we propose an extension of the UML 2.0 profiling mechanism with stereotypes for composite concepts. These stereotypes should leave the existing metamodel unmodified. Second, we describe how rules can be derived from the stereotypes to transform a composite concept into the corresponding composition of (elementary) concepts. Such transformation rules can be used to generate tools supporting the dedicated modelling languages that use the composite concepts, based on existing tools for the general-purpose language.

This paper is further structured as follows. Section 2 describes the profiling mechanisms and the trade-off between profiling and metamodelling. Section 3 introduces stereotypes for specifying composite concepts. Section 4 explains how model transformation can be used to implement these stereotypes. Section 5 illustrates some applications of our ideas. And section 6 presents conclusions and future work.

2 Profiling

Profiling allows one to extend an existing language metamodel with specializations of metaclasses and with constraints. The purpose of such an extension is to adapt a language for a particular application domain, development platform or design method. For example, one may want to support specific concepts, notation or terminology. An important restriction is that profiling does not allow one to modify the existing metamodel. Profiling in UML 2.0 can be applied to any MOF-compliant metamodel.

2.1 Profiles Package

Figure 1 depicts the Profiles package from the Infrastructure specification [10]. A *profile* is a kind of package that extends an existing metamodel or profile. A profile contains stereotypes. A *stereotype* extends (specializes) an existing metaclass or stereotype. This *extension* is defined by a specialized association between the stereo-

type and the extended metaclass. Through the extension each instance of the stereotype is associated with an instance of the metaclass that it extends. A *profile application* defines which profiles have been applied to some package.

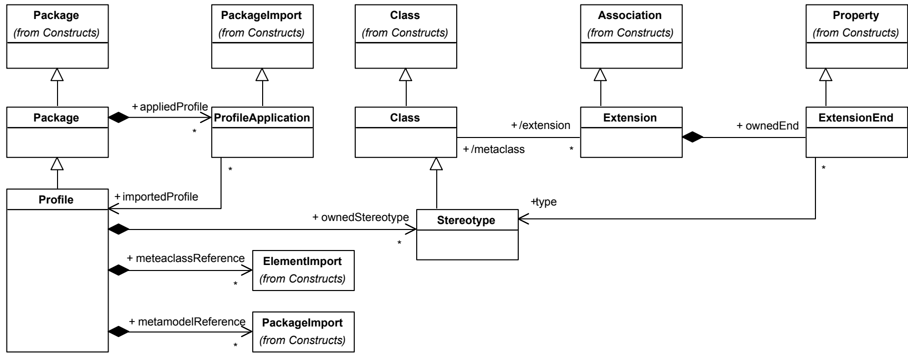

Fig. 1. The classes defined in the Profiles package.

Figure 2 depicts an example of the definition of an EJB profile. A profile is defined as a package stereotyped <<profile>>. A stereotype is denoted by the keyword <<stereotype>> above the stereotype name. An extension association is represented by a filled arrow pointing from the stereotype to the metaclass. The constraint {required} defines that the extension is required, which means that an instance of Bean, i.e., an instance of Entity or Session, must always be linked to an instance of Component. In general, constraints can be associated with stereotypes to specify rules and restrictions on their use. Just like a class, a stereotype may have properties (attributes). These properties extend the properties of the extended metaclass or stereotype. For example, attribute state of stereotype Session defines whether a session object is stateful or stateless. The values of stereotype properties are also referred to as *tagged values*. Package Conference illustrates how the EJB profile can be applied, which is represented by an import association stereotyped <<apply>>. Because state is a meta-attribute of stereotype Session, its value can not be set directly by the ConferenceManagement class, but can be set in a comment box that starts with the name of the stereotype.

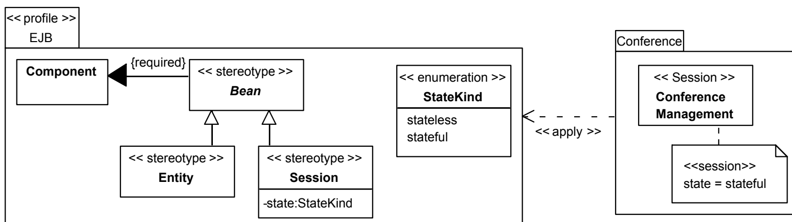

Fig. 2. Example of profiling.

We would like to stress that the Profiles package only provides a way to extend the metamodel, i.e., the abstract syntax, of some language. Language extension also involves the definition of the semantics and concrete syntax for the metamodel extension. This has to be done separately.

2.2 Profiling Versus Metamodelling

In general, two approaches to metamodel extension can be distinguished, which are often referred to as ‘profiling’ and ‘metamodelling’. Profiling refers to the extension mechanism described in section 2.1. Metamodelling refers to the definition of metamodels. An essential difference between both approaches is that profiling starts from an existing metamodel and does not modify this metamodel, whereas metamodelling involves the creation of a new or the modification of an existing metamodel.

The metamodelling approach can always be used instead of profiling. Metamodelling has to be used in case some of the modelling concepts that have to be represented by the metamodel can not be obtained as specializations of existing concepts. Furthermore, if one has a stable set of modelling concepts, one may want to create a separate metamodel and develop dedicated tools, since this pays off by having better modelling and tool support.

Instead, the profiling approach is meant to provide a lightweight extension mechanism that is more easy to use by language developers and more easy to support by tools. This approach can only be used in case the required modelling concepts are specializations of existing concepts. From the MDA perspective this seems sufficient to facilitate the development of transformations from general models (PIMs) to more specific models (PSMs). However, a more expressive profiling mechanism may facilitate the MDA approach even further. In particular, we claim that an extension mechanism for composite concepts is useful and can be introduced while maintaining the lightweight character of profiling. Section 5 discusses some applications of such an extension mechanism for composite concepts.

The characteristic that an existing metamodel is left unmodified has been an important motivation to propose an extension of the profiling approach. The profiling approach avoids that an existing metamodel is compromised, helps to shield distinct language extensions from each other, and facilitates re-use of tool support. One could argue that the same benefits can be achieved by structuring metamodels and their extensions properly, but this would require much more expertise from the language developer. Furthermore, the choice to extend the profiling mechanism should not be considered as a (strong) preference for stereotypes to define language extensions. In fact, some of the ideas underlying the definition of stereotypes for composite concepts can also be used when following a metamodelling approach (see also section 3.3).

Several papers [1,2,3,4,8,15] discuss the principles of and problems associated with metamodelling and profiling in more detail.

3 Specification of Composite Concepts

A concept represents some system property that is considered essential in the development of (software) systems. Concepts form the building blocks for constructing models. A model consists of one or more concept instances, representing the system properties that are conceived by the developer and considered relevant in relation to the purpose of the model in the development process.

An *elementary concept* represents an elementary system property, and forms the smallest unit for constructing models. We define a *composite concept* as a composition of concept instances, where a concept can be an elementary or a composite concept. We define a *structure concept* as a composition of concepts (rather than concept instances). The difference between a structure concept and a composite concept is that, if we want to use a structure concept in a model, we still have to decide on what instances of its constituents we want to use and how we want to associate them. Consequently, a structure concept represents a set of composite concepts, i.e., one for each possible composition of instances of the structure concept.

Composite and structure concepts are commonly used during a development process, either explicitly or implicitly. Examples are compositions, patterns or groupings of model elements; e.g., a transaction that consists of multiple related operation calls is an example of a composite concept, and the StructuredActivityNode in UML's activities that represents a group of activity nodes and edges is an example of a structure concept.

3.1 Representing Composite and Structure Concepts

We represent a composite or structure concept as a class that is related to its constituents by composite aggregations. For example, figures 3(i) and 3(ii) depict metamodels representing the structure concepts ATask and BTask, respectively, which consist of the elementary concepts Action and Flow.

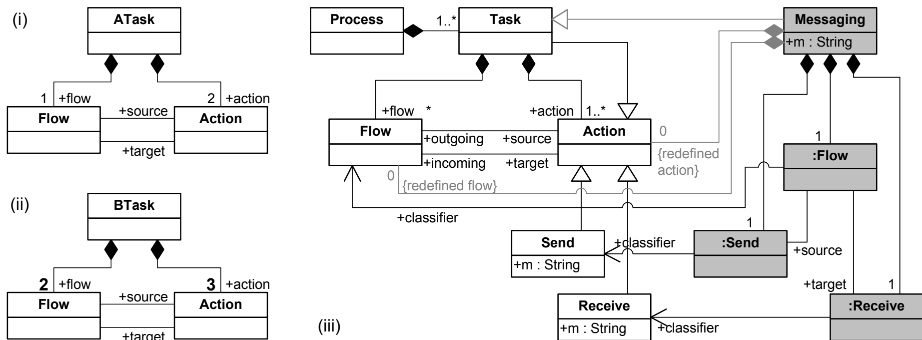

Fig. 3. Composite concepts.

One may be tempted to interpret the metamodel of figure 3(i) at an instance level, such that it represents: a task consisting of two actions that are related by a flow. However, the metamodel of figure 3(i) can only be interpreted at type level, such that

it represents: a task consisting of two actions and a flow between (any) two actions. The difference between both interpretations becomes clearer in case of the metamodel of figure 3(ii), which represents: a task consisting of three actions and two flows, but does not define which actions are related by a flow. Also the metamodel of figure 3(i) does, strictly speaking, not define which actions are related by a flow.

We conclude that a composite aggregation between a structure concept and a constituent concept can be used to represent that an instance of the structure concept contains instances of the constituent concept, where the number of instances is determined by the multiplicity constraint. In addition, associations can be defined between the constituent concepts, but these associations represent associations at type level and can not be used to define associations between instances of the constituent concepts. Consequently, this way of specifying a structure concept does not allow one to define how the constituents of a composite concept are related at instance level.

To represent the instances that a composite concept consists of as well as their associations, we use the notion of *instantiation*. An instantiation represents a particular instance, but at a higher meta-level than the instance itself[2]. This allows one to define a composite concept as a composition of instantiations, which define the instances that should be created upon instantiation of the composite instance.

To represent instantiation, we use the UML metaclass InstanceSpecification, as defined in [10]. An InstanceSpecification represents an instance in a modelled system. Instances of any classifier can be specified, so not only instances of a class but also of an association. Furthermore, values can be specified for the structural features of the instance. Figure 3(iii) depicts the definition of composite concept Messaging, which consists of an instance of a Send action, a Receive action and a Flow, where Send and Receive are defined as specialized actions. An instance specification is expressed using the same notation as its classifier, with the classifier name replaced by the concatenation of the instance name (if any), a colon symbol and the classifier name. Constraints can be added, e.g., to specify that the contents of the message in the anonymous instances :Send and :Receive must be equal to the message specified in Messaging.

3.2 Extended Profiles Package

Figure 4 depicts an Extended Profiles package that supports the extension of metamodels with stereotypes for composite concepts. A composite concept is defined using the metaclasses CompositeStereotype, ConstituentClass(End), ConstituentAssociation(End), ClassInstantiation and AssociationInstantiation. A CompositeStereotype represents the composite concept and inherits from Stereotype to define that it extends an existing metaclass or stereotype. A ConstituentClass represents a composite aggregation between a composite stereotype and an instantiation of one of its constituent classes. An instantiation is defined as a kind of InstanceSpecification. Similarly a ConstituentAssociation represents a composite aggregation between a composite stereotype and an instantiation of one of its constituent associations. A ConstituentAssociation is related to the class instantiations that it will associate.

[2] In Merriam-Webster Online, *instantiate* is defined as “to represent (an abstraction) by a concrete instance”.

The package defines how stereotypes can be defined, but does not enforce one to define how these stereotypes can be used in relation to existing metaclasses and stereotypes. However, both in case of a ‘regular’ stereotype and in case of a composite stereotype this is no issue, since a stereotype is defined as an extension of an existing metaclass, thereby ‘inheriting’ via the extended metaclass the associations that are defined between this metaclass and other metaclasses.

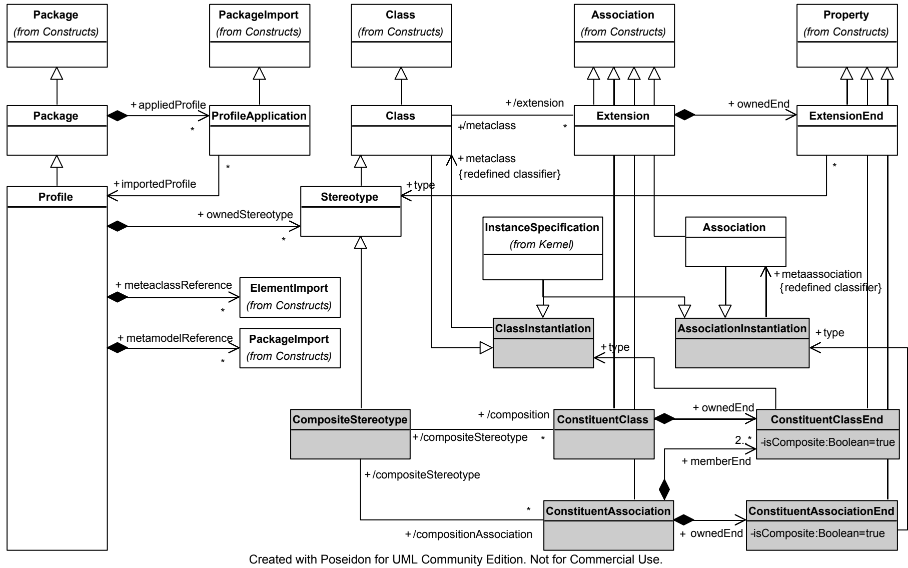

Fig. 4. The classes defined in the Extended Profiles package.

Hence, by extending an existing metaclass, a composite stereotype defines its possible associations with other existing metaclasses implicitly. The possible associations of the stereotype’s constituents and these metaclasses are however not defined in this way. We call these associations the *context relations* of a stereotype. The context relations define how associations between the composite stereotype and other metaclasses must be replaced by associations between the composite’s constituents and other meta-classes. We note, however, that associations between a composite’s constituents and other meta-classes can only exist in the model, after the composite concept is replaced by its constituents. Otherwise, an inconsistent model may be the result. We represent context relations as OCL constraints. Since context relations represent changes to a model, we define the OCL constraints as constraints on operations that define these changes.

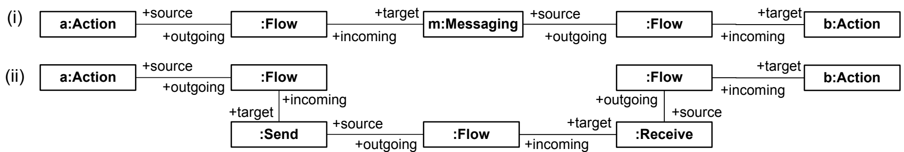

Fig 5. Example object models.

For example, consider the object model in figure 5(i). The context relations between the constituents of messaging task m and actions a and b are not defined by the definition of Messaging in figure 3(iii). We define the following context relations. Each association that relates an incoming flow to an instance of Messaging, must relate that incoming flow to the Send action of that instance instead. Each association that relates an instance of Messaging to an outgoing flow, must relate the flow to the Receive action of that instance instead. In addition, if a messaging concept instance is defined as part of a process or task, its constituents must be added to this process or task instead. Figure 5(ii) depicts the object model that results from replacing object m by the corresponding composition of elementary concept instances. We express the context relation regarding incoming flows in OCL as follows, where operation processContextRelations is assumed to implement the context relations when replacing a Messaging object by its constituents:

```
context Messaging::processContextRelations()
  post initial_actions:
    let incomingflows = self.incoming in
      self.Send.incoming->includesAll(incomingflows)  and
      incomingflows->forAll(f|f.target = self.Send)
```

3.3 Example: Operation Call with Time-Out

As an example we consider the definition of composite concept TimedOperationCall, as introduced in section 1. The activity diagram in figure 6 defines the behaviour of a timed operation call. For brevity, information aspects are not considered, which could be modelled through input and output pins.

When a timed operation call is invoked, actions CallOperationAction and AcceptTimeEventAction are enabled. CallOperationAction represents the transmission of an operation call request to the target object. The receipt of this request is represented by AcceptCallAction, which enables the actual handling of the operation call. ReplyAction represents the returning of the operation result, for which it uses return information produced by the AcceptCallAction. Action AcceptTimeEventAction represents the occurrence of a timeout after some time has expired. In this case an exception is generated through RaiseExceptionAction, which may interrupt the action sequence CallOperationAction, AcceptCallAction and ReplyAction.

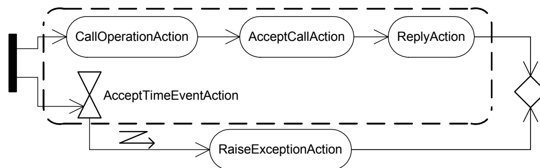

Fig. 6. Activity diagram of a TimedOperationCall.

Figure 7 depicts the metamodel definition of a timed operation call as a composite stereotype. The keywords <<composite>> and <<instantiate>> denote a composite stereotype and an instantiation, respectively. A line between two instantiations denotes the

instantiation of an association between those instantiations. For clarity, a single Constituent association between TimedOperationCall and a grey box is used to represent all ConstituentClass and ConstituentAssociation associations between TimedOperationCall and the instantiations in the box.

Stereotype TimedOperationCall has been defined as an extension of metaclass StructuredActivityNode to define the way in which it can be composed with other metaclasses in the action semantics. Since an StructuredActivityNode is a kind of ActivityNode, it can be connected to other ActivityNodes via ActivityEdges. The context relations for TimedOperationCall are the following. An association that relates an incoming ActivityEdge to the TimedOperationCall must relate that ActivityEdge to the ForkNode that is labelled initial instead. An association that relates an outgoing ActivityEdge to the TimedOperationCall must relate that outgoing ActivityEdge to the MergeNode that is labelled final instead.

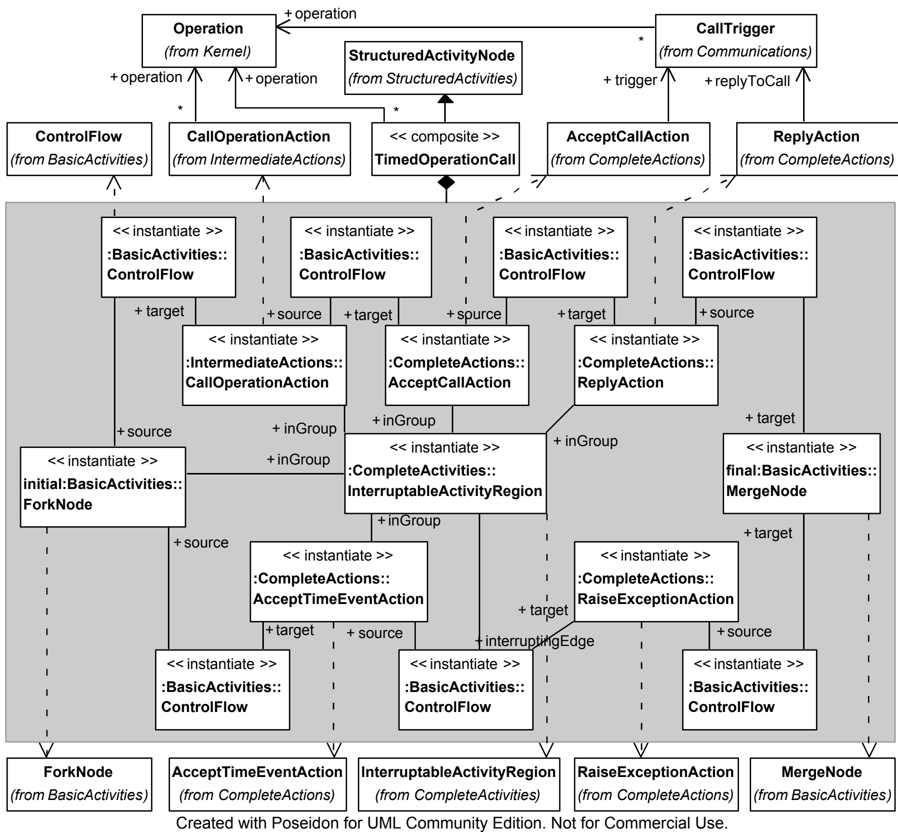

Fig. 7. Stereotype definition of composite concept TimedOperationCall.

Metamodelling. The composite concept of timed operation call can also be defined using metamodelling. We indicate two possible approaches to do this: a *constructive* and a *constraint-oriented* approach. In both approaches a TimedOperationCall is defined as a specialization of a StructuredActivityNode. Both approaches differ however in the

way the composition is defined. The constructive approach defines the composition explicitly in terms of its constituents. This approach resembles the approach followed in section 3.3. The constraint-oriented approach defines the composition by adding OCL constraints to the composite concept, which define the constituents of the composition implicitly. We expect this approach is much more difficult to apply and understand than the constructive approach.

4 Transformation of Composite Concepts

In order to use design techniques that are defined on elementary concepts, such as simulation, analysis and validation, we have to transform each stereotype into the concept or concepts that it consists of. In this way, existing tools can largely be reused and the need for tool modification is minimized. Section 5 presents an example of how tool support can be extended through model transformation.

In this section we focus on the transformation of composite stereotypes. The transformation of 'regular' stereotypes as defined by the UML 2.0 Profile package is rather straightforward. Regular stereotypes can be transformed directly to the metaclasses they extend. We note, however, that in this transformation any specialized design information added by the stereotype is lost. It depends on the particular design technique whether this loss of information is acceptable and existing tools can be reused.

4.1 Transformation Rules

A composite stereotype completely defines how an instance of the corresponding composite concept is composed of instances of existing concepts and associations. In addition, context relations (see section 3.2) define how this composition is embedded in a model that contains the composite concept, i.e., how the constituent concept instances and associations are related to other concept instances in the model. This means that the definition of a composite concept, including its context relations, provide all the information that is required to define rules for transforming its instances to existing concept instances and associations. In principle, these rules can be derived automatically. The following transformation steps are distinguished:

1. creation of the constituent concept instances. For each instance of ConstituentClass, create an instance of the metaclass defined by the instantiation. In addition, each ConstituentClass may define the instance name and attribute values;
2. creation of associations between concept instances. For each instance of ConstituentAssociation, create an instance of the meta-association defined by the instantiation. Relate this instance to the classes to which the ConstituentAssociation is related via ConstituentClassEnd.
3. replacement of associations between the composite concept instance and other concept instances in the model by associations between the constituent concept instances as created in step 1 and the other concept instances. This replacement is defined by the context relations associated with the composite stereotype.

4.2 YATL

We use the transformation language YATL [9] to define transformations, because tool support exists for this language and because it is compliant to the MOF. YATL makes extensive use of the Object Constraint Language (OCL) [11], a language that can be used to describe constraints on how concepts can be used. It can also be used to query a design to verify that a constraint holds on that design or to yield a particular set of concept instances as indicated by the query. Here, we assume the reader is familiar with the basic properties of OCL.

A YATL transformation has a name and consists of a set of transformation rules. These rules are performed in the order in which they are invoked by the rule that is declared the start rule. Each rule has a name, it optionally has a match part and it has a body part. The match part identifies a MOF Class by its name and optionally defines an OCL expression over that concept. The body part of the rule is evaluated over each instance that is selected in the match part. For each execution of the body part self takes the value of one of these instances.

The body part contains a sequence of statements that must be performed. A let statement, let <name>: <classifier name>;, declares a variable by the given name of the type given by the classifier name. An assignment statement, <expr$_1$> := <expr$_2$>, assigns the value of <expr$_2$> to <expr$_1$>. A track statement is used to store and recall a temporary relation between two concept instances. track(<ci$_1$>, <relation name>, <ci$_2$>) stores a relation between the concept instances <ci$_1$> and <ci$_2$> in the relation identified by <relation name>. The relation must be functional, such that each <ci$_1$> can be assigned to at most one other concept instance. track(<ci$_1$>, <relation name>, null) returns the concept instance that is related to concept instance <ci$_1$> by the relation identified by <relation name>. A tracking relation is visible in each rule in an entire transformation. A new statement, new <class name>, creates a new instance of the Class by the specified name.

YATL transformations can be structured by defining them in the context of namespaces. A namespace identifies the Packages that contain the Classes that are the source and the target of the transformation, respectively.

4.3 Example Transformation

As an example we have defined a YATL transformation for composite concept Messaging in figure 3(iii), which transforms a source model into a target model, such that each instance of Messaging in the source model is replaced by its corresponding composition of elementary concept instances in the target model. We assume that the metamodel of figure 3(iii) has been defined in a package named messagingpackage. Furthermore, for convenience, directed composite aggregations have been used.

The following excerpt describes the main transformation rule, which consists of the sequential invocation of 9 other rules. The first 7 rules basically define the copying of concept instances from the source model to the target model, excluding instances from the composite Messaging concept. For brevity, we don't illustrate these rules here, but the complete transformation can be obtained from [16].

```
rule main (){
  pureaction2pureaction();
  sendaction2sendaction();
  receiveaction2receiveaction();
  taskaction2taskaction();
  flowrelation2flowrelation();
  taskcontainment2taskcontainment();
  process2process();
  messaging2basic();
  messagingassociations2basicassociations();
}
```

Rule `messaging2basic` defines the creation of the constituent concept instances and the associations between them. This corresponds to steps 1 and 2 from section 4.1.

```
rule messaging2basic match messagingpackage::Messaging (){
  let dstsend: messagingpackage::Send;
  let dstreceive: messagingpackage::Receive;
  let dstflow: messagingpackage::Flow;
  dstsend := new messagingpackage::Send;
  dstreceive := new messagingpackage::Receive;
  dstflow := new messagingpackage::Flow;
  dstsend.m := self.m;
  dstsend.outgoing := dstsend.outgoing->including(dstflow);
  dstreceive.m := self.m;
  dstreceive.incoming := dstreceive.incoming->including(dstflow);
  dstflow.source := dstsend;
  dstflow.target := dstreceive;
  track(self, tmessage2send, dstsend);
  track(self, tmessage2receive, dstreceive);
  track(self, tmessage2flow, dstflow);
}
```

Finally, rule `messagingassociations2basicassociations()` implements the context relations for Messaging. This corresponds to step 3 from section 4.1. The following code excerpt describes part of the rule, which defines that any incoming flow of a Messaging instance must be an incoming flow for its constituent Send instance. In addition, if a messaging concept instance is defined as part of a process or task, its constituents must be added to this process or task.

```
rule messagingassociations2basicassociation match
    messagingpackage::Messaging (){
  --if messaging is part of process p, its constituents are part of p
  let dstsend: messagingpackage::Send;
  let dstreceive: messagingpackage::Receive;
  let dstflow: messagingpackage::Flow;
  dstsend := track(self, tmessage2send, null);
  dstreceive := track(self, tmessage2receive, null);
  dstflow := track(self, tmessage2flow, null);
  foreach p: messagingpackage::Process in
      Process.allInstances()->select(p| p.task->includes(self)) do {
    let dstp: messagingpackage::Process;
    dstp := track(p, tprocess2process, null);
    dstp.task := dstp.task->including(dstsend);
    dstp.task := dstp.task->including(dstreceive);
  }
  --if messaging is part of a task t, its constituents are part of t
  foreach t: messagingpackage::Task in
      Task.allInstances()->select(t| t.action->includes(self)) do {
    let dstt: messagingpackage::Task;
    dstt := track(t, ttask2task, null);
    dstt.action := dstt.action->including(dstsend);
    dstt.action := dstt.action->including(dstreceive);
    dstt.flow := dstt.flow->including(dstflow);
  }
```

```
--if messaging has an incoming flow f, its send action has f
foreach f: messagingpackage::Flow in self.incoming do {
  let dstsend: messagingpackage::Send;
  dstsend := track(self, tmessage2send, null);
  let inflow: messagingpackage::Flow;
  inflow := new messagingpackage::Flow;
  inflow.target := dstsend;
  dstsend.incoming := dstsend.incoming->including(inflow);
  let srcsourceaction: messagingpackage::Action;
  srcsourceaction := f.source;
  if (f.source.oclIsTypeOf(messagingpackage::Messaging)) then
    let dstsourcemessage: messagingpackage::Messaging;
    dstsourcemessage :=
      track(srcsourcemessage, tmessage2receive, null);
    inflow.source := dstsourcemessage;
    dstsourcemessage.outgoing :=
      dstsourcemessage.outgoing->including(inflow)
  else
    let dstsourceaction: messagingpackage::Action;
    dstsourceaction:=track(srcsourceaction, taction2action, null);
    inflow.source := dstsourceaction;
    dstsourceaction.outgoing :=
      dstsourceaction.outgoing->including(inflow)
  endif;
}
```

5 Example Applications of the Extended Profiles Package

This section further motivates and illustrates the use of the Extended Profiles package by presenting two possible applications: (i) relating modelling languages and (ii) structuring modelling languages.

Relating modelling languages. In earlier work [5], we presented an approach to relate different viewpoints and viewpoint models via a basic viewpoint (see Figure 8). A conceptual model represents the set of concepts that is used in a particular viewpoint and forms the basis for modelling languages that are used to express models (views) of a system as conceived from this viewpoint. The approach is based on the assumption that the concepts from each viewpoint can be considered as extensions of a common set of basic, i.e., elementary and generic, modelling concepts, as represented by the basic viewpoint. Two types of extensions are considered: (i) a viewpoint concept is a specialization of a basic viewpoint concept, or (ii) a viewpoint concept is a composition of (possibly specialized) basic concepts. These assumptions allow one to map different models from the same or different viewpoints onto basic viewpoint models. In this way, relationships between different viewpoint models, e.g., refinement and consistency relationships, can be analysed within the scope of a

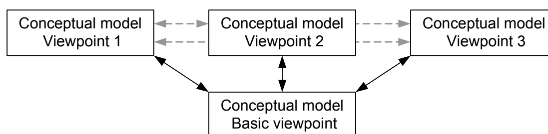

Fig. 8. Relating viewpoints via a basic viewpoint.

single conceptual model and by using the same set of analysis tools. By defining viewpoints as extensions (profiles) of a basic viewpoint, the Extended Profiles package provides a technique to implement the approach described above.

Structuring modelling languages. Another application of the Extended Profiles package is to structure and extend existing modelling languages, using the profiling mechanism. Using the Extended Profiles package we can structure a language into a small set of basic, i.e., elementary and generic, concepts and sets of composite (and specialized) concepts as extensions of those concepts. Having a small set of basic concepts helps to keep a language clear and consistent, while the definition of composite concepts helps us to increase the language's suitability and ease of use for some application domain. Although the existing profiling mechanism already helps us to structure a language in this way, the addition of composite stereotypes extends our possibilities.

An interesting case for this approach is UML, which consists of different languages supporting different modelling viewpoints. As is shown in [6], these languages can be divided into two main categories: structural languages (for class and component diagrams) and behavioural languages (for use case, collaboration, state, activity and sequence diagrams). Furthermore, it is shown that for each of these categories a basic conceptual model can be defined.

We also applied this structure to our behaviour modelling language ISDL [12, 13]. This language has originally been based on a small set of basic concepts [14]. To facilitate a designer in modelling frequently used compositions of ISDL concepts, we are currently introducing shorthand notations to express composite concepts more conveniently. Since each composite concept can be transformed into the basic concepts, we are able to reuse tools that we developed for the basic concepts to support the extended concepts. For example, in this way we have been able to reuse the ISDL simulator for ISDL models that contain instances of composite concepts. The same holds for our technique to assess the conformance between two ISDL models [16].

6 Conclusions

The use of the UML's 2.0 profiling mechanism allows one to combine the efficiency of general purpose languages with the intuitive clarity and ease of use of dedicated languages. Since the profiling mechanism leaves the language metamodel unmodified and introduces stereotypes as extensions of existing metamodel elements, modelling tool support can be reused. This benefit of profiling can be exploited further by allowing one to specify stereotypes for composite concepts, representing (frequently used) compositions of existing concepts. An extension of the UML's Profiles package is presented that supports the specification of composite stereotypes.

At the time of writing, we are not aware of other work that proposes metamodel extension mechanisms for composite concepts, particularly based on the UML profiling mechanism. However, many contributions can be found in literature on classifications of metamodel extension mechanisms and approaches, and on guidelines to use and interpret stereotypes [1,2,3,4,8,15]. This paper is orthogonal to this work and

makes a further contribution by extending the use of stereotypes in a general way. The notion of composite stereotype we introduce can be seen as a restrictive kind of stereotype as described in [4]. Furthermore, this notion is used for type classification as described in [3], since it is meant to introduce new language elements. Although general metamodelling techniques can be used to support the introduction and application of composite concepts, we have extended the UML 2.0 Profiles package because it does not allow a language developer to modify an existing metamodel. But, in principle, this restriction can also be obtained through, or actually is, a restrictive form of metamodelling.

We believe that tool support for the specification of composite stereotypes as described in this paper can be developed rather easily. In addition, we have illustrated how transformation rules can be derived systematically from the specification of a composite stereotype to transform a composite concept instance to the composition of the constituent concept instances it represents. Such a transformation can be used to implement the composite concept using existing tool support.

A question that remains to be resolved is the expressive power of the proposed Extended Profiles package compared to metamodelling. To answer this question, the ideas presented in this paper should be applied to multiple cases from different application areas. In particular, attention should be paid to the systematic definition of the context relationships of a composite stereotype. This future work should lead to a precise set of rules for specifying stereotypes and deriving transformations, which should guarantee both the consistent use of stereotypes by language developers as well as the correct implementation of tool support.

References

1. Atkinson, C. and Kühne, T. Strict Profiles: Why and How. In *Proceedings of <<UML>> 2000*, York, UK, October 2000, pp. 309-322.
2. Atkinson, C., et al. To Meta or Not to Meta – That is the Question. In *Journal of Object Oriented Programming*, Vol. 13, No. 8, December 2000, pp. 32-35.
3. Atkinsion, C. et al. Stereotypical Encounters of the Third Kind. In *Proceedings of <<UML>> 2002*, Dresden, Germany, September 2002, pp. 100-114.
4. Berner, S., et al. A Classification of Stereotypes for Object-Oriented Modeling Languages. In *Proceedings of <<UML>> '99*, Fort Collins, CO, USA, October 1999, pp. 249-264.
5. Dijkman, R.M., et al. An Approach to Relate Viewpoints and Modeling Languages. In *Proceedings of the 7th IEEE Enterprise Distributed Object Computing (EDOC) Conference*, Brisbane, Australia, pp. 14-27, 2003.
6. Evans, A., et al. A unified superstructure for UML. In *Journal of Object Technology*, Vol. 4, No. 1, January-February 2005, pp. 165-181.
7. ISDL. http://isdl.ctit.utwente.nl.
8. Jiang, Y., et al. On the Classification of UML's Meta Model Extension Mechanism. In *Proceedings of <<UML>> 2004*, Lisbon, Portugal, October 2004, pp. 54-68.
9. Patrascoiu, O. YATL: Yet Another Transformation Language. In *Proceedings of the 1st European MDA Workshop, MDA-IA*, pages 83-90. University of Twente, the Netherlands, January 2004.
10. OMG. UML 2.0 Infrastructure Specification. OMG Adopted Specification ptc/03-09-12.
11. OMG. UML 2.0 OCL Specification. OMG Adopted Specification ptc/03-10-14.

12. Quartel, D., et al. Methodological support for service-oriented design with ISDL. In *Proc. of the 2nd Int. Conf. on Service Oriented Computing*, New York City, NY, USA, 2004.
13. Quartel, D. et al. On architectural support for behavior refinement in distributed systems design. *Journal of Integrated Design and Process Science*, 6(1), March 2002.
14. Quartel, D. et al. On the role of basic design concepts in behaviour structuring. In *Computer Networks and ISDN Systems*, No. 29, 1997, pp. 413-436.
15. Schleicher, A. and Westfechtel, B. Beyond Stereotyping: Metamodeling Approaches for the UML. In: *Proceedings of HICSS 34*, 2001, pp. 3051-3060.
16. http://wwwhome.cs.utwente.nl/~dijkman/downloads/messagingtransformation.yatl.