University of Kent

Computing Laboratory

# Model Driven Language Engineering

Octavian Patrascoiu

March 2005

A thesis submitted to The University of Kent for the degree of Doctor of Philosophy

# Abstract

Modeling is a most important exercise in software engineering and development and one of the current practices is object-oriented (OO) modeling. The Object Management Group (OMG) has defined a standard object-oriented modeling language – the Unified Modeling Language (UML). The OMG is not only interested in modeling languages; its primary aim is to enable easy integration of software systems and components using vendor-neutral technologies. This thesis investigates the possibilities for designing and implementing modeling frameworks and transformation languages that operate on models and to explore the validation of source and target models. Specifically, we will focus on OO models used in OMG's Model Driven Architecture (MDA), which can be expressed in terms of UML terms (e.g. classes and associations).

The thesis presents the Kent Modeling Framework (KMF), a modeling framework that we developed, and describes how this framework can be used to generate a modeling tool from a model. It then proceeds to describe the customization of the generated code, in particular the definition of methods that allows a rapid and repeatable instantiation of a model. Model validation should include not only checking the well-formedness using OCL constraints, but also the evaluation of model quality.

Software metrics are useful means for evaluating the quality of both software development processes and software products. As models are used to drive the entire software development process it is unlikely that high quality software will be obtained using low quality models. The thesis presents a methodology supported by KMF that uses the UML specification to compute the design metrics at an early stage of software development.

The thesis presents a transformation language called YATL (Yet Another Transformation Language), which was designed and implemented to support the features provided by OMG's Request For Proposal and the future QVT standard. YATL is a hybrid language (a

mix of declarative and imperative constructions) designed to answer the Query/Views/Transformations Request For Proposals issued by OMG and to express model transformations as required by the Model Driven Architecture (MDA) approach.

Several examples of model transformations, which have been implemented using YATL and the support provided by KMF, are presented. These experiments investigate different knowledge areas as programming languages, visual diagrams and distributed systems. YATL was used to implement the following transformations:

- UML to Java mapping
- Spider diagrams to OCL mapping
- EDOC to Web Services

# Acknowledgments

This work would not have been possible without the constant dedication, guidance and advice of my supervisors, Stuart Kent and Peter Rodgers, to whom I am deeply grateful. Many thanks for their support and invaluable advice throughout the duration of the research and writing of this thesis.

I am grateful to the members of the Supervisory Panel, Peter Linington and David Shrimpton for their helpful comments and advice. Especially, my thanks to Peter for showing me how to form a coherent argument from my initial collection of ideas.

Many thanks to the academic staff of the Systems Engineering Research Group – David Akehurst, Eerke Boiten, Ana Cavalcanti, Nigel Dalgliesh, Rogerio de Lemos, and Jim Woodcook – who have provided an atmosphere of great intellectual stimulation, a pleasant and comfortable working environment. Many thanks to David Barnes and Tim Hopkins for their comments regarding the UML models measuring.

Thanks to both my parents Elena and Dumitru Patrascoiu for their support, encouragement and tutelage during my early years, without which I would have never got to the point of being able to attempt a doctoral degree. Thanks to both my parents-in-law Maria and Nicolae Mitroi for supporting me in the early years of my academic career.

Finally, thanks to my family – Giana and Virgil Patrascoiu – for their tolerance and patience during the final period of writing up.

The work presented in this thesis was partly funded by:

- *The UK Engineering and Physical Sciences Research Council (EPSRC)* as part of the Reasoning with Diagrams project (no. GR/R63509/01).
- *International Business Machines (IBM)* as part of IBM Faculty Partnership Award No. 220 20858.

# Contents

# List of Figures

# List of Tables

# Chapter 1. INTRODUCTION

The development of software requires an adequate description of the problem domain. Involved in the development of such a description are not only software engineers, but also users and domain experts. The members of such teams must communicate with each other using documents. The aim is to provide a representation of an application domain that is understandable for all persons involved in the software engineering process.

This representation, called a model, shows only the essential parts of the planned system. As models are intended to be used during the entire software development process, implementation details should be supported, too. To achieve this, a suitable modeling language is required. Such a language must be easy to understand and support a certain level of abstraction and formalization. For instance, programming languages are not suitable because they are implementation-oriented. Furthermore, not all team members, especially domain experts, easily understand programming languages. On the other side, natural language is not an alternative because it is ambiguous. Therefore, unambiguous languages with a certain level of abstraction are required.

It has been shown that visual modeling languages can be used successfully to achieve the above aims. A modeling language should contain not only diagrammatic components but also textual notation. This combination increases the expressiveness of modeling languages. The aim is to add support for both a visual and a textual description of a problem domain. The diagrammatic representation can be used to describe the visual information while the textual representation can be used to augment the visual information with written information. The augmentation can be used for different purposes such as providing comments, indicating further details or adding a formal description to a visual description.

Generating new models is relatively easy. But over time, responding to ever changing requirements gets more and more difficult. Hence, tools for model processing are required. Such tools include text editors, pretty printers, type checkers, diagram editors, parsers,

evaluators, simulators, execution engines and so on. On the other hand, changing the model implies changing the software. Every time a model is changed the software must be changed. The aim is to develop a system architecture solid enough to allow reliable code development. Automated code generation leads to solid code faster as long as the code generators are thoroughly tested. The aim is to automate the generation of code starting from a given model. The generation of design-level code for an application greatly increases both the quality of the components and the speed of their availability. For example, the code for model tools can be automatically generated.

## 1.1. Model Driven Engineering

Modeling is one of the foundations of software engineering and development and one of the current practices is object-oriented (OO) modeling. The Object Management Group (OMG) has defined a standard object-oriented modeling language – the Unified Modeling Language (UML).

The OMG is not only interested in modeling languages; its primary aim is to enable easy integration of software systems and components using vendor-neutral technologies. The last step towards this goal is its announcement of the Model Driven Architecture (MDA) as the basis for future OMG standards.

The quality of abstract descriptions is vital for MDA as it provides the possibility of generating software from abstract descriptions. While the current OMG standards such as UML and MOF provide a well-established foundation for defining OO models, no such foundation exists for describing transformations between models. The process of transformation between language models is based on a large body of research in the field of compilation. The OMG's recently initiated standardization process called Queries/Views/ Transformations will provide also the missing link of MDA: the transformation language.

The aim of this thesis is to investigate the possibilities for designing and implementing transformation languages that operate on models and to explore the validation of source and target models. Specifically, we will focus on OO models used in MDA, which can be expressed in terms of MOF/UML concepts (e.g. classes and associations). We think that model validation should include not only checking the well-formedness using OCL

constraints, but also the evaluation of model quality. As models are used in MDA to drive the entire software development process it is unlikely that high quality software can be obtained using invalid or low quality models. Evaluation of the quality of UML models at early stages of the software development process should reduce the overall cost of the software development process.

## 1.2. Objectives

The main objectives of this thesis are:

1) To investigate efficient and usable techniques for specifying transformations from a source UML model instance to a target UML model.
2) To illustrate whether or not this style of specification can be used to provide a transformation engine implementation that can be, at least partially, automated.
3) To investigate the validation of OO models by checking the OCL constraints on source and target model instances, and evaluating the quality of the source and target models using software metrics.

## 1.3. Thesis overview

To achieve the objectives described in 1.2 the thesis follows the following format:

Chapter 2 Background: this chapter starts by discussing the OMG's Model Driven Architecture (MDA), presenting the main features of the framework for software development. It also describes other work related to the area of language translation. It includes an overview of topics that support the understanding of the research presented in the following chapters. The last section presents a description of some object-oriented programming patterns used as part of the concepts, techniques, and tools proposed in this thesis.

Chapter 3 Kent Modeling Framework: this chapter starts with the presentation of the requirements for a modeling framework. Then it describes the modeling framework that we developed (*Kent Modeling Framework*) and how this framework can be used to generate a modeling tool from a model. It then

proceeds to describe the customization of the generated code, in particular the definition of methods that allows a rapid and repeatable instantiation of a model.

Chapter 4 Model Quality Measuring: this highlights a methodology that uses the UML specification to compute the design metrics at an early stage of software development. The first section gives a brief description of the background, object-oriented metrics, and problems of the measuring UML models using software metrics. The second section describes our set of metrics and algorithms. The third section describes the measuring problem for UML models and describes the methodology that we have used. The fourth section gives an example. The last two sections contain an overview of the related work, and the conclusions and future work.

Chapter 5 YATL Specification: this chapter presents the current version of YATL (Yet Another Transformation Language), which was designed and implemented to support the features provided by OMG's Request For Proposal and the future QVT standard. The first subsection provides a quick overview of the YATL language. Subsequent sections present the features of YATL in more details.

Chapter 6 Model Transformations in YATL: this chapter describes three examples of model transformations, which have been implemented using YATL and the Kent Modeling Framework. The three examples are:

- UML to Java mapping
- Spider diagrams to OCL mapping
- EDOC to Web Services

Chapter 7 Conclusions: highlights the contribution of the work presented in this thesis, showing how transformation specification techniques and the implementation approaches meet the objectives outlined in Chapter 1. This chapter also proposes some future research that could lead on from the results of the work presented here.

## 1.4. Contribution

This thesis defines a modeling framework that caters for the specification of OO model validation and model transformations in the context of OMG's MDA. The argument of the thesis is novel, in that current systems and frameworks do not provide adequate support for software development using OMG's MDA concepts. Furthermore, very few frameworks make use of the concepts specific to MDA for supporting the specification and development of large scale software systems. The transformation framework described in this thesis is a contribution to forming an adequate basis for supporting MDA software development. We already made some steps in this direction by providing validation support for UML models [ALP03] [AP03][OCL2P].

The modeling framework proposed in this thesis allows UML model instances to be validated before transformation takes place. This is important as models are the driving concepts in MDA and we are unlikely to obtain high-quality software from incorrect model instances or models that were designed poorly. In the classic approach UML models validation is performed by checking well-formedness rules described using OCL constraints. This approach fails to cover other aspects regarding the UML models such as the quality of the design and the effort required to understand and maintain a model. This thesis proposes a set of design metrics that can be used to evaluate the quality of UML models from a design perspective.

Although the major contribution of this thesis lies in the definition and specification of a transformation language called Yet Another Transformation Language (YATL), we also propose a modeling framework that supports, among other features, model transformations. We also present tools that have been implemented to support the modeling process. Modeling activities such as:

- Java and C# code generation to instantiate UML models
- Model persistence using XMI,
- UML model instance validation by checking OCL constraints
- UML model instance validation using design metrics

are supported by the KMF-Studio tool. Code generation is performed in KMF-Studio using an original template language called X Template Language (XTL) for which language

processors are implemented. The proposed transformation framework and language is implemented by YATL-Studio tool, which uses the code generated by KMF-Studio to implement YATL transformations.

To summarize, the contributions are presented below:

- Development of KMF (Kent Modeling Framework), a modeling framework that provides support for software development using MDA techniques. The main characteristics of KMF are:
  - All the modeling features described in this thesis (e.g. code generation, creating model instances, OCL validation, quality evaluation, and transformation support) are integrated in KMF.
  - Code generation is performed using templates described in XTL, a template language that was designed and implemented to provide code for flexible code generation in the KMF.
  - The OCL support is highly portable as it is structured using OO programming patterns such as adapter, bridge, visitor and observer. As a consequence of this approach, the initial implementation of OCL support in KMF was easily ported to IBM's EMF.
  - As KMF is using MDA concepts to develop software, it allows the integration of applications at the metadata and model levels.
- Designing of a transformation language called YATL that provides the missing link in the OMG's MDA framework. This is vital as transformations are key concepts in MDA. The main characteristics of YATL are:
  - YATL is a rule-based transformation language and structured in OO style using namespaces. A YATL transformation rule consists of two parts: a left-hand side (LHS) and a right-hand side (RHS). The LHS accesses the source model, whereas the RHS expands in the target model.
  - The LHS of a YATL transformation is specified using a filtering expression written either in OCL or native code such as Java, C#, and scripts. This approach allows filter expressions to include both modeling information (e.g. navigational expressions, property values, collections) and platform dependent properties (e.g. special conversion functions), which makes them extremely powerful.
  - The RHS of a YATL transformation rule is specified using a procedural approach (e.g. decision and iteration actions and new/delete syntactic constructs).
  - YATL supports a mechanism to store and retrieve source to target mappings using *track* actions. *Native* actions support interaction with the host platform. To provide deterministic behavior and flexibility, YATL rules are invoked explicitly using their names and providing the required arguments.
  - YATL is implemented both as a compiler and an interpreter to provide support both for static and dynamic model transformations.

- To test YATL's descriptive power and its expressiveness we performed several transformations. YATL was used to experiment with transformations between various models, from different knowledge domains (e.g. spider diagrams to OCL and UML's profile Enterprise Distributed Object Computing to Web Services). The experiments have shown that YATL is simple, easy to learn and use, and can be used to described transformations from various knowledge domains. The experiments also proved that the transformation engine that supports YATL is very efficient.
- KMF proposes two approaches to validate the source and target models involved in a transformation. The first approach uses the OCL support to check if the OCL constraints attached to the source and target model instance are satisfied, thus checking the well-formedness of models. The second approach provides the evaluation of the quality of the source and target model using a set of software metrics. This thesis proposes a framework to evaluate the quality of UML models and a set of design metrics to evaluate the maintainability of UML models. The validation of models is vital as in the OMG's MDA framework, software development process is driven by models.

## 1.5. Summary of publications

The work presented in Chapter 3 and Chapter 4 proposes a modeling framework that supports the software development process using OMG's MDA approach. The framework does not only supports classic modeling activities such as code generation, model element instantiation, storage and persitance through XMI, but also model validation using OCL constraints and design metrics to evaluate the model quality. The results of the investigations have been published in the following papers:

[ALP03] Akehurst, D., Linington, P., and Patrascoiu, O. (2003) OCL 2.0- Implementing the Standard. Technical Report No. 12-03, Computer laboratory, University of Kent, UK.

[AP03] Akehurst, D. and Patrascoiu, O. (2003). OCL 2.0 – Implementing the Standard for Multiple Metamodels. In OCL2.0-"Industry standard or scientific playground?" - Proceedings of the UML'03 workshop, page 19. Electronic Notes in Theoretical Computer Science.

[AP04a] Akehurst, D. and Patrascoiu, O. (2004). Prototyping Metamodels: Automated Generation of Modeling Tools with support for Checking Well-Formedness Constraints. Submitted to UML 2004.

[Pat02a] Patrascoiu, O. (2002) A quality model for Java programs maintenance. In Else Software Journal, University of Craiova.

[Pat02b] Patrascoiu, O. (2002) Software systems quality. In Else Software Journal, University of Craiova.

The work presented in Chapter 5 and Chapter 6 proposes a technique for model transformation and presents several experiments that were performed using this technique. The results have been published in the following papers:

[AKP03] Akehurst, D., Kent, S., and Patrascoiu, O. (2003). A relational approach to defining and implementing transformations between metamodels. In *Journal of Software and Systems Modeling* (SoSym), 2(4), 215-239.

[Pat04a] Patrascoiu, O. (2004) YATL:Yet Another Transformation Language. In *Proc. of First European Workshop MDA-IA*, University of Twente, the Nederlands.

[Pat04b] Patrascoiu, O. (2004) YATL:Yet Another Transformation Language. Reference Manual. Version 1.0. Technical Report 2-04, University of Kent, UK.

[Pat04c] Patrascoiu, O. (2004) Model transformations in YATL. Studies and Experiments. Technical Report 3-04, University of Kent, UK.

[Pat04d] Patrascoiu, O. (2004) Mapping EDOC to Web Services using YATL. In Proc. of 8th IEE International Enterprise Distributed Object Computing Conference, EDOC 2004.

[PR04] Patrascoiu, O. and Rodgers, P. (2004). Embedding OCL expressions in YATL. In Proc. of "OCL and Model Driven Engineering" workshop, UML 2004.

[PR05] Patrascoiu, O. and Rodgers, P. (2005). Model transformations in YATL. Submitted to Journal of Software and Sytems Modeling, January 2005.

# Chapter 2. BACKGROUND

This chapter starts by discussing the OMG's Model Driven Architecture (MDA), presenting the main features of the framework for software development. This chapter then describes other work related to the area of language translation. It also includes an overview of topics that support the understanding of the research presented in the following chapters.

The first section discusses the MDA, presenting the main features of the OMG's initiative. The second section presents the theoretical and practical aspects of the translation process. The last section presents a description of some object-oriented programming patterns used as part of the concepts, techniques, and tools proposed in this thesis.

## 2.1. Unified Modeling Language

Modeling is a principal exercise in software engineering and development and one of the current practices is object-oriented (OO) modeling. In 1996, the Object Management Group (OMG), an international consortium of computer vendors, end users and consultants, adopted the well-known Unified Modeling Language (UML). UML has since become far-and-away the dominant standard for software modeling. Today, nearly every software development tool has incorporated some form of UML-style modeling into its development process, and the number of commercially available UML tools is growing.

Based on the success of UML, the OMG has subsequently developed a number of other broad-based software industry standards around UML, including the Meta Object Facility (MOF), used primarily to manage metadata and integrate tools; the Common Warehouse Model (CWM), used primarily in data warehousing; the XML Metadata Interchange (XMI), used in mapping MOF to XML; and the Enterprise Distributed Object Computing (EDOC) standard, used for the modeling of enterprise computing.

UML has evolved since 1996 in successive versions. There is ongoing work on finalizing the latest version, UML 2.0. UML 2.0 is divided in several parts:

- UML 2.0 Superstructure: The superstructure defines the six structure diagrams, three behavior diagrams, four interaction diagrams, and the elements that comprise them, and so is the part of the language that you'll encounter
- UML 2.0 Infrastructure: The infrastructure defines base classes that form the foundation not only for the UML 2.0 superstructure, but also for MOF 2.0.
- UML 2.0 Object Constraint Language (OCL): This allows setting of pre- and post-conditions, invariants, and other conditions.
- UML 2.0 Diagram Interchange: This specification extends the UML metamodel with a supplementary package for graph-oriented information, allowing models to be exchanged or stored/retrieved and then displayed as they were originally.

The OMG also has begun to develop derivative standards for specific business domains (e.g. real-time, healthcare, financial services, telecom, transportation, manufacturing) by defining the following UML Profiles:

- UML Profile for CORBA
- UML Profile for CORBA Component Model (CCM)
- UML Profile for Enterprise Application Integration (EAI)
- UML Profile for Enterprise Distributed Computing (EDOC)
- UML Profile for OoS and Fault Tolerance
- UML Profile for Schedulability Performance, and Time
- UML Testing Profile.

and one related specification:

- UML Human-Usable Textual Notation (HUTN)

## 2.2. Model Driven Architecture

The Object Management Group (OMG) was formed with the declared purpose of accelerating the introduction of standardized object software. The Object Request Broker was one of the first important standards. Two other standards, the Object Management Architecture (OMA) and the Common Object Request Broker Architecture (CORBA), were designed to provide the standard framework for distributed systems. This framework is in the same spirit as the OSI Reference Model and the Reference Model of Open Distributed Procession (RM-ODP or ODP).

To keep up with its expanding focus, in 2001 OMG adopted a second framework, the Model Driven Architecture (MDA). MDA is not, like the OMA and CORBA, a framework for implementing distributed systems. It is an approach to using models in software development. It is based on other standards including MOF, UML, XMI, and CWM.

[MDA] introduces a number of concepts used by the OMG's MDA initiative. The definitions of these concepts are presented below.

| | |
|---|---|
| System | A system is a collection of elements and a set of relations between elements. An element can be anything. For example: a program, a computer, a network of computers, a human or an enterprise. |
| Model | A model is a description of a system and its environment. A model can be described using a modeling language or a textual language. |
| Viewpoint | Is an abstraction of a system using a set of architectural concepts and structuring rules. |
| View | A view is a representation of a system using a chosen viewpoint. |
| Computation Independent Model | The computation independent viewpoint (CIV) focuses on the requirements of a system and its environment. A computation independent model (CIM) of a system describes the domain and requirements of the system. A CIM might consist of a model that captures information about the data of a system. |
| Platform Independent Model | The platform independent viewpoint (PIV) focuses on the operation of the system discarding the details specific to a given platform. A platform independent model (PIM) is a description of a system from the platform independent viewpoint. |
| Platform Specific Model | The platform specific viewpoint (PSV) combines the platform independent viewpoint with details specific to the implementation on a specific platform. A platform specific model (PSM) is a description of a system from the platform specific viewpoint. |
| Model Transformation | A model transformation is the process of transforming a model of a given system into another model of the same system. |

Typically the process of software development using OMG's MDA approach is performed in several steps, described below.

**Requirements specification**. The requirements for the system are described using modeling languages that are computation independent. The resultant model, sometimes called the domain model or business model, describes the system and its interaction with the environment in which it operates. A CIM might be described using UML and additional information regarding the viewpoints used to describe the system.

**Platform modeling.** The architect will then choose a platform model that allows the implementation of the system with the desired architectural features. Usually, this model is described in software and hardware manuals and is based on the architect's experience.

**PIM modeling**. Starting from the CIM a PIM model is built. This model describes the system discarding the details specific to the platform on which it will be implemented.

**PSM modeling.** The mapping from PIM to PSM describes the transformation of PIM into PSM for a given platform. The platform model is used to determine the exact form of the transformation. The resulting PSM specifies the same system as PIM and describes how the model uses the platform.

**Generate deployable code.** To produce an implementation of the system, deployable code is generated starting from resulting PSM. Deployable code can be generated directly from PIM, without producing a PSM. This approach has the benefit of being more efficient. In some cases, using a direct code generation, could affect drastically the efficiency of further stages (e.g. debugging). Unless the PIM and the platform are close, the development of an intermediate PSM is recommended.

The MDA approach promises a number of benefits [MDA][CH03]:

- Improved portability due to separating the application knowledge from the mapping to a specific implementation technology.
- Increased productivity due to automating the mapping.
- Improved quality due to reuse of well-proven patterns and best practices.
- Improved maintainability due to better separation of concerns.

- Enables different applications to be integrated by explicitly relating their models: this facilitates integration and interoperability and supports system evolution as platform technologies change.

While the current OMG standards such as UML and MOF provide a well-established foundation for defining PIMs and PSMs, no such well-established foundation exists for transforming PIMs to PSMs [GLRSW02]. In 2002, in its effort to define the transformations, OMG initiated a standardization process by issuing a Request for Proposal (RFP) on Query / Views / Transformations (QVT) [QVT02]. This process will lead to an OMG standard for defining model transformations, which will be of interest not only for PIM-to-PSM transformations, but also for defining views on models and synchronization between models. Driven by practical needs and the OMG's request, a large number of approaches to model transformation have been recently proposed [CH03].

## 2.3. Modeling frameworks

Many current UML CASE-tools, both commercial (e.g.Rational Rose [RAT], Together [TOG], Poseidon [GEN]) and non-commercial (e.g. ArgoUML [ARG]) offer extensibility and interoperability capabilities, for example by providing a proprietary API for model repository access, by introducing a scripting language, or by providing libraries for tool developers. However, these CASE-tool dependent solutions are not generally well-suited for performing a chain of transactions or queries on the models. One of the main goals of modeling frameworks and tools is to support the combining of small model operations to achieve higher-level functionality, customizable for a given process, domain, or a platform. Of the existing UML model processing platforms, the IBM's Eclipse Modeling Framework (EMF), the NetBeans Metadata Repository (MDR), and FUJABA [FUJ] come close to our approach. In comparison, the system described in this thesis, KMF, supports model validation, using OCL and design metrics, and model transformations using a transformation language called YATL (Yet Another Transformation Language).

### 2.3.1. Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) is an open source framework targeting model-driven architecture development. The Eclipse Modeling Framework unifies Java, XML and UML-enabling developers to rapidly construct robust applications based on simple models. It can be used for both modeling and code generation. It creates Java code for graphically editing, manipulating, reading, and serializing data based on a model specified in XML Schema, UML, or annotated Java. EMF is the basis for many of the tools within IBM's WebSphere Studio and Eclipse projects.

In addition to generating Java code, EMF can also generate Eclipse plug-ins and graphical, customizable editors. EMF keeps the code synchronized with the model. The EMF-generated code supports the standard create, retrieve, update, and delete operations, and it also supports cardinality constraints, complex relationships and inheritance structures, containment definitions, and a suite of attribute descriptions. The generated code provides notification, referential integrity, and customizable persistence to XMI.

EMF incorporates several of the MDA concepts and standards. Behind both EMF and MDA is the key concept of using models as input to development and integration tools, transforming those models into executable implementations. In terms of the MDA standards, EMF uses XMI as primary serialization format for the models and meta-models. EMF's meta-model, called Ecore, roughly corresponds to the EMOF (Essential MOF) subset of the recently accepted MOF 2.0 standard. EMF also provides tools for transforming model forms like UML, XML Schema and simple annotated Java interfaces into Ecore and powerful code generator tools, which are used to produce high-quality Java code from Ecore model descriptions.

The project is implemented in Java and based on the Eclipse platform. To integrate EMF's various modules Eclipse's plug-in mechanism is used. For example, the basic code generation components have no Ecore modeling dependency, which makes them ripe for reuse in other code generation applications. In fact, EMF is modularized in such a way that many parts of it can even be used without Eclipse itself.

### 2.3.2. Metadata Repository

The Metadata Repository (MDR) implements the OMG's MOF standard based metadata repository and integrates it into the NetBeans Tools Platform. It contains an implementation of MOF repository, including persistent storage mechanism for storing the metadata. The interface of the MOF repository is based on and fully compliant with JMI (Java Metadata Interface). MDR also defines additional features that help to incorporate it into the IDE (e.g. its event notification mechanism).

MDR has the following features:

- Ability to save the contents of any package into an XMI 1.2 document.
- Generate Java APIs for accessing metadata described by the specified MOF metamodel.
- MOF metamodels loaded into the MDR can be instantiated.
- A metamodel can be accessed using both reflective and metamodel specific APIs.
- The generated APIs are implemented automatically during the MDR run-time as they are needed.
- MDR can work as a standalone application by using a command line access.
- Part of MDR is integrated in NetBeans by exposing the repository contents and actions that can be performed on the repository.

### 2.3.3. Fujaba

The primary topic of the Fujaba Tool Suite project is to provide an easy way to extend a UML and Java development platform with the ability to add plug-ins. The Fujaba Tool Suite combines UML class diagrams and UML behavior diagrams to create a powerful, easy to use, yet formal system design and specification language. Furthermore the Fujaba Tool Suite supports the generation of Java source code from the UML model. The result of the code generation could be an executable prototype, if the model contains all the relevant information. It provides also, to some extent, reverse engineering. The Fujaba Tool Suite is configured with plug-ins for Reverse Engineering and Design Pattern recognition. The Fujaba Tool Suite project is located at the Software Engineering Group, Computer Science Department at the University of Paderborn.

The Pattern Specification plug-in provides a graphical editor for the specification of patterns. They specify patterns as graph transformation rules, with respect to the abstract syntax graph (ASG) of a system's source code. Applying pattern rules results in enriching the ASG with annotation nodes that may be linked to an arbitrary number of ASG elements. Thus, annotation nodes mark pattern implementations recognized by pattern rules.

A pattern rule is defined by a left-hand side (LHS) and a right hand side (RHS). The LHS of the rule describes the structure that has to be found in the ASG if an instance of the pattern exists. The LHS may also contain annotations created by other pattern rules, thereby permitting a composition of rules. Rules requiring annotations created by other rules depend on those rules. The right-hand side (RHS) of a pattern rule defines an annotation node and links to certain ASG elements that are to be created when the LHS could be matched.

The pattern rules are applied by an inference algorithm that is implemented by the Inference Engine plug-in. The inference engine uses a pattern dependencies net (PDN) in which pattern rules are organized in levels according to their dependencies and trigger relationships. Based on the PDN the inference engine applies rules scheduled in priority queues. It starts with rules that are independent from other rules.

Successfully applied rules create annotations that in turn trigger other rules at higher levels. This is called the bottom-up mode of the inference engine. Newly triggered rules are scheduled according to their levels in descending order. Thus, high level rules, which produce meaningful results, are executed as early as possible. The analysis results (e.g. the annotations) are displayed in class diagrams that can be directly obtained from the ASG.

The inference engine works semiautomatically because it involves the software engineer in the reverse engineering analysis. The reverse engineer may pause the inference at any time and inspect the results produced so far. Furthermore the engineer may modify or manually add hypothetical results and continue the inference. The changes are then considered in the further analysis.

### 2.3.4. Rational Software Modeler

The IBM Rational Software Modeler (RSM) is a UML2 modeling tool based on the Eclipse framework, which is a part of the IBM Software Development Platform (SDP), a set of modeling and model-driven engineering tools.

The diagram editor provided by RSM supports the 13 official diagram types of UML2 and several extra types of diagrams (e.g browse, freeform and topic diagrams). Selecting the range of diagrams on which to perform given operations (e.g. printing) proves to be somewhat cumbersome. Customized UML profiles are supported and model constraints can be defined (e.g., OCL). HTML and XML-based data exportation and reporting are provided out of the box, but the documentation for advanced topics is sometimes not available or is minimal. For example, although it generates quality reports, the set of metrics that is used measures only various dimensions of the model (e.g. number of classes and number of packages) and not the complexity of dynamic diagrams (e.g. sequence diagrams). Features for advanced documentation and quality evaluation can be included in RSM using the plug-in mechanism provided by Eclipse.

RSM provides team support with multi-model support, compare, merge, and system versioning integrations. It supports model versioning but only at the model file level. A model cannot contain several versions of a UML element (e.g. classes). Having several versions of a UML element in the same model is a useful feature when modeling large scale systems.

RSM provides support for two types of transformations: model to model and model to text. It also provides three predefined transformations: to Java, C++, or EJB code. The transformations are implemented programmatically. User transformations can be integrated into RSM using the Eclipse plug-in mechanism. RSM does not provide support for a transformation language, which increases the effort of writing transformations.

### 2.3.5. Comparison

In this section we compare the modeling frameworks described in the previous sections by analysing the features provided by their specification. To achieve this comparison we analyse the languages on the basis of several features. The results of the comparison are summarized in Table 2.1.

| **Feature/ Language** | **EMF** | **MDR** | **FUJABA** | **RSM** |
|---|---|---|---|---|
| Graphic Editor | No | No | No | UML 2 diagrams |
| Model Quality | No | Local and global | No | Local and global |
| Prototype | Graphic | Programmatic | Programmatic | Programmatic |
| Well-formedness | Static & Dynamic OCL | Static OCL | No | Static OCL |
| Transformations | No | No | No | Predefined transformations to Java, C++ and EJB |
| Versioning | No | No | No | Model level |

*Table 2.1. A comparison of modelling frameworks*

In this table the rows represent features that are used for comparison and the columns represent the modeling frameworks that are compared. The table indicates how each feature is supported by a modelling framework. The features are explained below.

**Graphic Editor.** Indicates if the modeling framework contains a built in graphic editor that supports the drawing of UML diagrams.

**Model Quality.** The quality of a model can be measured using software metrics. This indicates if the modelling framework supports quality evaluation both at local and global level. Local evaluation allows the modeler to focus on a quality attribute or a particular

element from the model, without being distracted by having to assess things that are not the current focus. Global evaluation allows the user to have a global view over the entire model.

**Prototype.** In order to provide sufficient data against which to validate the model, the modelling framework must be capable of setting up potential populations of the model. The population can be created either by using a graphic interface or writing programs.

**Well-formedness.** To check if a given population is well-formed, the rules that validate a population must be verified. Hence, modeling frameworks could support both the static and the dynamic evaluation of the well-formedness rules specified in the model.

**Transformations.** Modeling frameworks can support transformations from one model instance to another and trace the mappings.

**Versioning.** It must be possible to handle multiple versions of the model. The versioning can be performed at model level or model element level. Versioning at the local level means that a model can contain several versions of a given element.

## 2.4. Transformation Languages

Transformation languages play an important part in the MDA framework. The process of translation between language models is based on a large body of research in the field of compilation. This section presents some of the existing transformation languages.

### 2.4.1. OMG's QVT

While the current OMG standards such as Unified Modelling Language (UML) and Meta Object Facility (MOF) provide a well-established foundation for defining PIMs and PSMs, no such well-established foundation exists for transforming PIMs to PSMs. In 2002, in its effort to define the transformations, OMG initiated a standardization process by issuing a Request for Proposal (RFP) on Query / Views / Transformations (QVT). This process will lead to an OMG standard for defining model transformations, which will be of interest not only for PIM-to-PSM transformations, but also for defining views on models and

synchronization between models. In response to the OMG's Request For Propoasal (RFP), 8 proposals were submitted:

1) Adaptive Ltd. (in the following abbreviated as ADAPTIVE)
2) DSTC/IBM (abbreviated as DSTC)
3) Compuware Corporation/Sun Microsystems (SUN)
4) Alcatel/Softeam/TNI-Valiosys/Thales (THALES)
5) Kennedy Carter (KC)
6) QVTPartners, which comprises Artisan Software, Kinetum, Kings College, and the University of York (QVTPartners)
7) Codagen Technologies Corporation (CODA)
8) Interactive Objects Software GmbH/Project Technology (IO)

#### *2.4.1.1. DSTC*

To satisfy the requirements of the RFP and those identified above, DSTC developed a transformation language that allows for the declarative specification of transformations without regard for rule application order. This language was prototyped based on a modified F-Logic interpreter [KLW95].

A declarative transformation describes what the result should be in terms of the input, but does not prescribe how to go about constructing the result. However, like Horn clauses in logic programming, instances of a transformation language should be a declarative specification, and also have an equivalent procedural interpretation, thus allowing the specification to be executed.

A transformation in DSTC's language consists of the following major concepts: pattern definitions, transformation rules and tracking relationships. Pattern definitions are used to label common structures that may be repeated throughout a transformation. A pattern definition has a name, a set of parameter variables, a set of local variables, and a term. Pattern definitions are used to name a query or pattern-match defined by the term. The result of applying a pattern definition via a pattern use is a collection of bindings for the pattern definition's parameter variables.

Transformation rules are used to describe the things that should exist in a target repository based on the things that are matched in a source repository. Transformation rules can be

extended, allowing for modular and incremental description of transformations. More powerfully, a transformation rule may also supersede another transformation rule. This allows for general case rules to be written, and then special cases dealt with via superseding rules.

Tracking relationships are used to associate a target element with the source elements that lead to its creation. Since a tracking relationship is generally established by several separate rules, they allow other rules to match elements based on the tracking relationship independently of which rules were applied or how a target element was created. This allows one set of rules to define what constitutes a particular relationship, while another set depends only on the existence of the relationship without needing to know how it was defined. This kind of rule decoupling is essential for rule reuse via extending and superseding to be useful.

#### *2.4.1.2. Thales*

The core of this proposal is a transformation language called TRL (Transformation Language). The language can be used for querying models as well as for transforming models. It reuses and extends the selection and filtering capabilities already available in OCL 2.0. The type of the data returned by a query may be a composite type (collection types, tuple types, dictionary types) or maybe provided by a metamodel (in which case the query is a special kind of transformation program).

TRL is based on metamodeling techniques. The rules express the relationship between source and target model elements in terms of the available metaclasses, metaattributes and metaassociations. In addition the language has a direct support of dynamic extensibility using through using stereotypes in profiles.

The abstract syntax is provided as a MOF 2.0 compliant metamodel and is independent of the proposed concrete syntax. A TRL program specification may have more than one source model as input. This allows the merging of distinct kinds of data that might be necessary to achieve a complete automated transformation. This applies in particular to marked models. In such a case the designer may declare what are the profiles that apply to a source or target model. In addition to this, a TRL program may have parameters.

### *2.4.1.3. QVT Partners*

This submission proposes a possibly extended version of OCL 2.0 to describe queries in the new QVT language, as OCL 2.0 resolves OCL 1.3's deficiencies as a query language.

A view is a projection on a parent model, created by a transformation. From this simple definition, the proposal builds the necessary machinery to cope with advanced technologies such as RM-ODP style viewpoints. Essentially, the viewpoints are analogous to a query which not only creates a view but also potentially restricts the meta-model of the view as well. Thus from each viewpoint one does not in general have enough information to rebuild the entire system. One possible mechanism for dealing with viewpoints in this proposal is to use a query to create a view of a model, and then use a transformation to alter the view to reflect the viewpoint's restricted meta-model.

This proposal defines the transformations using two distinct layers. Similar to UML2 concepts, they are named the *infrastructure* and *superstructure* layer. The proposal defines a simple infrastructure which has a small extension to the MOF meta-model and whose semantics are easily defined in terms of existing OMG standards. The infrastructure is necessarily low-level and not of particular importance to end users of transformations. The superstructure contains a much higher-level set of transformation types suitable for end users. Some parts of the infrastructure are effectively included 'as is' in the superstructure. Concepts that exist in the superstructure but not in the infrastructure have a translation into the infrastructure. This superstructure contains plug points to allow it to be easily extended with new features.

The proposal's overall framework for transformations allows the use of a variety of different transformation styles: *relations* and *mappings*.

Relations are multi-directional declarative specifications. In general they are non-executable, but some restricted types of bi-directional relations can be automatically refined into mappings. Relations are written in any valid UML constraint language, OCL being the best choice. In general, relations are used in the specification stages of system development.

Mappings are transformation implementations. Hence they are operational. Unlike relations, mappings are potentially uni-directional. Mappings are expressed in the Actions Semantics Language (ASL) and thus encompass all programming language implementations. Mappings

can implement any number relations, in which case the mapping must be consistent with the relations it refines.

### 2.4.2. ATLAS Transformation Language

The ATL is a QVT-based transformation language, developed by the INRIA Atlas team. An implementation of ATL is currently available as open source under an Eclipse project called Generative Model Transformer (GMT) project. It is developed as a set of Eclipse plugins and works as a development IDE for transformations, with execution and debugging. Currently integrates with EMF and MDR.

It is described by an abstract syntax (a MOF meta-model), a textual concrete syntax and an additional graphical notation allowing modelers to represent partial views of transformation models. A transformation model in ATL is expressed as a set of transformation rules. The recommended style of programming is declarative. Transformations from Platform Independent Models (PIMs) to Platform Specific Models (PSMs) can be written in ATL to implement the MDA.

The declarative part of ATL is based on the notion of *matched rule*. Such a rule consists of a source pattern matched over source models and of a target pattern that gets created in target models for every match. Traceability links are automatically created. Rule inheritance and polymorphic rule reference are available. Navigation is performed using OCL expressions.

Transformation programs written in ATL are inherently unidirectional. Source models, which are only navigable (e.g. read-only), and target models, which are not navigable (e.g. write-only), are clearly identified at development time.

ATL offers two imperative constructs: *called rule* and *action block*. A called rule is explicitly called, like a procedure, but its body may be composed of a declarative target pattern. Matched rules and called rules may be used together in a single transformation program. Action blocks are sequences of imperative instructions that can be used in either matched or called rules. The recommended style is declarative (e.g. no called rules and no action blocks). Imperative style should only be used when no declarative language construct provides the capabilities required by a particular case.

There are two modes in which the declarative part of an ATL program can operate: *standard* and *refining*. In standard mode, elements are only created when a rule is matched. However, since models cannot be transformed in-place (source models are read-only), transformations that only modify small parts of a model and leave most of the rest unchanged are complex to write in this mode. As a matter of fact, there must be roughly at least one copy rule for each type declared in the metamodel. This is not required in the refining mode where unmatched elements are automatically copied by the engine. In most cases, developers may assume they are actually modifying a source model with the difference that every navigation expression always operates on the original source model.

### 2.4.3. Other Transformation Frameworks

Below are some open source tools of different character:

- UMT (UML Model Transformation Tool) - UMT is an open source UML/XMI-based tool for model transformation and code generation purposes, which uses XSLT and Java for generation [UMT].
- The IBM Model Transformation Framework (MTF) is an EMF based model transformation framework, now available at alphaWorks. It provides a declarative means of specifying metamodel relationships, similar to that of QVT relations [MTF].
- Generative Model Transformer (GMT) an Eclipse project that will provide model transformation technology for the Eclipse platform. Currently the FUUT-je tool, a code generator tool, is the primary GMT deliverable. (ATL, mentioned above, provides core transformation technology.) [GMT]
- MTL Engine. Another QVT-like implementation, by the INRIA Triskell team. Uses the MTL language. Integrates with Netbeans MDR and Eclipse EMF.
- MOdel transformation Language (MOLA) is combination of traditional structured programming in a graphical form with pattern-based rules. The loop concepts enable the iterative style for transformation definitions, while other languages rely on recursion [MOLA].
- MOFScript, a model to text transformation tool, based on one of the OMG MOF Model to Text Transformation submissions. It is implemented as an Eclipse plugin, based on metamodels/models in EMF [MOFS].
- ModFact. A MOF Repository and QVT-like engine from LIP6, Paris. Based on the TRL language. LIP6 are also working on an open source ModelBus implementation, which will enable MDD tools interoperability [MODF].
- OpenArchitectureWare, a flexible, template-based generator framework integrated with XMI [OAW].

- OpenMDX, an open source MDA environment, which integrates with several tools through XMI and supports code generation towards several target platforms (J2EE, .Net) [OMDX].
- AndroMDA, an open source template-based tool for J2EE code generation from UML/XMI. Uses VTL (Velocity Template Engine) as scripting language and Netbeans MDR as a model API [AMDA].
- XDoclet an open source, attribute based code generation tool for J2EE. Not really model-based, but can be combined with generation tools such as UMT to achieve good model-based value [XDOC].
- Middlegen, an open source, database driven code generator based on JSBC, Velocity, Xdoclet and Ant [MID].

## 2.5. Languages and Translators

There is a communication gap between humans and computers. Computer hardware operates in terms of bytes and locations while humans express themselves in terms of natural languages such English or using high-level concepts. A translation process bridges the human-machine communication gap. Language translation is the process of restating some text written in one language in a different language. In other words, to translate is to examine some original text, written in what is termed the source language, and write a corresponding text in a different language, termed the target language, with the goal of preserving the meaning of the original text.

### 2.5.1. Languages, grammars, and automata

Programming languages used for the purpose of computer programming (such as C# or Java) do not resemble human languages very much. They are described using tools termed *formal languages*. Formal languages lack questions, exclamations, simile, metaphor, and other features of human language. [Sal73] provides a general treatment of formal languages.

In computer science a *formal language* is a set of *strings* over a given *alphabet*. A *grammar* is a way of describing *formal languages*. These systems are named *grammars* by analogy with the concept of grammar for human languages. The basic idea behind these formal systems is that strings contained in a language can be generated by starting from a special

start symbol and then apply rules that indicate how certain combinations of symbols can be *rewritten* by replacing them with other combinations of symbols.

A grammar *G* is an algebraic system consisting of the following components:

- A finite set *N* of *nonterminal symbols*.
- A finite set T of *terminal symbols* that is disjoint from *N*.
- A finite set *P* of *production rules* where a rule is of the form

  $\alpha \rightarrow \beta$ where $\alpha$ and $\beta$ are strings from the language $(T \cup N)^*$

  (where * is the Kleene star operator and $\cup$ is set union) with the restriction that the left-hand side of a rule (i.e., the part to the left of the $\rightarrow$) must contain at least one nonterminal symbol.
- A symbol *S* in *N* that is indicated as the *start symbol*.

Usually such a grammar *G* is simply summarized as (*N, T, P, S*).

A grammar is a *rewriting* system that generates strings from other strings by applying the grammar's productions. The string $\gamma_1$ derives directly to $\gamma_2$ , denoted as $\gamma_1 \Rightarrow \gamma_2$, if there is a production rule $\alpha \rightarrow \beta$ in G such as $\gamma_1 = \alpha_1\alpha\beta_1$ and $\gamma_2 = \alpha_2\beta\beta_2$, where $\alpha_1$, $\beta_1$, $\alpha_2$, and $\beta_2$ are arbitrary strings over the alphabet $(N \cup T)^*$. The notation can be extended to $\Rightarrow^+$ and $\Rightarrow^*$ using Kleene's operators.

The *language* described by a formal grammar *G* = (*N, T, P, S*), denoted as *L*(*G*), is the set of strings over T that can be generated by starting with the start symbol *S* and then applying the production rules in *P* until no more nonterminal symbols are present:

$$L(G) = \{w \in T^* \mid S \Rightarrow^* w \}$$

Languages can also be described using concepts from *automata theory*. The automata are abstract models of computer execution and storage. The best-known automata are the Turing machines, pushdown automata, and finite state machines. [Gin75], [Sal69], and [HU79] comprise a general treatment of automata and languages.

Turing [Tur36] introduced in 1936 a machine termed since than the *Turing Machine*. The purpose of this machine was to give a precise definition of algorithm or "mechanical procedure". Turing machines are widely used in theoretical computer science, especially in the theory of computation and theory of algorithm complexity.

The origin of *pushdown* concept is not clear and is attributed by most to [BWW54] and [NS57]. A little later the term LIFO storage was used explicitly in the literature by [SB60], who used it to translate the ALGOL formulas into machine code. *Pushdown automata* are best known for accepting the family of context-free languages, which was independently proved by [Cho62] and [Eve63].

A *finite-state automaton* is an abstract machine that has only a finite, constant amount of memory and an internal state. There are several types of finite state machines: acceptors, recognizers, and transducers. Acceptors either accept the input or do not by producing a “yes” or “no” answer. Recognizers are used to categorise the input and transducers are used to generate an output from a given input. Apart from theory, finite state machines like Moore and Mealy machines occur in hardware circuits.

Noam Chomsky introduced in [Cho56] a containment hierarchy of grammars. Table 2.2 summarizes each of Chomsky’s four types of languages, the class of grammars it generates and the type of automaton that recognizes it.

| **Language** | **Grammar** | **Automaton** |
|---|---|---|
| Recursively enumerable | Type-0 | Turing machine |
| Context-sensitive | Type-1 | Linear-bounded non-deterministic automaton |
| Context-free | Type-2 | Non-deterministic pushdown automaton |
| Regular | Type-3 | Finite state automaton |

*Table 2.2 Chomsky’s hierarchy*

### 2.5.2. Language processors

A *translator* is a program that accepts as input a program written in a language, termed the *source language*, and produces a program written in another language, termed the *target language*, preserving the meaning of the original program. Translators typically distinguish translation from interpretation, which is live translation of speech.

If the source language is a high-level language such as C# or Java and the target language is a low-level language such as assembly language or machine language, the translator is a *compiler*. The machine language of a computer is sometimes termed *object code*.

An *assembler* is a translator from an assembly language, which is very close to the machine, to the object code of a given machine.

An *interpreter* is a program that accepts a source program written in the source language and executes it. The interpreter does not produce an object program to be executed; it performs all the operations implied by the source program.

In theory an interpreter has to follow the control graph attached to the source program, analyse, and execute each action. This approach is very inefficient and therefore it is not used in real scale systems. The usual method is to split the interpretation process into two phases. The first phase analyses the entire source program and builds an internal representation. The second phase executes the internal form of the source program, following the control graph.

Among practitioners, a distinction is generally made between translation, where the compiler generates object code which is then executed, and interpreting or interpretation, where the interpreter analysis and executes the source program. From the point of view of analyzing the processes involved (translation studies), it is perhaps more useful to treat interpreting as a subcategory of translation.

Many software tools that manipulate source programs first perform some kind of analysis similar to that of a compiler. Some examples of such tools include: structure editors, pretty-printers, static checkers, text formatters, query interpreters, and preprocessors. Practical aspects of the translation process are presented in more detail in [ASU86], [WG84], [AP02], and [FL91].

## 2.6. Object Oriented Design Patterns

Mature engineering disciplines have handbooks that describe successful solutions to known problems. For instance, rail track designers do not design rail tracks by starting from scratch and using the laws of physics and geometry. Instead, they reuse standard designs with

successful track records regarding functionality and safety. The extra few percent of performance available by starting from scratch is not worth the cost.

Object-oriented developers wrote the first software patterns, so they focused on object-oriented design and programming [GHJV95] or on object-oriented modeling [Coa92]. Since then new trends appeared, for instance creating patterns in concurrent, parallel, and distributed programming systems [CVK96] [Gra02].

This thesis makes use of several of these patterns with respect to providing an implementation of models. These patterns are described in the following subsections.

### 2.6.1. Factory Method Pattern

Very often one needs to construct an object without knowing the class of object it must create. The Factory Method pattern is a creational pattern that “Define an interface for creating an object, but let subclasses decide which class to instantiate” [GHJV95]. The Factory Method pattern delegate the responsibility of choosing the class that must be created to subclasses. The participants involved in this software pattern are described in Figure 2.1.

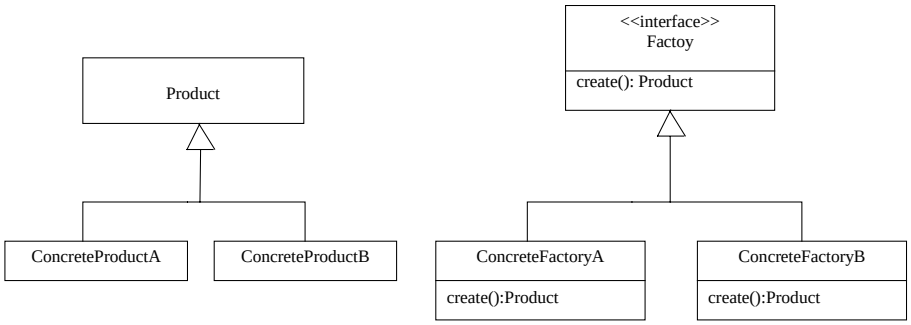



*Figure 2.1 Participants of the Factory Method Pattern*

### 2.6.2. Abstract Factory Pattern

The Abstract Factory pattern is one level of abstraction higher than the Factory Method pattern. The Abstract Factory pattern provides a way of encapsulating a group of individual

factories that create similar products that belong to different families of products. This pattern separates the details of implementation of a family of objects from their general usage. The participants involved in this pattern are presented in Figure 2.2.

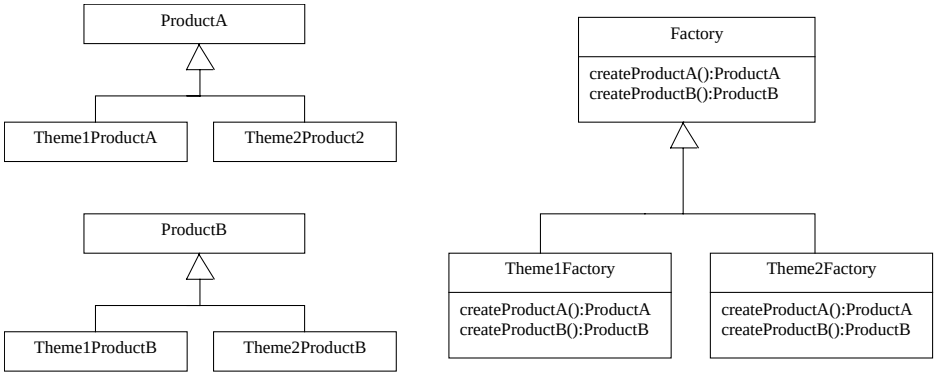



*Figure 2.2 Participants of the Abstract Factory pattern*

### 2.6.3. Builder Pattern

In many cases the algorithm for creating a complex object must be independent of the parts that make up the object. As the Builder pattern separates the construction of a complex object from its representation, a variety of representations can be created using the same construction process. This creational pattern it is intended “to decouple the process of building complex objects from parts that make up the object” [GHJV95]. The Builder pattern has two main participants called *director* and *builder*. The director, which responsible for the overall organization of the creation process, makes calls to the builder. The builder constructs the complex object under the control of the director. The structure of the pattern is presented in Figure 2.3.

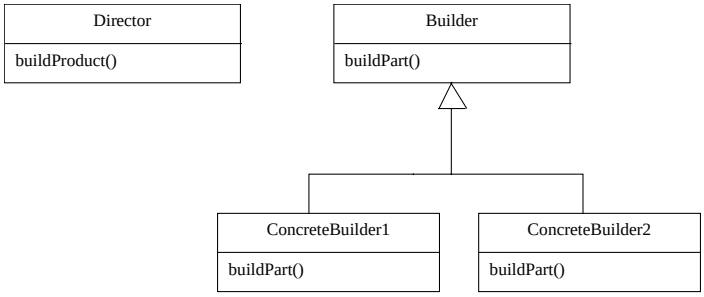


*Figure 2.3 Participants of the Builder pattern*

### 2.6.4. Visitor Pattern

The Visitor pattern is a behavioural pattern that lets you to define and perform a new operation on all the elements of the object structure, without changing the classes of the elements on which it operates. In the visitor pattern, the operations are seen as objects as themselves. The participants involved in this pattern are presented in Figure 2.4. The visitor pattern is characterized by the following:

1) Two interfaces are defined: *Visitable* and *Visitor*.
2) Each element of the object system implements the *Visitable* interface.
3) For each new operation a concrete visitor is defined that implements the *Visitor* interface.
4) The parameters of the operations are stored in *Data*.

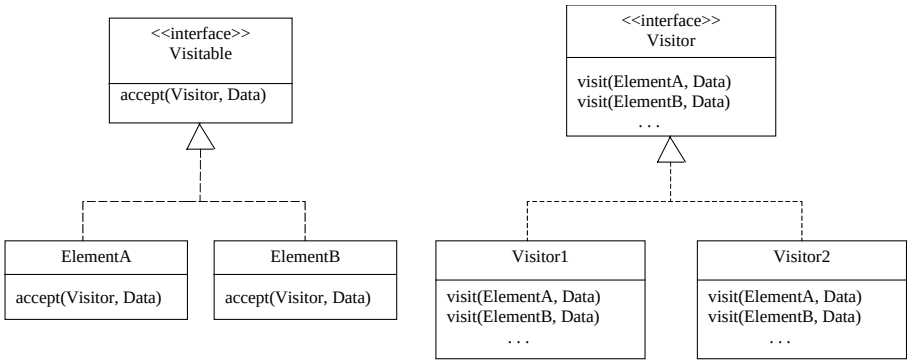



*Figure 2.4 Participants of the Visitor Pattern*

### 2.6.5. Observer Pattern

Sometimes partitioning a system into a collection of cooperative classes looses the consistency between related objects. Consistency can be achieved either by making the classes tightly coupled or using the Observer pattern. The Observer pattern is a behavioural pattern that defines the dependency relations between cooperative classes.

The key concepts in this pattern are *subject* and *observer*. A subject may have any number of dependent observers. The subject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own. After being informed of a change in the subject, an observer may query the subject for information.

The participants involved in this pattern are presented in Figure 2.4. The visitor pattern is characterized by the following:

1) Two objects are defined: *Subject* and *Observer*.
2) Each element of the object system that must be observed is a subtype of the *Subject* object.
3) For each subject there zero or more observers.
4) The parameters of the operations are stored in *Data*.

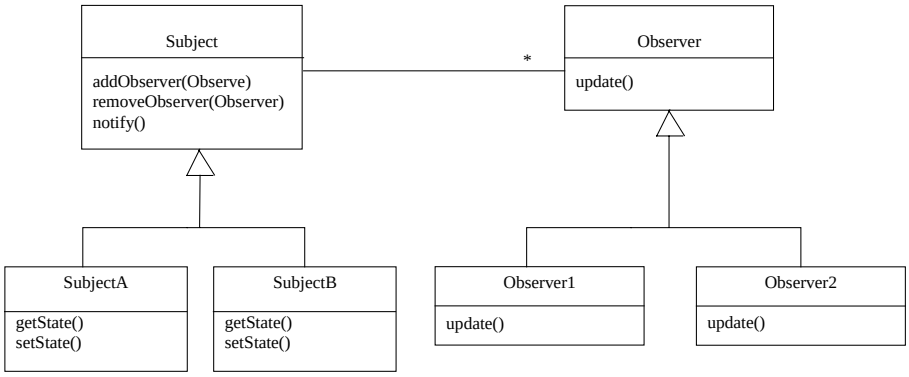



*Figure 2.5 Participants of the Observer Pattern*

## 2.6.6. Adapter Pattern

Sometimes objects with different interfaces need to communicate with each other and work together in a single program. In such cases the adapter pattern is a solution. The Adapter pattern is a structural pattern that converts the interface of a class into another interface that clients expect.

The key objects in this pattern are *target*, *adapter* and *adaptee*. Target defines the interface that the client is using. An adaptee defines an existing interface that needs to be adapted. An adapter adapts the interface of the adaptee to the target interface.

The adapter pattern can be implemented in two ways, as object adapters or class adapters. The difference between these two implementations is given by the strategy used to solve the problems: composition versus inheritance.

**Object Adapters**

Object adapters use a compositional strategy to adapt one interface to another. The adapter inherits the target interface that the client expects to see and contains an instance of the adaptee. When the client calls a method on the adapter, the method is translated into the corresponding specific request on the adaptee. The structure of object adapters is presented in Figure 2.6.

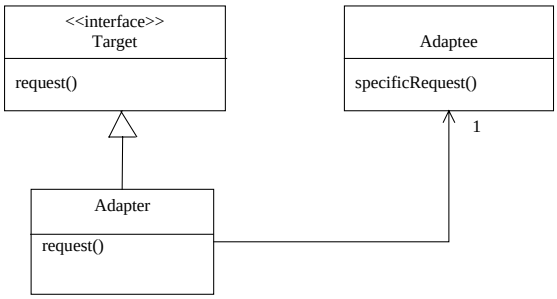



*Figure 2.6 Participants of Object Adapters*

**Class Adapters**

Class adapters use multiple inheritance to achieve their goals. As in the object adapter, the class adapter inherits the interface of the client's target. It also inherits the interface of the adaptee as well. The participants of the class adapters are presented in Figure 2.7.

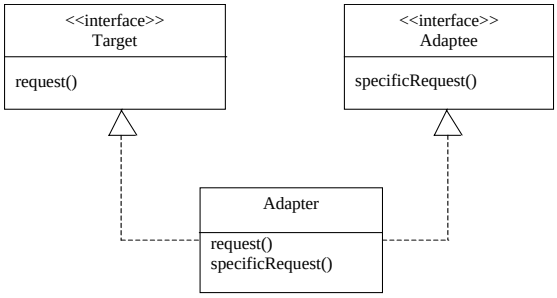



*Figure 2.7 Participants of Class Adapters*

A class adapter adapts adaptee to target by implementing a concrete class. Thus a class adapter is not capable to adapt a class and all its subclasses. Object adapters are capable of adapting a class and all its subclasses.

### 2.6.7. Bridge Pattern

The Bridge pattern is a structural design pattern whose intension is to "decouple an abstraction from its implementation so that the two can vary independently" [GHJV95]. The Bridge pattern encourages loose coupling of objects through the use of delegation.

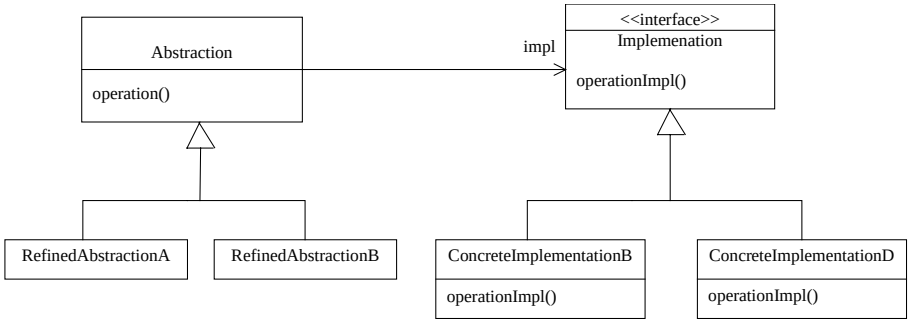



*Figure 2.8 Participants of Bridge Patterns*

The key concepts in this pattern are *abstraction, refined abstraction, implementation,* and *concrete implementation*. The abstraction defines the interface that the client uses for the interaction with the abstraction. The abstraction object maintains a reference to an implementation object that is used to forward the client request to the implementation. A refined abstraction is any of the abstract class extensions. The implementation defines the

interface for any of the implementations of the abstraction. Typically the implementation interface "provides only primitive operations, and Abstraction defines higher-level operations based on these primitives" [GHJV95]. The concrete implementation simply implements the interface defined by the implementation, defining a concrete implementation of the abstraction. The participants of the bridge pattern are presented in Figure 2.8.

## 2.7. Summary

In this chapter we have given an overview of the topics that form together the foundation of the research contained in the rest of this thesis. Modeling languages are vital for the process of software development using the model-driven approach. They are vital mainly for specifying the models used during the development process: computation independent model, platform model, platform independent model, and platform specific model. As models are a key part of the MDA framework their quality is very important, as is the validation of a model over a population of model instances.

Transformation languages play a very important part in the MDA framework. The process of translation between language models is based on a large body of research in the field of compilation. The finalization of OMG's initiated standardization process of QVT [QVT02] will also provide the missing link of MDA [GLRSW02]. A more detailed description of approaches taken so far for model transformation is presented in [CH03].

Software development, like any other mature engineering discipline, should be based on software patterns. Software patterns may vary from object-oriented design and programming patterns [GHJV95], object-oriented modeling patterns [Coa92] to more general and sophisticated patterns, as in concurrent, parallel, and distributed programming [CVK96] and object-oriented software environments [Gra02].

# Chapter 3. KENT MODELING FRAMEWORK

Modeling and metamodeling has become popular because it aids the derivation of implementation from a definition. Software tools for automatically generating an implementation of the structural part of the definition are now publicly available. Unfortunately, these tools do not tend to be used when a metamodel is developed, as the tools are not appropriate for supporting the definition process, and well-formedness rules of model instances tend to be ignored.

## 3.1. Modeling Tools Requirements

Currently, the focus of modeling is to capture the abstract syntax of a language, although models can also be to define other aspects of a language, such as semantics and evaluation. This thesis will focus on concrete syntax, abstract syntax, semantics, and the appropriate mappings between them.

The general problem is to support the activity of modeling, by providing a means to check during the process of a model development if the model is fit for purpose, well-formed and error-free. The general approach we have adopted is to generate modeling tools from a model. Another approach might be to provide an interpreter for the model. We have followed the first approach, because we would like to move on from generating prototypes to generating industrial-strength modeling tools or at least fragments of modeling tools. We do not believe that the interpretive approach can be used to deliver industrial-strength tools, mainly because such an approach is time consuming. With this in mind, we can now consider more specific requirements, both for the generator, and for the generated prototype.

In order to check if a model is fit-for-purpose we have to ensure that what needs to be expressed in the language it describes can be represented as an instance of the model, and that only instances which represent valid expressions of the language are valid instances. Hence, the generated tool should support a process of validation, which allows potential instances of the model to be explored and checked against the model.

This leads to the following requirements for a prototype tool generated from a model:

1) **Evaluating the quality of the model.** The quality of a model influences the entire process of software development because it is unlikely that a low quality model can be used to automatically generate a high quality software product. Tools should support quality evaluation both at local and global level. Local evaluation or selective evaluation allows the modeler to focus on a quality attribute or a particular element from the model, without being distracted by having to assess things that are not the current focus. Global evaluation allows the user to have a global view over the entire model. The quality checker must provide clear feedback to the user, which is vital in order to detect and fix errors.
2) **Rapid and repeatable input and editing of populations.** A population of a given model is a set of instances of the elements described in the model, representing items from the described language. In order to provide sufficient data against which to validate the model, the tools must be capable of setting up potential populations of the model quickly, in several ways (e.g. using a graphic interface or writing programs). The populations may include examples (valid constructions) and counter-examples (invalid constructions). It must be possible to set up sophisticated populations, representing complex constructions and subtle boundary cases. For instance, a tool that only allows you to set up a model instance object by object, link by link, would not meet this requirement very well.
3) **Viewing and exploring populations.** Tools must be capable of viewing and exploring a population easily. This facility is extremely important in certain situations, for instance when debugging well-formedness rules.
4) **Evaluation of well-formedness rules over populations.** It must be possible to evaluate well-formedness constraints over populations. Tools should support both local evaluation and global evaluation. Local evaluation or selective evaluation allows the user to focus on a particular rule or a particular element from the population, without being distracted by having to evaluate rules that are not the current focus. Global evaluation allows the user to have a global view over the entire population. The rule checker must provide clear feedback to the user, which is vital in order to detect and fix errors.
5) **Model transformation.** It must be possible to create transformations from one model instance to another and trace the mappings. Tools should support transformation both at a local and a global level. Transformations at a global level allow the user to have a global view over the entire population, while local transformations or selective transformations allow him to focus on a particular rule or a particular element from the population, without being distracted by having to perform rules that are not the current focus. The clarity of the feedback provided by the transformation engine is vital in order to detect and fix errors.

6) **Smooth process.** We would like the process of developing and editing models, applying transformations, compiling and launching the generated code, working with and obtaining feedback from test populations, then cycling back to the model, to be as smooth as possible. It is important that a generated prototype can work with other tools, especially ones that might provide a means of representing constructions in the language being defined in some concrete syntax.
7) **On-the-fly behavior.** It should be possible to input constraints or transformation rules and have them evaluated on-the-fly against sample populations. The feedback from this evaluation should be as helpful as possible.
8) **Round-trip engineering.** We have found that 100% generation of code is very difficult, especially when we consider some of the requirements on the generated tool that have to be met. So it is necessary to assume that the generated code will be supplemented by some hand-written code. On the other hand the model might be changed in the future, and the code regenerated.
9) **Model versioning.** It must be possible to handle multiple versions of the model in a way that does not require major changes to hand-written code, just because the version number (e.g. in the model name) changes.

## 3.2. The Kent Modeling Framework

This section describes the *Kent Modeling Framework* [KMF] and how it can be used to generate a modeling tool from a model. It then proceeds to describe the customization of the generated code, in particular the definition of methods that allow a rapid and repeatable input of population.

### 3.2.1. About KMF and KMF-Studio

KMF provides a set of tools to support model driven software development. At the core of KMF is KMF-Studio, a tool that generates modeling tools from the definition of languages expressed as models. KMF-Studio is supported by OCLCommon and OCL4KMF, two Java libraries that allows dynamic evaluation of OCL2 constraints; and XMI, a Java implementation of the XMI standards. Tools generated using KMF-Studio use OCLCommon and OCL4KMF to provide built in support for checking well-formedness of models, amongst other things; they use XMI to write and read models in XMI format. XMI is also used by KMF-Studio to read in models in standard UML 1.3 XMI 1.0 and XMI 1.2 format. The code generated by KMF Studio for a particular model is summarized in Table 3.1.

| Metamodel | Generated Java code |
|---|---|
| m:Model | User can choose the location of the generated code, and also the name of the model.<br><br>Licensing support for generated code is also provided.<br><br>A common set of boilerplate interfaces (e.g. Visitable, XElement, where X is the name of the model).<br><br>GUI code, XMI readers and writers and code for constructing and populating a repository.<br><br>Factory and Visitor interfaces for generating and navigating the model elements.<br><br>A repository storing all generated elements. |
| For all p:Package in m | Corresponding interfaces and classes are generated in a Java package, whose pathname follows the nesting structure of packages in the metamodel. |
| For all c:Class in p | A lifecycle class that includes a factory method for creating instances of the Java class generated from this class.<br><br>A repository contains one instance of the lifecycle class for each class, and the factory method stores the object it creates in that repository.<br><br>Lifecycle classes can be specialized using hand-written Java code, and repositories can be configured with objects of the specialized |

| | |
|---|---|
| | versions. |
| For all c:Class in p | Interface<br><br>Extends interfaces from superclasses, standard library classes such as X.XElement, where X is the name of the model.<br><br>Class<br><br>Implements interface generated from class. Includes boilerplate code required for GUI, XMI reading/writing and to support repository services. |
| For att:Attribute in c | Interface<br><br>A get method with the name getX, where X is the name of the attribute.<br><br>A set method with name setX, where X is the name of the attribute.<br><br>Class<br><br>An attribute whose name is derived from the name of the attribute.<br><br>Implementations for the get and set methods in the interface, that make use of the attribute. |
| For all q:Query operation in c | Interface<br><br>A method with corresponding signature.<br><br>Class<br><br>An implementation of the method, whose body is |

| | |
|---|---|
| | derived from the (OCL) expression that is the body of the operation. |
| For all inv:Invariant in c | A visit method included in XParseAllVisitor and XEvaluateAllVisitor classes, where X is the model name. |
| For all assoc:Association in p | If association is bidirectional, then two constraints are generated, one in each class connected by the association, to capture the bidirectionality constraint. |
| For all ae:Association End (only navigable ones) in assoc | Treated as attributes of the class at the source of the association end, where the type of the attribute is governed by the multiplicity of the end. If the target of the end is class X then if the cardinality is 1, the type is whatever X is mapped to; if the cardinality is greater than 1 and the association end is ordered the type is List from java.util package; else the type is Set from java.util package. |
| Type of attributes, arguments and result of operations, and association ends | When a class or datatype is used as the type of an attribute, parameter or operation in the metamodel, if the type is a class then interface matching the class is used as the type. If it is a primitive type X, where X is Integer, String, Boolean, Set, Sequence or Bag then the type is the corresponding Java primitive types. All basic types such as *int*, *float* and *double* are mapped to corresponding reference types such as Integer and Double etc. |

*Table 3.1 Outline of code generated by KMF Studio*

The generated code can be executed directly, which will launch a tool that provides the following functionality:

- The ability to populate the metamodel, to explore populations, and to edit and view specific elements of the population through a forms style interface.
- The ability to check well-formedness of a population in memory according to well-formedness constraints expressed in OCL on the metamodel.
- The ability to dynamically evaluate OCL expressions over the population in memory.
- The ability to save and load populations to/from XMI files.
- The ability to save populations to a "Human Usable Textual Notation" [HUTN] format.

### 3.2.2. About OCL support

OCLCommon and OCL4KMF are two libraries used in tools generated by KMF Studio to check constraints in the metamodel over populations, and to support dynamic evaluation of OCL expressions entered by the user through the GUI.

The OCL libraries provide support both for compilation and interpretation of OCL expressions. Implementing both a compiler and an interpreter maximizes the efficiency of the implementation by reducing the runtime. The compiler is used by KMF to generate code to check the constraints that are described into the metamodel, while the interpreter is used in the generated code to allow the user to explore and discover other useful constraints that are not present in the metamodel and evaluate them on-the-fly. If new constraints are discovered, they can be added into the metamodel and the compiler will generate code for them if the tool is regenerated using KMF.

The libraries provides support for all the standard OCL data-types (including collections) and all of the defined operations for those types. The evaluation of OCL expressions can be performed within Java code, by calling a method and passing the expression string and context objects as parameters, or by invoking an evaluator GUI with a defined context into which OCL expression strings may be typed. The former method of evaluation is used by the generated code to construct invariants defined on model elements; these invariants can be evaluated separately or 'on mass' from within the generated tool. The latter, GUI, method of

evaluation is provided to enable evaluation of expressions that are not part of the defined model, but which may be useful in exploring the model and testing parts of invariants.

OCLCommon contains elements (e.g. classes and methods) that are platform/tool independent. The platform/tool specific elements are contained in the OCL4KMF library. This approach increases the portability of the OCL support to other modelling platforms and tools (e.g. Eclipse Modeling Framework). OCLCommon is divided into the following packages: Syntax, Semantics, Evaluation, and Bridge. The Syntax package contains an OCL parser and APIs for the OCL abstract syntax tree model. The Semantics package contains a semantic analyzer for OCL and APIs for the OCL semantic model. The Evaluation package contains the compiler and the interpreter. The semantic analyzer uses a bridge to connect to a specific description of the model, in our case a bridge to UML1.x. In order to evaluate an OCL expression for a different model a new bridge implementation has to be written. The Bridge package contains the interfaces that must be implemented in the platform specific library (e.g. OCL4KMF and OCL4EMF).

Most of the code contained in the OCLCommon, around 85-90%, was developed using MDA techniques. KMF-Studio and a parser generator called CUP have been used to generate Java code starting from abstract description: a UML model of OCL's abstract syntax and a BNF description of OCL's concrete syntax.

## 3.3. About XMI and UML support

Both KMF-Studio and generated code need support for persistence and other common behaviour. The XMI package provides supports for reading and writing XMI files, supporting the standards XMI 1.1 and XMI 1.2. KMF-Studio reads in information from an XMI file that describes the model and creates instances of UML elements. The required UML API is provided by the UMLModel package. Initially, this package contained only a part the UML model, which was hand written. When KMF-Studio became mature enough the UMLModel API was generated automatically using KMF-Studio.

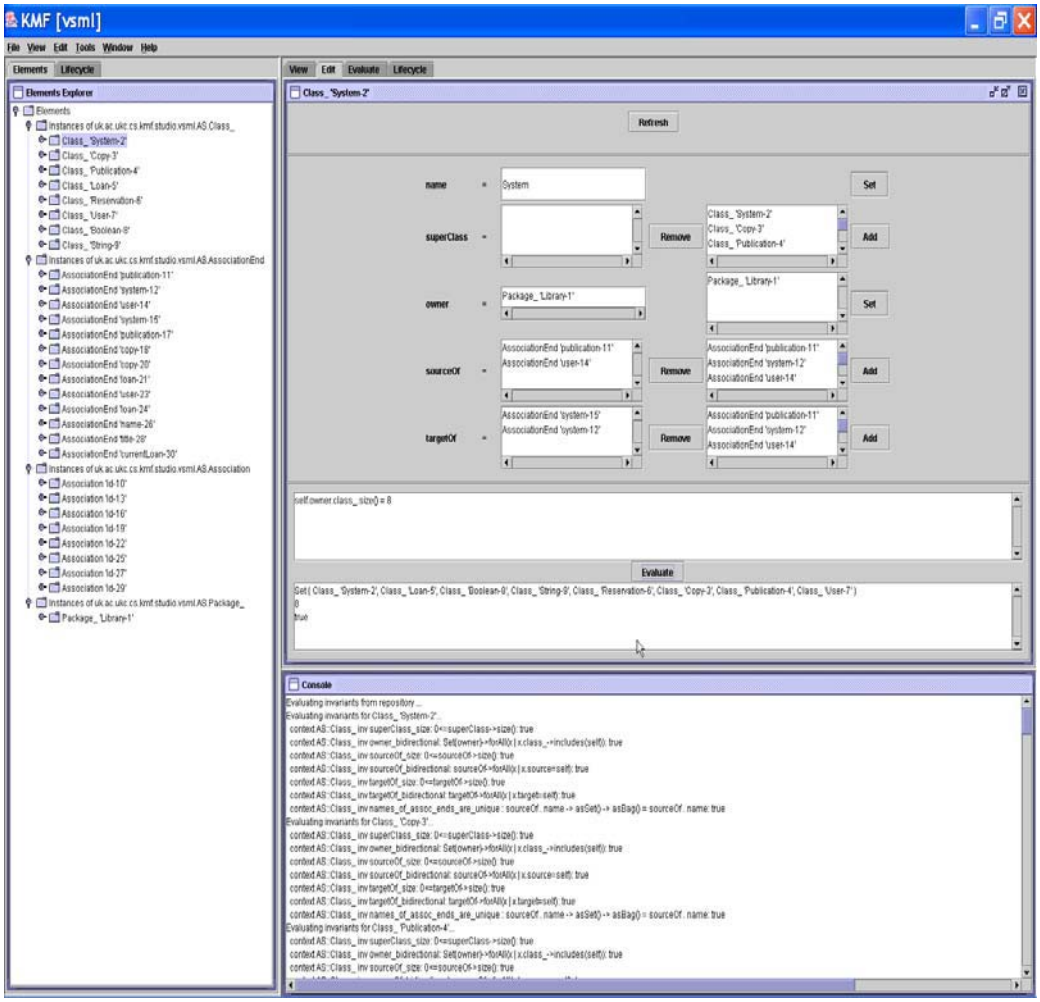



*Figure 3.1 Screen shot for generated tool*

## 3.4. The generated tool

The screen shot shown in Figure 3.1 illustrates the generated tool for a fragment of the UML language that contains packages, classes, associations and association ends. The left hand side shows the objects populating the specified model. The right hand side shows the facility for editing properties of a *Class_*. The generated tool must also deal with details regarding the underlying programming language (e.g. Class is a default Java class).

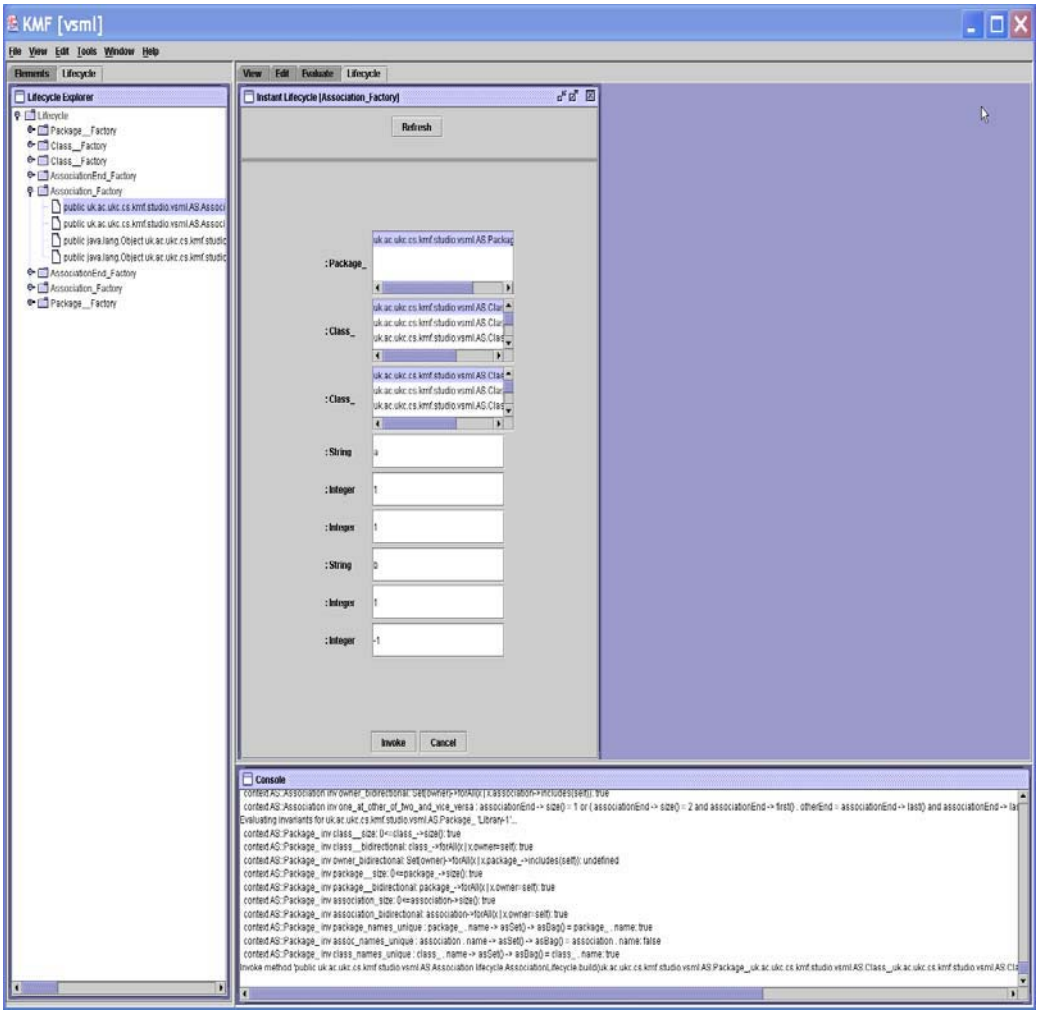



*Figure 3.2 Screen shot for builders*

The middle shows the evaluation of the OCL expression *self.owner.class_.size()=8*, which has been entered into the dynamic OCL evaluator in order to debug the expression. We can see the value of the subexpressions *self.owner.class_* and *self.owner.class_.size()* and the final result.

The bottom of the right hand side shows the evaluation of the invariants over the entire population. Evaluation can also be performed on subsets of the population, by selecting the desired elements using the explorer window.

## 3.5. Creating populations

Populations of the model can be constructed directly through the generated GUI, which provides access to the lifecycle-builder methods. This is illustrated by Figure 3.2, which shows the available lifecycle-builders, with one of them highlighted, on the left hand side. The right hand side shows the automatically created window that enables the highlighted lifecycle-builder to be invoked with various arguments. The builder shown is actually a bespoke one; it has been coded by hand. The construction of bespoke builders is discussed in the next subsection. Generated lifecycle-builder methods in general don't take any arguments, though they are invoked in a similar way.

Another way of building populations is to write code that initializes the repository before the GUI is launched. To aid the writing of such code, a default *Startup* class is generated. This includes two methods, *replaceDefaultLifecycles* and *initialisePopulation*, which can be overridden by subclasses. A sample *initialisePopulation* methods in a bespoke startup class that extends the default one, is given below.

```
protected void initialisePopulation() {
  //get required Lifecycles from repository
  String path = "Vsml.AS.";
  ClassLifecycle class_b =
     (ClassLifecycle)(rep.getLifecycle(path+"Class_"));
  AssociationLifecycle assoc_b =
     (AssociationLifecycle)(rep.getLifecycle(path+"Association"));
  PackageLifecycle package_b =
    (PackageLifecycle)(rep.getLifecycle(path+"Package_"));
  //Build population
  Package_ pkg = (Package_)package_b.build("example");
  Class_ clsA = (Class_)class_b.build(pkg, "A");
  Class_ clsB = (Class_)class_b.build(pkg, "B");
  Class_ clsC = (Class_)class_b.build(pkg, "C");
  Class_ clsD = (Class_)class_b.build(pkg, "D"));
  Association ass1 = (Association)assoc_b.build(pkg, clsA, clsB,
     "a", new Integer(1), new Integer(1),
     "b", new Integer(1), new Integer(1));
  Association ass2 = (Association)assoc_b.build(pkg, clsA, clsC,
     "a", new Integer(1), new Integer(1),
     "c", new Integer(1), new Integer(1));
}
```

This method begins by extracting the lifecycle objects for the classes that will populate the repository. It then continues to use these to build a population, in this case one comprising a package (example) that contains three classes (A, B, C, and D). The package also contains two associations, with ends labeled (a, b) and (a, c,) respectively.

We have found writing *initialisePopulation* methods inside a startup class to be an extremely efficient way of building test populations to check and validate a model through the prototype, especially in combination with bespoke lifecycle-builder methods. It is very quick to construct new tests or alter existing ones, simply by editing the code, or by more sophisticated means (e.g. by having methods that set up fragments of population and calling these from the main initialization method). It is also possible to evaluate constraints programmatically and check whether or not they pass or fail.

## 3.6. Augmenting the generated code

As an alternative to using the generated code on its own, we can augment it with additional bits of program. We could provide an alternative GUI and reuse the repository and OCL evaluation parts, or write an initialization program that populates the model programmatically rather than by invoking the generated lifecycle-builders from within the generated GUI.

A particularly useful option is to write bespoke lifecycle-builders that can greatly simplify the construction of a population. For example, when creating an *Association* it is necessary to create the association's ends, link and set the attributes and add the association to a package. This requires a number of individual steps that have to be repeated each time an association is constructed. If we perform these steps by writing initialization code to do them, or working through the GUI, we discover that they are time consuming and error-prone.

We can create a bespoke lifecycle-builder that does all of these things when called with the appropriate argument. Additionally, we can register the lifecycle with the existing, generated, repository and subsequently use it from within the generated GUI.

The following code is an example bespoke lifecycle build method for Associations:

```
//one end
public Association build(
          vsml.AS.Package_ p,
          vsml.AS.Class_ source,
          vsml.AS.Class_ target,
          String name, Integer lowerBound, Integer upperBound){
  Association assoc = (Association)build();
  lifecycle.AssociationEndLifecycle end_b =
    (lifecycle.AssociationEndLifecycle)
    (repository.getLifecycle("Vsml.AS.AssociationEnd"));
  AssociationEnd end =
    (AssociationEnd)
     end_b.build(source, target, name, lowerBound, upperBound);

  end.setOtherEnd(null);
  end.setOwner(assoc);
  assoc.getAssociationEnd().add(end);
  if (p!=null) p.getAssociation().add(assoc);
  assoc.setOwner(p);
  return assoc;
}

// two ends
public Association build(
          vsml.AS.Package_ p,
          vsml.AS.Class_ one_class,
          vsml.AS.Class_ other_class,
          String one_name, Integer one_lowerBound,
          Integer one_upperBound,
          String other_name, Integer other_lowerBound,
          Integer other_upperBound){
  Association assoc = (Association)build();
  lifecycle.AssociationEndLifecycle end_b =
     (lifecycle.AssociationEndLifecycle)
     (repository.getLifecycle("Vsml.AS.AssociationEnd"));
  AssociationEnd one_end =
     (AssociationEnd)
     end_b.build(one_class, other_class,
                 one_name, one_lowerBound, one_upperBound);
  AssociationEnd other_end =
     (AssociationEnd)
      end_b.build(other_class, one_class,
                  other_name, other_lowerBound, other_upperBound);
```

```
  other_end.setOtherEnd(one_end);
  one_end.setOtherEnd(other_end);
  other_end.setOwner(assoc);
  one_end.setOwner(assoc);
  assoc.getAssociationEnd().add(one_end);
  assoc.getAssociationEnd().add(other_end);
  if (p!=null) p.getAssociation().add(assoc);
  assoc.setOwner(p);
  assoc.setName("");
  return assoc;
}
```

The *build* method constructs the *Association*, gets the registered lifecycle for *AssociationEnd*, uses this to build each end of the association, adds each of the ends to the *Association* and finally adds the association to the passed in package.

To instruct the generated code to use this bespoke builder we must register it with the repository, as shown below:

```
rep.addLifecycle("Vsml.AS.Association",
                 new AssociationLifecycle(rep));
```

Subsequently, when the generated GUI is executed we are able to use the bespoke lifecycle-builder, as illustrated in Figure 3.2.

Such code can be included in the body of the method *replaceDefaultLifecycles* in a bespoke startup class, as discussed in the previous subsection. This also allows the bespoke lifecycle methods to be accessed by any *initialisePopulation* code included in that startup class.

## 3.7. Code generation

Model driven software engineering requires powerful, efficient, and flexible code generation mechanisms. OO methods help the developer to analyze and understand a system without code generation; however the benefits of object modeling seldom extend throughout a software product’s lifecycle, because developers of a pressing upgrade typically bypass the model and just modify the code. Models fall out-of-date and become less relevant. An efficient, flexible, and maintainable code generation for object models means that they retain their usefulness. This section begins with a discussion regarding the requirements of a code

generation framework and a presentation of different mechanisms that can be used to generate code from UML models along with a discussion about the differences between these mechanisms. This subsection also provides a description of the code generation framework used by KMF-Studio to generate code.

### 3.7.1. Code generation framework requirements

Currently UML is used mainly for the modeling of software systems. It also has potential to be applied in the implementation and testing phases. Generating code from UML models or other platform-independent models reduces coding errors, enforces compliance with coding standards and rules, reduces the time spent to develop software products, increases the quality of both software products and software development processes, and raises the abstraction levels for software architects.

A code generation framework has to meet the following requirements [Bel98][SVB02]:

- **Efficiency.** Code generation should be performed rapidly and without consuming too many resources of the underlying physical machine. Code generation should be performed using a fine tuning mechanism able to detect and regenerate only those parts of the model that have been changed and need to be regenerated in order to keep the system consistent.
- **Customization.** The code generation process needs to provide a mechanism to customize the generated code according to programming rules and user's taste. Customization ranges from low-level features, like changing the indentation and choosing prefixes or suffixes for names, to high-level features like creating targets that do not exist at the abstract level.
- **Extensibility.** Code generation systems should allow the user to rapidly and simply add new features to the generated code. The addition of generation targets should be performed without affecting the existing code generation framework. The user should be able to add new code generation rules without having to recompile the code generation framework.
- **Flexibility.** The code generation system should allow the user to rapidly change the features of the generated code. Adding and removing generated targets, and changing the attributes of the generation targets should be done in a rapid and simple manner.
- **Maintainability.** The code generation process should have a high level of maintainability. This reduces not only the costs for updating the code generation framework but also the costs of the evolution of the resulting software products.

### 3.7.2. Code generation mechanisms

According to [Bel98] three ways of bridging the gap between a model and the running code can be distinguished. Each subsequent mechanism is more complex and more powerful than its precedent.

1) **Structural approach.** This approach is based on code generation from the static structure of the model. In practice this approach generates code from class diagrams. Because class diagrams do not describe the behavior of the system, automated synchronization mechanisms between model and generated code are required, as model and code can easily become inconsistent.
2) **Behavioral approach.** The approach is based on models that contain state machines augmented with action specifications (e.g. SDL and UML state machines). The code generation produces a prototype of the system that can be tested and debugged by changing the model and not the generated code. This approach does not need a synchronization mechanism as both the static structure and the behavior is generated from a model.
3) **Translative approach.** This approach requires a complete application model that describes the object structure, the behavior and communication. The architecture of the target platform is also modeled. A translation engine then generates code for the application according to the mapping rules from the application model to the architecture model. This approach potentially allows one to generate code as well as documentation and test units.

In the next subsections, we will examine four ways of generating code using the translative approach.

### 3.7.3. Programmatic translation

Code generation can be performed in any programming language that can describe a model, gain access to the model description and manipulate basic data objects like integer, strings, and files. If the model is described in UML or other object-oriented modeling language, then object-oriented programming languages like C# or Java are ideal candidates because they can easily represent the model. The code generation module reads in the model and generates the equivalent code in text files.

This approach has following characteristics:

- The code generation rules are described using low-level concepts from the underlying programming languages. Hence, they are hard to read and understand.
- Coding the generation rules into low-level concepts decreases their maintainability.
- Customization is limited to the options provided by a code generation tool. To add to the options the entire tool needs to be analyzed, changed, and recompiled.
- Flexibility is poor because adding, removing or changing some features of the generated code implies analyzing, updating, and compiling the entire module responsible for the code generation.

### 3.7.4. Translation by XSLT

Along with the XML language, the W3C organization provides the Extensible Stylesheet Language (XSL). In essence, XSL is two languages, not one. The first language, called XSLT, is a transformation language, the second a formatting language. XSLT is useful independent of the formatting language. Its ability to move data from one XML representation to another makes it an important component of XML-based electronic commerce, electronic data interchange, metadata exchange, and any application that needs to convert between different XML representations of the same data. These uses are also united by their lack of concern with rendering data on a display for humans to read. They are purely about moving data from one computer system or program to another.

Although the primary goal of XSLT is to translate from one XML dialect into another, its is not limited to that. Stylesheets that translate from a UML model described using XMI, a dialect of XML, to code decouples the translation process from the modeling tool.

Although, this approach represents a step ahead from the programmatic approach, there are still several adverse characteristics:

- The stylesheets are very hard to read and maintain as the translation with XSLT is based on navigation through a tree.
- The code generation rules are still expressed in low-level concepts and hard to read, understand, and maintain.
- As XSLT maps one XML file to another XML file, an input file needs to be generated when generated code contains more than one file. Partitioning the model into small pieces does not solve the efficiency, as the partitioning algorithm is time consuming, especially for large- scale models.

### 3.7.5. Translation by templates

Most of the web pages on the Internet are *static* pages. They are just HTML or text files that are downloaded to your browser and displayed immediately. However, many web pages are *dynamic* pages. They are actually programs which produce HTML as their output, and then send that HTML to your browser. These pages are created using a template-based approach. The creation process uses the generation code to retrieve the required information in the model and fills in templates of HTML code with it.

This approach is not limited to dynamic web pages and it can be used to generate code from models. In the context of code generation the model is the source of the information that is used to fill in the empty slots from templates. The templates are used to describe the skeleton of the generated code. Applying the model to the templates returns a number of files containing the generated code associated with the UML model.

This approach has the following features:

- This approach is flexible as templates can be changed easily without affecting the model and the modeling tool.
- Most of the template engines are interpreted, and hence the code generation process can be slow. However, writing a compiler can solve this issue.
- Directly accessing model information from template languages is possible but complicated. The resulting template is hard to read and understand.
- Adding, removing, and updating templates can be done easily, if the template engine is implemented correctly.

This approach is used to generate code in environments/tools like Eclipse and Poseidon [SVB02].

### 3.7.6. Translation using transformation languages and templates

The above code generation approaches do not fully satisfy all characteristics of a model-driven engineering framework. The template-based approach seems to be the best approach for code generation from models due to its characteristics. The main problem that still needs to be solved is the fact that the code generation rules are still described using low-level concepts, and hence are hard to read and understand.

Hence, the best approach is the following:

- A transformation language like YATL (see Chapter 5) is used to perform a transformation from the model to a model of a target language/platform.
- A set of templates is used to generate the set of files that form the generated code.

This approach has the following features:

- The code generation rules are described using high-level concepts from the model. Hence, they are easy to read and understand. The templates that are used to map from model elements to code are also easy to read and understand because they are very simple.
- Coding the generation rules in high-level concepts increases their maintainability.
- Customization is not limited to the options provided by code generation tool. The transformation rules can be easily changed, and so can the templates. The code generation engine does not need to analyzed, changed, or recompiled if new options are to be added.
- Flexibility is increased because adding, removing or changing some features of the generated code implies analyzing, updating, and compiling the entire module responsible for the code generation.

## 3.8. KMF-Studio's code generation framework

Generating source code can save time in software development and reduce the amount of tedious redundant programming. Generating source code can be powerful, but the program that writes the code can quickly become very complex and hard to understand. One way to reduce complexity and increase readability is to use templates.

The Kent Modeling Framework (KMF) project contains a very powerful tool for generating source code: TLP (Template Language Processor). With TLP one can use a JSP-like syntax that makes it easy to write templates that express the code that one wants to generate.

In this subsection we present the XTL (XTemplate Language) language and the TLP tool that implements the processors for the XTL template language. An overview of XTL is presented in Appendix 2.

### 3.8.1. XTL an introduction

XTL is meant to provide the easiest, simplest, and cleanest way to generate code from UML models. A XTL *program* consists of one or more source files, known formally as *translation units*. A source file is an ordered sequence of Unicode standard characters. Conforming implementations must accept Unicode source files encoded with the UTF-8 encoding form [UNI], and transform them into a sequence of Unicode characters. Implementations may choose to accept and transform additional character encoding schemes, such as UTF-16, UTF-32, or non-Unicode character mappings.

Conceptually speaking, a XTL program is analysed in five steps:

1) Character conversion, which converts a file from a particular character repertoire and encoding scheme into a sequence of Unicode characters.
2) Lexical analysis, which translates a stream of Unicode input characters into a sequence of tokens.
3) Syntactic analysis, which translates the sequence of tokens into an abstract representation of the input structure.
4) Semantic analysis, which checks if the input follows the semantic rules, and produces an internal representation of both syntax and semantics.
5) Code generation or interpretation where the semantic representation is either used to generate code for the underlying machine or directly evaluated on the same machine.

### 3.8.2. Grammars

This section presents the syntax of XTL language using two grammars, structured on two levels. On the first level, the *lexical grammar* defines how Unicode characters are combined to form line terminators, white space, comments, and XTL tokens. At the second level, the *syntactic grammar* defines how the tokens resulting from the lexical grammar are combined to form XTL programs. Both grammars are described in Appendix 3, using the notation given in Appendix 1. Every source file in a XTL program must conform both to the *input* production of the lexical grammar and the *translation-unit* production of the syntactic grammar.

### 3.8.3. Comments

Comments allow descriptive text to be included that is not placed into the output of the template engine. Comments are a useful way of reminding and explaining what XTL actions are doing, or any purpose one finds useful. Below is an example of a comment in XTL.

```
## This is a single line comment
```

A single line comment begins with ## and finishes at the end of the line. Multi-line comments, which begin with #* and end with *#, are available to handle the scenario when one wants to write a few lines of commentary:

```
#*
 First line of comment
 Second line of comment
 ...
*#
```

There is a third type of comment, the XTL comment block, which can be used to store such information as the document author and versioning information:

```
#**
 This is XTL comment and may be used to store such information as
 the document author and versioning information
 @author Octavian Patrascoiu
 @version 5
**#
```

### 3.8.4. Expression action

XTL provides expression actions that one can use to communicate to the surrounding context. The expression with the action is evaluated and the result of the evaluation is inserted into the generated source code at the location where the expression action is defined.

```
public class <% exp context.className %> {
. . .
}
```

An XTL expression uses boolean, integer, real, and string literals, variable and properties as operands and a wide range of operators to communicate with the surrounding environment. The XTL operators are presented in Table 3.2.

### 3.8.5. Compound action

A compound action is used to group for syntactical purposes several actions:

```
<% foreach Classifier c in context.classes %> <% begin %>
public class <% exp context.className %> {
<%include template::java::generateExtension(context.class,“\t”)%>
<%include template::java::generateInterfaces(context.class,“\t”)%>
{
<%include template::java::generateMembers(context.class, “\t”)%>
}
<% end %>
```

| **Operators** | **Example** |
|---|---|
| Unary operators:<br>+ - ! | +3 -4 !true |
| Selection operator | class.name |
| Call operator | f(x, y) |
| Arithmetic operators<br>+ - * / % | a + b |
| Relational operators:<br>== != < <= > >= | 3 <= 4 |
| Logical operators:<br>&& \|\| | true && false |

*Table 3.2 XTL operators*

### 3.8.6. *include* action

The *include* action allows the template designer to invoke a template that was previously written. The text resulted after the invocation of the template is then inserted into the location where the *include* action is defined.

```
public class <% exp context.className %>
<%include template::java::generateExtension(context.class,“\t”)%>
```

```
<%include template::java::generateInterfaces(context.class,“\t”)%>
{
<%include template::java::generateMembers(context.class, “\t”)%>
}
```

generates Java code corresponding to a UML class, by filling in the inherited class, implemented interfaces, and contained members.

### 3.8.7. *if-elif-else* action

The *if-elif-else* action in XTL allows for text to be included when the source code is generated, if a certain condition is true. For example,

```
<% if (x == 1) %>
  int option = 1;
<% elif (x == 2) %>
  int option = 2;
<% else %>
  int option = 0;
<% end %>
```

generates code that declares and initializes an integer variable with a given value.

### 3.8.8. *foreach* action

The *foreach* action allows looping over the elements of a collection. For example,

```
<% foreach Classifier x in context.self.ownedElements %> <%begin%>
  class <% exp x.name.body %> {
  }
<% end %>
```

generates code for every classifier from collection *context.self.ownedElements*.

### 3.8.9. Namespaces

A XTL program consists of one or more translation units, each contained in a separate source file. When a XTL program is processed, all of the translation units are processed together.

Thus, translation units can depend on each other, possibly in a circular fashion. A translation unit consists of zero or more import directives followed by zero or more declarations of templates.

```
<% import java::util %>
<% import lib %>
<% namespace templates::java %>
<% template generateClass () %> <% begin %>
class <% exp context.name %> {
 . . .
}
<% end %>
```

The concept of namespace was introduced to allow XTL programs to solve the problem of names collision that is a vital issue for large-scale transformation systems. Namespaces are used both as an “internal” organization system for a program, and as an “external” organization system - a way of presenting program elements that are exposed to other programs.

## 3.9. Analysis of KMF: does it meet the requirements?

This section considers how well the current version of KMF, as described in Section 3.2 and illustrated in Section 3.4, meets the requirements set out in Section 3.1. We’ll deal with each in turn.

1) **Evaluating the quality of the model.** The features that allow KMF to support this characteristics are presented in Chapter 4.
2) **Rapid and repeatable input and editing of populations.** A KMF generated prototype allows populations to be input through the GUI, which can then be saved as XMI and reloaded. The API of the generated code is readily accessible, and it is possible to customise the default Startup class to initialise populations from code. It is also possible to customise the generated code with bespoke lifecycle classes that include bespoke builder methods. These methods can be accessed through the GUI or, of course, in code. The use of customised startup classes, in combination with bespoke lifecycle-builders, is a particularly efficient and repeatable way of setting up many sophisticated populations. As constraint checking can be invoked through the API, this also provides a scaleable approach to automated testing of the metamodel through the generated prototype. By way of contrast, the USE tool [RG00], which also supports OCL checking over UML models, only supports instantiation of a model (which could be a metamodel) by drawing object diagrams through the GUI or by feeding in a

text representation of an object diagram read from a file. Not only is this inefficient it is also error-prone and does not lend itself to automated testing.

3) **Viewing and exploring populations.** A KMF generated prototype allows populations to be explored, viewed, and edited through the GUI. We have also found that the dynamic evaluation of OCL expressions provides a convenient way of navigating the population.

4) **Evaluation of well-formedness constraints over populations.** A KMF generated prototype allows all the constraints over the metamodel to be evaluated, either using the API or through the GUI. Selective evaluation is supported through the GUI and, of course, through the API. Selective evaluation could be improved by, for example, allowing constraints to be evaluated on all objects obtained by walking the containment tree from a particular starting object.

5) **Model transformation.** KMF supports the transformation language called YATL (Yet Another Transformation Language), presented in Chapter 5.

6) **Smooth process.** The process of using KMF requires one to have a Java development environment (such as Eclipse), for compiling and executing the generated code, and a UML modelling tool, such as Poseidon, for editing the metamodel. We have found that, with all three tools open at the same time, the process is fairly quick and smooth. The inclusion of projects in KMF studio, has meant that regeneration of code is usually no more than a couple of mouse-clicks away. KMF can also be launched from the command line or within Eclipse to generate code, given a particular project file as argument.

7) **On-the-fly evaluation of constraint expressions.** The KMF generated prototype supports on-the-fly evaluation of OCL expressions, through the GUI or the API.

8) **Round-trip engineering.** We have organised the generated code so that it is possible to customise the code in ways that ensures hand written code is not overwritten on regeneration. We are aware that other frameworks, such as EMF [EMF], do a better job of this, largely because of the substantial support for Java (parsers and the like) provided by Eclipse.

9) **Model versioning.** It is possible to change the name of the model before code generation takes place, which means that if the model name provided contains version information, it can be overridden. Another issue here is how to port populations of a previous version of a metamodel to a new version, where the new version refactors the metamodel in significant ways. If the test populations are set up using code, then it can require the code to be updated to take account of refactoring. We are have also found that, if the refactoring is not too major, the generated XMI readers are robust enough to load populations of previous versions, even if there is some information that cannot be understood.

## 3.10. Conclusions

There is a need for tools to support the activity of modeling and metamodeling, per se, especially since metamodeling is being used to define major industry standards such as UML. This chapter has identified a set of requirements for such tools, based in the idea of generating a prototype modeling tool from a metamodel. It has described the Kent Modeling Framework, which can be used to generate prototypes, in a way that meets most of these requirements.

The prototyping tool generation facility offered by KMF is actively being used in the construction and development of a number of meta-models. In particular, KMF is being used in two research projects at the University of Kent to prototype modeling tools. The first is a project entitled “Reasoning with Diagrams” RWD, which is tasked with developing tools to support reasoning with mixed visual/textual constraint languages, employing fragments of UML, OCL and constraint diagrams. The second is a project entitled “Design Support for Distributed Systems (DSE4DS)” [DSE4DS], which is building tools to support the model driven development of distributed systems. Potentially, the tool could be used to test and validate the new UML 2 and MOF 2 standards.

The grand vision for KMF is to move beyond the generation of prototypes to the generation of industrial strength modelling tools. We are beginning to investigate the generation of graphical and textual editors from appropriately extended metamodel definitions, and even the generation of semantic analysis tools. See [ASP03], for early ideas in this direction. [ASP03], [Pat04a], [Pat04b], and [Pat04c] also discuss issues with the definition and (automated) implementation of mappings between modelling languages, a keystone of the MDA edifice.

# Chapter 4. MODEL QUALITY MEASURING

Software metrics are a useful means for evaluating the quality of both software development processes and software products. With the growing popularity and adaptation of object-oriented programming languages and object-oriented methodologies in software development, the existence of specific and effective software metrics for object-oriented characteristics is essential to the improvement of software development. To obtain the design metrics of the software product most of the existing approaches measure the metrics by parsing the source code of the software product. Such approaches can be performed only in the late phases of the software development and hence cannot directly affect the design process.

In this chapter, we present the framework provided by KMF-Studio to support the computation of software metrics at the early stages of software development from UML specifications. This is important especially in OMG's Model Driven Architecture framework for software development. As models are used to drive the entire software development process it is unlikely that high quality software will be obtained using low quality models.

The current version of KMF-Studio uses UML diagrams exported in XMI files and computes OO metrics that have been shown to be good indicators for evaluating the quality of object-oriented systems (e.g. [CK91][CK94]). It also provides a set of forty-four original metrics that can be computed to measure a given UML model. This set of metrics measure both the internal attributes of UML models (e.g. inheritance depth tree and inherited complexity of a class) and the external attributes of UML models (e.g. maintainability and changeability). The user can select a set of predefined metrics to evaluate, but he cannot change the way the values of the metrics are computed. The tool also allows the modeller to extend the set of existing metrics by defining new metrics using OCL and choose only a subset of the predefined metrics. The result of evaluating the metrics over a model can be used to identify

the weak points of UML models and give on the fly diagnostics about the design quality of the model.

This chapter is organized as follows. The first section gives a brief description of the background and existing object-oriented metrics. The second describes how UML models are measured in KMF, describing the problems of UML model measuring, the proposed set of metrics, and the measuring methodology used in KMF-Studio. The third section gives an example. The last section contain the conclusions and future work.

## 4.1. Background

Measuring has a long tradition in the area of natural sciences. At the end of the 19th century, the great physicist Lord Kelvin said the following about measuring:

> *"When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind".*

Measuring has been studied in the area of software engineering for about thirty years. The size of the costs for the development and maintenance of software products amplifies the need for a theoretical foundation for software developing standards and management decisions, using measurements. In 1980 Curtis [Cur80] stated that in order to transform programming into an engineering discipline, software products must be developed using sound scientific methods. The foundation of these methods requires the development of measuring techniques and establishing the cause-effect relations.

The need for software measurements is presented very clearly in [GC87][Gra90]. Software metrics can be used to measure attributes not only of software products, but also software development processes.

The true value of a software metrics suite comes from their capability to measure important external attributes [ISO96]. An external attribute is measured according to the way the software product interacts with its environment [Fen91]. Testability, reliability, portability, and maintainability are examples of external attributes. However, these attributes can be measured directly only quite late in the software development process. Therefore, software

metrics can be used to offer good indicators regarding important external attributes. For example, if we know that keeping the depth of the inheritance tree within some limits ensures good maintenance, we can optimize inheritance during design because we know that in doing so we are reducing the costs of maintenance as far as it is linked to the depth of inheritance tree.

In the last years much effort has been spent in the software engineering research community in developing software metrics both for procedural and object-oriented system. Usually, these software metrics compute the value of internal attributes of the software systems (e.g. number of lines of code, number of variables used and number of parameters). After a metrics suite has been designed, the relationship between the metric values and the external attributes needs to be studied. This process, called the validation of the metric, is usually performed using empirical studies [ET02]. For example [WH98] provides an interpretation and critique of [CK94] metrics, including the use of two traditional metrics ([McC76] and [Hal77]) by observing the evolution, over a two and a half year period, of one commercial grade C++ application comprising 114 classes with 25,000 lines of code. Once a set of metrics has been validated, software companies and programmers can use it as a guideline for the software development process.

### 4.1.1. An overview of object-oriented metrics

A considerable number of object-oriented metrics have been developed to measure the quality of software; for example see [FBC94], [BM99], [BDM97], [CS00], [CK91], [CK94], [HS96], [LI93], [LK94], and [TKC99]. By far, the most popular of these is the metric suite developed by Chidamber and Kemerer [CK94], known as the CK metrics. For historical reasons the CK metrics are the most referenced ones, being easy to compute and useful, and many commercial tools compute these metrics. Another comprehensive set of metrics that capture important structural characteristics has been defined by [BDM97]. The CK metrics have also received a considerable amount of empirical study. A summary of the CK metrics can be found in Table 4.1.

*Table 4.1. Summary of CK metrics*

| Metric Acronym | Description |
|---|---|
| DIT | [CK94] defines this metric as follows: "the depth of inheritance of the class is the DIT metric for the class. In cases involving multiple inheritance, the DIT will be the maximum length from the node to the root of the tree". |
| NOC | The number of children metric is defined as the "number of immediate subclasses subordinated to a class in the class hierarchy" [CK94] |
| WMC | The weighted method per class metric is defined as the sum of the complexity of methods in a class. |
| RFC | The response for a class metric measures the cardinality of "a set of methods that can potentially be executed in response to a message received by an object of that class" [CK94]. A variant of RFC excludes methods indirectly invoked by a method of a class [CK91]. |
| CBO | The coupling between objects metric is defined as "a count of the number of other classes to which it is coupled". A class is coupled to another if it uses the member functions and/or instance variables of the other class. [CK94]. |
| LCOM | The different definitions of the lack of cohesion in methods metrics were given by [CK91] and [CK94]. The original definition of LCOM metric measures the number of disjoint sets of a class' local methods as indicated by their access to class variables [CK91]. The LCOM metric was later revised and a new definition was given [CK94]. The revised LCOM metric measures the number of pairs of methods in the class that have no attributes in common, minus the number of pairs of methods that do. If the difference is negative, the metric is set to zero. |

The higher the DIT values are, the harder it is to predict the behaviour of a class due to interaction between inherited and local features. High NOC values may indicate an appropriate abstraction in the design while moderate NOC values indicate the scope for reuse of behaviour and features. The DIT and NOC metrics measure the shape and size of the class

structure. Well-designed object-oriented systems tend to be built as forests of classes, rather than one very large inheritance tree. [CK94] states that such forests of classes should not be deeper than seven classes and not wider than seven classes.

The WMC metric can be measured using different weighting functions and traditional complexity metrics (e.g. number of lines of code, McCabe's cyclomatic complexity [McC76], number of decision points, and number of paths from the entry points to exit points) to measure the complexity of methods. If all the methods are considered to have the same complexity, equal to one, the metric is called WMC1 and represents the number of methods. The WMC1 metric can be used to evaluate the effort that a user has to make in order to use the class properly, while WMC can be used to evaluate the effort to understand and maintain the class.

High CBO values may indicate a poor encapsulation and a low reusability. The idea behind CBO is that a software system with higher CBO values is error-prone as the behaviour of a class is affected by the activities performed by other coupled classes. High values for RFC indicate that the number of classes that could potentially respond to a message is high, hence it measures the complexity of the class. High LCOM values may indicate high complexity of classes, inappropriate abstraction, and poor encapsulation.

As these metrics measure the software at the source level and not at the model level, they cannot be used in the early stages of the software development processes. [TC02] presents a methodology that can be applied to UML specifications to obtain design information and to compute the design metrics at an early stage of software development. It proposes, in addition to the CK metrics, a set of four classes of metrics that can be used to further evaluate the complexity of OO designs.

In order to adapt OO metrics to models, we propose another metric suite, which is particularly suitable for OMG's MDA framework. The new metric suite is defined in the next section.

## 4.2. Measuring UML models in KMF-Studio

This section contains a description of particularities of software measurement on UML models, proposes a set of metrics to measure UML models, and presents the methodology and the framework provided by KMF-Studio to measure UML models.

### 4.2.1. Measuring UML models

Evaluating the *quality* of UML models is very important in the framework of MDA, as UML models are the key concepts in the software development process using MDA techniques. A system derived from a poorly designed model, although it can be built quickly to process the inputs correctly, may cost more in the long run because of the additional costs of the maintenance. Thus, improving the quality of models is a major research goal in software development using MDA. This goal will be difficult to achieve unless we can define and measure the components of model quality. In a restricted sense, the quality of a software product is often considered synonymous with the presence or absence of errors. However, most users disregard or do not consider that other software attributes, such as the effort to understand, use and modify software, should have high quality. The same also goes for models: the quality of a model is evaluated using external attributes such as complexity, maintainability and reliability.

UML has gained great popularity both in the software design process and the whole software development lifecycle. In order to apply software metrics early in the software lifecycle, object-oriented metrics should be incorporated into UML modeling tools. This ensures that object-oriented metrics can be applied both in high-level design and more detailed design phases. Most commercial software development tools only apply object-oriented metrics at the source code level, although some tools, such as TogetherSoft [TOG], provide support for the evaluation of object-oriented metrics for a given UML diagram.

UML uses specific diagrams such as class diagrams, collaboration diagrams, and activity diagrams to describe specific views of a system. A static diagram describes the internal structure of a class and relationships among classes (e.g. attributes, operations, associations and generalizations). A collaboration diagram describes the dynamic structure of the system,

the objects that interact and the messages that are exchanged between the objects, the sequence of messages in time and the roles of objects contained within the system. The transitions within a model element, which are triggered by events, are described using activity diagrams.

Computing metrics for only one type of UML diagram is imprecise. Computing object-oriented metrics for class diagrams can be useful to measure the static structure of software systems, but will not capture the dynamic structure of the system. Developing metrics to measure all the UML specific diagrams is required. On the other hand, every UML model potentially contains OCL constraints that are attached to model elements. Hence software metrics to evaluate attributes of the OCL expressions are required (e.g. number of variables used in an OCL expression and the complexity of an OCL expression). In conclusion, in order to measure effectively a UML model one needs to consider software metrics for all the elements that are present inside the model.

To see how object-oriented metrics need to be changed in order to measure various attributes of models each metric needs to be analyzed separately. [TC02] gives a study of the CK suite. The results of this study are presented below:

- The DIT and NOC metrics from the CK suite, which measure the static structure of the software systems, can be computed easily from class diagrams. Due to lack of information describing the body of methods in UML models measuring WMC using for example the McCabe cyclomatic complexity is not possible, as the body of the methods is not always specified. Instead we can compute WMC1 or we can consider the complexity of a method to be proportional to the number of parameters including the returned type, as UML models contain a description of each method's signature.
- To evaluate the CBO metric on UML models, two issues need to be resolved: the unit used for the measurement and the definition of the coupling concept. Although there is no difficulty in proposing the "class" as the unit for the metric, because the metric measures how many classes are coupled with a given class, there is no standard definition of the coupling concept for object-oriented systems. There are, however, different forms of coupling such as inheritance, coupling by association, by attributes, or by message passing.
- The RFC metric measures the response of a UML class. Hence, to compute the metric for a given class one needs access to the methods that are defined inside a class and to methods that are invoked by these methods. Methods can be accessed easily from the UML class diagram, but counting the number of methods from other classes invoked in a given method, requires a precise description of the interaction among classes, which is not described in class diagrams.

- The revised LCOM metric measures the number of pairs of methods in the class that have no attributes in common, minus the number of pairs of methods that have attributes in common. As the information on the use of instance variables inside the body of an operation is not available at the early stages of the development, only parameters can be used as input data to evaluate the metric. However, when the model contains more details about the dynamic behavior of the system, such as activity diagrams, reasonable values for LCOM metrics can be computed.

### 4.2.2. The KMF metrics suite

This section contains a description of the metrics that we have designed to measure the quality of UML models in KMF-Studio.

The set of metrics was designed to achieve the following objectives:

- Measure both internal attributes (e.g. number of methods declared in a class) and external attributes (e.g. maintainability).
- Measure all the types of elements present in a UML model: model, namespaces, classes, and OCL expressions.
- Measure all the relations that are present in UML models: inheritance and associations.
- Measure the nesting of containment elements: model, namespaces, and classes.
- Measure the complexity of classes and methods.
- Measure the complexity of OCL expressions by adapting well-know metrics used for procedural languages.
- Measure the average of relevant metrics (e.g. complexity of class).

The metrics are organized on two levels. The first level contains metrics to measure the internal attributes of the model (e.g. number of local methods and the height of the inheritance graph). The second level contains metrics to measure external attributes of the model such as testability and maintainability. They are also structured on several levels, according to the type of OO element that is measured: model, namespace, class, and OCL level. The metrics are summarized briefly in. Table 4.2 and Table 4.3.

*Table 4.2. KMF metrics suite- first level*

| Metric Acronym | Metric Name | Description |
|---|---|---|
| MODEL-HNT | Height of Nesting Tree | Measures the vertical nesting of namespaces in the model. |
| MODEL-HIG | Height of Inheritance Graph | Measures the maximum height of the inheritance graph, considering all the connected components. |
| MODEL-NCN | Number of Contained Namespaces | Measures the size of namespace nesting in the model. |
| MODEL-ANCPN | Average Number of Classes Per Namespace | Measures the horizontal nesting of namespaces in the model. |
| MODEL-ADIG | Average Depth of Inheritance Graph | Measures the average height of connected parts of the model's inheritance graph. |
| MODEL-ACC | Average Class Complexity | Measures the average complexity of classes in the model. |
| MODEL-AMC | Average Method Complexity | Measures the average complexity of the methods within the model. |
| MODEL-AOCC | Average OCL Constraint Complexity | Measures the average complexity of OCL constraints in the model. |
| NS-NDCN | Number of Directly Contained Namespaces | Measures the horizontal nesting of the namespace. |
| NS-NCN | Number of Contained Namespaces | Measures the size of the nesting of the namespace. |
| NS-NDCC | Number of Directly Contained Classes | Measures the local dimension of the namespace. A dimension means a measurement of the model in a particular direction (e.g. the number of included namespaces). |
| NS-NCC | Number of Contained Classes. | Measures the global dimension of the namespace. |
| NS-DNT | Depth of Nesting Tree | Measures the nesting level of the namespace. |

| | | |
|---|---|---|
| CLS-NLP | Number of Local Properties | Measures the local dimension of local properties. |
| CLS-NP | Number of Properties | Measures the dimension of all the properties. |
| CLS-NLO | Number of Local Operations | Measures the dimension of local operations. |
| CLS-NO | Number of Operations | Measures the dimension of all the operations. |
| CLS-ACLO | Average Complexity of Local Operations | Measures the average complexity of local operations. |
| CLS-ACO | Average Complexity of Operations | Measures the average complexity of all the operations. |
| CLS-DIG | Depth of Inheritance Graph | Measures the inheritance level of the class. |
| CLS-NDA | Number of Direct Ancestors | Measures the dimension of local ancestors. |
| CLS-NA | Number of Ancestors | Measures the dimension of all ancestors. If a class is inherited more than once, it counts all its appearances. |
| CLS-NDD | Number of Direct Descendants | Measures the dimension of local specialization. |
| CLS-ND | Number of Descendants | Measures the dimension of all specializations. |
| CLS-NMI | Number of Multiple Inheritances | Measures the dimension of repeated inheritances. |
| CLS-NRDC | Number of Referred Classes. | Measures the dimension of references to other classes. |
| CLS-NRE | Number of Referees | Measures the dimension of references from other classes. |
| CLS-LC | Local Complexity | Measure the local complexity of the class. |
| CLS-C | Complexity | Measures the global complexity of the class. |
| OPER-MCC | McCabe Complexity | Measures the McCabe complexity. |
| OPER-NP | Number of parameters | Measures the dimension of the prototype |

| | | |
|---|---|---|
| | | associated to a method. |
| OCL-NDP | Number of Decision Points | Measures the complexity of the methods using decision points. |
| OCL-HNT | Height of Nesting Tree | Measures the nesting of the OCL constraints. |
| OCL-MCC | McCabe complexity | Measures the McCabe complexity of the OCL constraint. |
| OCL-HALC | Halstead Complexity | Measures the Halstead complexity of the OCL constraint by computing the total number of operator occurrences and total number of operand occurrences. |
| OCL-NV | Number of Variables | Measures the complexity of the OCL constraint. |

*Table 4.3. KMF metrics suite-second level*

| **Metric Acronym** | **Metric Name** | **Description** |
|---|---|---|
| MODEL-MAIN | Model Maintainability | Measures the effort required to maintain the model. |
| MODEL-CHAN | Model Changeability | Measures the effort required to change a model. |
| MODEL-TEST | Model Testability | Measures the effort required to test the system described by a model. |
| CLS-MAIN | Class Maintainability | Measures the effort required to maintain the class. |
| CLS-ANAL | Class Analyzability | Measures the effort required to analyze the class |
| CLS-CHAN | Class Changeability | Measures the effort required to change the class |
| CLS-STAB | Class Stability | Measures the stability of the class after changing partially some of the inner components |

| | | |
|---|---|---|
| CLS-TEST | Class Testability | Measures the effort required to test the class. |
| CLS-USAB | Class Usability | Measures the effort required to use the class. |
| CLS-SPEC | Class Specialization | Measure the effort required to specialize the class. |

The metrics suite and the way the metrics are used to measure the quality of model elements is described in more details in Appendix 4. New metrics can be added in an XML style using OCL. For example,

```
. . .
<metric namespace='OCL' key='OCL-NDD'
        name='Number of Direct Descendants'
        type='ocl' min='0' max='POSITIVE_INFINITY'>
   <body>
        context uml::Foundation::Core::Class inv ndd:
             self.specialization->size()
   </body>
   <diagnostic>
        Reduce the number of direct children
   </diagnostic>
</metric>
. . .
```

computes the number of direct ancestors of a class.

More details about these metrics, including a brief description of the algorithms and proposed boundaries are presented in Appendix 4.

### 4.2.3. Methodology

In the KMF software suite metrics are collected on individual components of a single model. Predictions given by elementary KMF metrics on individual model elements are then composed in global KMF metrics to give predictions for the entire system. The same approach was taken by [EBGR01] to predict the proportion of faulty classes in a whole system. [BDW99] used object-oriented metrics to predict the effort to develop each class, and these were then composed to produce an estimate of the overall system. Both [EBGR01] and [BDW99] consider the implementation level and the modeling level.

The metrics are collected and composed in KMF into *quality models.* The results of measurement are also used to classify components according to their quality category into excellent, good, acceptable, and poor using the proposed boundaries and accepted deviations. Once instantiated a quality model takes as input the values of a set of metrics ($M_1$, $M_2$, …, $M_n$) for a particular model element, and computes its quality category. An overview of the quality model behavior is given in Figure 4.1. The quality model is described in more detail in Appendix 4.

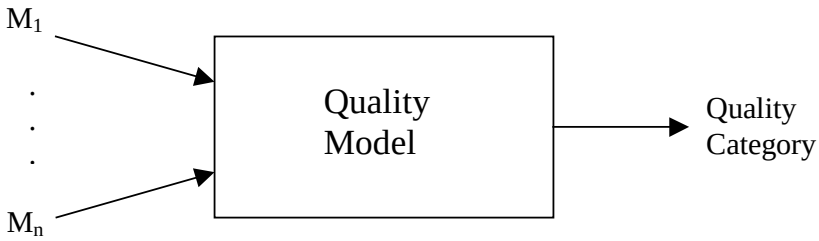



*Figure 4.1. Quality model*

## 4.3. An example

This section contains the description of an experiment that was performed in order to illustrate our methodology. The validation of the proposed metrics is outside the scope of this experiment.

The methodology presented above was applied on the OCL model that is fully described in [ALP03]. Figure 4.2, Figure 4.3, and Figure 4.4 show two of the class diagrams used to describe the OCL expression. With these diagrams, KMF-Studio computes the selected metrics for each class and displays it using Kiviat diagrams, pie charts and HTML, as shown in Figure 4.5. We decided to use this mechanism to visualize the result of the measurement as it provides excellent visual feedback regarding the critical points of the design. Using the values computed for the metrics that measure the internal attributes, KMF-Studio generates an HTML quality report that evaluates the maintainability of the system, as shown in Figure 4.6. The report displays, for each model element, the value of the metrics and groups the elements into several categories: Excellent, Good, and Acceptable (see Appendix 4). Both the value of the metrics associated with a model element and the final quality report are

generated by KMF-Studio in HTML format as HTML allows quick navigation. To draw a Kiviat diagram and a pie chart, KMF-Studio generates HTML text that contains applet invocations with given arguments. The final quality report contains a pie chart that describes the percentage of excellent, good, acceptable, and poor elements, according to the criteria specified in Appendix 4. One can identify the elements that violate the boundaries of attached metrics by following the provided HTML links. To provide useful feedback the violations are displayed in Kiviat diagrams using colors and visual effects.

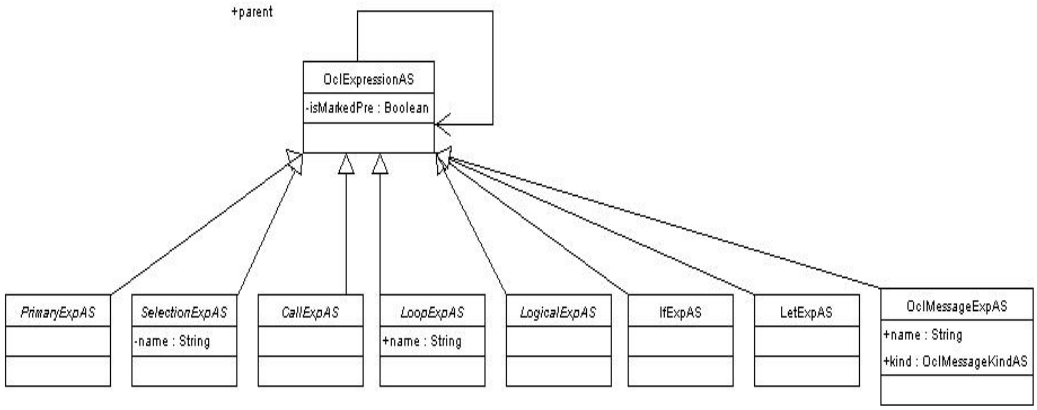



*Figure 4.2. OCL expressions*

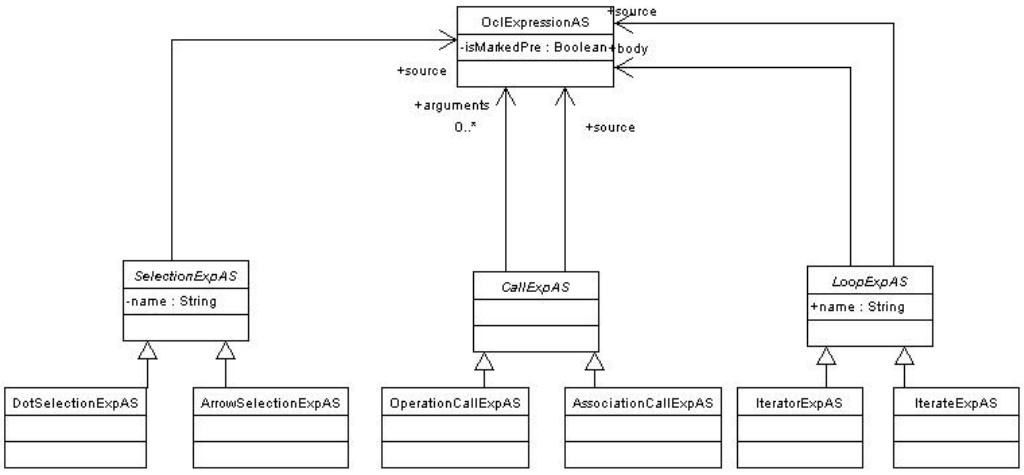



*Figure 4.3. OCL selection, call, and loop expressions*

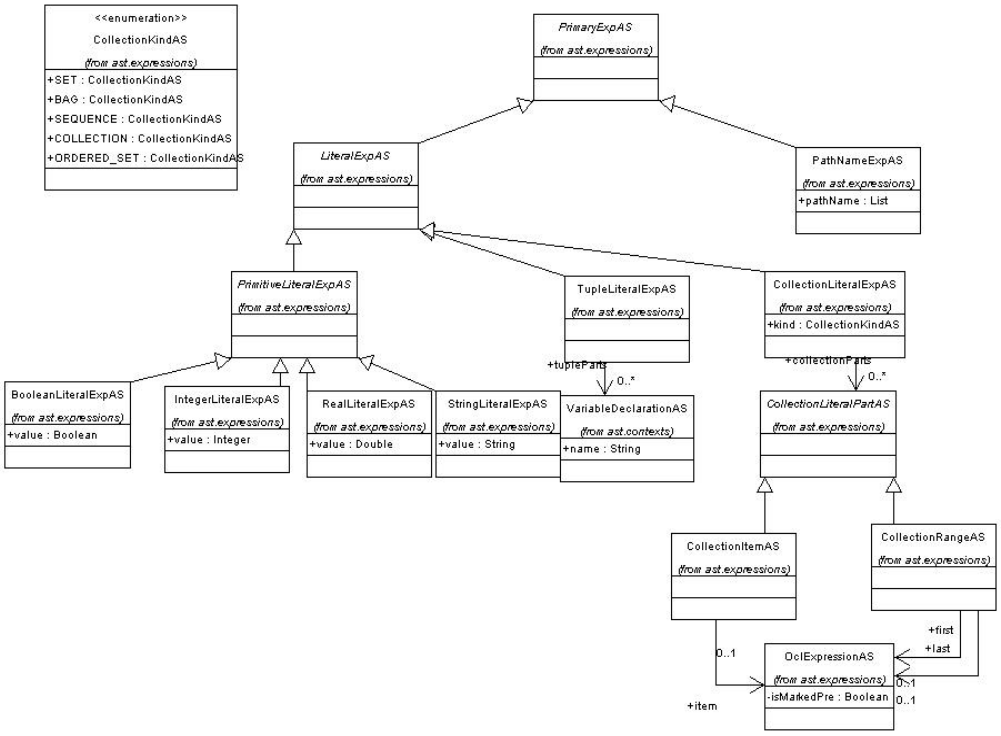
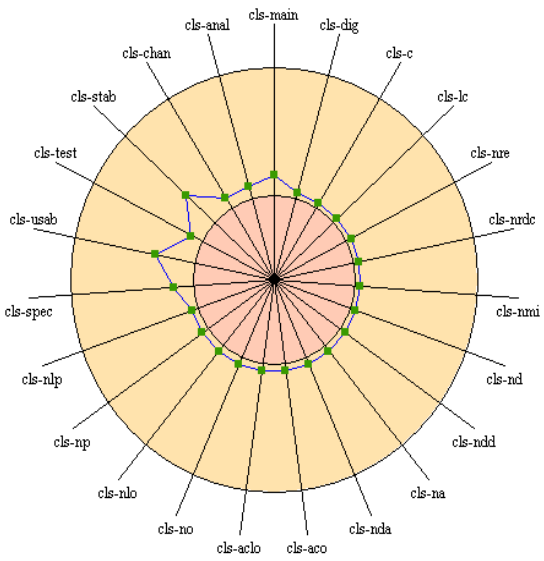
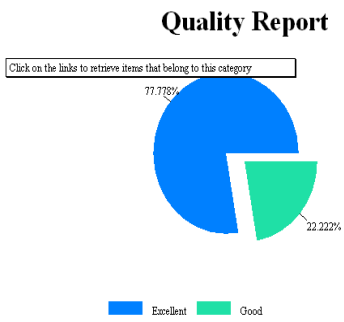



*Figure 4.4. OCL Primary expressions*




| Metric namespace | Metric name | Metric key | Minimum value | Maximum value | Value |
|---|---|---|---|---|---|
| KMF | Class Maintainability | cls-main | 0 | 400 | 47 |
| KMF | Class Analysability | cls-anal | 0 | 100 | 6 |
| KMF | Class Changeability | cls-chan | 0 | 100 | 6 |
| KMF | Class Stability | cls-stab | 0 | 100 | 29 |
| KMF | Class Testability | cls-test | 0 | 100 | 6 |
| KMF | Class Useability | cls-usab | 0 | 10 | 3 |
| KMF | Class Specilizability | cls-spec | 0 | 25 | 3 |
| KMF | Number of Local Properties | cls-nlp | 0 | +oo | 3 |
| KMF | Number of Properties | cls-np | 0 | +oo | 3 |
| KMF | Number of Local Operations | cls-nlo | 0 | +oo | 0 |
| KMF | Number of Operations | cls-no | 0 | +oo | 0 |
| KMF | Average Complexity of Local Operations | cls-aclo | 0 | +oo | 0 |
| KMF | Average Complexity of Operations | cls-aco | 0 | +oo | 0 |
| KMF | Number of Direct Ancestors | cls-nda | 0 | +oo | 0 |
| KMF | Number of Ancestors | cls-na | 0 | +oo | 0 |
| KMF | Number of Direct Descendants | cls-ndd | 0 | +oo | 8 |
| KMF | Number of Descendants | cls-nd | 0 | +oo | 29 |
| KMF | Number of Multiple Inheritances | cls-nmi | 0 | +oo | 0 |
| KMF | Number of Referred Classes | cls-nrdc | 0 | +oo | 9 |
| KMF | Number of Referees | cls-nre | 0 | +oo | 0 |

*Figure 4.5. Kiviat diagram for class OclExpressionAS*




**Excellent**

**Good**

*Figure 4.6. Quality report for OCL expressions*

## 4.4. Conclusions and future work

This chapter's contribution is the presentation of the framework provided by KMF-Studio to support software measurement from UML models. The framework structures the measurement on two levels. KMF-Studio measures at the first level the internal attributes of models (e.g. depth of inheritance graph, number of operations, number of properties, and average complexity of OCL constraints). The second level is responsible for measuring the external attributes of the software system (e.g. maintainability, testability, changeability, and usability). The resulting quality report can be used to identify model elements that violate the boundaries of metrics and thus provides an indication of the elements that are likely to consume most of the cost of implementation and maintenance. The quality evaluation system that we designed and implemented is usable, flexible, and extensible. For example, a user can choose the set of metrics that is used to prepare a quality report, either by choosing some of the predefined metrics or writing its own metrics written in OCL. An OCL metric navigates the model and computes a numeric value.

We are currently working to incorporate the measures of other software metrics in KMF-Studio. The intention is to extend the set of predefined metrics to include other well-known metric suites such as [LH93][TC02]. We also intend to provide support for measuring other

elements that appear in the UML2.0 standard (e.g. stereotypes, sequence diagrams, and activity diagrams) and discover which metrics are worth using.

# Chapter 5. YATL SPECIFICATION

This chapter presents the current version of YATL (Yet Another Transformation Language), which is evolving in order to support all the features provided by [QVT02] and the future QVT standard. The first subsection provides a quick overview of the YATL language. Subsequent sections present the features of YATL in more details.

## 5.1. YATL Overview

YATL is a hybrid language (a mix of declarative and imperative constructions) designed to answer the Query/Views/Transformations Request For Proposals [QVT02] issued by OMG and to express model transformations as required by the MDA [MDA] approach.

YATL formulates queries to interrogate the model using constructions from the OCL 2.0 standard. A YATL query is a syntactic construct that contains the description of the request in terms of OCL 2.0 (see Appendix 6). The YATL processor invokes the OCL processor to process the query and supply the results of interrogation.

A YATL transformation describes a mapping between a source MOF metamodel S, and a target MOF metamodel T. The transformation engine uses the mapping to generate a target model instance conforming to T from a source model instance conforming to S. The source and the target metamodels may be the same metamodel. Navigation over models is specified using OCL.

Each transformation contains one or more transformation rules. A transformation rule consists of two parts: a left-hand side (LHS) and a right-hand side (RHS). The LHS of a YATL transformation is specified using a filtering expression written either in OCL or native code such as Java, C#, and scripts. This approach allows filter expressions to include both modeling information (such as navigational expressions, properties values, collections) and

platform dependent properties (such as special conversion functions), which makes them extremely powerful. A compound action specifies the effect of the RHS. The LHS and RHS for the YATL transformation are described in the same syntactical construction, called a transformation rule. A rule is invoked explicitly using its name and with parameters.

The abstract syntax of YATL namespaces, translation units, queries, views, transformations, and transformations rules is described in Figure 5.1.

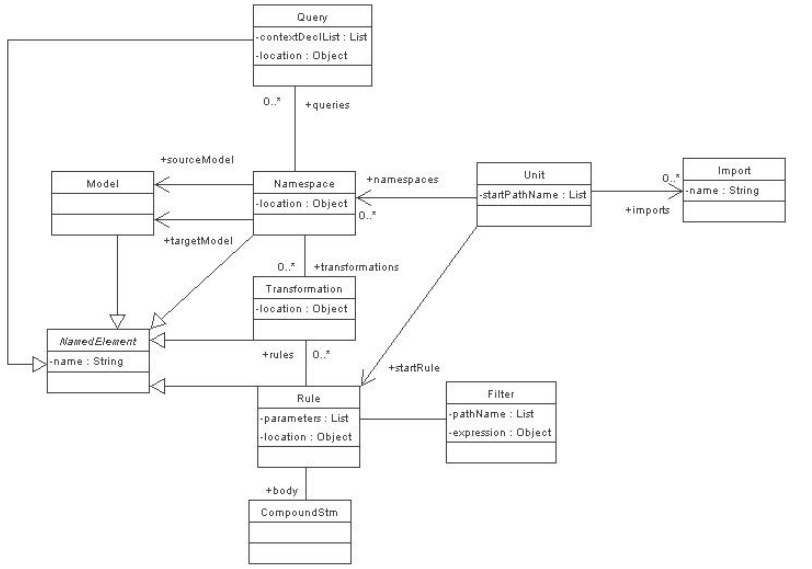



*Figure 5.1 Abstract Syntax*

## 5.2. An example

Let us consider the following two models:

- Model *M1* contains class *A*.
- Model *M2* contains class *B*.
- Class *A* has a property called *name*.
- Class *B* has a property called *value*.

and the transformation rule:

> "For each instance of class A in M1, which is named John, create an instance of class B with a value property equal to 5".

The YATL program in Figure 5.2 expresses the above transformation.

```
start kmf::edoc2ws::main;

namespace kmf(m1, m2) {
    transformation m1Tom2 {
        --
        -- A to B
        --
        -- Map an A to a B
        rule a2b match m1::A[self.name = 'John'] () {
            -- Create B
            let b: m2::B;
            b := new m2::B;
            b.value := 5;
        }
        -- main rule
        rule main () {
            -- Map individual elements
            apply a2b();
        }
    }
}
```

*Figure 5.2 A transformation example in YATL*

The YATL program starts with the invocation of the rule *main*, which invokes rule *a2b*. The rule iterates over all the instances of *A* in *M1*'s repository and filters them using the OCL expression *self.name = 'John'*. If the filter returns true, the body of rule *a2b* is used to build the corresponding instance of B and set the value to 5; otherwise the rule does nothing.

### 5.2.1. Main features

The declarative features come mainly from OCL expressions and the description of the LHS of transformation rules. YATL acts in a similar way to a database system that uses SQL to interrogate the database and the imperative host language to process the results of the query.

We choose OCL to describe the matching part of YATL rules because it is a well defined language for querying the UML models. It provides a standard library with an acceptable computational expressiveness, it is a declarative language, and it is a part of the OMG's standards.

YATL supports several kinds of imperative features, used in the RHS of transformation rules, which are presented later in this chapter. These features were selected so that YATL can provide lifecycle operations like creation and deletion, operations to change the value of properties, declarations, decisions, and iteration actions, native actions to interact with the host machine, and build actions to ease the construction of target model instance. Compound actions contain a sequence of instructions, which are to be executed in the given order. These syntactic constructions make use of OCL expressions to specify basic operations such as adding two integer values. YATL uses the same type system as OCL 2.0 [OCL].

YATL is described by an abstract syntax (a MOF metamodel) and a textual concrete syntax. It does not yet have a graphical concrete syntax as QVT RFP suggested. A transformation model in YATL is expressed as a set of transformation rules. Transformations from Platform Independent Models (PIMs) to Platform Specific Models (PSMs) can be written in YATL to implement the MDA.

A YATL transformation is unidirectional. We believe that a model transformation language should be unidirectional, otherwise it cannot be used for large scale models. The main difficulty with a bidirectional transformation language is that it needs some reasoning to perform the transformation. For example, DSTC's proposal [QVTD] uses mechanisms similar to Prolog-unification to perform a bidirectional mapping. The reverse transformation can be described as any other transformation using YATL.

For a real model-to-model transformation, traceability is necessary to make the approach workable. To trace the mapping between source and target model instances, YATL comprises an operator called *track*. Track expressions are, from the concrete syntax point of view, similar to DSTC's track constructions [QVTD]. The main difference is that YATL's tracks are defined using concepts like relation name, domain, and imagine, and not Prolog-like concepts (e.g. unification). This approach makes the traceability system of YATL suitable for large-scale systems.

## 5.3. Programs

A YATL *program* consists of one or more source files, known formally as *translation units*. A source file is an ordered sequence of Unicode standard characters. Conforming implementations must accept Unicode source files encoded with the UTF-8 encoding form [UNI], and transform them into a sequence of Unicode characters. Implementations may choose to accept and transform additional character encoding schemes, such as UTF-16, UTF-32, or non-Unicode character mappings.

Conceptually speaking, a YATL program is analysed in five steps:

1) Character conversion, which converts a file from a particular character repertoire and encoding scheme into a sequence of Unicode characters.
2) Lexical analysis, which translates a stream of Unicode input characters into a sequence of tokens.
3) Syntactic analysis, which translates the sequence of tokens into an abstract representation of the input structure.
4) Semantic analysis, which checks if the input follows the semantic rules, and produces an internal representation of both syntax and semantics.
5) Code generation or interpretation where the semantic representation is either used to generate code for the underlying machine or directly evaluated on the same machine.

## 5.4. Grammars

This section presents the syntax of YATL language using two grammars, structured on two levels. On the first level, the *lexical grammar* defines how Unicode characters are combined to form line terminators, white space, comments, and YATL tokens. At the second level, the *syntactic grammar* defines how the tokens resulting from the lexical grammar are combined to form YATL programs. Both grammars are described using the notation comprised in Appendix 1.

### 5.4.1. Lexical grammar

The lexical grammar of YATL is presented in Appendix 5. The terminal symbols of the lexical grammar are the characters of the Unicode character set, and the lexical grammar specifies how characters are combined to form white spaces, comments, and tokens.

The lexical processing of a YATL source file consists of reducing the file into a sequence of tokens that becomes the input to the syntactic analysis. Line terminators, white space, and comments can serve to separate tokens, but otherwise these lexical elements have no impact on the syntactic structure of a YATL program.

When several lexical grammar productions match a sequence of characters in a source file, the lexical processing always forms the longest possible lexical element. For example, the character sequence is processed as the beginning of a single-line comment because that lexical element is longer than a single token.

Every source file in a YATL program must conform to the *input* production of the lexical grammar.

### 5.4.2. Syntax grammar

The syntactic grammar of YATL is presented in Appendix 6 and the following sections. The terminal symbols of the syntactic grammar are the tokens defined by the lexical grammar, and the syntactic grammar specifies how tokens are combined to form YATL programs.

Every source file in a YATL program must conform to the *translation-unit* production of the syntactic grammar.

## 5.5. Types and variables

The types of the YATL language are derived from the OCL's types [OCL2],[AP03],[ALP03]. They can be used to encapsulate logical values, numbers, collections, tuples, and user types. The type hierarchy of YATL is described in Figure 5.3 and derives from [ALP03].

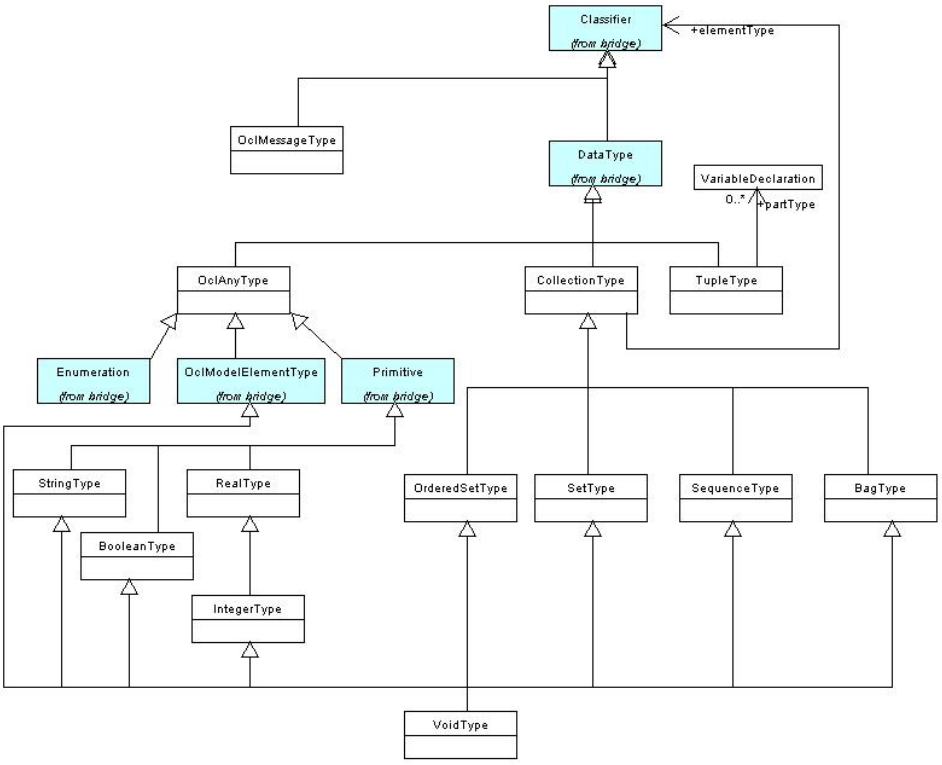



*Figure 5.3 YATL types*

YATL's type system is unified such that a value of any type can be treated as a *Classifier.* Every type in YATL directly or indirectly derives from the *Classifier* class type, which is the ultimate base class of all types. Undefined values are represented using *VoidType*.

YATL defines two categories of variables: local variables and value parameters. In the example

```
transformation T {
    rule r match java::Class (String s) {
         let i: Integer = 3;
    }
}
```

*s* is a value parameter and *i* is a local variable.

Variables represent storage locations. Every variable has a type that determines what values can be stored in the variable. YATL is a type-safe language, and the YATL processor guarantees that values stored in variables are always of the appropriate type. The value of a

variable can be changed through assignment. If the value of a variable is not specified by an initialization or assignment, it is considered to be the undefined value from OCL.

A variable must be *definitely assigned* before its value can be obtained. A variable is said to be *definitely assigned* at a given location in the executable code, if the compiler can prove, by a particular static flow analysis that the variable has been automatically initialized or has been the target of at least one assignment.

Variables are either *initially assigned* or *initially unassigned*. An initially assigned variable has a well defined initial value and is always considered definitely assigned. An initially unassigned variable has no initial value. For an initially unassigned variable to be considered definitely assigned at a certain location, an assignment to the variable must occur in every possible execution path leading to that location.

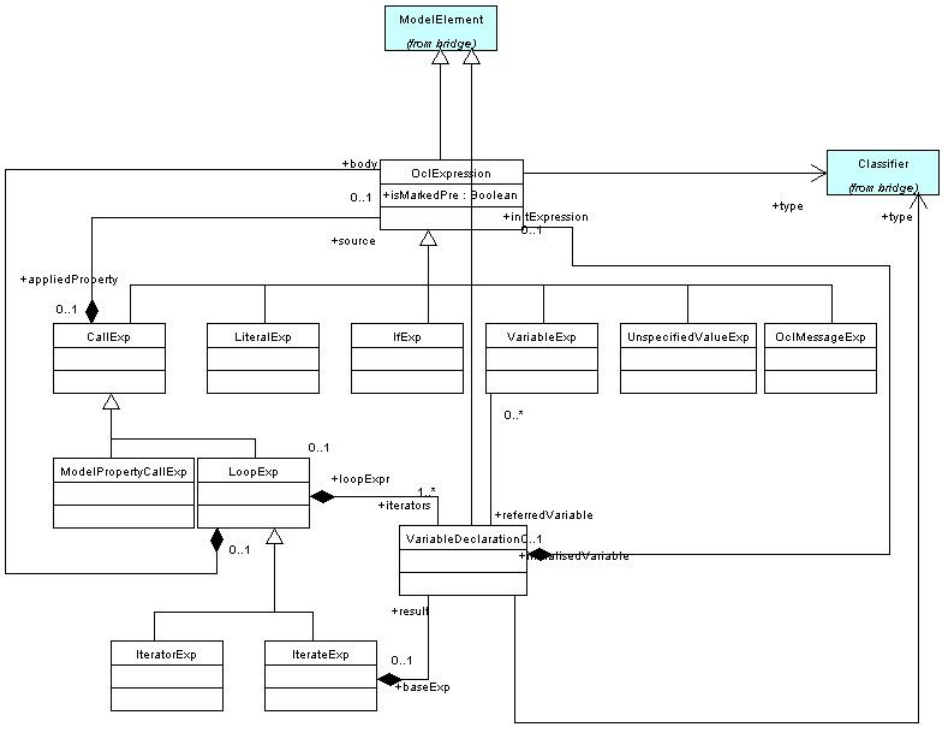



*Figure 5.4 YATL expressions*

## 5.6. Expressions

This section defines the syntax, order of evaluation of operands and operators, and meaning of expressions. YATL expressions are extensions of OCL 2.0 expressions presented in Figure 5.4 [ALP03].

More details about the expressions supported by OCL (e.g. concrete syntax, abstract syntax, and semantics) and the way they are implemented can be found in [OCL2][ALP03].

The extensions specific to YATL are presented in the following subsections.

### 5.6.1. The assignment operator

The assignment operator assigns a new value to a variable or a property.

*assignment-expression* →
    *ocl-expression* ‘:=’ *rhs-expression* .
*rhs-expression* →
    *ocl-expression* |
    *new-expression* |
    *build-expression* |
    *track-expression* .

The left operand of an assignment must be an expression classified as a variable or a property.

In an assignment, the right operand must be an expression of a type that is compatible to the type of the left operand [OCL2]. The operation assigns the value of the right operand to the variable or property given by the left operand.

The result of a simple assignment expression is the value assigned to the left operand. The result has the same type as the left operand and is always classified as a value.

### 5.6.2. The *new* operator

The *new* operator is used to create new instances of model element types [OCL2].

*new-expression* →
    *'new' path-name* .

The *new* operator implies creation of an instance of the *path-name* type.

### 5.6.3. The *build* operator

The *build* operator is used to create new instances of model element types and set their properties in the same time.

*build-expression* →
    *'build' path-name '{' list-pair '}'.*
*list-pair*→
    *λ* |
    *pair ',' list-pair* .
*pair* →
    *name ':=' rhs-expression* .

The *new* operator implies creation of an instance of the *path-name* type and sets the values for the properties specified in *list-pair*. If there is at least one *name* for which there is no such property in type *path-name,* a compile-error is reported.

### 5.6.4. The *track* operator

The track operator is used to store and retrieve mappings during and after the transformation process.

*track-expression* →
    *'track' '(' ocl-expression ',' simple-name ',' ocl-expression ')'* |
    *'track' '(' 'null' ',' simple-name ',' ocl-expression ')'* |
    *'track' '(' ocl-expression ',' simple-name ',' 'null' ')'* .

Given a relation R and two objects X and Y, the meaning of the track operator is the following:

- *track(X, R, Y)* stores the relation R(X, Y).
- *Y := track(X, R, null)* retrieves the element related to X by R.
- *X := track(null, R, Y)* retrieves the element related to Y by R.

The type of X and Y can be any OCL 2.0 type (e.g. integer, real, boolean, string, model element type, collection, or tuple).

## 5.7. Actions

This section contains the description of the actions supported by YATL and other basic concepts such as: end point, reachability, name lookup, rule resolution etc. The abstract syntax tree of YATL actions is described in Figure 5.5.

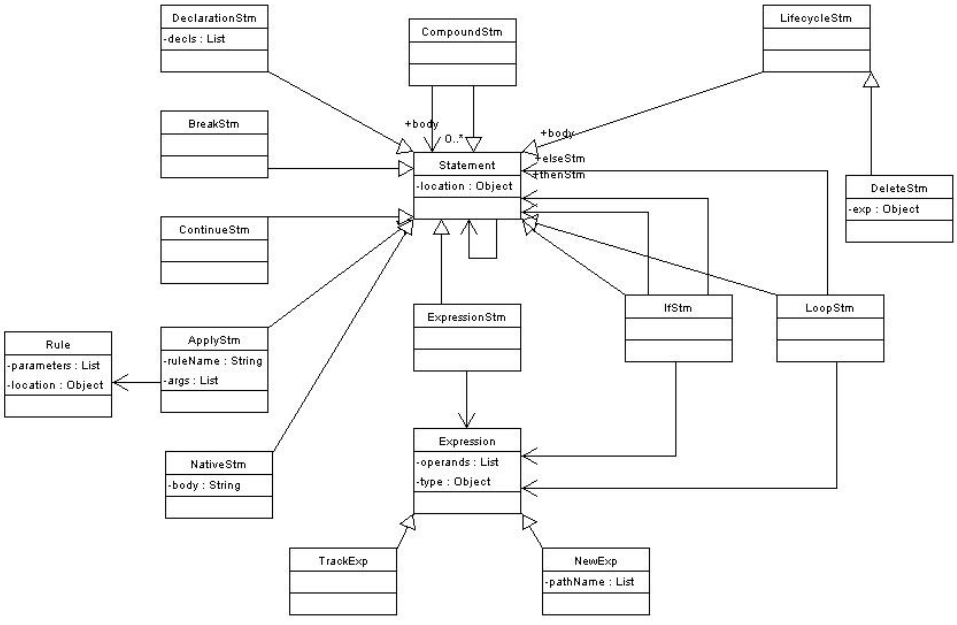



*Figure 5.5 YATL actions*

### 5.7.1. End points and reachability

Every action has an *end point*. In intuitive terms, the end point of an action is the location that immediately follows the action. The execution rules for composite actions (actions that contain embedded actions) specify the action that is taken when control reaches the end point of an embedded action. For example, when control reaches the end point of an action in a block, control is transferred to the next action in the block.

If an action can possibly be reached by execution, the action is said to be *reachable*. Conversely, if there is no possibility that an action will be executed, the action is said to be *unreachable*. In the following example

```
rule r() {
     while ( … ) {
       -- reachable
       let i:Integer=3;
       break;
       -- unreachable
        i := i+1;
    }
}
```

the action `i := i + 1` is unreachable because of the `break` action.

### 5.7.2. Blocks

A *block* permits multiple actions to be written in contexts where a single action is allowed.

*block* →
‘*{*‘ ‘*}*’
|
‘*{*‘ *action-list* ‘*}*’ .

A *block* consists of an optional *action-list*, enclosed in braces. If the action list is omitted, the block is said to be empty.

A block may contain declaration actions. The scope of a local variable or constant declared in a block is the block. Within a block, the meaning of a name used in an expression context must always be the same.

A block is executed as follows:

- If the block is empty, control is transferred to the end point of the block.
- If the block is not empty, control is transferred to the action list. When and if control reaches the end point of the action list, control is transferred to the end point of the block.

The action list of a block is reachable if the block itself is reachable.

The end point of a block is reachable if the block is empty or if the end point of the action list is reachable.

### 5.7.3. Action lists

An *action-list* consists of one or more actions written in sequence. Action lists occur in *block*s.

*action-list* →
*action* |
*action-list* *action* .

An action list is executed by transferring control to the first action. When and if control reaches the end point of an action, control is transferred to the next action. When and if control reaches the end point of the last action, control is transferred to the end point of the action list.

An action in an action list is reachable if at least one of the following is true:

- The action is the first action and the action list itself is reachable.
- The end point of the preceding action is reachable.

The end point of an action list is reachable if the end point of the last action in the list is reachable.

## 5.8. The empty action

An *empty-action* does nothing.

*empty-action*→

*';'* .

An empty action is used when there are no operations to perform in a context where an action is required.

Execution of an empty action simply transfers control to the end point of the action. Thus, the end point of an empty action is reachable if the empty action is reachable.

## 5.9. Declaration actions

A *declaration-action* declares a local variable. Declaration actions are permitted in blocks.

*declaration-action*→
*local-variable-declaration* .

### 5.9.1. Local variable declarations

A *local-variable-declaration* declares one or more local variables [OCL2], [ALP03].

*local-variable-declaration* →
*'let' variable-declaration-list ';'*
*variable-declaration-list* →
*variable-declaration* |
*variable-declaration-list ',' variable-declaration* .
*variable-declaration* →
*simple-name [':' type] ['=' init-expression]* .

The *type* of a *local-variable-declaration* specifies the type of the variables introduced by the declaration [OCL2][ALP03]. The *init-expression* gives the initial value of the variable. Both type and initial value are optional [OCL2].

The value of a local variable is obtained in an expression using a *simple-name*, and the value of a local variable is modified using an *assignment*. A local variable must be definitely assigned at each location where its value is obtained.

The scope of a local variable declared in a *local-variable-declaration* is the block in which the declaration occurs. It is an error to refer to a local variable in a textual position that precedes the *local-variable-declarator* of the local variable. Within the scope of a local variable, it is a compile-time error to declare another local variable with the same name.

A local variable declaration that declares multiple variables is equivalent to multiple declarations of single variables with the same type. Furthermore, a variable initializer in a local variable declaration corresponds exactly to an assignment action that is inserted immediately after the declaration.

The example

```
rule r() {
    let x : Integer = 1, y : Integer, z : Integer = x * 2;
}
```

corresponds exactly to

```
rule r() {
    let x : Integer;
    x := 1;
    let y : Integer;
    let z : Integer;
    z := x * 2;
}
```

## 5.10. Expression actions

An *expression-action* evaluates a given expression. The value computed by the expression, if any, is discarded.

*expression-action* →
    *expression* ‘;’ .
*expression* →
    *assignment-expression* |
    *ocl-expression* |
    *track-expression* .

Execution of an expression action evaluates the contained expression and then transfers control to the end point of the expression action.

## 5.11. The *apply* action

An *apply-action* is used to invoke a rule.

*apply-action* →
    *'apply' path-name'(' argument-list ')' ';'* .
*argument-list* →
    *λ* |
    *argument ',' argument-list* .
*argument* →
    *ocl-expression* .

For a rule invocation, the compiler must first identify the one rule to invoke or the group of overloaded rules from which to choose a specific rule to invoke. In the latter case, determination of the specific rule to invoke is based on the context provided by the types of the arguments in the *argument-list*.

The compile-time processing of a method invocation of the form *R(A)*, where *R* is a rule group and *A* is an optional *argument-list*, consists of the following steps:

- The set of candidate rules for the rule invocation is constructed. The set of rules associated with *path-name*, which are found by a name lookup operation, is reduced to those rules that are applicable with respect to the argument list *A*. The set reduction consists of applying the following rules to each rule *T::R* in the set, where *T* is the transformation in which the rule *R* is declared:
    - If *R* is not applicable with respect to *A*, then *R* is removed from the set.
    - If *R* is applicable with respect to *A*, then all rules declared in a base type of *T* are removed from the set.
    - If the resulting set of candidate rules is empty, then no applicable methods exist, and a compile-time error occurs.
- The best rule of the set of candidate rules is identified using the overload resolution rules. If a single best rule cannot be identified, the rule invocation is ambiguous, and a compile-time error occurs.

Once a rule has been selected and validated at compile-time by the above steps, the actual run-time invocation is processed according to the rules of invocation.

### 5.11.1. Name lookup

A name lookup is the process whereby the meaning of a name in the context of a transformation is determined. A rule lookup may occur as part of evaluating a *simple-name* in an apply action.

A lookup of a name *N* in a transformation *T* is processed as follows:

- The set of all accessible rules named *N* declared in *T* and the base transformations of *T* is constructed.
- If no members named *N* exist and are accessible, then the lookup produces no match.
- Otherwise, this group of rules is the result of the lookup.

### 5.11.2. Rule applicable to A

A rule is said to be an *applicable rule* with respect to an argument list *A* when all of the following are true:

- The number of arguments in *A* is identical to the number of parameters in the function member declaration.
- For each argument in *A*, the type of the argument is compatible with the type of the corresponding parameter, according to OCL 2.0 specification [OCL2].

#### *5.11.2.1. Better function member*

Given an argument list $A = A_1, A_2, \ldots, A_N$ with a set of argument types $T_1, T_2, \ldots, T_N$ and two applicable rules $R_P$ and $R_Q$ with parameter types $P_1, P_2, \ldots, P_N$ and $Q_1, Q_2, \ldots, Q_N$ , $R_P$ is defined to be a *better rule* than $R_Q$ if

- For each argument, the implicit conversion from $T_I$ to $P_I$ is not worse than the implicit conversion from $T_I$ to $Q_I$, and
- For at least one argument $A_J$, the conversion from $T_J$ to $P_J$ is better than the conversion from $T_J$ to $Q_J$.

#### *5.11.2.2. Better conversion*

Given an implicit conversion $C_1$ that converts from a type $S$ to a type $T_1$, and an implicit conversion $C_2$ that converts from a type $S$ to a type $T_2$, the *better conversion* of the two conversions is determined as follows:

- If $T_1$ and $T_2$ are the same type, neither conversion is better.
- If $S$ is $T_1$, $C_1$ is the better conversion.
- If $S$ is $T_2$, $C_2$ is the better conversion.
- If an implicit conversion from $T_1$ to $T_2$ exists, and no implicit conversion from $T_2$ to $T_1$ exists, $C_1$ is the better conversion.
- If an implicit conversion from $T_2$ to $T_1$ exists, and no implicit conversion from $T_1$ to $T_2$ exists, $C_2$ is the better conversion.

### 5.11.3. Rule invocation

This section describes the process that takes place at run-time to invoke a particular rule *R*. It is assumed that a compile-time process has already determined the particular rule to invoke, possibly by applying overload resolution to a set of candidate rules.

The run-time processing of a rule member invocation consists of the following steps:

- The argument list is evaluated from left to right.
- The resulting values are used to build an activation record.
- The body of rule *R* is applied over every source model element for which the filter attached to rule *R* is true. If the source model and target model are identical, the elements added by other previous rules are discarded.

For example, the rule

```
rule r match A(self.name='John') {
       let x:B;
       x := new B;
       ...
}
```

creates a B instance for each A instance whose property *name* has the value *John*. The filter expression can be any OCL expression (e.g. navigation expressions, operation on primitive types and collections, and iterator expressions such as *select* and *forall*).

## 5.12. The *delete* action

A *delete-action* destroys an object created by a *new-expression*.

*delete-action* →
*'delete' ocl-expression ';'* .

The operand must have a model element type [OCL20].

## 5.13. Decision actions

Selection actions select one of a number of possible actions for execution based on the value of some expression.

*selection-action* →
*if-action.*

### 5.13.1. The *if* action

The *if* action selects an action for execution based on the value of a boolean expression.

*if-action* →
*'iff' expression 'then' action ['else' action] 'endif'* .

An *else* part is associated with the lexically nearest preceding *iff* that is allowed by the syntax. Thus, an *if* action of the form

```
iff x iff y then y:= x; else x:=y;
```

is equivalent to

```
iff x then
    if y then
        y:=x;
    else
        x:=y;
    endif
endif
```

An `if` action is executed as follows:

- The *expression* is evaluated.
- If the expression yields *true*, control is transferred to the first embedded action. When and if control reaches the end point of that action, control is transferred to the end point of the *if* action.
- If the expression yields *false* and if an *else* part is present, control is transferred to the second embedded action. When and if control reaches the end point of that action, control is transferred to the end point of the *if* action.
- If the expression yields *false* and if an *else* part is not present, control is transferred to the end point of the *if* action.

The first embedded action of an *if* action is reachable if the *if* action is reachable and the expression does not have the constant value *false*.

The second embedded action of an *if* action, if present, is reachable if the *if* action is reachable and the expression does not have the constant value *true*.

The end point of an *if* action is reachable if the end point of at least one of its embedded actions is reachable. In addition, the end point of an *if* action with no *else* part is reachable if the *if* action is reachable and the expression does not have the constant value *true*.

## 5.14. Iteration actions

Iteration actions repeatedly execute an embedded action.

*iteration-action* →
*while-action* |
*do-action* |
*foreach-action.*

### 5.14.1. The *while* action

The *while* action conditionally executes an embedded action zero or more times.

*while-action* →
*‘while’ expression ’do’ action* .

A while action is executed as follows:

- The *expression* is evaluated.
- If the expression yields *true*, control is transferred to the embedded action. When and if control reaches the end point of the embedded action (possibly from execution of a *continue* action), control is transferred to the beginning of the *while* action.
- If the expression yields *false*, control is transferred to the end point of the *while* action.

Within the embedded action of a *while* action, a *break* action may be used to transfer control to the end point of the *while* action (thus ending iteration of the embedded action), and a *continue* action may be used to transfer control to the end point of the embedded action (thus performing another iteration of the *while* action).

The embedded action of a *while* action is reachable if the *while* action is reachable and the expression does not have the constant value *false*.

The end point of a *while* action is reachable if at least one of the following is true:

- The *while* action contains a reachable *break* action that exits the *while* action.
- The *while* action is reachable and the expression does not have the constant value *true*.

### 5.14.2. The *do* action

The *do* action conditionally executes an embedded action one or more times.

*do-action→*

*'do' action 'while' '(' expression ')' ';'*

A do action is executed as follows:

- Control is transferred to the embedded action.
- When and if control reaches the end point of the embedded action (possibly from execution of a *continue* action), the *expression* is evaluated. If the expression yields *true*, control is transferred to the beginning of the *do* action. Otherwise, control is transferred to the end point of the *do* action.

Within the embedded action of a *do* action, a *break* action may be used to transfer control to the end point of the *do* action (thus ending iteration of the embedded action), and a *continue*

action may be used to transfer control to the end point of the embedded action (thus performing another iteration of the *do* action).

The embedded action of a *do* action is reachable if the *do* action is reachable.

The end point of a *do* action is reachable if at least one of the following is true:

- The *do* action contains a reachable *break* action that exits the *do* action.
- The end point of the embedded action is reachable and the boolean expression does not have the constant value *true*.

### 5.14.3. The *foreach* action

The *foreach* action enumerates the elements of a collection, executing an embedded action for each element of the collection.

*foreach-action*→
*'foreach' variable-declaration 'in' expression 'do' action*

The *variable-declaration* contains the declaration of the *iteration variable* of the action. The iteration variable corresponds to a read-only local variable with a scope that extends over the embedded action. During execution of a *foreach* action, the iteration variable represents the collection element for which an iteration is currently being performed. The iteration variable can be modified or passed as an argument.

The type of the *expression* of a *foreach* action must be a collection type (as defined below), and an explicit conversion must exist from the element type of the collection to the type of the iteration variable. If *expression* has the undefined value, a dynamic semantics error is reported.

A type C is said to be a *collection type* if it is declared as an OCL collection type or implements the *collection pattern* by meeting all of the following criteria:

- C is the type of a UML attribute whose multiplicity describes a set of at least 2 elements.
- C is the type of a UML association end whose multiplicity describes a set of at least 2 elements.

### 5.14.4. The *break* action

The *break* action exits the nearest enclosing *while*, *do*, or *foreach* action.

*break-action* →
*'break' ';'*

The target of a *break* action is the end point of the nearest enclosing *while*, *do*, or *foreach* action. If a *break* action is not enclosed by a *while*, *do*, or *foreach* action, a compile-time error occurs.

When multiple *while*, *do*, or *foreach* action actions are nested within each other, a *break* action applies only to the innermost action. To transfer control across multiple nesting levels, decision actions and boolean flags must be used.

A *break* action is executed as follows:

- Control is transferred to the target of the *break* action.

Because a *break* action unconditionally transfers control elsewhere, the end point of a *break* action is never reachable.

### 5.14.5. The *continue* action

The *continue* action starts a new iteration of the nearest enclosing *while*, *do*, or *foreach* action.

*continue-action* →
*'continue' ';'*

The target of a *continue* action is the end point of the embedded action of the nearest enclosing *while*, *do*, or *foreach* action. If a *continue* action is not enclosed by a *while*, *do*, or *foreach* action, a compile-time error occurs.

When multiple *while*, *do*, or *foreach* actions are nested within each other, a *continue* action applies only to the innermost action. To transfer control across multiple nesting levels, decision actions and boolean flags must be used.

A *continue* action is executed as follows:

- Control is transferred to the target of the *continue* action.

Because a *continue* action unconditionally transfers control elsewhere, the end point of a *continue* action is never reachable.

## 5.15. Namespaces and translation units

A YATL program consists of one or more translation units, each contained in a separate source file. When a YATL program is processed, all of the translation units are processed together. Thus, translation units can depend on each other, possibly in a circular fashion. A translation unit consists of zero or more import directives followed by zero or more declarations of namespace members: queries, views, or transformations.

The concept of namespace was introduced to allow YATL programs to solve the problem of names collision that is a vital issue for large-scale transformation systems. Namespaces are used both as an “internal” organization system for a program, and as an “external” organization system - a way of presenting program elements that are exposed to other programs. A YATL program can reuse a transformation by importing the corresponding namespaces and invoking the appropriate rules.

A YATL query is an OCL expression, which is evaluated into a given context such as a package, classifier, property, or operation. The returned value can be a primitive type, model elements, collections or tuples. Queries are used to navigate across model elements and to interrogate the population stored in a given repository. YATL uses the OCL implementation that was initially developed under KMF and then under Eclipse as an open source project [OCLP].

A YATL transformation is a construct that maps a source model instance to a target model instance by matching a pattern in a source model instance and creating a collection of objects with given properties in the target model instance. The matching part is performed using the declarative features of OCL, while the creation of target instances is done using the imperative features provided by YATL. YATL provides also the possibility of interacting with the underlying machine using *native* actions. Although we do not encourage the use of such

features, they were provided to support the modeller when some operations are not available at the metamodel level (e.g. the standard library of OCL 2.0 does not provide a function to convert lowercase letters to uppercase letters).

## 5.16. Comparison

In this section we compare YATL and other transformation languages by analysing the features provided by their specification. The other transformation systems are discussed in more detail in 2.4.1, 2.4.2 and 2.4.3.To achieve this comparison we analyse the languages on the basis of several features. The features are derived from [CH03] and [Gra03]. The results of the comparison are summarized in Table 5.1.

| **Feature/ Language** | **DSTC** | **QVT Partners** | **YATL** | **ATL** | **UMT** |
|---|---|---|---|---|---|
| Abstraction Level | Model (UML) | Model (UML) | Model (UML) | Model (UML) | Data (XML) |
| Transformation Style | Declarative | Declarative | Hybrid | Hybrid | Declarative |
| Directionality | Bidirectional | Unidirectional | Unidirectional | Unidirectional | Unidirectional |
| Cardinality | Many to many | Many to many | Many to many | Many to many | One to one |
| Traceability Links | Manual | Automatic | Manual | Automatic | No support |
| Matching style | Logic matching patterns | Relations & Logic | OCL & Logic | OCL & Logic | XSLT & Logic |
| Queries | No support | Superset of OCL | Embedded OCL | No support | No support |
| Views | No support | Readonly Views | No support | No support | No support |
| Definitions | Yes | No support | No support | No support | No support |

*Table 5.1 A comparison of transformation languages*

In this table the rows represent features that are used to compare the transformation languages. The table indicates how the particular language supports each feature. The features are explained in the remaining part of this section.

**Abstraction Level**. Transformation definitions can be expressed at the XML level via XSLT or at the UML level using model concepts. Specifying transformations at the UML level makes the communication human-machine easier.

**Transformation Style**. Transformations can be described using various description styles. We distinguish imperative, hybrid and declarative transformation styles. The hybrid approach uses both declarative and imperative constructs to specify transformations.

**Directionality**. This feature indicates the direction in which the transformations can be executed. We distinguish unidirectional and bidirectional transformations. Unidirectional transformations can be executed in one direction only, which means that the target model is created or updated. Bidirectional transformations can be executed either from the source model to target model or from the target model to source model.

**Cardinality**. Cardinality indicates the number of input and output models for a transformation.

**Traceability**. This feature provides support for keeping records of relations between source and target elements during and after the execution of a transformation. The traceability is dealt with in two ways: automatic and manually.

**Matching Style**. This feature indicates the style that is used to match the transformation rules over the source repository. We distinguish the following styles: variable-based, graph-based and logic. Variable-based styles uses variables hold elements from the source or target models. Graph-based styles use graph patterns as model fragments with zero or more variables. Logic styles describe computations and constraints on model elements using logic.

**Queries.** A query is an expression that is evaluated over a model. The result of a query is one or more instances of types defined in the source model, or defined by the query language.

**Views**. A view is a model that is entirely derived from another model, called the source model. A view cannot be modified separately from the model from which it is derived. Changes to the base model cause corresponding changes to the view. If changes are

permitted to the view then they modify the source model. Views are typically not persisted independently of their source models, except perhaps for caching. Views are often read only. If views are editable a change made in the view results in a change in the source model.

**Definitions**. A definition is a specification of a relation between elements in the left-hand side and right-hand side models. A definition may contain sufficient information to describe the transformation from left to right, right to left or both.

## 5.17. Conclusions

This section contains a description of the compliance to RFP requirements, other design requirements, and related work in this area.

### 5.17.1. Compliance to RFP requirements

OMG's QVT RFT [QVT02] comprises a set of mandatory and optional requirements for the Query/Views/Transformations proposal. Meeting these requirements, especially the mandatory ones, is very important, because they are crucial for describing model transformations in the model driven engineering framework. This section presents these requirements and analyzes YATL's compliance with them.

#### *5.17.1.1. Mandatory requirements*

> *"1. Proposals shall define a language for querying models. The query language shall facilitate ad-hoc queries for selection and filtering of model elements, as well as for the selection of model elements that are the source of a transformation."*

YATL queries described using OCL 2.0 concepts can be used to query the source model instance. The data returned by a query can be any OCL value: number, string, boolean value, collection, tuple, or any value from the metamodel. The selection and filtering of model elements that are the source of transformation is done through the LHS of transformation rules.

> *"2. Proposals shall define a language for transformation definitions. Transformation definitions shall describe relationships between a source MOF metamodel S, and a target MOF metamodel T, which can be used to generate a target model instance conforming to T from a source model instance conforming to S. The source and target metamodels may be the same metamodel."*

The relations between source metamodel S and target metamodel T are described in YATL by translation rules with LHS and RHS. Current instances of relations can be stored so that they can be retrieved latter, using the *track* mechanism. YATL can be used to describe transformations for which the source model is identical with the target model. To avoid unnatural behavior in this particular case, the transformation engine applies the transformation rules only on the elements contained initially in the source model instance. The model elements that are added into the model instance by invoking transformation rules are not considered when the LHS of a rule is matched against the model instance.

> *"3. The abstract syntax for transformation, view and query definition languages shall be defined as MOF (version 2.0) metamodels."*

The abstract syntax of YATL is described using MOF concepts and is independent of the concrete syntax. The abstract syntax of YATL is described in Figure 5.1. There is an ongoing research on the graphical syntax of YATL.

> *"4. The transformation definition language shall be capable of expressing all information required to generate target model from a source model automatically."*

Both the LHS and RHS of the rules are capable of expressing all the necessary information for transformations. The LHS is used to match a specific pattern against the source model instance, while the RHS is capable of describing the objects which are added into the target model instance.

> *"5. The transformation definition language shall enable the creation of a view of a metamodel."*

YATL does not support views yet. This is an area of ongoing research.

> *"6. The transformation definition language shall be declarative in order to support transformation execution with the following characteristic:*
>
> *• Incremental changes in a source model may be transformed into changes in a target model immediately."*

YATL is partially declarative, containing a mixture of declarative and imperative features. The declarative features are inherited from OCL while the imperative features are provided mainly by YATL actions.

> *"7. All mechanisms specified in Proposals shall operate on model instances of metamodels defined using MOF version 2.0."*

Both LHS and RHS of the transformation rules operate on model instances using names, pathnames, and concepts specific to the metamodels and not to their specific implementation on a given platform.

#### *5.17.1.2. Optional requirements*

> *"1. Proposals may support transformation definitions that can be executed in two directions. There are two possible approaches:*
>
> *• Transformations are defined symmetrically, in contrast to transformations that are defined from source to target.*
>
> *• Two transformation definitions are defined where one is the inverse of the other."*

The transformations described by YATL are executed in one direction, usually from source model to target model. If a reverse transformation is needed, the modeler must write that transformation.

> *"2. Proposals may support traceability of transformation executions made between source and target model elements."*

The current version of YATL supports only explicit traceability of the execution, through explicit use of *track* constructions. Adding implicit traceability mechanisms is an ongoing research area.

> *"3. Proposals may support mechanisms for reusing and extending generic transformation definitions. For example: Proposals may support generic definitions of transformations between general metaclasses that are automatically valid for all specialized metaclasses. This may include the overriding of the transformations defined on base metaclasses. Another solution could be support for transformation templates or patterns."*

To support the reusability of the code YATL programs are organized in translation units and namespaces. Future versions of YATL will support abstract, overridden, and virtual transformation rules.

> *"4. Proposals may support transactional transformation definitions in which parts of a transformation definition are identified as suitable for commit or rollback during execution."*

Future versions of YATL will support transactional transformations for which all contained transformation rules are either committed or rolled back together.

> *"5. Proposals may support the use of additional data, not contained in the source model, as input to the transformation definition, in order to generate a target model. In addition proposals may allow for the definition of default values for this data."*

YATL allows the invocation of the transformation rules by passing additional data as arguments.

> *"6. Proposals may support the execution of transformation definitions where the target model is the same as the source model; i.e. allow transformation definitions to*

*define updates to existing models. For example a transformation definition may describe how to calculate values for derived model elements."*

YATL allows the definition of transformations for which the source model is identical to the target model. For example, YATL transformations can be used to change properties' values or remove objects. To avoid unnatural behavior in this particular case, the transformation engine applies the transformation rules only on the elements contained initially in the source model instance. The model elements that are added into the model instance by invoking transformation rules are not considered when the LHS of a rule is matched against the model instance.

### *5.17.1.3. Issues to be discussed*

*"1. The OMG CWM specification already has a defined transformation model that is being used in data warehousing. Submitters shall discuss how their transformation specifications compare to or reuse the support of mappings in CWM."*

YATL uses the concept of repository and warehouse to store source and target model instances. These concepts are mapped into an implementation by KMF-Studio, a tool from KMF. Mapping support in CWM can easily be reformulated using YATL.

*"2. The OMG Action Semantics specification already has a mechanism for manipulating instances of UML model elements. Submitters shall discuss how their transformation specifications compare to or reuse the capabilities of the UML Action Semantics."*

A YATL program specification can be described in terms of the Action Semantics.

*"3. How is the execution of a transformation definition to behave when the source model is not well-formed (according to the applicable constraints?). Also should transformation definitions be able to define their own preconditions. In that case: What's the effect of them not being met? What if a transformation definition applied to a well-formed model does not produce a well-formed output model (that meets the constraints applicable to the target metamodel)?"*

YATL does not check implicitly if the source model instance or if the generated target model instance are well formed. YATL queries can be used explicitly before and after the transformation to check the pre and post conditions associated with a transformation.

> *"4. Proposals shall discuss the implications of transformations in the presence of incremental changes to the source and/or target models."*

YATL and YATL-Studio cannot automatically detect if the source or the target model instance suffered incremental changes. At this stage it is the modeler's task to keep track of the changes. In the near future, mechanisms to detect automatically if a model instance suffered some changes will be added to the KMF warehouse and repository concepts.

### 5.17.2. Other design features

As well as supporting the ongoing QVT requirements, we designed YATL to support the following additional requirements:

- The syntax and semantics of YATL must be well defined.
- The process of applying the transformation rules must be deterministic.
- Queries, views, and transformations are organized in namespaces to provide reusability and avoid name collision.
- The transformation engine must be capable of performing efficient transformation for large-scale systems.
- YATL must provide adequate computational expressiveness power, regardless of the host platform or language. For example, YATL should support a complete set of operations on basic types like strings, integers, or floating point numbers.

### 5.17.3. Relationship to existing OMG specifications

**Object Constraint Language** OCL forms the basis of the query language and is also used to match the LHS of the transformation rules.

**Meta Object Facility** The abstract syntax of YATL and OCL are both described in terms of MOF; the superstructure is a slightly more involved extension of MOF.

**Common Warehouse Metamodel** Concepts like warehouse and repository are used to store source and target model instances.

### 5.17.4. Comparison to QVT submissions

Since OMG launched its QVT RFP [QVT02] in 2002, several submissions were made. DSTC's submission [QVTD] contains a declarative definition of QVT and uses high-level concepts that are similar with those from Prolog. Unfortunately it cannot cope with large-scale transformations because its concepts make the implementation very slow. QVT Partners submission [QVTP] considers that transformations are special cases of relations and describes them using a graphical syntax. This approach is similar to the one presented in [ASP03]. This submission provides a mechanism for relation refinement. In the near future YATL will provide a similar support, although it will be described in textual way. The French submission [QVTF] has similarities with the approach that we took. However, there are a lot of differences such as the concrete syntax, the semantics of the rules, the tracking mechanism, the support for interaction with the host machine and creation of the target model instance.

# Chapter 6. MODEL TRANSFORMATIONS IN YATL

This chapter describes three examples of model transformations, which have been implemented using YATL and the support provided by Kent Modeling Framework [KMF]. Model transformations are supported in KMF by a set of tools such as YATL-Studio, KMF-Studio, OCLCommon, and OCL4KMF. The core of the model transformations in KMF is YATL-Studio, a software environment used to create YATL projects and perform model transformations on them. The implementations of the source and target model are generated by KMF-Studio. The OCL 2.0 support is provided by OCLCommon and OCL4KMF, described in more details in [AP03][ALP03], which implement the OCL 2.0 standard.

## 6.1. Transformation environment

The OMG's MDA is a new approach to develop large software systems. The core technologies of MDA are the Unified Modeling Language (UML), Meta-Object Facility (MOF), XML Meta-Data Interchange (XMI) and Common Warehouse Metamodel (CWM). These standards are used to facilitate the design, description, exchange, and storage of models. MDA also introduces other important concept: Platform-Independent Model (PIM), Platform-Specific Model (PSM), transformation language, and transformation engine. The relations and interactions between these concepts in KMF is depicted in Figure 6.1.

In our approach, the source and target models are described using the MOF language, which in this case acts like a metalanguage. The transformation language, in our case YATL, is described using two metalanguages: BNF and MOF. BNF is used to describe the concrete syntax, while MOF is used to describe the abstract syntax. The transformation engine

performs the mapping from a source model instance to a target model instance, executing a YATL program, which is an instance of the YATL transformation language.

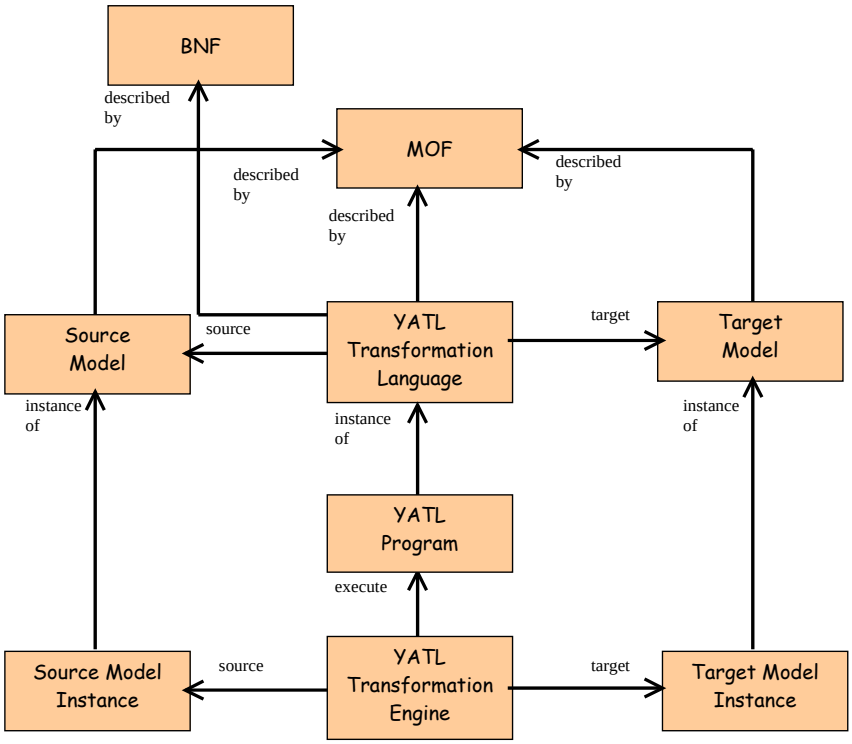



*Figure 6.1 Transformation Environment*

The entire transformation process is performed in KMF following the steps:

- The source and target models are defined using a MOF editor (e.g. Rational Rose or Poseidon)
- KMF-Studio is used to generate Java implementations of the source and target models.
- The source model repository is populated used either Java hand-written code or a GUI provided by the modelling tool generated by KMF-Studio.
- YATL-Studio is used to create a YATL project and perform the requested transformation.

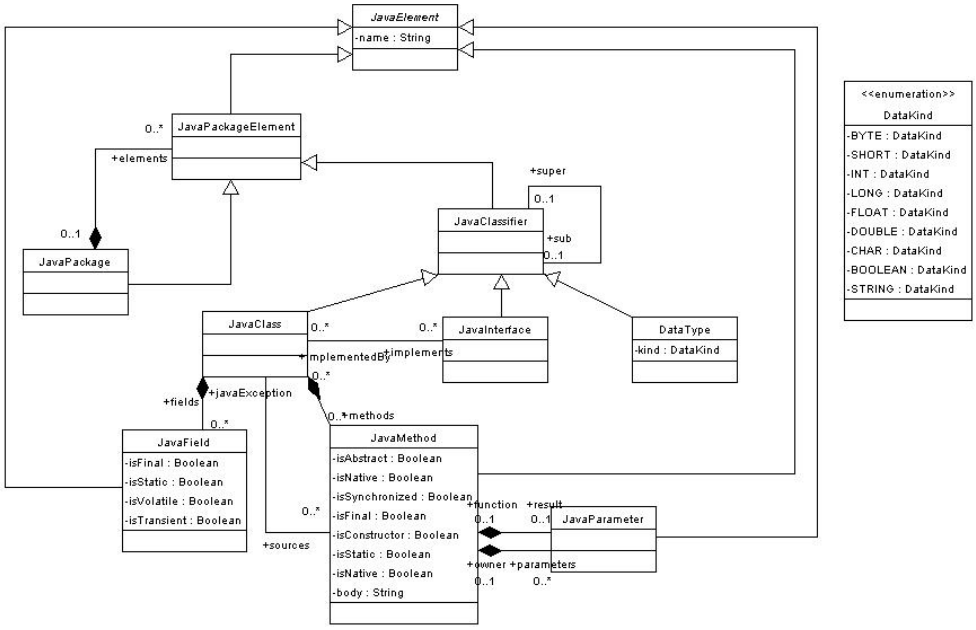



*Figure 6.2 A possible Java model*

## 6.2. Transformation from the UML model to the Java model

Figure 6.2 contains a possible model of the Java programming language. This model is derived from the Java standard [Java] and covers only a subset of the language. The main elements of the Java model are:

- *JavaElement* denotes a generic element in the Java language and represents a generalization of all the elements from Java.
- *JavaPackageElement* denotes a *JavaElement* that can be included in a package.
- *JavaClassifier* denotes a generalization of the types used in Java
- *JavaPackage*, *JavaClass*, and *JavaInterface* denote Java packages, classes, and interfaces.
- Members contained within a class are represented by *JavaField* and *JavaMethod*.
- Parameters of Java operations are described using *JavaParameter*.
- Basic types are described using *DataType*.

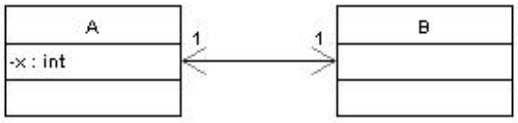
The transformation that maps from UML model to Java model is performed in two phases. In the first phase 1-1 mappings are established between equivalent concepts:

- For every UML *Package* rule *umlPkg2JavaPkg* creates an instance of *JavaPackage.*
- For every UML *Class* rule *umlClass2JavaClass* creates an instance of *JavaClass.*
- For every UML *Attribute* rule *umlAttribute2JavaField* creates an instance of *JavaField*.
- For every UML *AssociationEnd* rule *umlAssociationEnd2JavaField* creates an instance of *JavaField*.
- For every UML *Operation* rule *umlOperation2JavaMethod* creates an instanmce of *JavaMethod.*

The above rules create new instances of the required types and store the mappings using *track* constructions. This information is required in the second phase, which is responsible for filling the containment fields of Java model elements:

- Rule *linkElement2Package* scans all the *ownedElement*s of all the UML *Packages*, retrieves the corresponding *JavaPackageElements* and includes them in the *elements* collection.
- Rules *linkAttribute2Class* and *linkAssociationEnd2Class* set the correct content of the *fields* property.
- Rule *linkOperation2Class* sets the value of the *methods* property.

The YATL program that performs this transformation is described in detail in Appendix 7. For example, the following UML class diagram




maps to the following Java program:

```
class A {
     int x;
     B b;
}
class B {
     A a;
}
```

The transformation is performed at the abstract syntax level. The concrete representation of the program is obtained by visiting the abstract syntax tree and printing the required information.

The above transformation rules were tested on a source model instance that was populated using the XMI file that describes the Java model. The result of the mapping of the UML model instance described in Figure 6.2 to a Java model instance, using YATL-Studio and the YATL program from Appendix 7, is described in Figure 6.3.

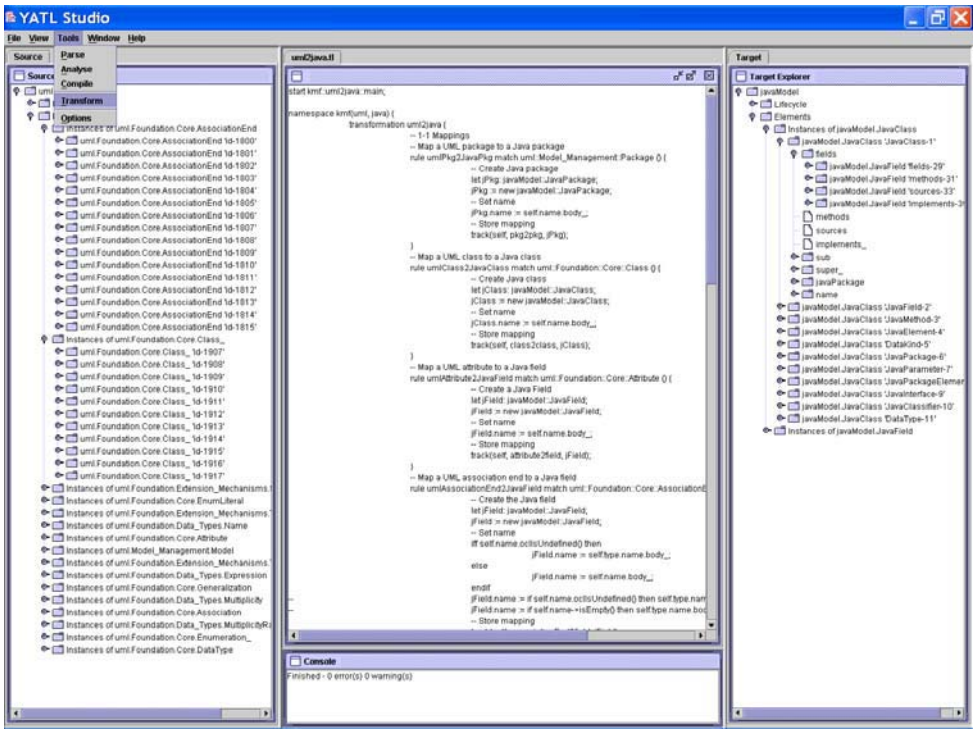



*Figure 6.3 Example of mapping from UML model to Java model*

## 6.3. Transformation from spider diagrams model to OCL model

This section contains the description of the transformation from the spider diagrams model to the OCL model. The first subsection contains a brief description of the concepts related to spider diagrams. The subsequent subsections briefly describe the mapping process.

### 6.3.1. Spider diagrams

This section introduces the main syntax and semantics of spider diagrams. Spider diagrams, introduced in [GHK99] are based on Euler diagrams rather than Venn diagrams. Spider diagrams considered here are adapted so that we can infer lower bounds for the cardinalities of the sets represented by the non-empty regions.

A *contour* is a simple closed plane curve. A *boundary rectangle* properly contains all other contours. A *basic region* is the bounded subset of the plane enclosed by a contour. A *region* is defined, recursively, as follows: any district is a region; if $r_1$ and $r_2$ are regions, then the union, intersection, or difference, of $r_1$ and $r_2$ are regions provided these are non-empty. A *zone* or *minimal region* is a region having no other region contained within it. Contours and regions denote sets. Every region is a union of zones. A region is *shaded* if each of its component zones is shaded. A shaded region denotes the empty set.

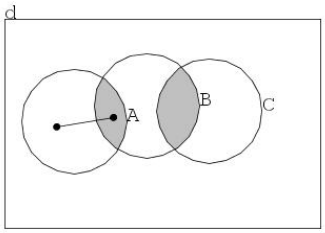



*Figure 6.4 A spider diagram*

A spider is a tree with nodes, called feet, placed in different zones. The connecting edges, called legs, are straight lines. A spider touches a zone if one of its feet appears in that region. A spider may touch a zone at most once. A spider is said to inhabit the region that is the union of the zones it touches. For any spider s, the habitat of s is the region inhabited by s. A spider denotes the existence of an element in the set denoted by the habitat of the spider. Two distinct spiders denote distinct elements.

Figure 6.4 contains a spider diagram with contours A, B, and C, six zones, two shaded zones, and a spider with one leg and two feet. The construction of the equivalent OCL expression, presented in Figure 6.5, is based on the following basic ideas:

- Every spider diagram maps to an OCL let expression.

- Every zone maps to a variable declaration of Set type.
- Every boundary condition regarding a zone maps to an OCL expression that checks the size of the corresponding variable.

```
context OclVoid inv:
let
       setA: Set(OclAny) = OclAny.allInstances()->select(x : OclAny |
              x.oclIsKindOf(RwD::A) and
              not x.oclIsKindOf(RwD::B) and  not x.oclIsKindOf(RwD::C)),
       setB: Set(OclAny) = OclAny.allInstances()->select(x : OclAny |
              x.oclIsKindOf(RwD::B) and
              not x.oclIsKindOf(RwD::A) and  not x.oclIsKindOf(RwD::C)),
       setA_B: Set(OclAny) = OclAny.allInstances()->select(x : OclAny |
              x.oclIsKindOf(RwD::A) and x.oclIsKindOf(RwD::B) and
              not x.oclIsKindOf(RwD::C)),
       setC: Set(OclAny) = OclAny.allInstances()->select(x : OclAny |
              x.oclIsKindOf(RwD::C) and
              not x.oclIsKindOf(RwD::A) and  not x.oclIsKindOf(RwD::B)),
       setA_C: Set(OclAny) = OclAny.allInstances()->select(x : OclAny |
              x.oclIsKindOf(RwD::A) and x.oclIsKindOf(RwD::C) and
              not x.oclIsKindOf(RwD::B)),
       setB_C: Set(OclAny) = OclAny.allInstances()->select(x : OclAny |
              x.oclIsKindOf(RwD::B) and x.oclIsKindOf(RwD::C) and
              not x.oclIsKindOf(RwD::A)),
       setA_B_C: Set(OclAny) = OclAny.allInstances()->select(x : OclAny |
              x.oclIsKindOf(RwD::A) and x.oclIsKindOf(RwD::B) and
              x.oclIsKindOf(RwD::C)),
       out :Set(OclAny) = OclAny.allInstances()->select(x : OclAny |
              not x.oclIsKindOf(RwD::A) and  not x.oclIsKindOf(RwD::B) and
              not x.oclIsKindOf(RwD::C))
in
       (setA_B->size() = 1) and (setB_C->size() = 0) or
       (setA->size() >= 1) and (setA_B->size() = 0) and (setB_C->size() = 0)
```

*Figure 6.5 OCL equivalent expression*

The transformation rules and their meaning are described briefly in Table 6.1.

| **Rule name** | **Rule description** |
|---|---|
| ud2let | Creates an OCL *LetExpression* for each spider diagram *Diagram* and stores the mapping using the *track* mechanism. |
| z2var | Creates an OCL *VariableDeclaration* for each spider diagram *Zone* and stores the mapping using the *track* mechanism. |
| ud2in | Creates an OCL *Expression*, representing the body of the *LetExpression*, for each spider diagram *Diagram* and stores the mapping using the *track* mechanism. |
| LinkLet2Variables | Sets the correct value for *variables* property for each OCL *LetExpression*. |
| LinkLet2In | Sets the correct value for *body* property for each OCL *LetExpression* |
| Main | Invokes the above rules in the following order:<br>`apply ud2var();`<br>`apply z2var();`<br>`apply ud2in();`<br>`apply linkLet2Variables();`<br>`apply linkLet2In();` |

*Table 6.1 Transformation rules from spider diagrams to OCL*

The entire YATL program that performs this transformation is described in detail in Appendix 8. Appendix 8 also contains the Java code that has been used to populate a source model instance. The result of the mapping of this spider diagram model instance to an OCL

model instance, using YATL-Studio and the YATL program from Appendix 8, is described in Figure 6.6.

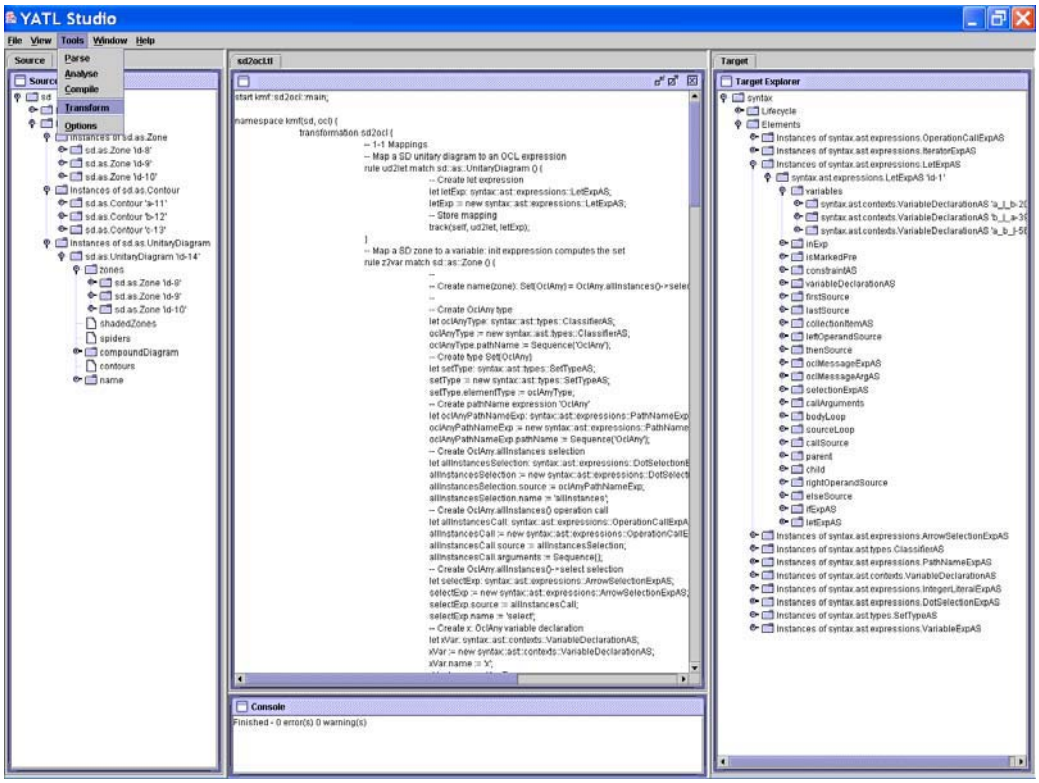



*Figure 6.6 Mapping spider diagrams to OCL*

## 6.4. Transformation from a subset of EDOC to Web Services

This section provides a mapping of a distributed system described using a subset of EDOC into an equivalent system described using Web Services. The subset contains only distributed systems described by EDOC's Model Document and Component Collaboration Architecture profiles. The equivalence between source and target system is established using the behavior of the system from the user's point of view. The first two subsections contain a brief description of EDOC and Web Services. The subsequent sections describe the system and the

transformation that performs the mapping. The entire transformation from Model Document to XML Schema is described in Appendix 9.

### 6.4.1. EDOC: the UML profile for Enterprise Distributed Object Computing Specification

The EDOC profile of UML was adopted by the OMG in November of 2001 as the standard for modeling enterprise systems. It is the modeling standard for Internet computing - providing for model driven development of enterprise systems based on the OMG's MDA.

EDOC is proposed as the *modeling framework for Internet computing*, integrating web services, messaging, ebXML, .NET and other technologies under a common technology-independent model. It comprises a set of profiles, which define the Enterprise Collaboration Architecture (ECA), the Patterns, and the Technology Specific Models and Technology Mappings.

The ECA allows the definition of PIMs and provides five UML profiles:

- The *Component Collaboration Architecture* (CCA) uses UML classes, collaborations, and activity graphs to model the structure and behaviour of components that are part of a system.
- The *Entity profile* describes a set of UML extensions that may be used to model entity objects.
- The *Events profile* describes a set of UML extensions that may be used to model event driven systems.
- The *Business Process* profile specializes the CCA and comprises a set of UML extensions that can be used to model business processes.
- The *Relationship* profile contains extensions of the UML core for rigorously specifying relationships.
- The *Patterns* profile defines a standard means, Business Function Object Patterns that can be used to describe object Models using the UML package notation.
- The *Technology Specific Models* and the *Technology Specific Mappings* take into account the mapping from ECA specification to technology specific models. It defines and EDOC profile for Enterprise Java Beans (EJB) and another for Flow Composition Model (FCM).

### 6.4.2. Web Service

The purpose of web services is to enable a distributed environment in which any number of applications, or application components, can communicate in a platform-independent, language-independent fashion. A web service is a piece of software application, located on the Internet, that is accessible through standard-based Internet protocols such as HTTP or SMTP.

Given this definition, several technologies used in recent years could have been classified as web service technologies, but were not. These technologies include win32 technologies, J2EE, CORBA, and CGI scripting. These technologies are not web services technologies mainly because they are based on a proprietary binary standard, which is not supported globally by most major technologies firms. The core of the web services technologies is made of eXtensible Markup Language [XML], Simple Object Access Protocol [SOAP], Web Service Description Language [WSDL], and Universal Description, Discovery and Integration [UDDI].

XML is a widely used standard from the World Wide Web Consortium (W3C) that facilitates the interchange of data between computer applications. XML is similar to the language used for Web pages, the HyperText Markup Language (HTML), both using markup codes (tags). Computer programs can automatically extract data from an XML document, using its associated DTD as a guide.

SOAP provides a standard packaging structure for exchanging XML documents over a variety of Internet protocols, including HTTP, SMTP, and FTP. The existence of a standard transport mechanism allows heterogeneous clients and servers to communicate. For example, .NET clients can invoke EJBs and Java clients can invoke .NET Components through SOAP.

WSDL is an XML technology that provides a standard description of web services. WSDL can be used to describe the representation of input and output parameters of an invocation, the function’s structure, the nature of the invocation, and the protocol used for transport.

UDDI provides a worldwide registry of web services for description, discovery, and integration purposes. Analysts and technologists use UDDI to discover available web services by searching for categories, names or identifiers.

### 6.4.3. Mapping from Document Model to XML Schema

Both EDOC and WS models describe business processes. A business process manipulates and exchanges information with other business processes. To describe the information that is manipulated or exchanged during a business process, both EDOC and WS have dedicated components: *Model Document* and *XML Schema* respectively.

The first step in the mapping from EDOC to WS is to map the models that are used to describe the information that is manipulated. This section contains the description of the mapping process from Document Model to XML Schema.

The Document Model package from the EDOC profile defines the information that can be manipulated by EDOC *ProcessComponent*s. The document model is based on *data elements* that can be either primitive *data types* or *composite data*. A CD data element contains several attributes. An *attribute* has a specific type, an initial value and can be marked as *required* or as *many* to indicate the cardinality. An *enumeration* defines a type with a fixed set of values. The document model is described in Figure 6.7.

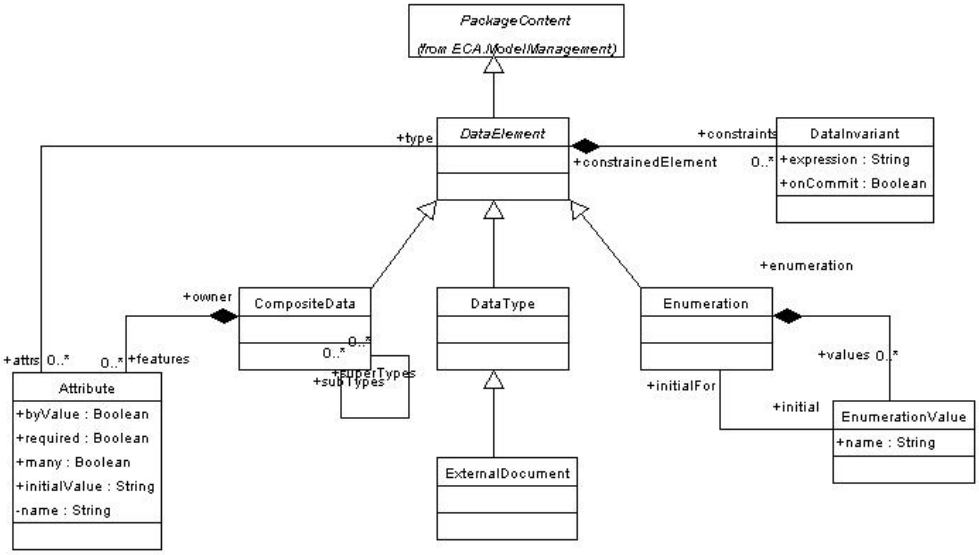



*Figure 6.7 Document Model profile*

The XML Schema [XMLS] describes the information that can be manipulated by web services. It contains types that can be *simple*, such as string or decimal, or *complex*. A *ComplexType* contains a sequence of *attributes*. An *Attribute* has a name and a given type. A partial model of XML Schema is given in Figure 6.8.

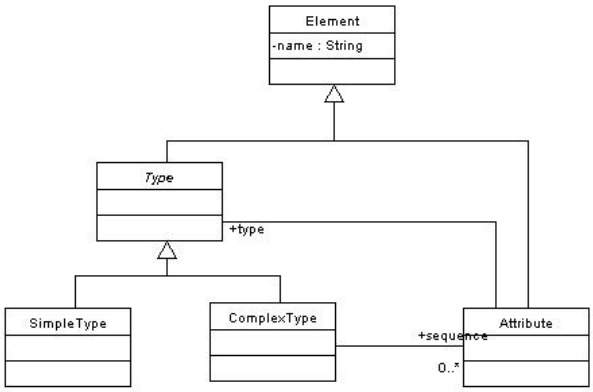



*Figure 6.8 XML Schema*

It is obvious that mapping from Model Document to XML Schema means mapping from DataElement, DataType and CompositeData to Type, SimpleType and ComplexType respectively. The transformation process and the rules that perform the mapping are described briefly in Table 6.2.

| **Rule name** | **Rule description** |
|---|---|
| dt2st | Creates a XML Schema *SimpleType* for each Document Model *DataType* and stores the mapping using the *track* mechanism. |
| cd2ct | Creates a XML Schema *ComplexType* for each Document Model *CompositeData* and stores the mapping using the *track* mechanism. |
| at2at | Creates a XML Schema *Attribute* for each Document Model *Attribute* and stores the mapping using the *track* mechanism. |

| **Rule name** | **Rule description** |
|---|---|
| linkAttribute2Type | Sets the correct value for the *type* property for each XML Schema *Attribute*. |
| linkComplexType2Attribute | Sets the correct value for *sequence* property for each XML Schema *CompositeType* |
| documentModel2xsd | Invokes the above rules in the following order:<br>`apply dt2st();`<br>`apply cd2ct();`<br>`apply at2at();`<br>`apply linkAttribute2Type();`<br>`apply linkComplexType2Attribute();` |

*Table 6.2 Transformation rules for Document Model to XML Schema mapping*

### 6.4.4. Mapping from CCA to WSDL

The CCA profile details how the UML concepts of classes and collaboration graphs can be used to model the structure and the behaviour of the components that comprise a system. In CCA *process components* interact with other process components using a set of *ports*. A *ProcessComponent* describes the contract for a component that performs actions. A *Port* defines a point of interaction between process components. Ports can be classified according to the complexity of the interaction into *FlowPorts*, *ProtocolPorts*, *OperationPorts, and MultiPorts.* A *FlowPort* is a port capable of producing and consuming a single data type. *ProtocolPorts* describe more complex interactions based on *Protocols*. A *Protocol* is a method by which two components can communicate. An *OperationPort* is a port that realizes a typical request/response operation. A *MultiPort* is a group of ports whose actions are tied together. The specification of a *ProcessComponent* may include a *Choreography* to specify the sequence of interactions performed through ports. Figure 6.9 describes the CCA profile.

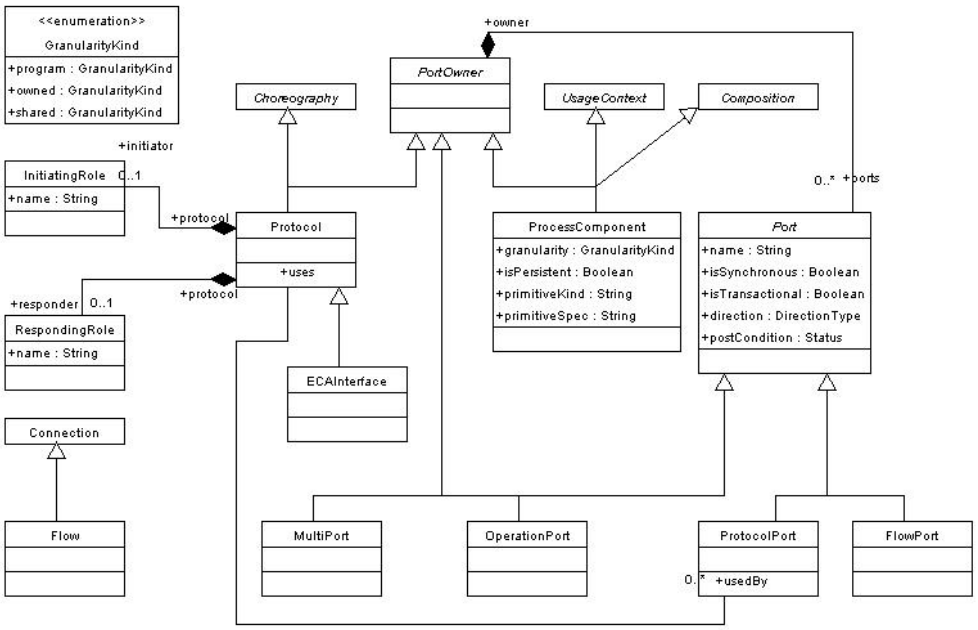



*Figure 6.9 CCA profile*

In WSDL the *Definition* element acts as a container for the service description. The *Import* element serves a purpose similar to the #include directive in the C/C++ programming language. It lets the modeller separate the elements of a service definition into separate documents and include them in the main document. The *Type* element acts as a container for the definition of datatypes that are used in the *Message* elements. The *Message* element is used to model the data exchanged in a web service. A message is made of several *parts*, each part having a name and a type. The *PortType* element specifies a subset of operations supported for an endpoint of a web service. The *Operation* element models an operation. A WSDL operation can have input, output, and fault messages as part of its action. The *Binding* element specifies the protocol and data format of a *PortType* element. The bindings can be standard - HTTP, SOAP, or MIME – or can be created by the user. The *Service* element typically appears at the end of a WSDL document and identifies a web service. The primary purpose of a WSDL document is to describe the abstract interface. A *Service* element is used only to describe the actual endpoint of a service. Figure 6.10 contains the WSDL model.

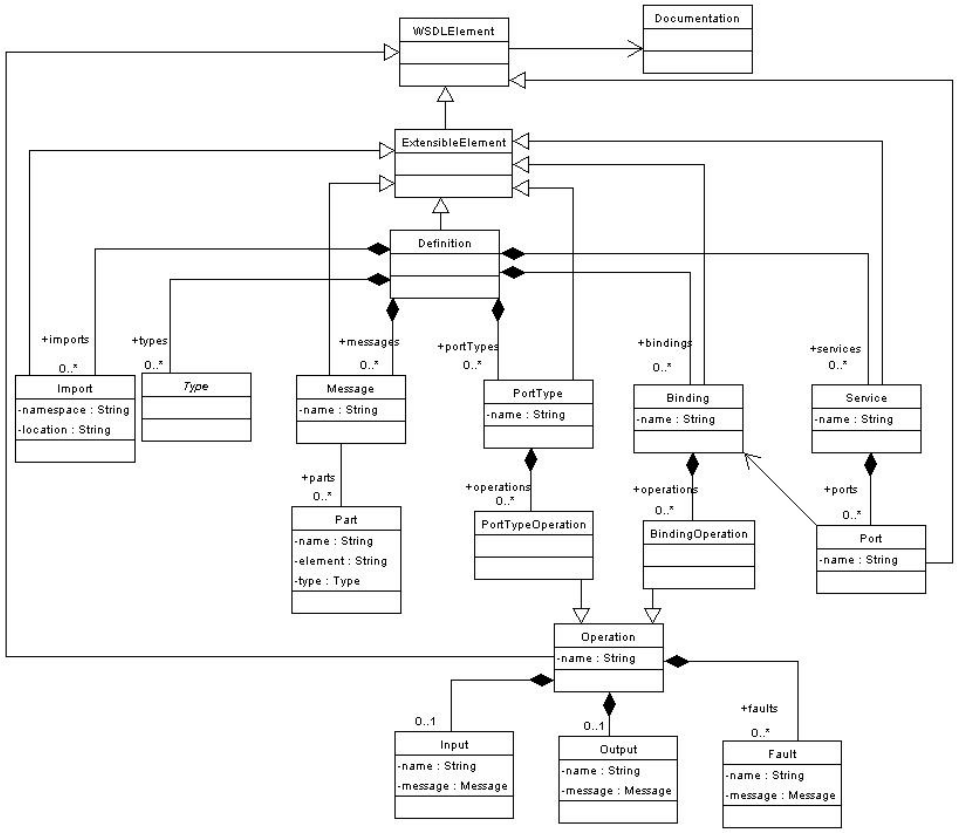



*Figure 6.10 WSDL model*

The transformation from CCA to WSDL obeys the well-known compositional principal of Frege [JB81], which states that "the meaning of a syntactic construct is a function of the meanings of its constituents". The transformation process and transformation rules are described in Table 6.3.

| **Rule name** | **Rule description** |
|---|---|
| flowPort2message | Creates a WSDL Message for each CCA FlowPort and stores the mapping using the track mechanism. |
| operationPort2operation | Creates a WSDL Operation for each CCA OperationPort and stores the mapping using the track mechanism. The input and output properties of the WSDL Operation are computed using the initiator and the responder port from the OperationPort. |

| | |
|---|---|
| protocolPort2portType | Creates a WSDL PortType for each CCA ProtocolPort and stores the mapping using the track mechanism. |
| processComponent2service | Creates a WSDL Service for each CCA ProcessComponent and stores the mapping using the track mechanism. The definition of the service is instantied by this rule. The values of the properties are assigned by the other rules. |
| LinkDefinition2X | Computes the types, messages, and portTypes properties for every WSDL Definition. Uses the track mechanism to retrieve the mapping information stored by previous rules. |
| cca2wsdl | Invokes the above rules in the following order:<br>`apply flowPort2message();`<br>`apply operationPort2operation();`<br>`apply protocolPort2portType();`<br>`apply processComponent2service();`<br>`apply linkDefinition2X();` |

*Table 6.3 Transformation from CCA to WDSL*

### 6.4.5. An example

To study the mapping from EDOC to WS using YATL and YATL-Studio we consider a simplified model of a travel agency. In general a travel agency provides services such as: reserves and purchases flights and charters tickets, reserves hotel rooms, rents cars, books holidays and cruises, and sells travel insurance. To provide such services a travel agency needs to establish business links with companies such as airlines, hotels, and banks.

Figure 6.12 contains the description of a travel agency community process. The activities in the TravelAgency Community Process start by the Client initiating the interactions on its Buy ProtocolPort, according to the BuySell protocol. The TravelAgency is connected through the Sell ProtocolPort with the Client and responds to the BuySell protocol initiated by the Client. The TravelAgency uses the dedicated ports BuyFlight, ReserveRoom, RentCar, and Payment to communicate with the other processes: Airline, Hotel,

CarCompany, and Bank. The TravelAgency initiates the communication through these ports, according to Client's requests. Figure 6.12 contains the description of choreographies for BuySell and BuyFlight protocols. Similar choreographies can be derived for ReserveRoom

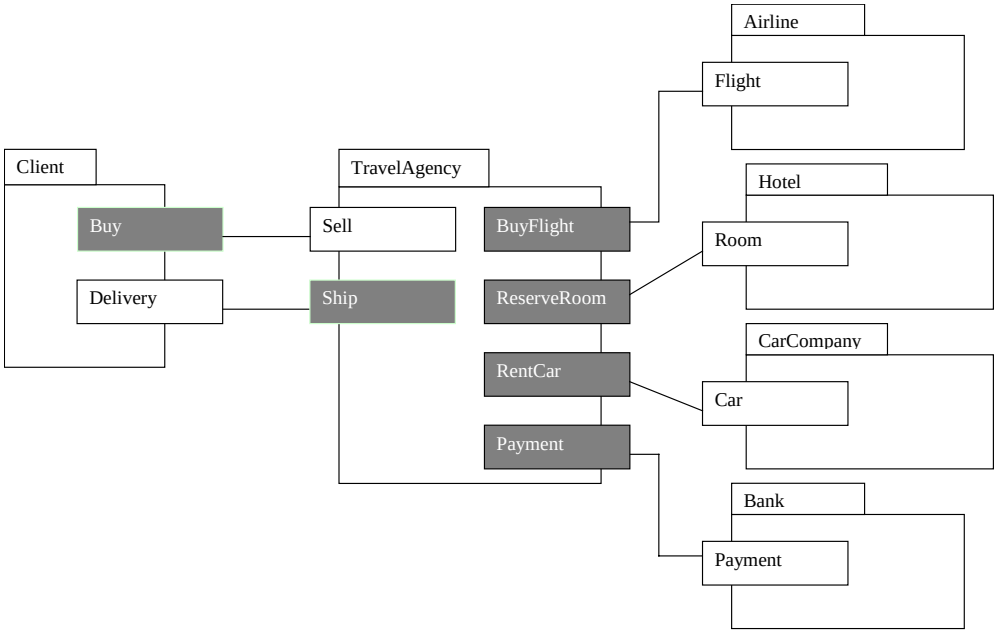
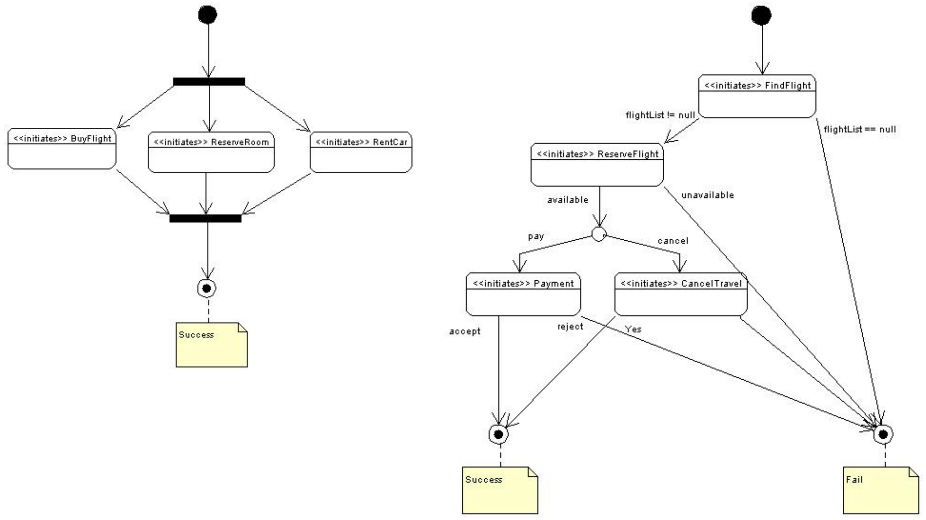



*Figure 6.11 Travel agency community process*




a) BuySell choreography b) BuyFlight choreography

*Figure 6.12 BuySell and BuyFlight coreography*

Appendix 9 contains the Java code that has been used to populate a source model instance. It also contains the entire description of transformation rules. The result of the mapping performed by the YATL program from Appendix 9 over this source model instance is described in Figure 6.13.

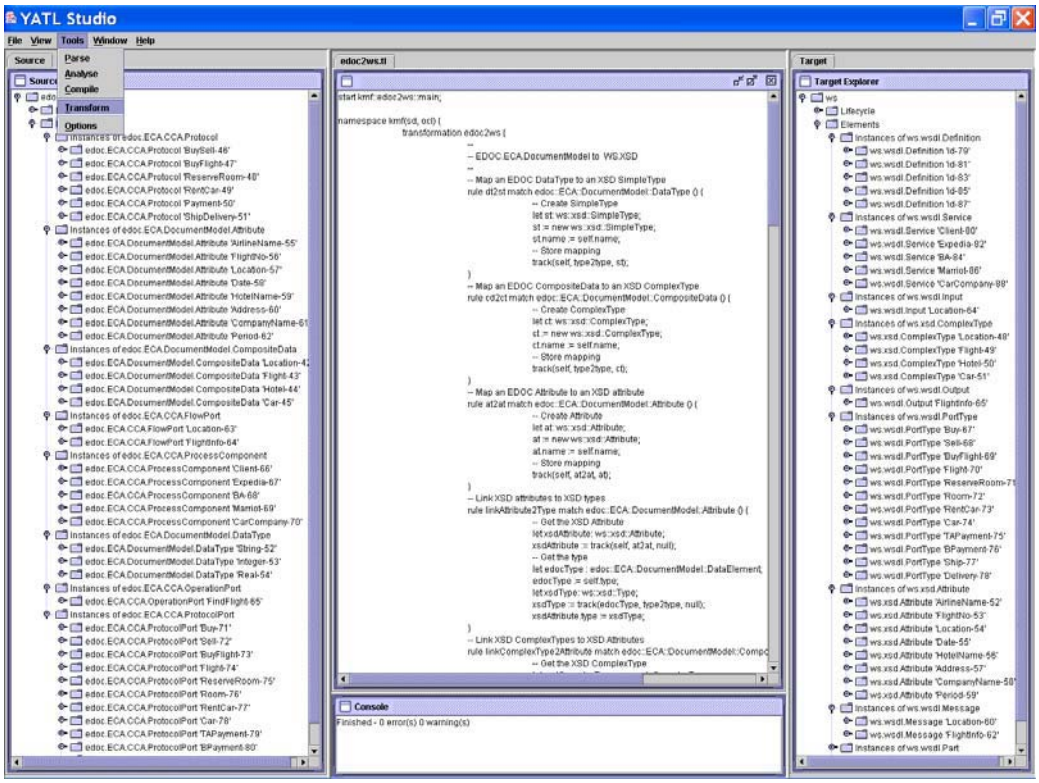



*Figure 6.13 Mapping the travel agency model to a WS model*

## 6.5. Conclusions

We have learned a lot during this work. The experiments forced us to add new features to YATL and improve the implementation, especially the mapping from spider diagrams to OCL because it is not a conventional mapping from a visual language to a textual language. YATL is still evolving because one of our main goals is to make it compliant to the QVT standard.

# Chapter 7. DISCUSSION AND CONCLUSIONS

Section 7.1 of this chapter summarizes the work presented in this thesis. Section 7.2 highlights the achievements in terms of the objectives defined in the introduction. Finally, section 7.3 proposes possible future research that continues from that presented in this thesis.

## 7.1. Thesis Summary

The thesis presents at the beginning the background of the research: model driven engineering, language translation, and object-oriented design patterns. The thesis is focused on the Object Management Group's (OMG's) Model Driven Architecture (MDA) initiative. As MDA is a software development framework in which the translation of one model into another forms an important part, this thesis is focused on model transformations and model quality evaluation.

This thesis has investigated and presented object-oriented techniques that can be used to represent and efficiently implement model transformations in the OMG's MDA framework.

The proposed technique is based on Yet Another Transformation Language (YATL). YATL is a hybrid language (a mix of declarative and imperative constructions) that has been designed and implemented to answer the Query/Views/Transformations Request For Proposals issued by OMG and to express model transformations as required by the MDA approach.

The technique that we have proposed in this thesis does not claim to be more powerful than graph transformations, but the implementation of this technique proved to be efficient.

The declarative features come mainly from OCL expressions and the description of the LHS of transformation rules. YATL acts in a similar way to a database system that uses SQL to

interrogate the database and the imperative host language to process the results of the query. We choose OCL to describe the matching part of YATL rules because it is a well-known language for querying the UML models; it provides a standard library with an acceptable computational expressiveness, it is a declarative language, and it is a part of the OMG's standards.

YATL supports several kinds of imperative features, used in the right hand side of transformation rules. These features were selected so that YATL can provide lifecycle operations like creation and deletion, operations to change the value of properties, declarations, decisions, and iteration actions, native actions to interact with the host machine, and build actions to ease the construction of target model instances. Compound actions contain a sequence of instructions, which are to be executed in the given order. These syntactic constructions make use of OCL expressions to specify basic operations such as adding two integer values. YATL uses the same type system as OCL 2.0.

YATL is described by an abstract syntax (a MOF metamodel) and a textual concrete syntax. It does not yet have a graphical concrete syntax as QVT RFP suggested. A transformation model in YATL is expressed as a set of transformation rules. Transformations from Platform Independent Models (PIMs) to Platform Specific Models (PSMs) can be written in YATL to implement the MDA.

A YATL transformation is unidirectional. We believe that a model transformation language should be unidirectional, otherwise it cannot be used for large scale models. The main difficulty with a bidirectional transformation language is that it needs some reasoning to perform the transformation. The reverse transformation can be described just as any other transformation using YATL.

The current version of KMF-Studio uses UML diagrams exported over XMI files and computes OO metrics that have been proved in time to be good indicators to evaluate the quality of object-oriented systems. KMF-Studio provides forty-four predefined metrics that can be computed to evaluate to measure a given model. The metrics supported by KMF-Studio are design metrics that evaluate and measure the maintainability of models. The result of evaluating the metrics over a model identifies the weak points of UML models and gives on the fly diagnostic about the current status of the model.

## 7.2. Achievements

The objectives laid out in Section Chapter 1 have been met by the content of this thesis as described below.

Objective 1 is met by the design and implementation of the YATL language for specifying model transformations described in Chapter 5. UML and YATL are both object-oriented specification methods and the transformation specification techniques enables the transformation relation to be defined between two models that have been specified using UML.

Objective 2 is met by the experimental studies presented in Chapter 6 and the proposed modeling framework that is presented in Chapter 3. The experimental studies cover a wide range of transformations: mapping UML to Java, visual descriptions of constraints to textual descriptions of constraints (spider diagrams to OCL), and different languages that are used to describe distributed processing (EDOC to Web Services). The discussion contained in the above chapters demonstrates how to create a transformation from a UML/YATL specification. The implementation consists of two parts. The first part, which implements the UML models, contains the code generated by KMF-Studio providing persistence, editing, and browsing facilities at model level. The second part contains the specification of transformations that is executed using the transformation engine implemented by YATL-Studio.

Objective 3 is met by the design and implementation of a suite of software metrics that can be used to evaluate the quality of UML models at early stages of software development process. This is very important especially in OMG's Model Driven Architecture framework for software development. As models are used to drive the entire software development process it is unlikely that high quality software can be derived from low quality models.

## 7.3. Future work

There are a number of possible areas for continuing the research presented in this thesis. Some of these are discussed in the following subsections.

### 7.3.1. Visual languages and YATL

Visual languages of many types are used in many disciplines for many purposes. The use of visual languages is compelling for many reasons, not the least of which is that their graphical nature can lead to a representation of the actual domain in a way that is not possible with purely textual systems.

The work presented in this thesis could be extended to study the relationship between YATL and visual languages. This could lead to a visual description of transformations described using YATL.

A suitable case study for this investigation would be the constraint diagrams defined by [GHK99]. These diagrams are based on the concepts of contours, regions, spiders, and arrows. Such diagrams cannot be mapped to a spatial relationship model based on directed graphs. Other work has been carried out in [GHK01] to identify the basic concepts of the notation.

The relationship of these concepts to the abstract YATL concepts could be defined using mapping rules specified using the specification technique proposed in this thesis.

Some initial work has been carried out in this area and published in [Pat04c] and investigated the relationship between spider diagrams, which are a subset of constraint diagrams, and OMG's Object Constraint Language (OCL), which is used in YATL to query the model instances. This work could be extended to include investigation into the specification and implementation of visual languages that are not based on the directed graph style of spatial relationship model associated with box and line based diagrams.

### 7.3.2. Relationship between graph transformations and YATL

Graph transformations and graph grammars are at this time the most mature technique for specifying transformations. Unfortunately graph transformations are not based on object-oriented concepts, and hence are not compatible with the OMG's Model Driven Architecture. Moreover, graph transformations proved to be hard to implement and usually the

implementation of such transformations is inefficient. This makes the graph transformation approach unsuitable for large-scale systems and hence for industrial use.

As future work we propose the investigation of the relationship between graph grammars and the technique that we proposed in this thesis. We think that UML class diagrams, with the addition of OCL and YATL are as expressive as graph grammars. There is no formal backing to this assertion and work to produce evidence in support of it could provide a useful bridge between the graph grammar and object-oriented communities.

Investigation regarding the expressiveness capabilities of graph grammars and UML/YATL technique for specifying model transformations could be a direction to follow. Additionally, the specification of translators between graph grammars and UML/YATL specifications would aid this work and enable known results from each area to be applied to the other.

To specify the translation it would be necessary to identify the abstract syntax model of both graph grammars and UML/YATL. The abstract syntax model of graph grammar should ideally be one that is widely accepted by the graph grammar community.

Based on these translators, tools can be built to provide both graph grammar and UML/YATL specification of model transformations. This approach would bring the experience and techniques of the graph grammar community into the industrial community using UML.

Some initial work has been carried out in this area, published in [Pat04b], which investigated the abstract syntax model of YATL. This work could be extended to include investigation into the specification of an abstract syntax model for graph grammars and specification of translation between graph grammars and UML/YATL.

### 7.3.3. Adding new features to YATL processors

One of the advantages of the model transformation technique proposed in this thesis is the use of the standardized languages such as UML and OCL, and object-oriented concepts. This makes the technique easily adoptable by the object-oriented community.

The implementation of the UML/YATL model transformation specification is based on a classical interpreter/complier approach. The main advantages of this approach are:

- On the fly evaluation of model transformations
- Efficiency of implementation
- Support for model transformation debugging

This approach also has disadvantages such as:

- Every time the transformation changes the entire transformation needs to be compiled or interpreted.
- Every time the source model changes the entire transformation needs to be re-executed.
- The runtime of the transformation execution is proportional to the size of the source model instance.

To address these disadvantages, a new implementation approach is required. The approach that we propose makes use of the observer pattern to monitor the source model instance for changes continuously. After detecting a change in the source model instance, the transformation environment alters the target model instance to be consistent with the new source model instance.

The first step to follow this path could be choosing the appropriate granularity of the observers used in the transformation environment. As YATL transformation rules are filtered according to the type of model element instances using dedicated observers for each types could be a useful approach.

Using such an approach to implement model transformations solves the above disadvantages:

- The observers detect any change in the transformation and trigger and compile/interpret only the parts that were modified.
- The observers detect any change in the source model instance and trigger a required local transformation that updates the target model instance according to the new source model instance. Hence, the transformation is not required explicitly when the model instance changes.
- The runtime of the transformation execution is no longer proportional to the size of the source model instance. The cost of updating the source model instance is now proportional with the size of the update and the complexity of the invoked local transformation.

# Appendix 1. GRAMMAR SPECIFICATION RULES

Grammar specification is done using the following rules:

1) Left hand-side and right hand-side are separated by symbol $\rightarrow$.
2) Each production ends with a dot.
3) Terminal symbols are written using capital letter or delimited by apostrophes.
4) The following shortcuts are permitted:

| Shortcut | Meaning |
|---|---|
| $X \rightarrow \alpha\ (\ \beta\ )\ \gamma\ .$ | $X \rightarrow \alpha\ Y\ \gamma\ .\ Y \rightarrow \beta\ .$ |
| $X \rightarrow \alpha\ [\ \beta\ ]\ \gamma\ .$ | $X \rightarrow \alpha\ \gamma \mid \alpha\ (\ \beta\ )\ \gamma\ .$ |
| $X \rightarrow \alpha\ u\ +\ \gamma\ .$ | $X \rightarrow \alpha\ Y\ \gamma\ .\ Y \rightarrow u \mid u\ Y\ .$ |
| $X \rightarrow \alpha\ u\ *\ \gamma\ .$ | $X \rightarrow \alpha\ Y\ \gamma\ .\ Y \rightarrow u \mid u\ Y \mid \lambda\ .$ |
| $X \rightarrow \alpha \mid\mid a.$ | $X \rightarrow \alpha\ (\ a\ \alpha\ )\ *\ .$ |

where $\alpha$, $\beta$ and $\gamma$ are strings over the language alphabet, $Y$ is a symbol which does not appear elsewhere in the specification, $u$ is either a unique symbol or an expression delimited by parentheses, and $a$ is a terminal symbol.

# Appendix 2. XTL-OVERVIEW

The KMF-Studio framework contains a powerful tool for generating source code: the XTL (X Template Language). With XTL one can use a JSP-like syntax to write templates that specify the output to be generated. KMF-Studio provides support for XTL through a generic template engine that can be used to generate various kinds of outputs (e.g. C/C++/Java/C# source code and XML).

This section describes how XTL templates are created and used to generate source code. This section also provides a short reference to the XTL syntax.

The code generation process is performed by KMF-Studio in two steps:

1. Create a Java class, called the template class, from the XTL description.
2. Create an instance of the template class and invoke the method that generates the code.

### 2.1.1. An Example

For example, in order to generate a Java file that contains a description of an interface, the following XTL template

```
--
-- Generate code for Java
--
<%namespace java %>
--
-- Template for interfaces
--
<%template Test (String pkgName, String interfaceName) %>
<%begin %>
package <%exp pkgName%>;


public  Test<%exp interfaceName%> {
}
<%end %>
```

corresponds to the following template class:

```
/**
 *
 *  Class Test.java
 *
 *  Generated by XTL compiler at 16 December 2004 16:16:05
 *  Visit http://www.cs.ukc.ac.uk/kmf
 *
 */

package test.scripts;

import uk.ac.kent.cs.kmf.*;
import uk.ac.kent.cs.kmf.*;

class Test {
     /** Constructor */
     public Test(java.io.PrintWriter out,
                 String pkgName,
                 String interfaceName) {
          this.out = out;
          this.pkgName = pkgName;
          this.interfaceName = interfaceName;
     }

     /** Generate code method */
     public void generate() {
          out.print("\npackage ");
          out.print(pkgName);
          out.print(";\n\npublic interface ");
          out.print(interfaceName);
          out.print(" {\n}\n");
     }

     //
     // Local variables
     //
     protected java.io.PrintWriter out;
     protected String pkgName;
```

```
        protected String interfaceName;
    }
```

If the template class is invoked using “test” and “A” as input arguments, the generated Java code is:

```
package test;

public interface A {
}
```

### 2.1.2. Supported Features

XTL provides support for the following features:

- Namespaces to group templates in hierarchies.
- Specify the import of packages used by the generated code.
- Specify the parameters of the template class.
- Support for control flow and computation through common statements and expressions (e.g. *foreach* statements and arithmetic expressions).

# Appendix 3. XTL-GRAMMAR

## 3.1. XTL Syntax

Five basic elements make up the lexical structure of a XTL source file: line terminators, white spaces, comments, and tokens. Of these basic elements, only tokens are significant in the syntactic grammar of a XTL program.

For compatibility with source code editing tools that add end-of-file markers, and to enable a source file to be viewed as a sequence of properly terminated lines, the following transformations are applied, in order, to every source file in a C# program:

- If the last character of the source file is a Control-Z character, this character is deleted.
- A carriage-return character is added to the end of the source file if that source file is non-empty and if the last character of the source file is not a carriage return, a line feed, a line separator, or a paragraph separator.

The *input* production defines the lexical structure of a XTL source file. Each source file in a XTL program must conform to this lexical grammar production.

*input* $\rightarrow \lambda$ | *input-element* | *input* *input-element.*

*input-element* $\rightarrow$ *line-terminator* | *whitespace*| *comment*| *token.*

Line terminators divide the characters of a C# source file into lines. YATL uses the following markers to indicate the end of a line:

- Carriage return character (*U+000D*)
- Line feed character (*U+000A*)
- Carriage return character (*U+000D*) followed by line feed character (*U+000A*)
- Next line character (*U+0085*)
- Line separator character (*U+2028*)
- Paragraph separator character (*U+2029*)

It adds only the following keywords:

| | | |
|---|---|---|
| elif | false | in |
| else | foreach | namespace |
| end | if | template |
| exp | import | true |

and the following special signs and sequences:

| | | | | |
|---|---|---|---|---|
| . | + | ! | == | , |
| () | - | && | != | <% |
| | * | \|\| | < | %> |
| | / | | <= | * |
| | % | | > | :: |
| | | | >= | |

The syntax grammar is described below:

*// Translation Unit*

*translation-unit → import* namespace*

*// Import*

*import → '<%' 'import' name '%>' | '<% 'import' name '.' '*' '%>'*

*// Namespace*

*namespace → '<%' 'namespace' simple-name '{' template* '}' '%>' | template**

*// Template*

*template → '<%' 'template' simple-name '(' param* ')' '%>' compound-stm*

*// Action*

*action → text-stm | exp-stm | include-stm | compound-stm | if-stm | foreach-stm*

*exp-stm → '<%' 'exp' exp '%>'*

*include-stm → '<%' 'include' name '(' args ')' '%>'*

*if-stm → '<%' 'if' '(' exp ')' '%>' stm ('<%' 'elif' '(' exp ')' '%>' stm)**

*['<%' 'else' '%>' stm]*

*'<%' 'end' '%>'*

*foreach-stm → '<%' 'foreach' type-name simple-name 'in' exp '%>' stm*

*// Expressions*

*exp → simple-name | 'true' | 'false' | 'integer' | 'real' | 'string'*

*exp → exp '.' simple-name*

*exp → exp '.' simple-name '(' args ')'*

*exp → ('+' | '-' | '!') exp*

*exp → exp ('*' | '/' | '%' ) exp*

*exp → exp ('+' | '-') exp*

*exp → exp ('==' | '!=') exp*

*exp → exp ('<' | '<=' | '>' | '>=') exp*

*exp → exp '&&' exp*

*exp → exp '||' exp*

*// Arguments*

*args → | exp (',' exp)**

*// Name*

*name → simple-name ('::' simple-name)**

# Appendix 4. THE QUALITY MODEL

The ISO/IEC 9126 standard defines the quality of software products considering the following six characteristics:

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability
- Portability

The quality model that we propose evaluates the maintainability of UML models according to the above ISO standard.

Maintainability is defined as a set of attributes that measure the effort to perform given changes. This characteristic can be reduced to the evaluation of the following attributes, also called subcharacteristics:

- Analyzability
- Changeability
- Stability
- Testability

These attributes together with the corresponding metrics are classified on four levels of quality.

We have classified model elements whose quality is *satisfactory* as

- Excellent: all the metrics of the quality model are within specified boundaries.
- Good: the metric values do not deviate too much from the specified boundaries.
- Acceptable: there are no major violations of the metrics boundaries.

A model element whose quality is *unsatisfactory* can be classified as

- Poor: the quality model cannot guarantee an efficient maintenance.

This model is based on the principles formulated in [Ghe91] [Som92].

**Metrics for internal attributes**

| Metric Acronym | Name | Description |
|---|---|---|
| MODEL-HNT | Height of Nesting Tree | Scan the nesting tree starting from the top using a depth first strategy and compute the height of the tree. The height of a tree with only one node is zero. |
| MODEL-HIG | Height of Inheritance Graph | Scan all the connected parts of the inheritance graph and compute its height using an algorithm similar with the one used in MODEL-HNT. Compute the maximum of the resulting values. |
| MODEL-NCN | Number of Contained Namespaces | Performs a depth first search and count the number of all contained namespaces, regardless of the nesting level. |
| MODEL-ANCPN | Average Number of Classes Per Namespace | Computes the number of classes for each namespace and then computes the arithmetic average. |
| MODEL-ADIG | Average Depth of Inheritance Graph | Computes the height for each inheritance graph and then computes the arithmetic average. |
| MODEL-ACC | Average Class Complexity | Computes the complexity for each class and then computes the arithmetic average. |
| MODEL-AMC | Average Method Complexity | Computes the complexity of every method and then computes the arithmetic average. |
| MODEL-AOCC | Average OCL Constraint Complexity | Computes the complexity of every OCL constraint and then compute the arithmetic average. |
| NS-NDCN | Number of Directly Contained Namespaces | Computes the number of directly owned namespaces. |

| | | |
|---|---|---|
| NS-NCN | Number of Contained Namespaces | Computes the number of all owned namespaces. |
| NS-NDCC | Number of Directly Contained Classes | Computes the number of classes defined inside the namespace. |
| NS-NCC | Number of Contained Classes. | Computes the number of classes owned by the namespace and all the contained namespaces. |
| NS-DNT | Depth of Nesting Tree | Computes the level of the namespace in the tree that describes the nesting relation between namespaces. The height of a node associated to a namespace that does not include another namespace is 0. |
| CLS-NLP | Number of Local Properties | Counts the attributes and the associated ends that are defined in a Class without considering the inherited properties |
| CLS-NP | Number of Properties | Counts all the properties of a class considering also the inherited properties, considering overridden properties only once. |
| CLS-NLO | Number of Local Operations | Similar to CLS-NLP |
| CLS-NO | Number of Operations | Similar to CLS-NP |
| CLS-ACLO | Average Complexity of Local Operations | Computes the ratio of the sum of complexity for every local operation and the number of local operations. |
| CLS-ACO | Average Complexity of Operations | Similar to CLS-ALCPO |
| CLS-DIG | Depth of Inheritance Graph | Computes the maximum height in the existing inheritance graph. |
| CLS-NDA | Number of Direct Ancestors | Computes the number of directly inherited classes |
| CLS-NA | Number of Ancestors | Computes the number of all the inherited classes. If a class is inherited more than |

| | | |
|---|---|---|
| | | once, this metric counts all its appearances. |
| CLS-NDD | Number of Direct Descendants | Computes the number of directed specializations. |
| CLS-ND | Number of Descendants | Similar to CLS-NA, except that specializations are counted. |
| CLS-NMI | Number of Multiple Inheritances | Computes the number of classes that are inherited more than once, considering all the appearances. |
| CLS-NRDC | Number of Referred Classes. | Computes the number of classes that are used directly as attributes' and association ends' types, and inside operations. Operations' signature and body are both checked for appearances. Primitive data types are not considered. |
| CLS-NRE | Number of Referees | Computes the number of classes that refer to a class. |
| CLS-LC | Local Complexity | 2*CLS-NLP + ΣMCC(o)<br>where o is a local operation. For each property both a getter and a setter is considered. |
| CLS-C | Complexity | 2*CLS-NP+ΣMCC(o) where o is a local or inherited operation. |
| OPER-MCC | McCabe Complexity | Computes the McCabe metric. |
| OPER-NP | Number of parameters | Counts the number of parameters including the return type. |
| OCL-NDP | Number of Decision Points | Counts the number of existing OCL iteration expressions. |
| OCL-HNT | Height of Nesting Tree | Counts the height of the nesting tree that describes the nesting relation. Nesting relations that appear in OCL iterations and let expressions are considered. |
| OCL-MCC | McCabe complexity | Computes the McCabe metric for the OCL |

| | | |
|---|---|---|
| | | expression, considering OCL iterations as loop actions. |
| OCL-HALC | Halstead Complexity | Computes the Halstead metric. |
| OCL-NV | Number of Variables | Counts the number of variables used in an OCL expression. |

**Metrics for external attributes**

A quality model implies a set of metrics and boundary limits for each metric. The maintainability of a UML model is measure at the model and class level according to the following formulas.

1. MODEL-MAIN = MODEL-CHAN + MODEL-TEST
2. CLS-MAIN = CLS-ANAL + CLS-CHAN + CLS-STAB + CLS-TEST

**Class level**

**Analyzability**: CLS-ANAL = CLS-LC + CLS-NA + ΣCLS-ANAL(c) where c is a referred class.

Definition: Measures the effort to diagnose the errors, the cause of errors, or the parts that need to be changed. The evaluation of this effort is in strong correlation with the value of other metrics: local complexity, number of ancestors, and referred classes.

**Changeability**: CLS-CHAN = CLS-USAB + CLS-SPEC
Definition: The changeability of a class is the sum of the usability and the specialization of the class.

**Usability**: CLS-USAB = CLS-NLP + CLS-NLO
Definition: The usability of a class is defined as the sum of:

- The number of local properties.
- The number of local operations.

Justification: This metric measures the effort required before a class is used. The number of local properties is multiplied by two because of the presence of get/set methods. The higher the value of the metric, the harder the class is to use.

**Specialization**: CLS-SPEC = CLS-NLP + CLS-NLO + 10*CLS-NA.

Definition: The specialization of a class is defined as the sum of

- The number of local properties.
- The number of local operations.
- Ten times the number of all inherited classes.

Justification: This metric measure the effort required before a class is specialized. The number of ancestors is multiplied by a factor as an inherited class defines a set of properties and operations that need to be analysed. The higher the specialization is the harder is to speciliaze the class.

**Stability**: CLS-STAB = CLS-ND + CLS-NRE

Definition: Measures the risk that an unexpected consequence appears after some changes are performed inside a class. The evaluation derives from the number of the classes that depend of the class (the descendants and the referees).

**Testability**: CLS-TEST = CLS-LC

Definition: Testability is the local complexity of class.

Justification: The higher the complexity of a class is, the harder the class is to test. Testability is based on the computation of McCabe cyclomatic complexity.

Limits

| Acronym | Min | Max |
|---|---|---|
| CLS-MAIN | 0 | 400 |
| CLS- ANAL | 0 | 100 |
| CLS-CHAN | 0 | 100 |
| CLS-STAB | 0 | 100 |
| CLS-TEST | 0 | 100 |
| CLS-USAB | 0 | 10 |

| | | |
|---|---|---|
| CLS-SPEC | 0 | 25 |

**Model level**

**Changeability**: MODEL-CHAN = MODEL-HIG + MODEL-ACC + MODEL-AOCC

Measures the effort required to change the model or to fix some defects. The evaluation of this effort depends of the depth of the inheritance graph and the average complexity of classes and OCL constraints.

**Testability:** MODEL-TEST = MODEL-AMC + MODEL-AOCC

Measures the effort required to validate the model. The effort of validation depends of the average complexity of methods and OCL constraints.

Limits

| Acronym | Min | Max |
|---|---|---|
| MODEL-MAIN | 0 | 200 |
| MODEL-CHAN | 0 | 100 |
| MODEL-TEST | 0 | 100 |

# Appendix 5. YATL-LEXICAL GRAMMAR

Five basic elements make up the lexical structure of a YATL source file: line terminators, white space, comments, and tokens. Of these basic elements, only tokens are significant in the syntactic grammar of a YATL program.

For compatibility with source code editing tools that add end-of-file markers, and to enable a source file to be viewed as a sequence of properly terminated lines, the following transformations are applied, in order, to every source file in a C# program:

- If the last character of the source file is a Control-Z character, this character is deleted.
- A carriage-return character is added to the end of the source file if that source file is non-empty and if the last character of the source file is not a carriage return, a line feed, a line separator, or a paragraph separator.

The *input* production defines the lexical structure of a YATL source file. Each source file in a YATL program must conform to this lexical grammar production.

*input* $\rightarrow \lambda$ | *input-element* | *input input-element.*

*input-element* $\rightarrow$ *line-terminator* | *whitespace*| *comment*| *token.*

Line terminators divide the characters of a C# source file into lines. YATL uses the following markers to indicate the end of a line:

- Carriage return character (*U+000D*)
- Line feed character (*U+000A*)
- Carriage return character (*U+000D*) followed by line feed character (*U+000A*)
- Next line character (*U+0085*)
- Line separator character (*U+2028*)
- Paragraph separator character (*U+2029*)

YATL's tokens are based on OCL tokens [OCL20],[ALP03]. It adds only the following keywords:

| | | | |
|---|---|---|---|
| apply | do | namespace | start |
| break | foreach | new | track |
| build | import | null | transformation |
| continue | in | query | while |
| delete | match | rule | |

and the assignment operator :=.

# Appendix 6. YATL-SYNTAX GRAMMAR

*translation-unit* →

*import-list starting-rule namespace-declaration-list .*

*import-list* →

*λ |*

*import-list import-declaration .*

*import-declaration* →

*‘import’ simple-name ‘.’ ‘*’ ‘;’ .*

*starting-rule* →

*‘start’ pathname ‘;’ .*

*namespace-declaration-list* →

*λ |*

*namespace-declaration-list namespace-declaration .*

*namespace-declaration* →

*'namespace' simple-name '(' models ')' '{' (query|transformation)* '}' .*

*models* →

*source-model [',' target-model].*

*transformation* →

*‘transformation’ simple-name ‘{‘ rule* ‘}’ .*

*rule* →

*'rule' simple-name filter '(' [param (',' param)*] ')' compound-stm .*

*filter* →

*'match' filter-path .*

*filterPath* →

*filter-step |*

*filter-path '::' filter-step .*

*filter-step* →

*simple-name ['[' ocl-expression ']']*

*action-list* →

*λ |*

*action-list action .*

*action →*

*declaration-stm |*

*expression-stm |*

*compound-stm |*

*if-stm |*

*loop-stm |*

*break-stm |*

*continue-stm |*

*apply-stm .*

*declaration-stm →*

*‘let’ variable-declaration-list ‘;’ .*

*expression-stm →*

*[ expression ‘;’ ] .*

*compound-stm →*

*‘{‘action-list:list ‘}’.*

*if-stm →*

*‘iff’ ocl-expression ‘then’ action [ ‘else’ action ] ‘endif’.*

*loop-stm →*

*‘while’ ocl-expression ‘do’ action |*

*‘do’ action ‘while’ ‘(‘ocl-expression ‘)’ ‘;’ |*

*‘foreach’ variable-declaration ‘in’ ocl-expression ‘do’ action .*

*break-stm →*

*‘break’ ‘;’ .*

*continue-stm →*

*‘continue’ ‘;’ .*

*apply-stm →*

*‘apply’ pathname ‘(‘ [ocl-expression (‘,’ ocl-expression)*] ‘)’ ‘;’*

*delete-stm →*

*‘delete’ ocl-expression ‘;’ .*

*expression →*

*assignment-expression |*

*ocl-expression |*

*track-expression .*

*assignment-expression →*

*ocl-expression ‘:=’ rhs-expression .*

*rhs-expression →*

*ocl-expression* |

*new-expression* |

*build-expression* |

*track-expression* .

*new-expression →*

*‘new’ path-name* .

*build-expression →*

*‘build’ path-name ‘{‘ [pair (‘,’ pair)*] ‘}’.*

*pair →*

*name ‘:=’ rhs-expression* .

*track-expression →*

*‘track’ ‘(‘ ocl-expression ‘,’ simple-name ‘,’ ocl-expression ‘)’* |

*‘track’ ‘(‘ ‘null’ ‘,’ simple-name ‘,’ ocl-expression ‘)’* |

*‘track’ ‘(‘ ocl-expression ‘,’ simple-name ‘,’ ‘null’ ‘)’* .

*query →*

*‘query’ simple-name ‘{‘ context-declaration-list ‘}’* .

Nonterminal ocl-expression, variable-declaration, and context-declaration-list are described in [OCL2] and [ALP03].

# Appendix 7. MAPPING FROM UML MODEL TO JAVA MODEL

```
start kmf::uml2java::main;

namespace kmf(uml, java) {
  transformation uml2java {
   -- 1-1 Mappings
   -- Map a UML package to a Java package
   rule umlPkg2JavaPkg
        match uml::Model_Management::Package () {
    -- Create Java package
    let jPkg: javaModel::JavaPackage;
    jPkg := new javaModel::JavaPackage;
    -- Set name
    jPkg.name := self.name.body_;
    -- Store mapping
    track(self, pkg2pkg, jPkg);
   }

   -- Map a UML class to a Java class
   rule umlClass2JavaClass
        match uml::Foundation::Core::Class () {
    -- Create Java class
    let jClass: javaModel::JavaClass;
    jClass := new javaModel::JavaClass;
    -- Set name
    jClass.name := self.name.body_;
    -- Store mapping
    track(self, class2class, jClass);
   }

   -- Map a UML attribute to a Java field
   rule umlAttribute2JavaField
        match uml::Foundation::Core::Attribute () {
    -- Create a Java Field
```

```
 let jField: javaModel::JavaField;
 jField := new javaModel::JavaField;
 -- Set name
 jField.name := self.name.body_;
 -- Store mapping
 track(self, attribute2field, jField);
}

 -- Map a UML association end to a Java field
rule umlAssociationEnd2JavaField
     match uml::Foundation::Core::AssociationEnd (){
 -- Create the Java field
 let jField: javaModel::JavaField;
 jField := new javaModel::JavaField;
 -- Set name
 iff self.name.oclIsUndefined() then
  jField.name := self.type.name.body_;
 else
  jField.name := self.name.body_;
 endif
 -- Store mapping
 track(self, associationEnd2field, jField);
}

-- Map a UML method to a Java operation
rule umlOperation2JavaMethod
     match uml::Foundation::Core::Operation () {
 -- Create a Java Method
 let jMethod: javaModel::JavaMethod;
 jMethod := new javaModel::JavaMethod;
 -- Set name
 jMethod.name := self.name.body_;
 -- Store mapping
 track(self, operation2method, jMethod);
}

-- Link all the elements to the corresponding package
rule linkElements2Pkg
     match uml::Model_Management::Package () {
 -- Get the corresponding JavaPackage
 let jPkg: javaModel::JavaPackage;
 jPkg = track(self, pkg2pkg, null);
 -- For each owned element
```

```
 foreach e:uml::Foundation::Core::Classifier
         in self.ownedElement do {
  -- Get the Java classifier
  let jCls: javaModel::JavaClassifier;
  jCls := track(e, class2class, null);
  jPkg.elements := jPkg.elements->including(jCls);
 }
}

-- Link all the fields to the corresponding class
rule linkAttribute2Class
     match uml::Foundation::Core::Attribute () {
 -- Get the Java Class that owns the corresponding field
 let umlOwner: uml::Foundation::Core::Classifier,
      jClass : javaModel::JavaClass;
 umlOwner := self.owner;
 jClass := track(umlOwner, class2class, null);
 -- Get the Java Field
 let jField: javaModel::JavaField;
 jField := track(self, attribute2field, null);
 -- Link field and class
 jClass.fields := jClass.fields->including(jField);
 jField.javaClass := jClass;
}
rule linkAssociationEnd2Class
     match uml::Foundation::Core::AssociationEnd () {
 -- Get the AssociationEnds
 let ends: Set(uml::Foundation::Core::AssociationEnd) =
            self.association.connection->asSet();
 let otherEnd: uml::Foundation::Core::AssociationEnd =
            (ends->asSet()-Set{self})->asSequence()->at(1);
 -- Get the Java Class that owns the corresponding field
 let umlOwner: uml::Foundation::Core::Classifier,
      jClass: javaModel::JavaClass;
 umlOwner := otherEnd.type;
 jClass := track(umlOwner, class2class, null);
 -- Get the Java Field
 let jField: javaModel::JavaField;
 jField := track(self, associationEnd2field, null);
 -- Link field and class
 jClass.fields := jClass.fields->including(jField);
 jField.javaClass := jClass;
}
```

```
    -- Link all the operations to the corresponding class
    rule linkOperation2Class
         match uml::Foundation::Core::Operation () {
     -- Get the UML Class that owns the attribute
     let umlOwner: uml::Foundation::Core::Classifier,
          jClass: javaModel::JavaClass;
     umlOwner := self.owner;
     jClass := track(umlOwner, class2class, null);
     -- Get the Java Method
     let jMethod: javaModel::JavaMethod;
     jMethod := track(self, operation2field, null);
     -- Link method and class
     jClass.methods := jClass.methods->including(jMethod);
     jMethod.javaClasses := jMethod.javaClasses->including(jClass);
    }

    -- main rule
    rule main () {
     -- Map individual elements
     apply umlPkg2JavaPkg();
     apply umlClass2JavaClass();
     apply umlAttribute2JavaField();
     apply umlAssociationEnd2JavaField();
     apply umlOperation2JavaMethod();
     -- Add element to Java packages
     apply linkElements2Pkg();
     -- Add fields to Java classes
     apply linkAttribute2Class();
     apply linkAssociationEnd2Class();
     -- Add operations to Java classes
     apply linkOperation2Class();
    }
   }
}
```

# Appendix 8. MAPPING FROM SPIDER DIAGRAMS MODEL TO OCL MODEL

**Java program that populates the spider diagram model instance**

```
SdRepository rep = new SdRepository$Class();
// Create contours
Contour a = (Contour)rep.buildElement("sd.as.Contour");
a.setName("a");
Contour b = (Contour)rep.buildElement("sd.as.Contour");
b.setName("b");
Contour c = (Contour)rep.buildElement("sd.as.Contour");
c.setName("c");
// Create zone (a | b)
Zone z1 = (Zone)rep.buildElement("sd.as.Zone");
z1.getContainingContours().add(a);
z1.getExcludingContours().add(b);
// Create zone (b | a)
Zone z2 = (Zone)rep.buildElement("sd.as.Zone");
z2.getContainingContours().add(b);
z2.getExcludingContours().add(a);
// Create zone (a, b |)
Zone z3 = (Zone)rep.buildElement("sd.as.Zone");
z3.getContainingContours().add(a);
z3.getContainingContours().add(b);

// Create diagram containing all the zones
UnitaryDiagram ud1 =
     (UnitaryDiagram)rep.buildElement("sd.as.UnitaryDiagram");
ud1.getZones().add(z1);
ud1.getZones().add(z2);
ud1.getZones().add(z3);
// Save repository
rep.saveXMI("src/test/scripts/sdRep.xml");
```

**YATL program**

```
start kmf::sd2ocl::main;

namespace kmf(sd, ocl) {
 transformation sd2ocl {
  -- 1-1 Mappings
  -- Map a SD unitary diagram to an OCL expression
  rule ud2let match sd::as::UnitaryDiagram () {
    -- Create let expression
    let letExp: syntax::ast::expressions::LetExpAS;
    letExp := new syntax::ast::expressions::LetExpAS;
    -- Store mapping
    track(self, ud2let, letExp);
  }

  -- Map a SD zone to a variable: init exppression computes the set
  rule z2var match sd::as::Zone () {
   --
   -- Create name(zone): Set{OclAny} = OclAny.allInstances()
   -- ->select(x:OclAny | x.isKindOf() and ... and not x.isKindOf()
   -- and ... and not )
   --
   -- Create OclAny type
   let oclAnyType: syntax::ast::types::ClassifierAS;
   oclAnyType := new syntax::ast::types::ClassifierAS;
   oclAnyType.pathName := Sequence{'OclAny'};
   -- Create type Set{OclAny}
   let setType: syntax::ast::types::SetTypeAS;
   setType := new syntax::ast::types::SetTypeAS;
   setType.elementType := oclAnyType;
   -- Create pathName expression 'OclAny'
   let oclAnyPathNameExp: syntax::ast::expressions::PathNameExpAS;
   oclAnyPathNameExp := new syntax::ast::expressions::PathNameExpAS;
   oclAnyPathNameExp.pathName := Sequence{'OclAny'};
   -- Create OclAny.allInstances selection
   let allInstancesSelection:
       syntax::ast::expressions::DotSelectionExpAS;
   allInstancesSelection :=
       new syntax::ast::expressions::DotSelectionExpAS;
   allInstancesSelection.source := oclAnyPathNameExp;
   allInstancesSelection.name := 'allInstances';
```

```
-- Create OclAny.allInstances() operation call
let allInstancesCall:
    syntax::ast::expressions::OperationCallExpAS;
allInstancesCall :=
    new syntax::ast::expressions::OperationCallExpAS;
allInstancesCall.source := allInstancesSelection;
allInstancesCall.arguments := Sequence{};
-- Create OclAny.allInstances()->select selection
let selectExp: syntax::ast::expressions::ArrowSelectionExpAS;
selectExp := new syntax::ast::expressions::ArrowSelectionExpAS;
selectExp.source := allInstancesCall;
selectExp.name := 'select';
-- Create x: OclAny variable declaration
let xVar: syntax::ast::contexts::VariableDeclarationAS;
xVar := new syntax::ast::contexts::VariableDeclarationAS;
xVar.name := 'x';
xVar.type := oclAnyType;
-- Create filters: isKindOf and notIsKindOf
let filters: Sequence(syntax::ast::expressions::OclExpressionAS);
filters := Sequence{};
let isKindOfSelection:
    syntax::ast::expressions::DotSelectionExpAS;
let isKindOfCall: syntax::ast::expressions::OperationCallExpAS;
let contourPathNameExp: syntax::ast::expressions::PathNameExpAS;
foreach c: sd::as::Contour in self.containingContours do {
 -- Create name(c) path name
 contourPathNameExp :=
    new syntax::ast::expressions::PathNameExpAS;
 contourPathNameExp.pathName := Sequence{c.name};
 -- Create x.isKindOf
 isKindOfSelection :=
    new syntax::ast::expressions::DotSelectionExpAS;
 isKindOfSelection.source := xVar;
 isKindOfSelection.name := 'isKindOf';
 -- Create x.isKindOf(c.name)
 isKindOfCall :=
    new syntax::ast::expressions::OperationCallExpAS;
 isKindOfCall.source := isKindOfSelection;
 isKindOfCall.arguments := Sequence{contourPathNameExp};
 -- Add it to filters
 filters := filters->including(isKindOfCall);
}
foreach c: sd::as::Contour in self.excludingContours do {
```

```
 -- Create name(c) path name
 contourPathNameExp :=
    new syntax::ast::expressions::PathNameExpAS;
 contourPathNameExp.pathName := Sequence{c.name};
 -- Create x.isKindOf
 isKindOfSelection :=
    new syntax::ast::expressions::DotSelectionExpAS;
 isKindOfSelection.source := xVar;
 isKindOfSelection.name := 'isKindOf';
 -- Create x.isKindOf(c.name)
 isKindOfCall :=
    new syntax::ast::expressions::OperationCallExpAS;
 isKindOfCall.source := isKindOfSelection;
 isKindOfCall.arguments := Sequence{contourPathNameExp};
 -- Create not x.isKindOf(c.name)
 let notSelection: syntax::ast::expressions::DotSelectionExpAS;
 notSelection := new syntax::ast::expressions::DotSelectionExpAS;
 notSelection.source := isKindOfCall;
 notSelection.name := 'not';
 let notCall: syntax::ast::expressions::OperationCallExpAS;
 notCall := new syntax::ast::expressions::OperationCallExpAS;
 notCall.source := notSelection;
 notCall.arguments := Sequence{};
 -- Add it to filters
 filters := filters->including(notCall);
}
-- Compute iterator's body
let itBody: syntax::ast::expressions::OclExpressionAS;
itBody := filters->at(1);
let i:Integer = 2;
while i <= filters->size() do {
 -- Create itBody.and
 let andSelection: syntax::ast::expressions::DotSelectionExpAS;
 andSelection := new syntax::ast::expressions::DotSelectionExpAS;
 andSelection.name := 'and';
 andSelection.source := itBody;
 -- Create itBody.and(args)
 let andCall: syntax::ast::expressions::OperationCallExpAS;
 andCall := new syntax::ast::expressions::OperationCallExpAS;
 andCall.source := andSelection;
 andCall.arguments := Sequence{filters->at(i)};
 -- Set new value for itBody
 itBody := andCall;
```

```
  -- Next filter
  i := i + 1;
 }
 -- Create iterator expression OclAny.allInstances()->select(...)
 let iteratorExp: syntax::ast::expressions::IteratorExpAS;
 iteratorExp := new syntax::ast::expressions::IteratorExpAS;
 iteratorExp.source := selectExp;
 iteratorExp.iterator := xVar;
 iteratorExp.loopBody := itBody;
 -- Compute zone's name
 let zName: String = '';
 foreach c: sd::as::Contour in self.containingContours do {
  zName := zName.concat(c.name);
  zName := zName.concat('_');
 }
 zName := zName.concat('|');
 foreach c: sd::as::Contour in self.excludingContours do {
  zName := zName.concat('_');
  zName := zName.concat(c.name);
 }
 -- Create name(zone):Set{OclAny} :=
 --    OclAny.allInstances()->select( ... )
 let var: syntax::ast::contexts::VariableDeclarationAS;
 var := new syntax::ast::contexts::VariableDeclarationAS;
 var.name := zName;
 var.type := setType;
 var.initExp := iteratorExp;
 -- Store mapping
 track(self, z2var, var);
}

-- Map a SD to let's body (in expression)
rule ud2in match sd::as::UnitaryDiagram () {
 -- Make a list of conditions for each zone
 let ands: Sequence(syntax::ast::expressions::OclExpressionAS) =
     Sequence{};
 -- For each zone
 foreach z: sd::as::Zone in self.zones do {
  -- Compute the number of spiders touching the zone
  -- All spiders are single footed
  let feetNo: Integer = 0;
  foreach s: sd::as::Spider in self.spiders do {
   iff s.habitat->includes(z) then
```

```
  feetNo := feetNo + 1;
 endif
}
-- Compute is shaded flag
let isShaded: Boolean = self.shadedZones->includes(z);
-- Make the expression that checks the size
-- name(z)->size() operator feetNo
-- Make name(z) expression
let varExp: syntax::ast::expressions::VariableExpAS;
varExp := new syntax::ast::expressions::VariableExpAS;
varExp.variableDeclarationAS := track(z, z2var, null);
-- Make name(z)->size
let selectExp: syntax::ast::expressions::ArrowSelectionExpAS;
selectExp := new syntax::ast::expressions::ArrowSelectionExpAS;
selectExp.source = varExp;
selectExp.name := 'size';
-- Make name(z)->size()
let callExp: syntax::ast::expressions::OperationCallExpAS;
callExp := new syntax::ast::expressions::OperationCallExpAS;
callExp.source := selectExp;
-- Make operator
let opName: String = '>=';
iff isShaded then
 opName := '=';
endif
-- Make name(z)->size() <=
let selExp: syntax::ast::expressions::DotSelectionExpAS;
selExp := new syntax::ast::expressions::DotSelectionExpAS;
selExp.source := callExp;
selExp.name := opName;
-- Make feetName exp
let argExp: syntax::ast::expressions::IntegerLiteralExpAS;
argExp := new syntax::ast::expressions::IntegerLiteralExpAS;
argExp.value := feetNo;
-- Make name(z)->size() <= feetNo
let relCall: syntax::ast::expressions::OperationCallExpAS;
relCall :=  new syntax::ast::expressions::OperationCallExpAS;
relCall.source := selExp;
relCall.arguments := relCall.arguments->including(argExp);
--
-- Add exp to ands
--
ands := ands->including(relCall);
```

```
 }
 -- Make a logical expression from ands
 iff ands->size() >= 1 then {
  let inExp: syntax::ast::expressions::OclExpressionAS;
  inExp := ands->at(1);
  let i:Integer = 2;
  while i<=ands->size() do {
   -- Make an and
   let andSel: syntax::ast::expressions::DotSelectionExpAS;
   andSel := new syntax::ast::expressions::DotSelectionExpAS;
   andSel.source := inExp;
   andSel.name := 'and';
   let andCall: syntax::ast::expressions::OperationCallExpAS;
   andCall := new syntax::ast::expressions::OperationCallExpAS;
   andCall.source := andSel;
   andCall.arguments := andCall.arguments->including(ands->at(i));
   -- Update inExp for next iteration
   inExp := andCall;
   -- Next
   i := i+1;
  }
  -- Store mapping
  track(self, ud2in, inExp);
 }
 endif
}

-- Link let expressions to variables
rule linkLet2Variables match sd::as::UnitaryDiagram () {
 -- Get let expression
 let letExp: syntax::ast::expressions::LetExpAS;
 letExp := track(self, ud2let, null);
 -- For each zone
 foreach z: sd::as::Zone in self.zones do {
  let var:syntax::ast::contexts::VariableDeclarationAS;
  var := track(z, z2var, null);
  letExp.variables := letExp.variables->including(var);
 }
}

-- Link let expressions to variables
rule linkLet2In match sd::as::UnitaryDiagram () {
 -- Get let expression
```

```
    let letExp: syntax::ast::expressions::LetExpAS;
    letExp := track(self, ud2let, null);
    -- Get in expression
    let inExp: syntax::ast::expressions::OclExpressionAS;
    inExp := track(self, ud2in, null);
    -- Link them
    letExp.inExp := inExp;
   }

   -- main rule
   rule main () {
    -- Create a let expression for each unitary diagram
    apply ud2let();
    -- Create a variable declaration for each zone
    apply z2var();
    -- Create the in expression
    apply ud2in();
    -- Link diagrams to variables
    apply linkLet2Variables();
    -- Link diagrams to in
    apply linkLet2In();
   }
  }
 }
```

# Appendix 9. MAPPING FROM EDOC TO WS

**Java code to populate the source model instance**

```
//
// Create EDOC population
//
protected static DataType makeDataType(Repository rep, String type) {
      DataType dt = (DataType)rep.buildElement("edoc.ECA.DocumentModel.DataType");
      dt.setName(type);
      return dt;
}
protected static Attribute makeAttribute(Repository rep, String name,
            DataElement type) {
      Attribute at = (Attribute)rep.buildElement("edoc.ECA.DocumentModel.Attribute");
      at.setName(name);
      at.setType(type);
      return at;
}
protected static CompositeData makeCompositeType(Repository rep, String name,
            List dataElements) {
      CompositeData dt =
            (CompositeData)rep.buildElement("edoc.ECA.DocumentModel.CompositeData");
      dt.setName(name);
      dt.setFeatures(dataElements);
      return dt;
}
protected static Protocol makeProtocol(Repository rep, String name) {
      Protocol p = (Protocol)rep.buildElement("edoc.ECA.CCA.Protocol");
      p.setName(name);
      return p;
}
protected static FlowPort makeFlowPort(Repository rep,String name,DataElement type) {
      FlowPort fp = (FlowPort)rep.buildElement("edoc.ECA.CCA.FlowPort");
      fp.setName(name);
      fp.setType(type);
      return fp;
}
protected static ProtocolPort makeProtocolPort(Repository rep, String name) {
      ProtocolPort pp = (ProtocolPort)rep.buildElement("edoc.ECA.CCA.ProtocolPort");
      pp.setName(name);
      return pp;
```

```
}
protected static OperationPort makeOperationPort(Repository rep, String name,
            FlowPort call, FlowPort ret) {
      OperationPort op =
            (OperationPort)rep.buildElement("edoc.ECA.CCA.OperationPort");
      op.setName(name);
      op.getPorts().add(call);
      op.getPorts().add(ret);
      return op;
}
protected static Repository initEDOCPopulation() {
      EdocRepository rep = new EdocRepository$Class();
      // Create simple types
      DataType stringType = makeDataType(rep, "String");
      DataType integerType = makeDataType(rep, "Integer");
      DataType realType = makeDataType(rep, "Real");
      // Create attributes
      Attribute airlineName = makeAttribute(rep, "AirlineName", stringType);
      Attribute flightNo = makeAttribute(rep, "FlightNo", integerType);
      Attribute location = makeAttribute(rep, "Location", stringType);
      Attribute date = makeAttribute(rep, "Date", stringType);
      Attribute hotelName = makeAttribute(rep, "HotelName", stringType);
      Attribute address = makeAttribute(rep, "Address", stringType);
      Attribute companyName = makeAttribute(rep, "CompanyName", stringType);
      Attribute period = makeAttribute(rep, "Period", integerType);
      // Create composite types
      List locationInfo = new Vector();
      locationInfo.add(location);
      locationInfo.add(date);
      CompositeData locationType = makeCompositeType(rep, "Location", locationInfo);
      List flightInfo = new Vector();
      flightInfo.add(airlineName);
      flightInfo.add(flightNo);
      flightInfo.add(date);
      CompositeData flightType = makeCompositeType(rep, "Flight", flightInfo);
      List hotelInfo = new Vector();
      hotelInfo.add(hotelName);
      hotelInfo.add(address);
      hotelInfo.add(date);
      hotelInfo.add(period);
      CompositeData hotelType = makeCompositeType(rep, "Hotel", hotelInfo);
      List carInfo = new Vector();
      carInfo.add(companyName);
      carInfo.add(address);
      carInfo.add(date);
      carInfo.add(period);
      CompositeData carType = makeCompositeType(rep, "Car", carInfo);
      // Create BuySell protocol
      Protocol buySellProt = makeProtocol(rep, "BuySell");
      ProtocolPort buyPort = makeProtocolPort(rep, "Buy");
      buyPort.setDirection(DirectionType$Class.Initiates);
      buyPort.setOwner(buySellProt);
      buyPort.setUses(buySellProt);
```

```
ProtocolPort sellPort = makeProtocolPort(rep, "Sell");
sellPort.setDirection(DirectionType$Class.Responds);
sellPort.setOwner(buySellProt);
sellPort.setUses(buySellProt);
buySellProt.getPorts().add(buyPort);
buySellProt.getPorts().add(sellPort);
// Create BuyFlight protocol
Protocol buyFlightProt = makeProtocol(rep, "BuyFlight");
ProtocolPort buyFlightPort = makeProtocolPort(rep, "BuyFlight");
buyFlightPort.setDirection(DirectionType$Class.Initiates);
buyFlightPort.setOwner(buyFlightProt);
buyFlightPort.setUses(buyFlightProt);
ProtocolPort flightPort = makeProtocolPort(rep, "Flight");
flightPort.setDirection(DirectionType$Class.Responds);
flightPort.setOwner(buyFlightProt);
flightPort.setUses(buyFlightProt);
buyFlightProt.getPorts().add(buyFlightPort);
buyFlightProt.getPorts().add(flightPort);
// Add operation protocols
FlowPort locationPort = makeFlowPort(rep, "Location", locationType);
locationPort.setDirection(DirectionType$Class.Initiates);
FlowPort flightFlowPort = makeFlowPort(rep, "FlightInfo", flightType);
flightFlowPort.setDirection(DirectionType$Class.Responds);
OperationPort    findFlightPort    =    makeOperationPort(rep,    "FindFlight",
locationPort, flightFlowPort);
buyFlightProt.getPorts().add(findFlightPort);
// Create reserveRoom protocol
Protocol reserveRoomProt = makeProtocol(rep, "ReserveRoom");
ProtocolPort reserveRoomPort = makeProtocolPort(rep, "ReserveRoom");
reserveRoomPort.setDirection(DirectionType$Class.Initiates);
reserveRoomPort.setOwner(reserveRoomProt);
reserveRoomPort.setUses(reserveRoomProt);
ProtocolPort roomPort = makeProtocolPort(rep, "Room");
roomPort.setDirection(DirectionType$Class.Responds);
roomPort.setOwner(reserveRoomProt);
roomPort.setUses(reserveRoomProt);
reserveRoomProt.getPorts().add(reserveRoomPort);
reserveRoomProt.getPorts().add(roomPort);
// Create rentCar protocol
Protocol rentCarProt = makeProtocol(rep, "RentCar");
ProtocolPort rentCarPort = makeProtocolPort(rep, "RentCar");
rentCarPort.setDirection(DirectionType$Class.Initiates);
rentCarPort.setOwner(rentCarProt);
rentCarPort.setUses(rentCarProt);
ProtocolPort carPort = makeProtocolPort(rep, "Car");
carPort.setDirection(DirectionType$Class.Responds);
carPort.setOwner(rentCarProt);
carPort.setUses(rentCarProt);
rentCarProt.getPorts().add(rentCarPort);
rentCarProt.getPorts().add(carPort);
// Create payment protocol
Protocol paymentProt = makeProtocol(rep, "Payment");
ProtocolPort taPaymentPort = makeProtocolPort(rep, "TAPayment");
```

```
taPaymentPort.setDirection(DirectionType$Class.Initiates);
taPaymentPort.setOwner(paymentProt);
taPaymentPort.setUses(paymentProt);
ProtocolPort bPaymentPort = makeProtocolPort(rep, "BPayment");
bPaymentPort.setDirection(DirectionType$Class.Responds);
bPaymentPort.setOwner(paymentProt);
bPaymentPort.setUses(paymentProt);
paymentProt.getPorts().add(taPaymentPort);
paymentProt.getPorts().add(bPaymentPort);
// Create ShipDelivery protocol
Protocol shipDeliveryProt = makeProtocol(rep, "ShipDelivery");
ProtocolPort shipPort = makeProtocolPort(rep, "Ship");
shipPort.setDirection(DirectionType$Class.Initiates);
shipPort.setOwner(shipDeliveryProt);
shipPort.setUses(shipDeliveryProt);
ProtocolPort deliveryPort = makeProtocolPort(rep, "Delivery");
deliveryPort.setDirection(DirectionType$Class.Responds);
deliveryPort.setOwner(shipDeliveryProt);
deliveryPort.setUses(shipDeliveryProt);
shipDeliveryProt.getPorts().add(shipPort);
shipDeliveryProt.getPorts().add(deliveryPort);
// Create Client
ProcessComponent client =
      (ProcessComponent)rep.buildElement("edoc.ECA.CCA.ProcessComponent");
client.setName("Client");
client.getPorts().add(buyPort);
client.getPorts().add(deliveryPort);
buyPort.setOwner(client);
deliveryPort.setOwner(client);
// Create Travel Agency
ProcessComponent travelAgency =
      (ProcessComponent)rep.buildElement("edoc.ECA.CCA.ProcessComponent");
travelAgency.setName("Expedia");
travelAgency.getPorts().add(sellPort);
travelAgency.getPorts().add(buyFlightPort);
travelAgency.getPorts().add(findFlightPort);
travelAgency.getPorts().add(reserveRoomPort);
travelAgency.getPorts().add(rentCarPort);
travelAgency.getPorts().add(taPaymentPort);
sellPort.setOwner(travelAgency);
buyFlightPort.setOwner(travelAgency);
reserveRoomPort.setOwner(travelAgency);
rentCarPort.setOwner(travelAgency);
taPaymentPort.setOwner(travelAgency);
// Create Airline
ProcessComponent airline =
      (ProcessComponent)rep.buildElement("edoc.ECA.CCA.ProcessComponent");
airline.setName("BA");
airline.getPorts().add(flightPort);
// Create Hotel
ProcessComponent hotel =
      (ProcessComponent)rep.buildElement("edoc.ECA.CCA.ProcessComponent");
hotel.setName("Marriot");
```

```
      hotel.getPorts().add(roomPort);
      // Create CarCompany
      ProcessComponent carCompany =
            (ProcessComponent)rep.buildElement("edoc.ECA.CCA.ProcessComponent");
      carCompany.setName("CarCompany");
      carCompany.getPorts().add(carPort);
      // Save repository into an xml
      rep.saveXMI("src/test/scripts/edocRep.xml");
      return rep;
}
```

**The YATL program**

```
start kmf::edoc2ws::main;

namespace kmf(sd, ocl) {
  transformation edoc2ws {
    --
    -- EDOC.ECA.DocumentModel to  WS.XSD
    --
    -- Map an EDOC DataType to an XSD SimpleType
    rule dt2st match edoc::ECA::DocumentModel::DataType () {
      -- Create SimpleType
      let st: ws::xsd::SimpleType;
      st := new ws::xsd::SimpleType;
      st.name := self.name;
      -- Store mapping
      track(self, type2type, st);
    }
    -- Map an EDOC CompositeData to an XSD ComplexType
    rule cd2ct match edoc::ECA::DocumentModel::CompositeData () {
      -- Create ComplexType
      let ct: ws::xsd::ComplexType;
      ct := new ws::xsd::ComplexType;
      ct.name := self.name;
      -- Store mapping
      track(self, type2type, ct);
    }
    -- Map an EDOC Attribute to an XSD attribute
    rule at2at match edoc::ECA::DocumentModel::Attribute () {
      -- Create Attribute
      let at: ws::xsd::Attribute;
```

```
  at := new ws::xsd::Attribute;
  at.name := self.name;
  -- Store mapping
  track(self, at2at, at);
}
-- Link XSD attributes to XSD types
rule linkAttribute2Type
     match edoc::ECA::DocumentModel::Attribute () {
  -- Get the XSD Attribute
  let xsdAttribute: ws::xsd::Attribute;
  xsdAttribute := track(self, at2at, null);
  -- Get the type
  let edocType : edoc::ECA::DocumentModel::DataElement;
  edocType := self.type;
  let xsdType: ws::xsd::Type;
  xsdType := track(edocType, type2type, null);
  xsdAttribute.type := xsdType;
}
-- Link XSD ComplexTypes to XSD Attributes
rule linkComplexType2Attribute
     match edoc::ECA::DocumentModel::CompositeData () {
  -- Get the XSD ComplexType
  let xsdComplexType: ws::xsd::ComplexType;
  xsdComplexType := track(self, type2type, null);
  -- Add every attribute
  foreach edocAttribute : edoc::ECA::DocumentModel::Attribute
          in self.features do {
    let xsdAttribute : ws::xsd::Attribute;
    xsdAttribute := track(edocAttribute, at2at, null);
    xsdComplexType.sequence :=
     xsdComplexType.sequence->including(xsdAttribute);
  }
}
-- Map concepts from EDOC.ECA.DocumentModel to WS.XSD concepts
rule documentModel2xsd() {
  -- Create a SimpleType for each DataType
  apply dt2st();
  -- Create a ComplexType for each CompositeData
  apply cd2ct();
  -- Create an XSD Attribute for each EDOC Attribute
  apply at2at();
  -- Link XSD Attributes to XSD Types
  apply linkAttribute2Type();
```

```
  -- Link XSD ComplexTypes to XSD Attributes
  apply linkComplexType2Attribute();
}

--
-- Map concepts from EDOC.ECA.CCA to WS:WSDL
--
-- Create a WSDL Message for each EDOC FlowPort
rule flowPort2message match edoc::ECA::CCA::FlowPort () {
  -- Create Message
  let m: ws::wsdl::Message;
  m := new ws::wsdl::Message;
  m.name := self.name;
  -- Create part and add it
  let part: ws::wsdl::Part;
  part := new ws::wsdl::Part;
  part.name := self.name;
  part.type := track(self.type, type2type, null);
  m.parts := m.parts->including(part);
  -- Store mapping
  track(self, fp2m, m);
}
-- Create a WSDL Operation for each EDOC OperationPort
rule operationPort2operation
     match edoc::ECA::CCA::OperationPort () {
  -- Get input and output port
  let iPort : edoc::ECA::CCA::OperationPort;
  iPort := self.ports->asSequence()->at(1);
  let oPort : edoc::ECA::CCA::OperationPort;
  oPort := self.ports->asSequence()->at(2);
  -- Create input
  let input: ws::wsdl::Input;
  input := new ws::wsdl::Input;
  input.name := iPort.name;
  input.message := track(iPort, fp2m, null);
  -- Create outpout
  let output: ws::wsdl::Output;
  output := new ws::wsdl::Output;
  output.name := oPort.name;
  output.message := track(oPort, fp2m, null);
  -- Create Operation
  let o: ws::wsdl::Operation;
  o := new ws::wsdl::Operation;
```

```
  o.name := self.name;
  o.input := input;
  o.output := output;
  input.operation := o;
  output.operation := o;
  -- Store mapping
  track(self, op2o, o);
}
-- Create a WSDL PortType for each EDOC ProtocolPort
rule protocolPort2portType
     match edoc::ECA::CCA::ProtocolPort () {
  -- Create a portType
  let pt: ws::wsdl::PortType;
  pt := new ws::wsdl::PortType;
  pt.name := self.name;
  -- Add operations
  let ps: Set(edoc::ECA::CCA::Port) = self.owner.ports->asSet();
  let fps: Set(edoc::ECA::CCA::Port) =
     ps->select(e | e.oclIsKindOf(edoc::ECA::CCA::FlowPort));
  let ops: Set(edoc::ECA::CCA::Port) =
     ps->select(e|.oclIsKindOf(edoc::ECA::CCA::OperationPort));
  foreach op: edoc::ECA::CCA::OperationPort in ops do {
    -- Find operation
    let o: ws::wsdl::Operation;
    o := track(op, op2o, null);
    pt.operations := pt.operations->including(o);
  }
  -- Store mapping
  track(self, pp2pt, pt);
}
-- Create a WSDL Definition for each EDOC ProcessComponent
rule processComponent2service
     match edoc::ECA::CCA::ProcessComponent () {
  -- Create Definition
  let d: ws::wsdl::Definition;
  d := new ws::wsdl::Definition;
  -- Create service
  let s: ws::wsdl::Service;
  s := new ws::wsdl::Service;
  s.definition := d;
  s.name := self.name;
  -- Store mapping
  track(self, pc2s, s);
```

```
}
-- Link Definition to Types
rule linkDefinition2X
     match edoc::ECA::CCA::ProcessComponent () {
  -- Get the WSDL Service
  let s: ws::wsdl::Service;
  s := track(self, pc2s, null);
  let d : ws::wsdl::Definition;
  d := s.definition;
  -- Add every portType
  let ps : Set(edoc::ECA::CCA::Port) = self.ports->asSet();
  let fps: Set(edoc::ECA::CCA::Port) =
     ps->select(e | e.oclIsKindOf(edoc::ECA::CCA::FlowPort));
  let ops: Set(edoc::ECA::CCA::Port) =
    ps->select(e|e.oclIsKindOf(edoc::ECA::CCA::OperationPort));
  let pps: Set(edoc::ECA::CCA::Port) =
    ps->select(e|e.oclIsKindOf(edoc::ECA::CCA::ProtocolPort));
  let m: ws::wsdl::Message;
  let ms: Set(ws::wsdl::Message);
  let ts: Set(ws::xsd::Type);
  foreach fp : edoc::ECA::CCA::FlowPort in fps do {
    m := track(fp, fp2m, null);
    ms := ms->including(m);
    foreach p:ws::wsdl::Part in m.parts do {
      ts := ts->including(p.type);
    }
  }
  foreach op : edoc::ECA::CCA::OperationPort in ops do {
    -- Get input and output port
    let iPort : edoc::ECA::CCA::OperationPort;
    iPort := op.ports->asSequence()->at(1);
    let oPort : edoc::ECA::CCA::OperationPort;
    oPort := op.ports->asSequence()->at(2);
    m := track(iPort, fp2m, null);
    ms := ms->including(m);
    foreach p:ws::wsdl::Part in m.parts do {
      ts := ts->including(p.type);
    }
    m := track(oPort, fp2m, null);
    ms := ms->including(m);
    foreach p:ws::wsdl::Part in m.parts do {
      ts := ts->including(p.type);
    }
```

```
      }
      let pts : Set(ws::wsdl::PortType);
      foreach pp : edoc::ECA::CCA::ProtocolPort in pps do {
        let pt : ws::wsdl::PortType;
        pt := track(pp, pp2pt, null);
        pts := pts->including(pt);
      }
      d.messages := ms->asSequence();
      d.types := ts->asSequence();
      d.portTypes := pts->asSequence();
    }
    --- Map CCA to WSDL
    rule cca2wsdl() {
      -- Create a WSDL Message for each EDOC FlowPort
      apply flowPort2message();
      -- Map Operation Ports
      apply operationPort2operation();
      -- Map Protocol Ports
      apply protocolPort2portType();
      -- Map ProcessComponent
      apply processComponent2service();
      -- Link Definition to types, messages, and portTypes
      apply linkDefinition2X();
    }

    -- main rule
    rule main () {
      -- Map DocumentModel to XSD
      apply documentModel2xsd();
      -- ECA to WSLD
      apply cca2wsdl();
    }
  }
}
```

# BIBLIOGRAPHY


[AC94] Abreu, F. B. and Carapuca, R. (1994). Object-oriented software engineering: measuring and controlling the development process. *In Proceedings of the 4th International Conference on Software Quality*.

[AJU75] Aho, A. V., Johnson, S. C., and Ullman, J. D. (1975) Deterministic parsing of ambiguous grammars. *Commun. ACM* 18(8), pp 441-452.

[AJU77] Aho, A. V., Johnson, S. C., and Ullman, J. D. (1977) Code generation for expressions with subexpressions. *JACM*, 24(1), pp 146-160.

[AKP03] Akehurst, D., Kent, S., and Patrascoiu, O. (2003). A relational approach to defining and implementing transformations between metamodels. In *Journal of Software and Systems Modeling* (SoSym), 2(4), pp 215-239.

[ALP03] Akehurst, D., Linington, P., and Patrascoiu, O. (2003). OCL 2.0 – Implementing the Standard. Technical Report No. 12-03, Computer Laboratory, University of Kent, UK.

[AMDA] AndroMDA Project Home Page, 2005. On-line at http://www.andromda.org/

[ARG] ArgoUML Project Home Page, 2004. On-line at http://argouml.tigris.org/

[AP02] Appel, A. W. and Palsberg J. (2002). *Modern Compiler Implementation in Java*. Second ed, Cambridge University Press.

[AP03] Akehurst, D. and Patrascoiu, O. (2003). OCL 2.0 – Implementing the Standard for Multiple Metamodels. In *OCL2.0-"Industry standard or scientific playground?" - Proceedings of the UML'03 workshop*, pp 19-25.

[AP04a] Akehurst, D. and Patrascoiu, O. (2004). Prototyping Metamodels: Automated Generation of Modeling Tools with support for Checking Well-Formedness Constraints. Submitted at UML 2004.

[ASU86] Aho, A., V., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles,Techniques and Tools*. Addison-Wesley, Reading Mass.

[AU72] Aho, A. V. and Ullman, J. D. (1972). *The theory of parsing, translation, and compiling*. Prentice Hall, Engl. Cliffs.

[Bac79] Backhouse, R. C. (1979). *Syntax of Programming Languages: Theory and Practice*. Prentice Hall, Engl. Cliffs.

[BDM97] Briand, L., Devenbu, P., and Melo, W. (1997) An investigation into coupling measurement for C++. In *Proceedings of the 19st International Conference on*

*Software Engineering*, pp. 334-344.

[BDW99] Briand, L., Daly, J., and Wuest, J. (1999) A unified framework for coupling measurement in object-oriented systems. IEEE Transactions on Software Engineering, 25(1), pp 91-121.

[Bel98] Rodney Bell (1998) Code Generation from Object Models.

http://embedded.com/98/9803fe3.htm

[BM99] Benlarbi, S. and Melo, M. (1999) Polymorphism measures for early risk prediction. In *Proceedings of the 21st International Conference on Software Engineering*, pp. 334-344.

[BWW54] Burks, A. W., Warren, D. W., and Wright, J. B. (1954). An analysis of logical machine using paranthesis-free notation. In *Mathematical Tables and Other Aids to Computation*, 8(46), pp. 53-57.

[CH03] Czarnecki, K., and Helsen, S. (2003). Classification of Model Transformation Approaches. In *Generative techniques in the context of MDA – Proceedings of OOPSLA 2003 workshop*.

[Cho56] Chomsky, N. (1956) *Three models for the description of language*, IRE Transactions on Information Theory, 2, pp. 113-124.

[Cho62] Chomsky, N. (1962) *Handbook of Mathematics Psychology*, volume 2, chapter Formal Properties of Grammars, pp. 323-418. Wiley & Sons, New York.

[CK91] Chidamber, S.R. and Kemerer, C.F. (1991). Towards a metrics suite for object-oriented design. In *Proceedings of The Sixth Object-Oriented Programming Systems, Languages, and Applications*, pp. 97-211.

[CK94] Chidamber, S.R. and Kemerer, C.F. (1994). A metrics suite for object oriented design. *IEEE Transactions Software Engineering*, 6, pp. 476-493.

[Con63] Conway, M. E. (1963). Design of a separate transition-diagram compiler. *Commun. ACM 6(7)*, pp. 396-408.

[CS00] Cartwright, M. and Shepperd, M. (2000) An empirical investigation of an object-oriented software system: An exploratory analysis. IEEE Transactions on Software Engineering, 24(8), pp. 629-639.

[Cur80] Curtis, B. (1980), Measurement and Experimentation. In *Software Engineering, Proceedings of the IEEE*, 68(9).

[CWM] OMG, Common Warehouse Metamodel Specification. OMG Document formal/2003-03-02, available at http://www.omg.org/cwm.

[DeR71] DeRemer, F. L. (1971). Simple LR(k) grammars. *Commun. ACM* 14, pp. 453-460.

[DseDS] Design Support Environments for Distributed Systems (DSE4DS) project. www.cs.kent.ac.uk/projects/dse4ds/

[EBGR01] El-Eman, K., Benlarbi, S., Goel, N., and Rai, S. (2001) The confounding effect of class size on the validity of object-oriented metrics. *IEE Transactions on Software*

*Engineering*.

[EDOC] OMG, Enterprise Distributed Object Computing Specification OMG Document formal available at http://www.omg.org/technology/documents/formal/edoc.htm

[EMF] IBM, Eclipse Modeling Framework. http://www.eclipse.org.

[ET02] Erdogamus, H. and Tanir, O. (2002). *Advances in Software Engineering. Comprehension, Evaluation, and Evolution*. Springer-Verlag.

[Evan97] Evanco, W. (1997) Poisson analysis of defects from small software components. Journal of Systems and Software, 38, pp. 27-35.

[Eve63] Evey, R. J. (1963) The Theory and Applications of Pushdown Store Machines. Ph.D thesis, Harvard University, Massachusetts.

[Fen91] Fenton, N. (1991) Software Metrics: A rigorous approach, Chapman and Hall.

[FL91] Fischer, C. N. and LeBlanc, R. J. Jr. (1991). *Crafting a compiler with C*. Benjamin Cummings.

[Fra03] Frankel, D. S. (2003) *Model Driven Architrecture: Applying MDA to Enterprise Computing*. John Wiley & Sons.

[FUJ] Fujaba Tool Suite Developer Team, University of Paderborn, 2004. On-line at http://www.fujaba.de/

[Gar03] Gardner, T., Griffin, C., Koehler, J., and Hauser, R. (2003) A reviewof OMG MOF 2.0 Query/Views/Transformations submissions and recommendations towards the final standard, 1st International Workshop on Metamodeling for MDA, York, UK, 2003.

[GC87] Grady, R. B. and Caswell, D., L., (1987) Software Metrics: Establishing a Company-Wide Program, Prentice-Hall.

[GEN] Gentleware AG, Poseidon, 2004. On-line at http://www.gentleware.com/

[Ghe91] Ghezzi C, Jazayeri, M., Mandrioli, D. (1991) Fundamentals of Software Engineering, Prentice Hall.

[GHJV95] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995) Design Pattens. Elements of Reusable Object-Oriented Software, Addison-Wesley.

[GHK99] Gil, J., Howse, J, and Kent, S. (1999) Formalising Spider Diagrams, In *Proccedings .of IEEE Symposium on Visual Languages (VL99)*, IEEE Press, pp. 130-137.

[GHK01] Gil J, Howse J, Kent S. (2001) Towards a formalization of constraint diagrams. In *Proceedings of EEE Symposia on Human-Centric Computing (HCC'01)*, Stresa, Italy, IEEE Computer Society Press, pp. 72-79.

[Gins75] Ginsburg, G. (1975) Algebraic and Automata-Theoretic Properties of Formal Languages. North-Holland. Amsterdam.

[GLRSW02] Gerber, A., Lawley, M., Raymond, K., Steel, J., and Wood, A. (2002). Transformation: The Missing Link of MDA, in A. Corradini, H. Ehring, H. J.

Kreowsky, G. Rozenberg (Eds): In *Proceedings. of Graph Transformation: First International Conference (ICGT 2002)*

[GMT] Generative Model Transformer Project Home Page. On-line at http://www.eclipse.org/gmt/

[Gra02] Grand, M. (2002) Java Enterprise design patterns, Wiley&Sons.

[Gra88] Gray R. W. (1988) γ-GLA – a generator for lexical analyzers that programmars can use. In *Proceedings of USENIX Conference*. USENIX Association, Berkley, CA, pp. 147-160.

[Gra90] Grady, R. B., (1990) Work-Product Analysis: The Philosopher's Stone of Software, IEEE.

[Hal77] Halstead, M. (1977) Elements of Software Science, Elsevier, Amsterdam.

[Hei81] Heilbrunner, S. (1981) A parsing automata approach to LR theory. *Theoretical Computer Science* 15, pp. 117-157.

[HS96] Henderson-Sellers, B. (1996) Object-Oriented Metrics: Measures of Complexity, Prentice-Hall.

[HU69] Hopcroft, E. and Ullman, J.D. (1969) Formal Languages and Their Relation to Automata. Addison-Wesley.

[HU79] Hopcroft, E. and Ullman, J.D. (1979) Introduction to Automata Theory, Languages, and Computations. Addison-Wesley.

[Iro61] Irons, E. T. (1961) A syntax directed compiler for ALGOL 60. *CACM* 4, 51-55.

[ISO96] ISO/IEC (1996) Information Technology – Software Product Evaluation; Part 1: Overview. ISO/IEC DIS 14598-1. (International Organization for Standardization and the International Elcctrotechnical Commission).

[Java] Java standard http://www.sun.com

[JB81] Janssen, T. M. V. and van Emde Boas, P. (1981) Some observations on compositional semantics. Report 81-11. University of Amsterdam.

[Jon75] Johnson, S. C. (1975). Yacc: yet another compiler compiler. Tech. Rep. CSTR-32, AT&T Bell Laboratories, Murray Hill, NJ.

[KLW95] Kifer M., Lausen G., and Wu J.. Logical Foundations of Object-Oriented and Frame-Based Languages. Journal of the ACM, 42(4):741 843, July 1995.

[Kle56] Kleene, S.C. (1956) Representation of events by nerve nets. In *Automata Studies*, ed. C.E. Shannon and J McCarthy, Princeton University Press, Princeton, pp. 3-42.

[KMF] Kent Modeling Framework project. http://www.cs.kent.ac.uk/projects/kmf.

[Knu65] Knuth, D. E. (1965) On the translation of languages from left to right. *Information and Control 8*, pp. 607-639.

[Knu67] Knuth, D. E. (1967) The Art of Programming, Vol. I: Fundamental Algorithms. Addison Wesley.

[Knu68] Knuth, D. E. (1968) Semantics of context-free languages Math. Syst. Theory 2, 127-145. Correcttion: Math. Syst. Theory 5, pp. 95-96.

[Les75] Lesk, M. E. (1975). Lex-a lexical analyzer generator. Tech. *Rep. Computing Science Technical Report 39*, Bell laboratories, Murray Hill, NJ.

[LH93] Li, W. and Henry, S. (1993) Object-oriented metrics that predict maintainability. Journal of Systems and Software, 23, pp. 111-122.

[LK94] Lorentz, M. and Kidd, J. (1994) Object-Oriented Software Metrics, Prentice-Hall.

[LS68] Lewis, P. M. and Stearns, R. E. (1968). Syntax-driven translation, *JACM* 15, pp. 464-488.

[McC76] McCabe, T. (1976) A complexity measure. IEEE Transactions Software Engineering, 2, pp. 308-320.

[MDA] MDA. Model Driven Architecture Specification. OMG document omg/03-06-01, available at. http://www.omg.org/mda.

[MID] Middlegen Project Home Page. On-line at http://sourceforge.net/projects/middlegen

[MODF] ModFact Project Home Page. On-line at http://modelware.inria.fr

[MOF] OMG, MOF Meta Object Facility Specification, OMG Document formal/2002-04-03, available at http://www.omg.org/mof

[MOFS] MOFScript Project Home Page. On-line at http://www.modelbased.net/mofscript/

[MTF] Model Transformation Framework Project Home Page. On-line at http://www.alphaworks.ibm.com/tech/mtf

[MTL] MTL Engine.Project Home Page. On-line at http://modelware.inria.fr/

[Myh57] Myhill, J. (1957) *Finite automata and the representation of events*. WADD TR-57-624, Wright Patterson AFB, Ohio, pp. 112-137.

[Ner58] Nerode, A. (1958) Linear automaton transformations. *In Proceedings of the American Mathematical Society*, 9, pp. 541-544.

[NS57] Newell, A. and Shaw, J. C. Programming the logic theory machine. In *Proceedings of the 1957 Western Joint Computer Conference*, pp. 230-240, Institute of Radio Engineers, New-York.

[OAW] Open Architecture Ware Project Home Page. On-line at http://www.openarchitectureware.org/

[OCL] OMG, OCL Object Constraint Language Specification Revised Submission, Version 1.6, January 6, 2003, OMG document ad/2003-01-07.

[OCL2P] Open source project: Object Constraint Language for Kent Modeling Framework and Eclipse Framework. http://www.cs.kent.ac.uk/projects/kmf.

[OMDX] OpenMDX Project Home Page. On-line at http://www.openmdx.org/

[OMG] Object Management Group. http://www.omg.org.

[Pat04a] Patrascoiu, O. (2004) YATL:Yet Another Transformation Language. In *Proc. of* First European Workshop MDA-IA, University of Twente, the Nederlands.

[Pat04b] Patrascoiu, O. (2004) YATL:Yet Another Transformation Language. Reference Manual. Version 1.0. Technical Report 2-04, University of Kent, UK.

[Pat04c] Patrascoiu, O. (2004) Model transformations in YATL. Studies and Experiments. Technical Report 3-04, University of Kent, UK.

[Pat04d] Patrascoiu, O. (2004) Mapping EDOC to Web Services using YATL. In *Procedings of 8th IEE International Enterprise Distributed Object Computing Conference, EDOC 2004*.

[Pat02a] Patrascoiu, O. (2002) A quality model for Java programs maintenance. In *Else Software Journal*, University of Craiova.

[Pat02b] Patrascoiu, O. (2002) Software systems quality. In *Else Software Journal*, University of Craiova.

[Pax95] Paxson, V. (1995) Flex-Fast lexical analyzer generator. Lawrence Berkley Laboratory, Berckeley, CA, http://www.icir.org/vern/

[PM91] Patrascoiu, O. and Marian, Gh. (1991). TDPG: A Parser Generator for Top-down Parsing Grammars. In *Proceedings of The International Conference on Applied and Theoretical Electrotechnics*, Craiova.

[PM94] Patrascoiu, O. and Marian, Gh. (1994). Translation Scheme for Regular Expression. In *Proceedings of the National Symposium on System Theory*, Craiova.

[PR04] Patrascoiu, O. and Rodgers, P. (2004). Embedding OCL expressions in YATL. *In Proc. of "OCL and Model Driven Engineering" workshop, UML 2004.*

[QVT02] OG, QVT Query/Views/Transformations RFP, OMG Document ad/02-04-10, revised on April 24, 202. http://www.omg.org/cgi-bin/doc?ad/2002-4-10

[QVTD] OMG, MOF Query/Views/Transformation, Initial submission, DSTC and IBM.

[QVTF] OMG, MOF Query/Views/Transformation, Initial submission, Alcatel, SoftTeam, Thales, TNI-Valiosys.

[QVTP] OMG, MOF Query/Views/Transformation, Initial submission, QVT Partners.

[RAT] Rational Software Corporation, Rational Rose, 2004. Online at http://www.rational.com

[RG00] Richters, M. and Gogolla, M. (2000) Validating UML models and OCL constraints. In *Proceeding of The Third International Conference on the Unified Modeling*

*Language* (UML'2000), LNCS. Springer.

[RJB99] Rumbaugh, J., Jacobson, I., and Booch, G.. (1999). The Unified Modeling Language – Reference Manual. Addison-Wesley.

[RS59] Rabin, M.O. and D. Scott. (1959) Finite Automata and their decision problem. In *IBM Journal of Research and Development* 3, pp. 114-125.

[RWD] Reasoning with Diagrams http://www.cs.kent.ac.uk/projects/rwd

[Sal69] Salomaa, A. (1969) Theory of Automata. International Series of Monographs in Pure and Applied Mathematics, Pergamon Press.

[Sal73] Salomaa, A. (1973) Formal Languages. Academic Press, Revised edition in the series "Computer Science Classics", Academic Press.

[SB60] Samelson, K. and Bauer, F. L. (1960). Sequential formula translation. *Communications of the ACM*, 3(20), pp. 76-83.

[SJF96] Schmidt, D. C., Johnson, R. E., and Fayad, M. (1996) Software patterns. In *CACM*, 39(10).

[SOAP] W3C, Simple Object Access Protocol http://www.w3.org/TR/soap

[Som92] Sommerville I. (1992) Software engineering, Addison-Wesley.

[SVB03] Sturm, T., von Voss, J., and Boger, M. (2003) Generating Code from UML with Velocity Templates, In *Proceedings of The Fifth International Conference on the Unified Modeling Language (UML'2002)*, Dresden, Germany.

[TC02] Tang, M.-H., and Chen, M.-H. (2002) Measuring OO Design Metrics from UML. In *Proceedings of the Fifth International Conference <<UML>> 2002 – The Unified Modeling Language. Model Engineering, Concepts, and Tools*, pp. 368-382.

[TKC99] Tang, M.-H., Kao, M.-H., and Chen, M.-H. (1999) An empirical study on object oriented metrics. In *Proceedings of the Sixth International Software Metrics Symposium*, pp. 242-249.

[TOG] TogetherSoft http://www.togetherSoft.com

[Tur36] Turing, A., *On Computable Numbers, With an Application to the Entschedungsproblem.* In, Proceedings of the London Mathematical Society, Series 2, Volume 42, 1936; reprinted in M. David (ed.), *The Undecidable*, Hewlett, NY: Raven Press, 1965; online: http://www.abelard.org/turpap2/tp2-ie.asp.

[UDDI] Universal Description, Discovery, and Integration http://uddi.org/specification.html

[UML] OMG, Unified Modeling Language Specification, Version 1.5, 2003, OMG Document formal/2003-03-01, available at. http://www.omg.org/uml.

[UMT] UML Model Transformation Tool Project Home Page. On-line at http://umt-qvt.sourceforge.net/

[UNI] Unicode standard. http://www.unicode.org

[WG84] Waite, W. M. and Goos, G. (1984) *Compiler construction*. Springer Verlag.

[WIK] Wikipedia The Free Encyclopedia http://www.wikipedia.org

[WH98] Wilkie F. G. and Hylands B. (1998) Measuring Complexity in C++ Application Software, Software Practice and Experience, 28(5), pp 513-546.

[WK99] Warmer, J. and Kleppe, A. (1999). *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley.

[WSDL] W3C, Web Service Description Language http://www.w3.org/TR/wsdl

[XDOC] XDoclet Project Home Page. On-line at http://xdoclet.sourceforge.net/xdoclet/index.html

[XMI] OMG, MOF Meta Object Facility Specification OMG Document 2003-05-02, available at http://www.omg.org/uml

[XML] W3C, Extensible Markup Language http://www.w3.org/TR/2004/REC-xml-20040204/

[XMLS] XML Schema http://www.w3.org/XML/Schema