# From Class Diagrams to Zope Products with the Meta-Modelling Tool AToM$^3$

**Andriy Levytskyy and Eugene J.H. Kerckhoffs**
**Delft University of Technology**
**Faculty of Information Technology and Systems, Mediamatica Department**
**Mekelweg 4, 2628 CD Delft, The Netherlands**
**a.levytskyy@cs.tudelft.nl**

**Keywords:** Class Diagrams, Metamodel, CASE Tool, Code Generation, Zope Product

## Abstract

This paper illustrates how meta-modelling and model-transforming can be used to create a highly specialised CASE tool for the homemade so-called Simplified Class Diagrams (SCD), and to provide automated code generation of Zope Products. Products provide a way to extend Zope with custom types of objects tailored to specific applications needs. We informally introduce Simplified Class Diagrams and consequently model them in the Entity Relationship formalism with the meta-modelling tool AToM$^3$ (A Tool for Multi-formalism and Meta-Modelling). Based on this metamodel, AToM$^3$ can generate a completely new SCD modelling tool. Finally, we describe a transformation that generates a Zope product for a given SCD class diagram, and provide an example of such a model-transforming.

## INTRODUCTION

The emergence of the world-wide web (WWW) and its popularity in the simulation community gave birth to the concept of *web-based simulation* [Fishwick 1996], which now includes (among others) activities that deal with the use of the WWW as infrastructure to support distributed simulation execution and encompasses research in tools, environments and frameworks that support the distributed, collaborative design and development of simulation models [Page 1998].

Within this domain, several years ago we started a Collaborative Simulation project in which a generic web environment is developed to support simulation and modelling components in multidisciplinary collaborative projects [Levytskyy et al., 2001]. The environment's functionality is similar to that of the DLR-IMF Virtual Laboratory [Ernst et al., 2003]. The practical application of our prototyped environment lies in the so-called NanoComp project, which investigates computing systems based on quantum devices; therefore the environment is named NanoComp Simulation Environment (NCSE).

NCSE runs under Zope (www.zope.org) and is based on two major types of remote *resources*: conventional tools and models, which are maintained by the collaborative groups that own them [Levytskyy and Kerckhoffs 2001]. The environment provides: (i) an infrastructure that connects remote resources to their respective web-façades (proxy objects accessible from the web) via a distributed object middleware; (ii) centralised control to access remote resources; and (iii) on-line services, such as registration, discovery and processing of resources (i.e. simulation of a registered model with an integrated simulation tool). These web-façades are containers for data describing properties of the remote counterpart tools and models, thus enabling the above-mentioned services. Since 2002, NCSE includes meta-modelling capabilities with the assistance of AToM$^3$ (A Tool for Multi-formalism and Meta-Modelling).

AToM$^3$ is a visual tool for meta-modelling and model-transforming. Meta-modelling refers to modelling formalism concepts at a meta-level, and model-transforming refers to automatic converting, translating or modifying a model of a given formalism into another model of the same or different formalism [Vangheluwe et al., 2002]. This allows AToM$^3$ to be used in the meta-CASE application domain [de Lara and Vangheluwe 2002b]. A meta-CASE tool allows users to specify their engineering method, code generation and produces a CASE-tool supporting that method. Examples of such tools are MetaEdit+ (http://www.metacase.com/) and KOGGE (http://www.uni-koblenz.de/~ist/kogge.en.html).

In NCSE, AToM$^3$ is used as meta-CASE Tool to specify meta-models for various formalisms to be supported by the environment. Given such a metamodel, AToM$^3$ is transformed into a CASE tool for the specified formalisms. Finally, we employ the model-transforming capabilities (a) to generate job descriptions for the NCSE controller and (b) given a formalism's metamodel, to synthesize Python code for the components of the NCSE environment.

This paper illustrates how meta-modelling and model-transforming can be used to create a highly specialised UML-based CASE tool. In contrast to conventional CASE tools, such as Rational Rose tools, Objecteering or Poseidon, that only support selected and rather complex

mainstream methods such as UML, Booch, OMT and OOSE and technologies such as Java, C++, VB, CORBA IDL, this AToM$^3$-generated tool is adapted to our homemade so-called Simplified Class Diagrams (SCD) and provides automated Python code generation for Zope Products.

In the rest of the paper we informally introduce Simplified Class Diagrams that are consequently meta-modelled. Based on this metamodel, AToM$^3$ can generate a completely new SCD modelling tool. The next section describes an SCD-to-ZProduct transformation that generates Python code of a Zope product for a given SCD model, followed by an example of model-transforming. We conclude the paper with final remarks.

## SIMPLIFIED CLASS DIAGRAMS

Our primary goal in this meta-modeling case is to be able to explicitly design the core components used in NCSE, at meta-level. Given our preference for the Object Orientated Framework, we choose to base our design method on the UML class diagrams. UML (Unified Modeling Language) is a common language for creating models of object-oriented systems [Rumbaugh et al., 1999].

We selected and adapted to our needs a subset of UML specifically related to the *Class Diagram* notation, which provides a static overview of a system by showing its classes and relationships among them. The resulting set of modified UML constructs is further on referred to as *Simplified Class Diagrams* (SCD) and consists of the elements Class, Association and Generalization. Figure 1 shows graphical presentations of the SCD constructs.

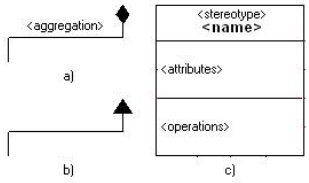


**Figure 1.** SCD notation

Unified Modeling Language is defined by two metamodels that serve different purposes. The UML metamodel is described in UML notation diagrams, OCL constraints, and text, and is intended to convey the concepts and semantics of UML to the human reader. The UML-based UML metamodel is thoroughly described in the semantics chapter of the UML specification [OMG 2001]. Alternatively, the UML metamodel is defined using the Meta-Object Facility (MOF) specified in [OMG 2002]. MOF provides a much narrower and more stringent set of modeling constructs than UML; these constructs serve as a basis for generating APIs, DTDs, repositories, and other computing activities.

We model the SCD constructs after and as close as possible to the (original) UML-based metamodel of UML as specified in the Foundation-Core-Backbone package of [OMG 2001]; however, we stress that SCD is fewer in quantity and simpler in features then UML Class Diagrams.

## METAMODEL

A metamodel of a given formalism specifies the syntax aspect of the formalism by defining the language constructs and how they are built-up in terms of other constructs. To construct an SCD metamodel we used Entity Relationship (ER) diagrams extended with constraints, a default meta-formalism of AToM$^3$. Constraints further restrict how a construct can be connected to another construct to be meaningful, and thus specify static semantics of the formalism.

Each construct is specified with a full *Descriptor* and *Relationships*, constrained with *Constraints* (Well-Formedness Rules in UML), annotated with a *Comment*, and appears according to its *Appearance* (PresentationElement in UML). Since ER does not feature the inheritance mechanism inherent to object-oriented languages, a full descriptor of a construct is obtained by combining the (modified) segment descriptors of the corresponding UML element and that element's ancestors.

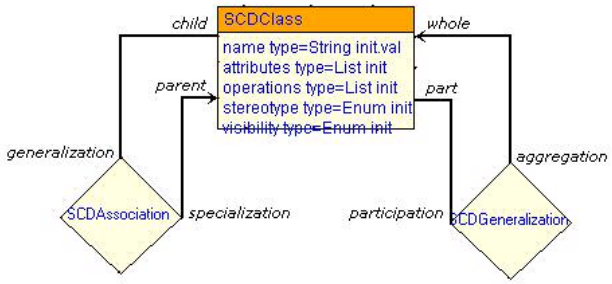


**Figure 2.** SCD metamodel

Figure 2 llustrates how the constructs SCDAssociation, SCDClass and SCDGeneralization build up the SCD metamodel. Please note that Constraints and Appearance are parts of each element in the metamodel and are not visible in the figure. The formal specification of SCD constructs is given below (Constraints and Comments are omitted to save space):

SCDASSOCIATION

*Descriptor*:

*name* is a unique name within the enclosing namespace: String = "Association"

*aggregation* is a specification of aggregation kind: Enum {aggregate, composite, none}

*documentation* is a documentation string: String

*Relationships*:

*whole* designates the "whole" participant of the association; Association-to-whole multiplicity (1:1)

*part* designates the "part" participant of the association; Association-to-part multiplicity (1:1..*)

*Appearance*:

An association is a solid connector with a diamond end pointing to the part containing the whole (Figure 1a); its aggregation kind is denoted with a label at the center of the arrow.

GENERALIZATION

*Descriptor*:

*discriminator* designates a partition to which the Generalization belongs: String = “”

*documentation* is a documentation string: String

*Relationships*:

*parent* designates a generalized version of the child class: Generalization-to-parent multiplicity (1:1)

*child* designates a specialized version of the parent class; Generalization-to-child multiplicity (1:1)

*Appearance*:

A generalization is a solid connector with a triangle pointing to the parent class (see Figure 1b).

SCDCLASS

*Descriptor*:

*name* is a unique name within the enclosing namespace: String = ”Class”

*attributes* is a collection of structural features owned by the Class (to be defined at the lower meta-level): Sequence (SCDAttributeType)

*operations* is a collection of behavioral features owned by the Class (to be defined at the lower meta-level): Sequence (SCDMethodType)

*stereotype* specifies an extension of the Classes: Enum {actor, …, utility}

*visibility* specifies access permission for the Class: Enum {public, protected, private}

*isAbstract* specifies if the Class may not have a direct instance: Boolean

*isActive* specifies if the Class is active or passive: Boolean

*isLeaf* specifies if the Class may not have descendants: Boolean

*isRoot* specifies if the Class may not have ancestors: Boolean

*documentation* is a documentation string: String

*Relationships*:

*aggregation* designates an Association connecting the “whole” element; Class-to-aggregation multiplicity (1:0..*)

*participation* designates an Association connecting the “part” element; Class-to-participation multiplicity (1:0..*)

*generalization* designates a Generalization whose parent is the immediate ancestor of the current class; Class-to-generalization multiplicity (1:0..*)

*specialization* designates a Generalization whose child is the immediate descendant of the current class; Class-to-specialization multiplicity (1:0..*)

*Appearance*:

Class is denoted with a rectangle divided into three horizontal sections containing class’s name in bold face, attributes and operations (see Figure 1c).

Along with the properties defined for each SCD construct, we also extend the global properties for the metamodel itself with regular attributes, such as *title*, *subject*, *description*, *author*, *version* and generative ones: *attributes* and *constraints*. They are used for documentation of models specified in this SCD formalism and for specifying global constraints. All global properties and regular attributes are to be filled-in at the lower meta-level by end-users of the SCD modeling tool to be generated.

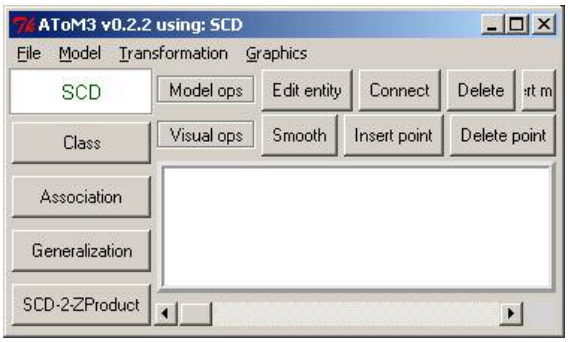


**Figure 3.** Generated SCD modeling tool

Given our metamodel, we can now generate in AToM$^3$ a meta-specification, which, when loaded into the meta-level of AToM$^3$, turns it into a new modeling environment for the designed SCD formalism. A part of this meta-specification is a specification of the User Interface. This specification is a model in its own right and can be edited in AToM$^3$ at any time under a so-called “Buttons” formalism. By default, this specification creates a button for every construct of the formalism. In addition, we created one extra button labeled “SCD-2-ZProduct”, which on click applies the code generation transformation to the model on the tool’s canvas. An instance of the generated SCD modeling tool is shown in Figure 3.

## CODE GENERATION TRANSFORMATION

Model transformation is related to dynamic semantics of a formalism, which defines the meaning of well-formed constructs. This meaning can be described in a number of ways, e.g.: formalism transformation, model optimization, code generation and simulator specification.

This section describes a Python code generation for Zope products. Products provide a way to extend Zope [Pelletier et al.] and in our case NCSE, with custom types of objects tailored to specific applications needs. The described automated product construction is based on the *mxm Easy product* module [Max 2002].

In AToM$^3$ model transformations are specified through Graph Grammars, and consist of *Initial Action*, *Final Action* and *Transformation rules*. Each rule consists of *Left Hand Side* (LHS) and *Right Hand Side* (RHS) graphs, and *Condition*, *Action* and *Priority* properties.

The *Initial Action* of the transformation creates a temporary global attribute *body* to store sequence of signatures of the classes to be generated in the product and iterates through all the elements of the current model (objects on the tool’s canvas) to augment them with temporary attributes: *isVisited* and *isAssociated* help to tell the elements that have been already processed from those

that have not yet. Attribute *isSelected* marks Classes that can be processed next. Attribute *isCurrent* marks a class among the selected ones to process next.

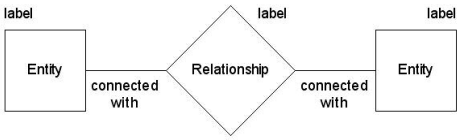


**Figure 4.** Subgraph match pattern

We designed the rules to match the pattern shown in Figure 4, where the entity element is an SCDClass, and the relationship element can be an instance of the other SCD constructs. The rightmost entity can be missing. Present elements are labeled with consequent numbers. In the following we briefly describe each rule:

RULEALLOWEDMETATYPES (priority 1) locates a pair of associated classes as shown in Figure 5 and copies the LHS to the RHS. Its *action* stores a reference to the participant of the association relationship or, if the participant is abstract, references to participant's non-abstract children in attribute *parts* of the current class; finally, it marks the participant as associated. An association can have multiple participants; therefore the rule may be applied more then once.

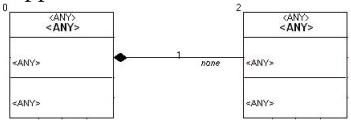


**Figure 5.** LHS graph for rule 1

```
Action
  pre:  LHS.element0.isCurrent = 1
          and LHS.element1.aggregation = #none
            and LHS.element2.isAssociated = 0
  post: RHS.element2.isAssociated = 1
```

RULELOCATEIMMIDIATEPARENT (priority 2) locates an immediate parent of the current class as shown in Figure 6 and copies the LHS to the RHS. Its *action* marks the parent as selected, stores its name in attribute *parents* of element 0 and elements 0 and 1 as visited. A child can have multiple parents, therefore the rule may be applied more then once to the same child and a new parent.

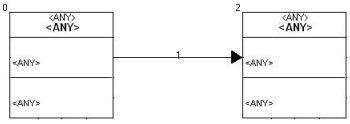


**Figure 6.** LHS graph for rule 2

```
Action
  pre:  LHS.element0.isCurrent = 1
          and LHS.element1.isVisited = 0
  post: RHS.element0.isVisited = 1
          and RHS.element1.isVisited = 1
            and RHS.element2.isSelected = 1
```

RULEMAKECLASSSIGNATURE (priority 3) locates the current class (see Figure 7) and copies the LHS to the RHS. Its *action* makes a signature for the class, adds the signature to the global attribute *body* and marks the class as not current.

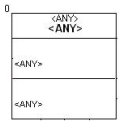


**Figure 7.** LHS graph for rule 3 and 4

```
Action
  pre:  LHS.element0.isCurrent = 1
  post: RHS.element0.isCurrent = 0
```

RULECHOOSENEWCURRENT (priority 4) picks up a class among the selected classes and copies the LHS (see Figure 7) to the RHS. Its *action* marks the class as current, unselects it and adds temporary attributes *parents* and *parts* to it. This rule may return multiple matches and therefore requires parallel execution of the transformation.

```
Action
  pre:  LHS.element0.isSelected = 1
          and LHS.element0.isVisited = 0
  post: RHS.element0.isCurrent = 1
          and RHS.element0.isSelected = 0
```

The *Final Action* creates a blank product structure in the file system, generates proper Python code based on the sequence of signatures in attribute *body*, and saves the code into the created product files. As the last step, it iterates through all the elements on the tool's canvas and removes the temporary attributes.

## MODEL-TRANSFORMING

During execution of a model transformation, AToM$^3$'s Graph Rewriting Processor (GRP) iterates through the list of rules sorted by their priority in an ascending order and tries to apply the current rule to the model. If the rule makes a match, i.e. LHS graph is found and conditions are met, it is executed and the GRP repeats trying each rule again from the beginning of the list. This continues until there are no more rules that can be applied, and then GRP completes the model transformation [de Lara and Vangheluwe 2002a].

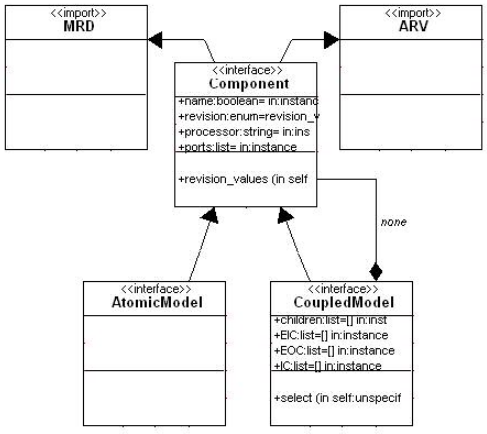


**Figure 8.** A model in SCD formalism

Figure 8 shows a model of the Coupled DEVS component created with the generated SCD modeling tool. The core of this diagram is a concrete class *CoupledModel* and abstract class *Component*. The concrete class *AtomicModel* is needed for its participation in the association between *CoupledModel* and *Component*. Since *Component* is an abstract class and may not have direct instances, that relationship implies an instance of a child of class *Component*, i.e. an instance of either *AtomicModel* or *CoupledModel*. Root classes *MRD* (Metadata for Resource Discovery) and *ARV* (Abstract Resource View) provide features that enable registration, discovery and processing of *CoupledModel* instances under NCSE [Levytskyy and Kerckhoffs 2001].

A model-transforming process in AToM$^3$ can be launched in a variety of ways, e.g. by clicking the "SCD-2-ZProduct" button of the graphical user interface. The transformation begins with class *CoupledModel*, which was selected (graphical attribute) by a modeler on the canvas. In case of this model, the sequence of executed rules is 4, 1, 2, 3, 4, 2, 2, 3, 4, 3, 4 and 3. The result of the model transformation is a valid Python package implementing a Zope product. Figure 9 illustrates core class *CoupledModel* of the synthesized product.

```
class CoupledModel(mxmObjectManager, MRD, ARV):
    """Coupled DEVS component."""

    meta_type = 'CoupledModel'

    _allowed_meta_types =('AtomicModel','CoupledModel')

    _properties = (
        {'type': 'string',    'id': 'name'},
        {'type': 'string',    'id': 'processor'},
        {'type': 'tokens',    'id': 'ports'},
        {'type': 'tokens',    'id': 'children'},
        {'type': 'tokens',    'id': 'EIC'},
        {'type': 'tokens',    'id': 'EOC'},
        {'type': 'tokens',    'id': 'IC'}
        {'type': 'selection', 'id': 'revision', \
         'select_variable': 'revision_values',},
        ) + MRD._properties + ARV._properties

    index_html = HTMLFile('www/index_html', globals())

    def revision_values (self):
        """Return list of DEVS revions."""
        return ['Classic', 'Parallel']

    def select (self):
        """Select an imminent component."""
        pass
```

**Figure 9.** Generated code

At this point a Zope developer can finalize the synthesized product by e.g. specifying an external implementation of the public interface `index_html` or implement internal methods (e.g. `revision_values`) and install it in the product directory of Zope installation. After Zope has been rebooted, a new type of objects, namely Coupled DEVS components, is added to NCSE, which just like the other NCSE components is easily documented, searchable and executable on-line in a standardized way. Client AToM3-generated tools can store abstract specifications of DEVS components in the NCSE Model Base, retrieve components for further reuse, and generate code for execution on one of the tools integrated in NCSE. More details about the environment's registration, discovery and processing services can be found in [Levytskyy et al., 2001].

## FINAL REMARKS

In this paper we have demonstrated how the concepts of meta-modelling and model-transforming are used to create a highly specialised CASE tool for the so-called Simplified Class Diagrams (SCD) and provide automated Python code generation for Zope products. Herein, the meta-modelling tool AToM$^3$ is used as (meta-) CASE tool and a code generator.

The current product construction is based on a minimum product framework and we plan to generalise it further in order to include such aspects as security, management interfaces, product packaging and documentation.

## ACKNOWLEDGEMENT

The research reported in this paper is done in the framework of the NanoComp project, sponsored by TU-Delft.

We would like to thank the Modelling, Simulation and Design Lab (MSDL) of the School of Computer Science of McGill University (Montreal, Canada), and especially Hans Vangheluwe and Juan de Lara, for providing and helping us with AToM$^3$.

## REFERENCES

de Lara, J. and Vangheluwe, H. (2002) AToM3: A Tool for Multi-Formalism Modelling and Meta-Modelling. In European Conferences on Theory And Practice of Software Engineering ETAPS02, Fundamental Approaches to Software Engineering (FASE). Lecture Notes in Computer Science 2306, pp.: 174 - 188. Springer-Verlag.

de Lara, J., and Vangheluwe, H. 2002. "Using AToM3 as a Meta-CASE tool." In Proceedings of 4th International Conference on Enterprise Information Systems, ICEIS'02, (Ciudad Real, Spain, April 2002). 642-649.

Ernst, T., Rother, T., Schreier, F., Wauer, J., and Balzer, W., 2003, "DLR's VirtualLab: Scientific Software Just a Mouse Click Away", Computing in Science & Engineering magazine, vol. 5, no. 1, Jan./Feb.: 70-79

Fishwick, P.A. 1996. "Web-Based Simulation." In Proceedings of the 1996 Winter Simulation Conference. 772 – 779.

Levytskyy, A., and Kerckhoffs, E.J.H. 2001. "Integration of Simulation Tools and Models in a Collaborative Environment". In Proceedings of 2001 European Simulation Interoperability Workshop, EuroSIW01 (London, United Kingdom, June 25-27), Simulation Interoperability Standards Organisation, 407-415.

Levytskyy, A., Kerckhoffs E.J.H., and Vangheluwe, H. 2001. "Sharing Simulation Models and Tools within a

Collaborative Research Project." In Proceedings of 15th European Simulation Multiconference (Prague, Czech Republic, June 6-9). SCS, 578-584.

Max, M. 2002. An easier way to write products: www.zope.org/Members/maxm/HowTo/easyProduct

OMG. 2001. "Unified Modeling Language Specification, version 1.4." Object Modeling Group. September 2001.

OMG. 2002. "MetaObject Facility (MOF) Specification, version 1.4." Object Modeling Group. April 2002.

Page, E.H. 1998. "The rise of Web-based simulation: implications for the high level architecture." In Proceedings of 1998 conference on Winter simulation (Washington, D.C., United States). 1663 – 1668.

Pelletier, M., Latteier, A., and McDonough, C. "The Zope Developer's Guide. Zope 2.4 edition": www.zope.org/Documentation/Books/ZDG/current/

Rumbaugh, J., Jacobson, I., and Booch, G. 1999. *The Unified Modeling Language Reference Manual*. Addison-Wesley.

Vangheluwe, H., de Lara, J., and Mosterman, P.J. 2002. "An introduction to multi-paradigm modelling and simulation." In Proceedings of the 2002 AI, Simulation and Planning in High Autonomy Systems Conference, AIS'2002, (Lisboa, Portugal, April 2002). 9-20.

## AUTHOR BIOGRAPHIES

**Andriy Levytskyy** graduated from Chernivtsi State University, Ukraine and holds an MSc-degree in Computer Science. Currently, he is a PhD student at Delft University of Technology, Faculty "Information Technology and Systems", Department "Mediamatica", Group "Knowledge-based Systems".

**Eugene J.H. Kerckhoffs** holds an MSc-degree from Delft University of Technology (1970, Physical Engineering, thesis on analogue and hybrid computer simulation) and a PhD-degree from the University of Ghent (1986, Computer Science, thesis on parallel continuous simulation). Currently, he is an associate professor at Delft University of Technology (Faculty "Information Technology and Systems", Department "Mediamatica", Group "Knowledge-based Systems"). He was also chairholder of the SCS Chair in Simulation Sciences at the University of Ghent, Belgium.