Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

DIPLOMARBEIT

MDA Support for Constraint Checking Framework in EJB

Ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Diplom-Ingenieurs unter der Leitung von

Univ.-Prof. Dr. Schahram Dustdar
und
Dr. Karl M. Göschka,

Dr. Lorenz Froihofer
als verantwortlich mitwirkende Assistenten am

Institutsnummer: E184-1
Institut für Informationssysteme
Arbeitsbereich für Verteilte Systeme

eingereicht an der Technischen Universität Wien
Fakultät für Elektrotechnik und Informationstechnik

von

Xuejun Ji, B.Sc. Bakk.phil.
Matr.Nr.: 0225254
Universumstrasse 26/9, 1200 Wien

Wien, im Januar 2011 _____

Abstract

In today's systems, the code for constraint checking is often tangled with other code (e.g. the code for the business logic). However, the approach with explicit constraint classes is more flexible. Metadata can be used to manage and to customize these constraint classes on applications with different purposes. In this diploma thesis, a code generator is investigated and implemented that provides MDA (Model Driven Architecture) support for a software development toolkit building upon EJB (Enterprise Java Bean) and explicit integrity constraints. This code generator supports the trade-off between availability and consistency as investigated by the European research project DeDiSys. Consequently, the tool produced within this diploma thesis supports metadata specification for constraints along with the ability to generate EJB source code, explicit constraint checking classes and metadata deployment information.

The evaluation results illustrate that the generated code of the new build code generator is helpful, but many enhancements such as performance improvement of the generated code are necessary to use the code generator in a production system without worrying. The generated constraints during the evaluation phase were ready-to-use without any changes, particularly the constraints with simple condition statements. However, when condition statements used loops or complex data types, the generated code was too slow with the increasing number of added objects. The evaluation shows that manual optimization was necessary to increase the efficiency. Another possibility is using other transformation languages such as SQL that is supported by the implemented OCL-to-stored-routines transformer. The reached effect was that the evaluation duration increased linearly with the number of added objects or could be limited to a maximum for each constraint validation.

Kurzfassung

In den heutigen Systemen ist der Quellcode für Bedingungsüberprüfung (Constraint) oft mit anderen Quelltexten vermischt (z.B. mit dem Code der Geschäftslogik). Der Ansatz mit expliziten Constraint-Klassen ist hingegen flexibler. Metadata werden eingesetzt um diese Constraint-Klassen zu managen und für Applikationen unterschiedlicher Zwecken anzupassen. Im Rahmen dieser Diplomarbeit wird ein Code-Generator erforscht und implementiert, um MDA-Unterstützung (Model Driven Architecture) für ein Software-Toolkit, das auf EJB (Enterprise Java Bean) und explizite Integritätsbedingungen basiert, anzubieten. Dieser Code-Generator unterstützt den Trade-off-Ansatz zwischen Verfügbarkeit und Konsistenz, nach dem im Rahmen des europäischen Forschungsprojekts DeDiSys geforscht wird. Infolgedessen unterstützt das Tool, dass während dieser Diplomarbeit entwickelt wurde, Metadata-Spezifikation für Constraints mit der Möglichkeit EJB-Quelltexte, explizite Constraint-Klassen und Metadata-Konfigurations-Informationen zu erzeugen.

Die Evaluierungsergebnissen zeigen, dass das generierte Code vom neuen implementierten Codegenerator hilfreich ist, aber weitere Erweiterungen, wie zum Beispiel Performanceverbesserung für das generierte Code, nötig sind um ein Codegenerator in ein Produktionssystem unbesorgt einsetzen zu können. Die generierte Constraints während der Evaluierungsphase waren direkt anwendbar ohne jegliche Änderungen, insbesondere jene Constraints mit einfachen Bedingungsanweisungen. Jedoch, wenn in Bedingungsanweisungen Schleifen oder komplexe Datentypen verwendet wurden, war der generierte Code zu langsam bei einer immer weiter steigenden Menge von eingefügten Objekten. Die Evaluierung zeigt, dass eine manuelle Optimierung nötig war um die Effizienz zu erhöhen. Eine weitere Möglichkeit ist andere Transformationssprachen zu verwenden wie zum Beispiel SQL, das vom implementierten „OCL-to-stored routines“-Transformer unterstützt wird. Der erreichte Effekt war, dass die Evaluierungsdauer linear mit der Anzahl der eingefügten Objekte stieg oder auf ein Maximum für jede einzelne Constraint-Validation begrenzt werden konnte.

Acknowledgements

First of all, I give special thanks to Dr. Lorenz Froihofer for his excellent guidance of my diploma thesis. His knowledge is impressive and without his valuable inputs this thesis would be end in nirvana. He displayed patience and understanding at every stage that was an endless help and always a motivation to work harder and greater.

Furthermore, special thanks go to Dr. Karl Michael Göschka who gave me the possibility to be a part of this interesting research project. His inimitable style of teaching will always stay in my memory.

Thanks go to my colleagues Dipl.-Ing. Dominik Ertl for the cooperation and the many discussions. His feedback was always worthwhile. Moreover, I wanted to thank Dipl.-Ing. Hubert Kraut and Ving Long Hua for the review and the feedback.

Finally, I give also thanks to my girlfriend, my whole family and friends. During the time I am working on the thesis, there were many ups and downs in my life. It was their helps that gave me a soft landing instead of a hard one. Nothing is more important for me to see them, especially my girlfriend, in good health.

Abbreviations and Acronyms

API	Application Programming Interface
AST	Abstract Syntax Tree
BMP	Bean managed persistence
BSD	Berkeley Software Distribution
CASE	Computer-Aided Software Engineering
CMP	Container managed persistence
CST	Concrete Syntax Tree
CWM	Common Warehouse Metamodel
DDL	Data Definition Language
DeDiSys	Dependable Distributed Systems
EAR	Enterprise Application Archive
EJB	Enterprise Java Beans
GIF	Graphics Interchange Format
GUI	Graphical User Interface
EJB	Enterprise JavaBeans
IBM	International Business Machines Corporation
IDL	Interface Definition Language
ISU	Iowa State University
JAR	Java Archive
Jass	Java with assertions
JDT	Java Development Tools
J2EE	Java 2 Platform Enterprise Edition
J2SE	Java 2 Platform Standard Edition
JSF	JavaServer Faces
MDA	Model Driven Architecture
MDR	Metadata Repository
MOF	Meta Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
PIM	Platform Independent Model
PNG	Portable Network Graphics
PSM	Platform Specific Model
SQL	Structured Query Language
UI	User Interface
UML	Unified Modeling Language
XLST	Extensible Stylesheet Language Transformations
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Contents

1 Introduction 1
1.1 Dependable Distributed Systems (DeDiSys) 1
1.2 Background and Motivation 2
1.3 Problem definition and Goal 3
1.4 Structure of this thesis 5
2 Mainly Technological Basis 6
2.1 DeDiSys framework 6
2.1.1 Consistency types 7
2.1.2 System states 7
2.1.3 Constraint Consistency Manager (CCMgr) 8
2.1.4 Constraints classification 9
2.1.5 Metadata 10
2.2 Enterprise JavaBeans 11
2.2.1 Session beans 13
2.2.2 Entity beans 14
2.3 Unified Modeling Language 14
2.4 Object Constraint Language 16
2.5 Dresden OCL Toolkit 17
2.5.1 Structure of the Dresden OCL Toolkit 17
2.5.2 OCL2 Parser 18
2.6 Model Driven Architecture (MDA) 20
2.6.1 Improvements through MDA 21
2.6.2 Basis technologies 22
2.7 AndroMDA 22
2.7.1 Development cycle 23
2.7.2 CASE tools 24
2.8 Related work 24
3 Realization 27
3.1 MDA approach 28
3.2 Realization overview 28
3.3 Solution of the code generation process 29
3.4 Modelling 31
3.5 Metadata integration in CASE tool 32
3.6 EJB generation 33
3.7 Textual constraint generation 34
3.7.1 Approach for Poseidon 35
3.7.2 Approach for ArgoUML 36
3.8 Constraint generation 36
3.8.1 Structure of an explicit constraint class 37
3.8.2 Integration of the Dresden OCL Parser 39
3.8.3 Code generation architecture 40
3.8.4 Java code generator 42
3.9 Generation of the metadata configuration document 45
3.10 Process integration 46
3.11 Optimization of the integrity constraint code generator 48
3.11.1 Structure of the advanced constraint code generator 49

3.11.2 Transformation of OCL to SQL 49
3.11.3 Transformation of OCL to store routines 50
3.11.4 Combined code generator 52
3.12 Related technologies 53
4 Evaluation and Results 56
4.1 Test environment 56
4.1.1 System and Evaluation environment 56
4.1.2 Test-Application 58
4.2 Test scenarios 60
4.2.1 Test scenario: No constraints 62
4.2.2 Test scenario: FlightNotEmpty 63
4.2.3 Test scenario: AllInstances 64
4.2.4 Test scenario: Select 66
4.2.5 Test scenario: Iterate 67
4.2.6 Scalability testing 68
4.2.7 Interpretation 71
4.2.8 Potential enhancements 71
4.3 Examples of real-life applications 72
5 Lessons Learned 74
6 Summary & Outlook 78
6.1 Summary 78
6.2 Future prospects 79
6.3 Conclusion 80
References 81
Appendix 84
A1. Link collection 84
A2. Description: constraint deployment descriptor 85
A3. Configuration document: andromda.xml 87
A4. XLST transformation: xmi2ocl.xsl 88
A5. XLST transformation: xmi2xmi.xsl 89

List of Figures

Figure 1: Trade-off between availability and consistency 2
Figure 2: *FlightNotEmpty*: constraint and metadata 2
Figure 3: Problem overview 4
Figure 4: DeDiSys system layer [Froi05] 6
Figure 5: Consistency types [Froi05] 7
Figure 6: System states [FOG07] 8
Figure 7: Static and dynamic constraints [Froi05] 9
Figure 8: Classification of constraints 9
Figure 9: Context object of a constraint 10
Figure 10: Enterprise JavaBeans 12
Figure 11: Stub and skeleton [RSB05] 13
Figure 12: UML diagrams 15
Figure 13: Example - UML class diagram 15
Figure 14: Structure of the Dresden OCL Toolkit [DOCL08] 19
Figure 15: OCL2 Parser [Kone05b] 20
Figure 16: MDA interoperability using bridges [KWB03] 21
Figure 17: Problem and goal description 29
Figure 18: MDA processing steps 30
Figure 19: *FlightTestSelect*: constraint and metadata 32
Figure 20: Class diagram: textual constraint generation 36
Figure 21: Class diagram: model transformation 40
Figure 22: Class diagram: code generator 41
Figure 23: Class diagram of *ConfigurationFileGenerator* 45
Figure 24: Flowchart of the combined code generator 52
Figure 25: Test UI of the flight booking application 58
Figure 26: Class diagram of the test application 59
Figure 27: Test case: "*no constraints*" 62
Figure 28: Test case: *FlightNotEmpty* 64
Figure 29: Test case: *AllInstances* 65
Figure 30: Test case: *Select* 67
Figure 31: Test case: *Iterate* 68
Figure 32: Scalability of constraint validation for the *AllInstances*-constraint 69
Figure 33: Scalability of constraint validation for the *Select*-constraint 69
Figure 34: Scalability of constraint validation for the *Iterate*-constraint 70
Figure 35: Scalability testing – *Iterate*-constraint (stored routines) 70

List of Listings

Listing 1: Constraint definition: *DemoConstraint* 10
Listing 2: *ccDefinition.xml:* DemoConstraint metadata 11
Listing 3: Example for an OCL expression 16
Listing 4: OCL syntax for invariant constraints 17
Listing 5: OCL syntax for pre- and postconditions 17
Listing 6: Apache Ant target for AndroMDA 34
Listing 7: Apache Ant target for textual constraint generation 35
Listing 8: Wrapper-based validation 38
Listing 9: Explicit constraint class 39
Listing 10: Constraint definition: *FlightTestSelect* 43
Listing 11: *FlightTestSelect*: handwritten version 43
Listing 12: *FlightTestSelect*: generated version 44
Listing 13: *ccDefinition.xml*: *FlightTestSelect* metadata 47
Listing 14: Constraint checking code of the SQL generator 50
Listing 15: Constraint definition: *DemoStoredFunction* 51
Listing 16: Stored function code of *DemoStoredFunction* 51
Listing 17: Java code of *DemoStoredFunction* 52
Listing 18: Constraint definition: *FlightNotEmpty* 63
Listing 19: Constraint definition : *AllInstances* 64
Listing 20: Normalized constraint definition: *AllInstances* 64
Listing 21: Constraint definition: *Select* 66
Listing 22: Normalized constraint definition: *Select* 66
Listing 23: Constraint definition: *Iterate* 67
Listing 24: AndroMDA configuration document 87
Listing 25: Transformation document: *xmi2ocl.xsl* 88
Listing 26: Transformation document: *xmi2xmi.xsl* 89

List of Tables

Table 1: AndroMDA configuration file 34
Table 2: Default values for *ccDefinitions.xml* 45
Table 3: Example for the AndroMDA translation library [AMDA08] 54
Table 4: Main facts of the used test computer 56
Table 5: Test result for "*no constraints*" 62
Table 6: Test result for *FlightNotEmpty* 63
Table 7: Test result for *Iterate* 68
Table 8: Constraint DTD main elements 86
Table 9: Constraint tag structure 86

1 Introduction

Distributed system becomes more interesting the more complex a software system is with respect to maintainability. Because the systems depend on each other, the data consistency plays a very important role. It is a big challenge to fulfil the condition that each system each time should have the same data. This thesis addresses this problem by providing tools to support a developer during the development process.

The Dependable Distributed Systems project (DeDiSys) – a research project supported by the European Union – is a project that deals with issues when a network failure happens. It follows the approach to allow the temporarily breaking of constraints while systems are not fully connected to each other but are able to serve request. During the DeDiSys project, a constraint checking framework has been implemented. In the sense of separation of concerns, the integrity constraints and constraint checking are treated explicitly within a system.

However, the development of such software applications is very complex and the factor time-to-market is critical. For this reason, the use of code generators will be more important in the future. A code generator helps to speed up the development process and allows people taking an active part in a project who are not experts in programming.

Currently, powerful tools with code generating engine are rare, particularly with regard to the support of explicit integrity constraints.

In this thesis, a code generator for explicit constraints in the context of the constraint checking framework of the DeDiSys project is realised. This approach uses metadata to specify further properties for constraints.

1.1 Dependable Distributed Systems (DeDiSys)

The DeDiSys project was a European research project to investigate the trade-off between availability and consistency in dependable distributed systems as shown in Figure 1.

The project consists of implementations in three different programming languages: EJB (Java), .NET (C#) and Corba (Java). This diploma thesis uses the EJB packet to prove its feasibility. Different applications are implemented by the DeDiSys project and one of them, the flight booking application, is used as the test application of this work.

The mission of DeDiSys is to research the feasibility of software systems for safety-critical applications, in which the availability of the system is more important for dependability than strict data integrity. Availability should be enhanced by disregarding data integrity temporarily when necessary. For example, this situation would happen when there is a link failure between the connected systems. Potential inconsistencies will be accepted on replicated copies when constraints are validated. A following reconciliation phase will resolve these inconsistencies [FOG06].

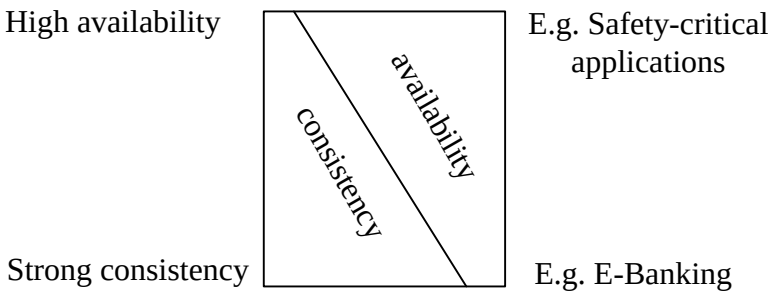

Figure 1: Trade-off between availability and consistency[1]

1.2 Background and Motivation

Under the pressure of economical aspects, the developer has less and less time for developing due to the ever-shortened innovation cycles. For this reason, code generators play an important role to increase productivity and their role could increase in the future. The developer should spend more time in design and less for implementation. This part could be performed by a code generator. The advantage is that the code of a generator would use predetermined algorithms and another developer could continue to work on the same implementation more easily if he/she knows the used algorithms of the code generator. However, a program that is written by a programmer is characterised by the style of this person and it would take lots of time to understand it by other people if that is possible at all. Of course, a code generator could also generate hardly understandable code. For this reason, it is necessary to provide a good documentation of the implemented code generator and the generated code should have been tested for readability if possible.

The uniformity of code will take an important role in the future. The cost for development increased in the past years continuously and the complexity of large projects increased too. The operation of external tools such as code generators will be an essential factor to decide if the software would be successful in the market. Simplifying the way of the development and reducing the cost will determine the future path of an innovated software package.

Constraints and constraints metadata are the main focus of this work. A constraint can be defined as a restriction of possibilities, and it provides an excellent mechanism to ensure data consistency. In this diploma thesis, constraints are specified in OCL. Every constraint has a wide range of properties that are specified as metadata such as the type or the priority of a constraint. An example is shown in Figure 2. The OCL expression of this constraint on the left side in Figure 2 means that the attribute "*flightnumber*" cannot be a null object.

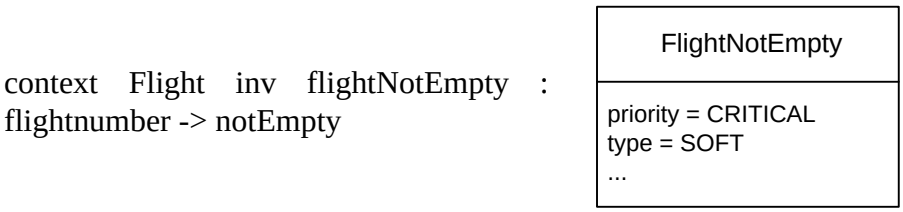

Figure 2: *FlightNotEmpty*: constraint and metadata

[1] Dependable Distribute Systems (DeDiSys): http://www.dedisys.org/ (14/06/2010)

Constraints vary greatly in applications. Besides the restrictions of a programming language, a developer has to take care that no mistakes exist in the code such as a wrong constraint validation statement. Error detection costs lots of effort and time. The risk of an error configuration increases with regard to constraints metadata. The code between constraints and constraints metadata has to match to each other. It is very tedious to write the code statements all manually such as the corresponding class with all its methods and parameters of a constraint. The susceptibility to errors increases with the number of used metadata. The effort increases disproportionate with the number of constraints and constraints metadata if everything has to be error free.

A code generator could not guarantee that always the best constraints code is generated, particularly with regard to the performance. However, it provides error free code that is evaluated several times. The developer could spend more time in writing of other parts such as the code for the business logic. Besides the constraints, their metadata could be also specified in a model so that the both parts are matched to each other. While the constraints code often requires an optimization manually, the generated metadata code is ready to use without any changes. Error free code and an easy way for verification will take an important role in software development. Mistakes could not be eliminated, but the probability of making mistakes could be reduced. Of course, it is also possible to make mistakes on specifying constraints directly in a model, but by providing clearly represented structure through using of GUIs together with a constraint language such as OCL it is possible to verify and if necessary to modify the specified constraints and metadata in an easy way. The generated constraints and constraints metadata use a structure of uniform code that can be followed if the way of the generation process is known.

Therefore, the code generator could support a developer during the development process. It could provide a basic structure with ready-to-run constraint classes. As mentioned above, the generated constraints may be not well optimized in performance, but they are already tested and declared as error free. The effort for changing the generated constraints is much smaller than to write this all manually. A developer has only to consider the query part for constraint validation. The metadata that are required by the approach of this diploma thesis are tedious to write, but with a code generator the process is simplified several times. It provides ready-to-use metadata so that this way is error proof. The effort for writing constraints metadata could be also reduced to a minimum. Careless mistakes could be avoided through the code generator. The structure of such metadata configuration document would be transparent for the developer. Consequently, a constraint code generator provides an instrument that supports a developer in development phase to reduce the effort for coding and to achieve error safe code for constraints and constraints metadata.

1.3 Problem definition and Goal

For this thesis, the feasibility of MDA support through a code generator is investigated. The code generator is based on explicit integrity constraint classes that are configured through metadata. Nowadays, there exist code generators that can generate code for different programming languages. The code for constraint checking is mostly mixed together with other code (e.g. the code for the business logic). As mentioned above, the new build generator of this work supports the approach of the constraint checking framework that is implemented by the DeDiSys project.

Figure 3 illustrates the problem this diploma thesis is confronted to solve. There exist modelling tools. Modelling tools can be used to model UML classes and to specify constraints in OCL. Moreover, the DeDiSys framework has integrated a constraint checking framework that uses explicit constraint classes for constraint validation. This constraint framework uses metadata to adapt constraints. Metadata contain a set of properties for each specified constraint.

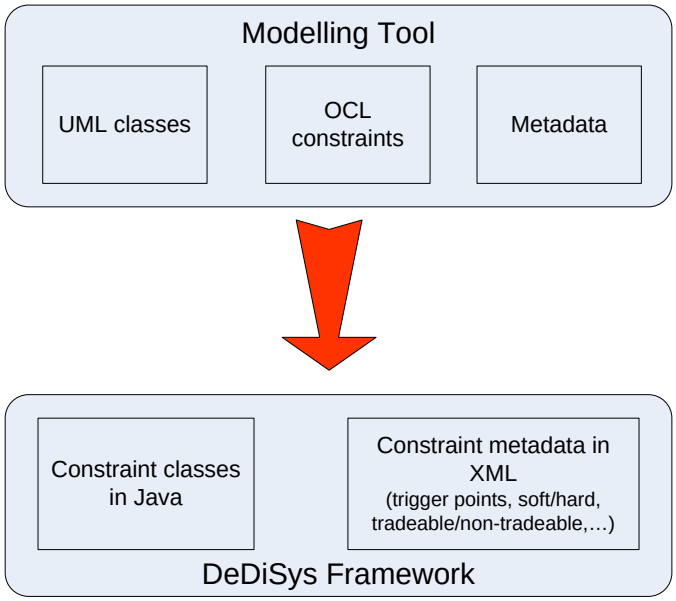

Figure 3: Problem overview

Consequently, the goal of this work is to bridge the gap between modelling tool and the DeDiSys Framework as illustrated in Figure 3. The solution contains a code generator that provides MDA support building upon EJB and explicit constraint classes including metadata specification. This new code generator is able to generate Java code from a UML model. The constraints are defined in OCL and are used together with UML in one model. Some functions are already supported by other tools and do not have to be reinvented in this diploma thesis. The reused tools are the Dresden OCL Toolkit and AndroMDA. The used functions are adapted with other functionalities if necessary. The first task is to make these tools work together to have the basis functionality that is required by this thesis. Potential incompatibilities are identified and – if possible – solved. Different approaches and tools are used as well to find a more appreciated solution. There is one limitation, namely, to use open-source tools only or tools that have a community edition for free.

OCL itself does not support metadata, but metadata are required by the constraint checking framework. The used constraint classes for this framework are normal Java classes that do not contain specifications such as when it can be used or which type of constraint it is. OCL has three types of constraints (invariants, pre- and post-conditions). Therefore, the second task is the integration of metadata that contain the constraint properties that are required during run-time.

The final solution includes a code generator that is able to generate code with explicit constraint classes from a UML-OCL model with metadata included in one process.

1.4 Structure of this thesis

The structure of this diploma thesis is as follows:

Chapter 1 is the introductory section that gives background information about the research project DeDiSys in which context the thesis is written. A short description is also given about code generators, the problems and the goals of this diploma thesis.

Chapter 2 gives an overview about the mainly technical baseline this thesis is based on. Important facts that would be needed for the following sections are presented, and too much details will be avoided.

Chapter 3 presents the solution for the code generator. It includes the implementation of different tools with a UML tool at the beginning to the integration of the code generator in the constraint checking framework of the DeDiSys project. The co-working of different independent applications is demonstrated.

Chapter 4 contains the evaluation section. The used test environment and the evaluation tests are illustrated. The test description and its results are discussed in detail.

Chapter 5 describes the development process of this diploma thesis. Problems, advantages and goals in context with the design and implementation phase are illustrated.

Chapter 6 concludes this diploma thesis. Additionally, it provides an outlook to the future and recommended future works are presented.

In the appendix, one can find explanation and examples for different kinds of configuration documents.

2 Mainly Technological Basis

This section gives a short introduction about technologies that are used for the implementation of this diploma thesis. Readers who are already familiar with them may jump directly to the next chapter.

This chapter provides information about the DeDiSys framework, EJB, UML including OCL, Dresden OCL Toolkit, MDA and AndroMDA. As this thesis is part of the European research project DeDiSys, it is annotated that this thesis has the focus on EJB, one of the three base technologies of DeDiSys.

2.1 DeDiSys framework

The DeDiSys project is a research project supported by the European Community. Its focus is the investigation of the trade-off between consistency and availability and how this relation is configurable to application-specific requirements.

The intention of DeDiSys is not a totally new middleware, but to extend existing middleware (e.g. EJB, .Net or CORBA) with special functionality to reach the target. As shown in Figure 4, the DeDiSys framework is a part of the middleware and supports the application at the top level in trading consistency for availability. It is located between the application and the middleware layer. The middleware is responsible for group communication, group membership, naming, transaction and persistence with databases. At the bottom is the operating system that includes the networking support.

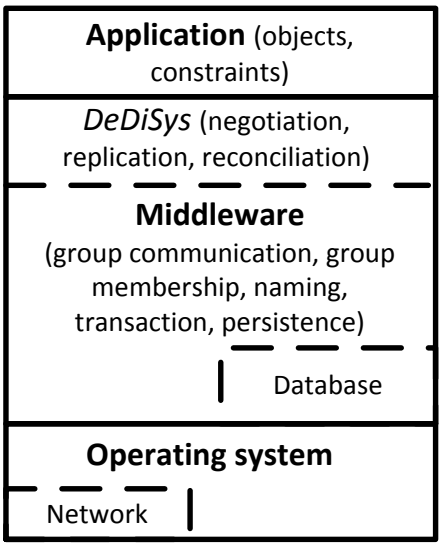

Figure 4: DeDiSys system layer [Froi05]

As mentioned, this diploma thesis is part of DeDiSys and covers the implementation of a code generator for explicit integrity constraints including metadata specification. The code generator that has to be implemented during this thesis must support the approach of the constraint checking framework that is implemented by DeDiSys. Consequently, the structure of the generated constraints from the new build code generator is predefined by this constraint checking framework.

2.1.1 Consistency types

The DeDiSys project regards three different consistency types [FOG07]:

- ***Integrity constraint consistency***
  Correctness of data is ensured by a number of data integrity constraints. If all defined constraints are checked successfully, then the system is fully constraint consistent.
  This thesis has the focus on integrity constraint consistency.

- ***Concurrency consistency (isolation)***
  Access to single data items is controlled by ordering of concurrent operations on objects to ensure data integrity. Concurrency control mechanism such as locking can be used.

- ***Replica consistency***
  It defines the consequence if different replicas of one entity have different states at the same time. The reasons may be a difference in the value of their attributes or of their relationships. A system can be declared fully replica-consistent, when all available replicas of each individual object have stored the same state in a primary-backup replication protocol. This criterion depends on the used replication protocol and is for example not valid for a quorum-based replication protocol.

The DeDiSys approach has the focus on constraint consistency and replica consistency. The concurrency consistency is not specifically addressed by the DeDiSys middleware add-on, but belongs to the sphere of a lower layer. The dependencies between these three different consistency types are illustrated in Figure 5.

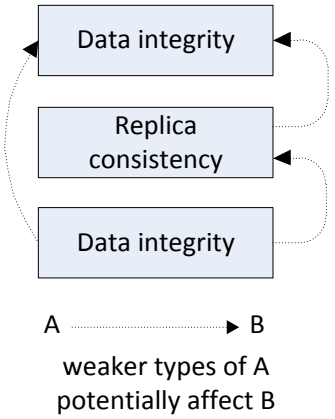

Figure 5: Consistency types [Froi05]

2.1.2 System states

The DeDiSys framework has three different system states. The three states are:

- ***Healthy***
  This is the intended working mode. A system in the “healthy” mode is in a fully consistent state. This is the preferred state of this thesis for validating integrity constraints in the evaluation section.

- ***Degraded***
  During system initialization, the system is in the “degraded” mode which it leaves after the connection between all defined nodes is established and which it re-enters in case of

detected node or link failures. During this state, inconsistencies are tolerated to a certain degree and controlled by integrity constraints.

- ***Reconciliation***
  After the failure has been resolved, the system starts the reconciliation phase to establish full consistency again. When both the replica consistency and the constraint consistency have been reached, the system changes to the "healthy" mode again.

The relationship between these three system states is illustrated in Figure 6:

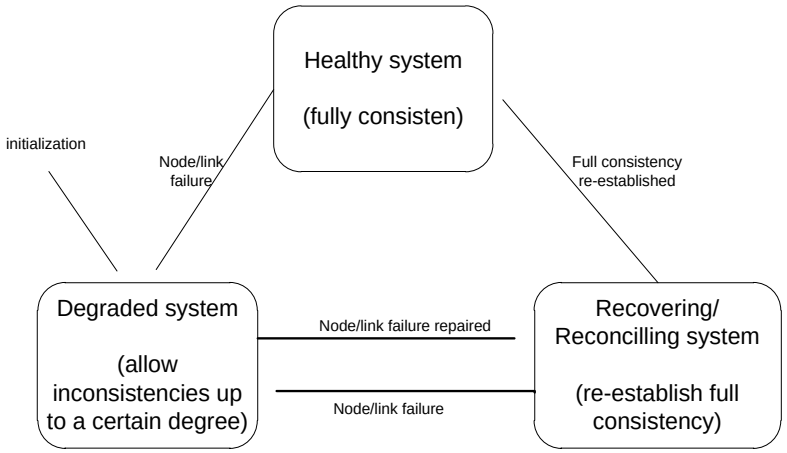

Figure 6: System states [FOG07]

2.1.3 Constraint Consistency Manager (CCMgr)

The use of constraint consistency management is a novel approach introduced by DeDiSys. Its role is monitoring the consistency of the defined constraints. It validates constraints, triggers the negotiation of consistency threats and supports the re-establishing of the constraints consistency after a node/link-failure through re-evaluating of constraints [OFG06].

The constraints that are generated by the code generator of this diploma thesis are validated by the Constraint Consistency Manager (CCMgr). The CCMgr uses a predefined structure for constraints. Because the code of all constraints uses the same structure, metadata are used to differentiate the constraints. Metadata together with explicit constraint classes allow a flexible use of constraint validation by CCMgr. The meaning of metadata is illustrated in Section 2.1.5

Constraints for the DeDiSys project can be differenced on different way, e.g. through the type or the priority of constraints. One way is the separation into static and dynamic constraints as shown in Figure 7. Static constraints constrain the state of an object. On the contrary, dynamic constraints are restricting the state changes of objects. There are also two types of dynamic constraints. Either, they specify the allowed difference between the state before the change and the state after the change (*state transition constraints*) or they consider a sequence of states so that require a history of the object states (*state sequence constraints*). Another way of differentiation is to declare intra-object or inter-object constraints. Intra-object constraints define constraints within a single object instance. In contrast to them, inter-object constraints define integrity constraints between different object instances [Froi05].

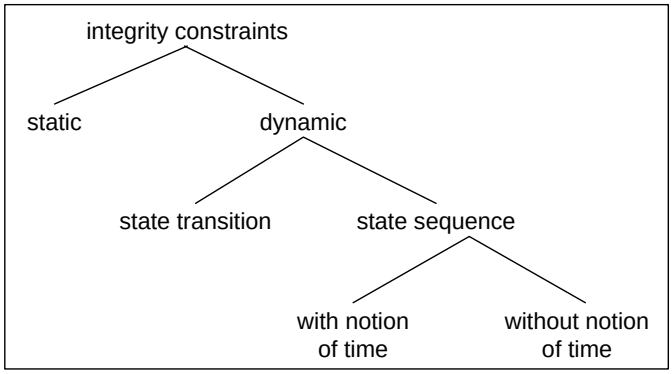

Figure 7: Static and dynamic constraints [Froi05]

2.1.4 Constraints classification

The constraint classifications within the DeDiSys project are described in Figure 8 [FOG07]. These constraint types that are identical to the OCL specification must also be supported by the new build code generator. Because OCL does not differentiate between hard and soft constraints for invariants, the distinction is implemented through metadata that are introduced by the constraint checking framework.

Constraints are classified in:

- ***Precondition***
  Preconditions have to be validated before a method validation.

- ***Postcondition***
  Postconditions are required to be satisfied after the call to a method returns.

- ***Invariant***
  Invariant constraints do not provide any timing conditions of when to validate the constraint. However, as soon as the context of an object that is constrained is changed, a new validation must be done. Two types of invariant constraints can be distinguished, *hard* and *soft* constraints. Hard constraints have to be checked as soon as the object is updated, and must be satisfied at any time. Soft constraints can be checked at the end of the transaction that causing the update.

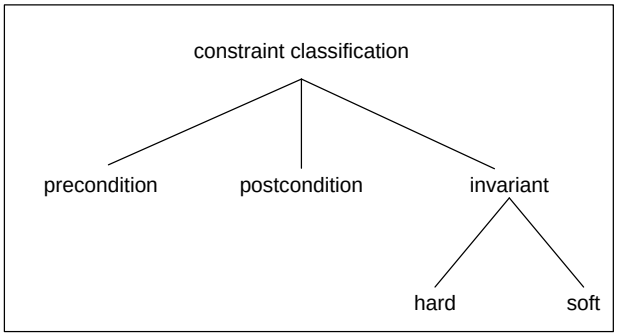

Figure 8: Classification of constraints

2.1.5 Metadata

All constraints that are used by the constraint checking framework of DeDiSys have the same structure independent of the type of constraints. Each constraint class contains only of a query statement for constraint validation, but has no properties such as at which time the constraint should be checked. To differentiate the constraints, metadata are introduced. Metadata contain several properties for each specified constraint. We distinguish between several kinds of metadata, like the type, the priority, the default reconciliation handler, affected methods or the context object.

In this diploma thesis, metadata from all specified constraints are stored to a configuration document named *ccDefinitions.xml* as required by the constraint checking framework. This document will be called by the DeDiSys middleware add-on during run-time for constraint validation. In other words, constraints validation works only if constraint classes and their metadata exist together at run-time.

Metadata such as the type or the priority of a constraint can be defined directly by the developer. However, there are some properties that depend on the context of constraints. It is important that these properties are set correctly. Otherwise, it could slow down the constraint validation process such as through unnecessary additional checks of the constraint. For example, the context object depends on the context of the corresponding constraint whether a context object is required. Figure 9 shows an example for a context object and an affected object.

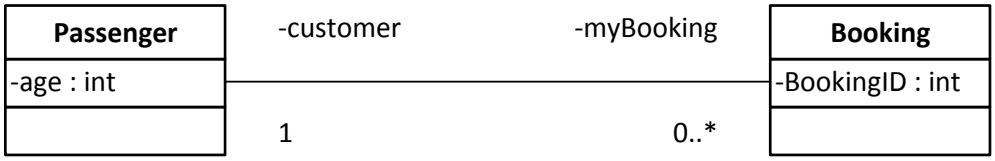

Figure 9: Context object of a constraint

For example, a person (→ passenger) who makes a booking must be older than 18 years and the OCL statement would be defined in the following way:

```
context Passenger inv demoConstraint:
myBooking->size() > 0 implies age>= 18
```

Listing 1: Constraint definition: *DemoConstraint*

The constraint in Listing 1 is an inter-object constraint because the references to *Booking* must be checked. *Passenger* is the context object. Because a reference to *Booking* is needed during constraint checking, *Booking* would be an affected object of that constraint. Therefore, for this example a context object is required for constraint validation.

Listing 2 shows the structure of the so-called configuration document *ccDefinitions.xml*. In this example, different metadata was set for the demonstration constraint *DemoConstraint*. The meanings of the metadata elements that are important for this diploma thesis are described in Appendix A2.

```
 <!DOCTYPE ccDefinitions SYSTEM "cc_def_1.0.dtd">
<ccDefinitions>
<persister class="org.dedisys.ccmgmt.persistence.CCMgrDefaultThreatPersister" />
<defaultconstraintreconciliationhandler class
        ="ejb.FlightbookingConstraintReconciliationHandler" />
<minSystemSatisfactionDegree value="POSSIBLY_SATISFIED" />
  <constraint name="DemoConstraint"
        type="SOFT"
        priority="RELAXABLE"
        negotiation="IMMEDIATE" contextObject="Y"
        minSatisfactionDegree="POSSIBLY_VIOLATED"
        latestAcceptedSatisfiedThreatRemovesIdenticalThreats="Y"
        intra-object="false">
    <class>Constraints.DemoConstraint </class>
    <context-class>ejb.PassengersBeanImpl</context-class>
    <expression></expression>
    <affected-methods>
      <affected-method>
        <context-preparation>
          <preparation-class>org.dedisys.ccmgmt.CalledObjectIsContextObject
                </preparation-class>
        <objectMethod name="setAge">
          <objectClass>ejb.PassengersBeanImpl</objectClass>
          <arguments>
            <argument>java.lang.Integer</argument>
          </arguments>
        </objectMethod>
      </affected-method>
    </affected-methods>
  </constraint>
</ccDefinitions>
```

Listing 2: *ccDefinition.xml:* DemoConstraint metadata

2.2 Enterprise JavaBeans

Enterprise JavaBeans (EJB) is one of the three base technologies of DeDiSys. The DeDiSys EJB implementation is based on the standard EJB 2.1 [DeMi03]. The constraint checking framework and all build applications of this diploma thesis are conformed to EJB.

Enterprise JavaBeans is a major part of the J2EE standard (Java 2 Platform Enterprise Edition). The J2EE platform is an umbrella standard Java's enterprise computing facilities and bundles technologies for a complete enterprise-class server-side development and deployment platform in Java. As described in [RSB05], J2EE is a robust suite of middleware services and builds on the Java 2 Standard Edition (J2SE).
EJB defines the way how components have to be written on the server-side. It itself uses several other J2EE technologies and provides a connection between these components and the application server.

An enterprise bean is a server-side software component that can be deployed in a distributed enviroment. There exists a well-defined interface (this must conform to the Enterprice JavaBeans spezification) for the client application to communicate with the bean. The client software is not predefined and can be anything such as a servlet, an applet or another enterprise bean.

There exist three different types of beans (see Figure 10):

- ***Session beans***
  Session beans are business process objects. They perform communications with other components. There exist two subtypes of session beans: stateful session beans and stateless session beans. Further descriptions about session beans and their subtypes can be find in Section 2.2.1.

- ***Entity beans***
  Entity beans are objects that represent persistent data. They do not contain business process logic like session beans, but handle the business processes. Entity beans model business data so that they can be seen as data objects. Two mechanism are used for persistent management of entity beans: bean managed and container managed persistence. Their difference and further details about entity beans are described in Section 2.2.2.

- ***Message driven beans***
  These beans are similar to session beans, but they can receive and send asynchronous messages. Message driven beans are not used by these thesis.

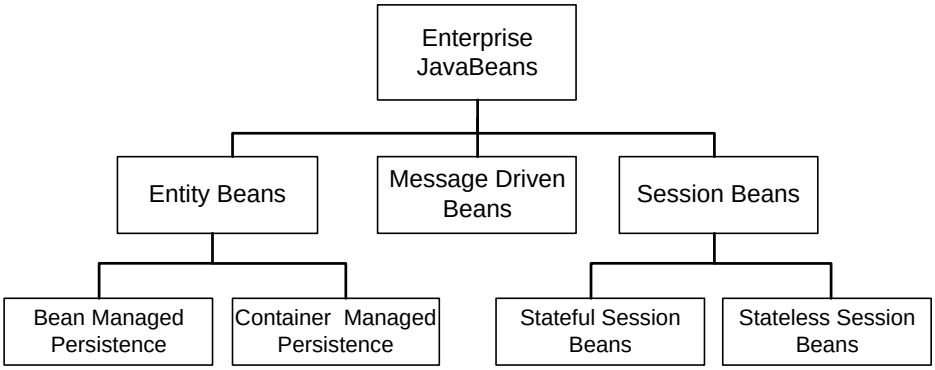

Figure 10: Enterprise JavaBeans

EJB components are based on “distributed objects” that are callable from a remote system. There are no strict restrictions where the client must be. It can be called from an in-process client or clients elsewhere on the network. The following part describes how a client can call a distributed object as illustrated in Figure 11 [RSB05]:

The communicaton represents a remote communication system. At first, the client calls the client-side proxy, named the stub. This proxy calls the skeleton as the corresponding server-side proxy. The skeleton has to forward the call to the distributed object. For the perfect case, a client should not see a difference between local and remote interactions but in practice this is very difficult to achieve because of the presence of network latency for example.

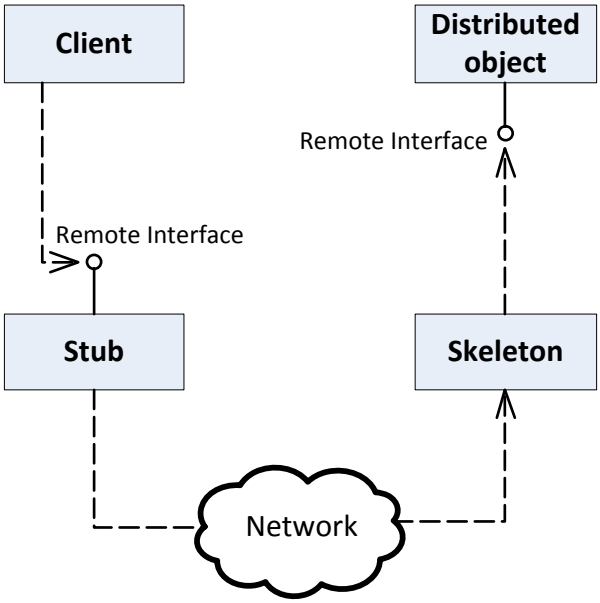

Figure 11: Stub and skeleton [RSB05]

Enterprise JavaBeans contains several Java classes and configuration files such as XML deployment descriptors. The basic interface that all session, entity and message-driven bean classes have to implement is the *javax.ejb.EnterpriseBean* interface. This interface defines the callback methods for the EJB container to manage the beans and its lifecycles. An interesting aspect is that this interface extends *java.io.Serializable*. This means that all beans can be converted to a bit-blob. See [RSB05] for further details. Each bean has its own specific interface that extends *javax.ejb.EnterpriseBean* to define its behavior distinct within the container.

The client never communicates with an EJB implementation directly. The communication goes though RMI (remote method invocation) and the stub and the skeleton are created automatically. The invocation is intercepted by the container that can perform middleware services implicitly, e.g. distributed transaction management, security, persistence, location transparency and so on. Hence, the component developer can concentrate on the implementation of application logic and must not spend time in writing code that calls middleware APIs. Thereby, the EJB container represents a layer of indirection between the client code and the bean that acts as a request interceptor and is called the “EJB object”.

Due to the fact that the implementation of this diploma thesis makes use of session beans and entity beans, they are described more in detail in the following sections. Message driven beans are not used in the context of this diploma thesis.

2.2.1 Session beans

A session bean represents work being performed for the client that is calling it. Session beans are business process objects that implements business logic, business rules, workflow and so on. These beans are called session beans because of their lifetime. They are shortliving and live as long as the “session” (or lifetime) – the time a client communicates with the session bean. In contrast to entity beans, session beans are not shared between different instances and can be used only by one client at a time. The lifetime of beans is managed by an application server (the container).

They are two subtypes of session beans: stateful session beans and stateless session beans. Stateful session beans are designed for business processes with multiple requests or transactions. A good example is the online shopping cart of a webstore where the state must be hold for the whole shopping process. On the other hand, stateless session beans are not preserved for later transactions. This implies that the state is not required to be utilized for a specific EJB client later. For further details see [RSB05].

2.2.2 Entity beans

Entity beans model persistent data that, e.g., can be stored within a relational database. They can survive critical failures and their lifetime might be completly independent from the client's session. They only depend on how long the data sits in the database.

There are two mechanisms for persistent management of entity beans:

- ***Bean managed persistence (BMP)***
  The bean developer himself/herself has to implement the persistence.

- ***Container managed persistence (CMP)***
  The EJB container is responsible for the persistence. All database calls are done automatically. The programer does not need to write any storing relevant code. This mechanism is the favoured persistence one within the DeDiSys project.

2.3 Unified Modeling Language

Today, modelling is an essential part of software projects, not only for large projects, but also helpful for small and medium projects. Modelling is used to design software applications before coding.

In this diploma thesis, the Unified Modeling Language[2] (UML) is used to specify a model. This model implies not only UML constructs, but also constraints in OCL as described in Section 2.4. Additionally, metadata are specified for OCL constraints within this model.

The Unified Modeling Language is a specification from the Object Management Group (OMG) and defines rules how to specify, visualize, construct and document software systems, including their structure and design. It is not only restricted to modelling software applications, but can also be used for business modelling and modelling of other non-software systems like organizational structures.

The UML Specification includes the following [Trus03]:

- a formal definition of the UML metamodel; that is the abstract language for specifying UML models
- a (non-normative) graphic notation for expressing UML models
- a set of CORBA Interface Definition Language (IDL) interfaces for representing and managing UML models
- XML Metadata Interchange (XMI): a format for UML model interchange

[2] Unified Modeling Language (UML): http://www.uml.org/#UML2.0 (14/06/2010)

In UML 2.2, there are 14 types of diagrams defined for graphic notation, divided into two groups: behaviour and structure diagrams. The relationship between these diagrams is illustrated in Figure 12. The main used diagram of this thesis is the class diagram. A class diagram shows the structure of a system by showing its classes, attributes, methods and the relations between the classes. The package diagram is used by this thesis to group the elements providing a hierarchical organization.

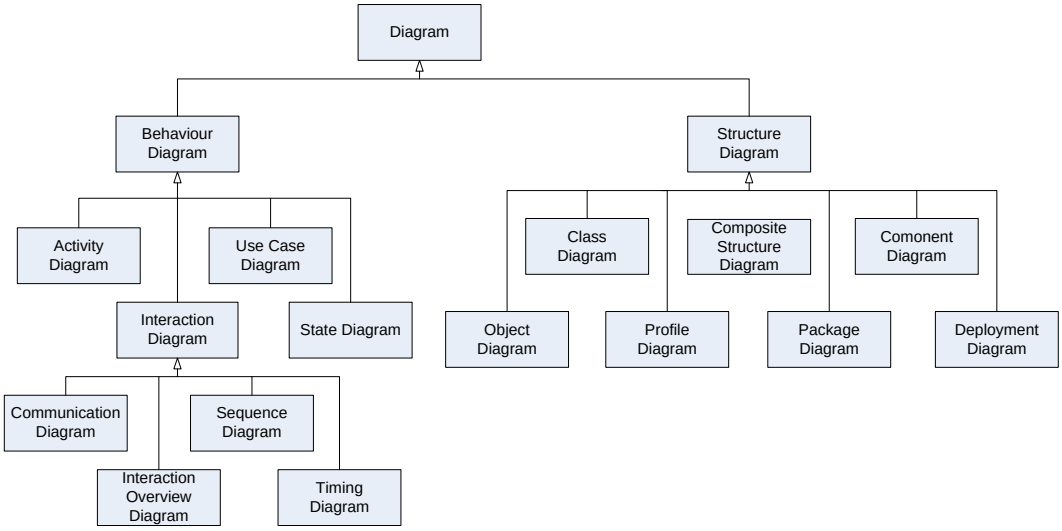

Figure 12: UML diagrams

Figure 13 shows a UML example that is used by this thesis in similar way for code generation. The class diagram is used by this diploma thesis to create the UML-OCL model that is the starting point of the code generation process. Figure 13 shows an example that is used by this thesis in similar way. For more details about the UML class diagram, please refer to the UML specification [UML09].

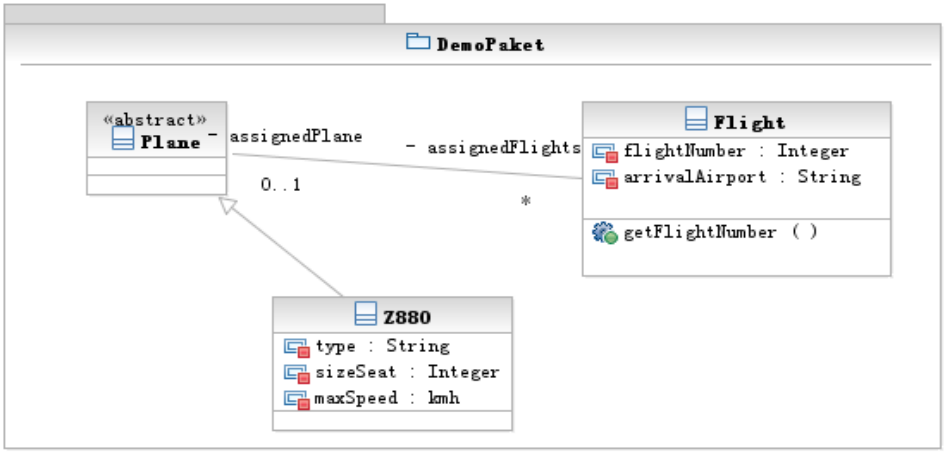

Figure 13: Example - UML class diagram

The class diagram in Figure 13 has a package, named *DemoPaket*, and it contains three classes (*Plane*, *Z880* and *Flight*). A stereotype can be defined by the class name, e.g. 《abstract》 to mark the class *Plane* as an abstract class. An abstract class is a class that cannot be instantiated and is used for inheritance. In this example, it is the class *Z880* that is derived from the class *Plane*.

A class does not have to, but can contain attributes and methods. For example, the class *Flight* has the method *getFlightNumber()* and two attributes, *flightNumber* and *arrivalAirport*. Each class in Figure 13 is connected with another one by a line. This kind of relationship is called an association. Generally, it is a binary association as used in Figure 13, but higher association with more than two classes is also possible. In such cases, the ends are connected to a central diamond. The relationship between the class *Plane* and *Z880* is a particular one because of the inheritance. It is called a generalization.

2.4 Object Constraint Language

The goal of this diploma thesis is to build a code generator for explicit integrity constraints. These constraints are specified in OCL (Object Constraint Language) within a UML model. This section presents general information about OCL.

Definition for a constraint [WK99]**:**
„A constraint is a restriction on one or more values of (part of) an object-oriented model or system."

The Object Constraint Language is a modelling language and not a programming language. Therefore, it has no support to write program logic or flow control in OCL. It is not possible to invoke processes or activate non-query operations within OCL [OCL06]. Firstly, OCL is developed by IBM[3] and now part of the UML standard of the Object Management Group.

OCL allows describing expressions on UML models. An OCL expression itself has no side effects if it is evaluated. It only returns a value after the evaluation. With OCL, it is possible to define constraints in a way that is understandable for modeller with no strong mathematical background. The modeller can specify constraints for their application-models in a simple way. In OCL, there is no possibility to differentiate between hard and soft constraints. It provides only the possibility to define pre-and post conditions or invariant constraints.

OCL can be used [OCL06]:

- as a query language
- to specify invariants on classes and types in the class model
- to specify type invariant for stereotypes
- to describe pre- and post conditions on operations and methods
- to describe Guards
- to specify target (sets) for messages and actions
- to specify constraints on operations
- to specify derivation rules for attributes for any expression over a UML model.

The following example shows that an OCL expression can be defined in different ways:

```
(1) context Meeting inv: end > start
(2) context Meeting inv: self.end> self.start
(3) context Meeting inv startEndConstraint: self.end> self.start
```

Listing 3: Example for an OCL expression

[3] IBM: http://www.ibm.com/ (14/06/2010)

The first keyword *context* introduces the beginning of the constraint. The class *Meeting* is the context in which the statement is valid. The keyword *inv* denotes that it is an invariant constraint. The keyword *self* here is an instance of type *Meeting*, and it belongs always to the object after *context*. Because the context is clear, *self* can be dropped. The third variant defines a name for the constraint after *inv*, but this is optional and not necessary.

Listing 4 and Listing 5 summarize the syntax definition for invariant constraints and pre- and postconditions.

```
context <classname>
inv [<constraintname>]: <OCL expression>
```

Listing 4: OCL syntax for invariant constraints

```
context <classname>::<operation> (<parameters>)
pre [<constraintname>]: <OCL expression>
post [<constraintname>]: <OCL expression>
```

Listing 5: OCL syntax for pre- and postconditions

2.5 Dresden OCL Toolkit

The Dresden OCL Toolkit[4] is developed and maintained by students and scientific staff of the Software Engineering Group at the Dresden University of Technology. It contains several modules. The module used to achieve the goal of this paper is only the OCL Parser. It is responsible for parsing the constraints that are specified within a UML-OCL model. The constraint code generation process of this work is based on this OCL Parser.

Dresden OCL was released in various different versions. During the research period of this diploma thesis, the toolkit in the version 1.x is stopped for concentrating the effort on the version 2. This version, called the Dresden OCL2 Toolkit, is used by this thesis and all described tools in the following section belong to this one.

After the development phase of this work, the third and latest version is released. This version is called Dresden OCL, formerly called Dresden OCL for Eclipse. Dresden OCL is released as a set of Eclipse plug-ins, but a standalone Java library distribution is available as well.

2.5.1 Structure of the Dresden OCL Toolkit

The Figure 14 gives an overview in which parts the toolkit is divided. There are four main parts:

- The MDR repository
  The Dresden OCL Toolkit is based on the Netbeans Metadata Repository (MDR).

- Base tools
  The base tools consist of the OCL2 Parser, the OCL Base Library and the code generator/constraint evaluator. These tools are described in the following.

[4] Dresden OCL Toolkit: http://dresden-ocl.sourceforge.net/ (14/06/2010)

- Tools working on the base tools
  This part consists of two tools. The first one is an OCL Editor plugin that provides a text editor for OCL Constraints. This editor is also adapted as an Eclipse plugin. The second one is the Declarative Code Generator that generates declarative target code for a given expression in OCL and used by the OCL22SQL tool.

- End user tools
  End user tools are standalone applications. There exist seven applications in the Dresden OCL2 Toolkit. These demonstration modules help to understand the way the toolkit works such as the OCL2 Workbench or the OCL2 Parser GUI as shown in Figure 14.

Dresden OCL Toolkit is not a standalone solution but a toolset. Most of these tools are used as libraries that can be integrated into existing standalone solutions. This thesis uses only the components provided by the base tools. For further information about additional tools that are not relevant for this thesis, please refer to [DOCL08].

The base tools are [DOCL08]:

- ***OCL2 Parser***
  The parser requires two steps. The first step creates a concrete syntax tree (CST) from the textual constraint. At step two, the attribute evaluator performs the transformation from CST to AST (abstract syntax tree). This is the only tool that is used by this thesis. For this reason, a detailed description is provided in Section 2.5.2.

- ***OCL Base Library***
  The well-proven "OCL-Basisbibliothek" of the Dresden University of Technology is adapted to fit the new metamodel-based architecture.

- ***Code generator/constraint evaluator***
  This tool is an abstraction of multiple general template engines (as StringTemplate) helpful for the reuse by the Transformation Framework and the OCL Declarative Code Generator.

2.5.2 OCL2 Parser

The OCL Parser in the Version 2.0 named OCL2 Parser is used by this diploma thesis. During the implementation phase of this work, this version of the Dresden OCL Toolkit is still in development. It provides support for classifier contexts and all kinds of invariants. Pre- and post-conditions should also be supported but there is no warranty for completeness [Ocke03].

The following describes how this OCL2 Parser works:
The OCL2 Parser of this toolkit is based on a tailored, hand-optimized L-attributed grammar of OCL2.0. The Dresden University of Technology feeds it to an enhanced version of the popular LALR(1) parser generator SableCC to create lexer, syntax analyzer ("parser") and an abstract attribute evaluator skeleton [DOCL08].

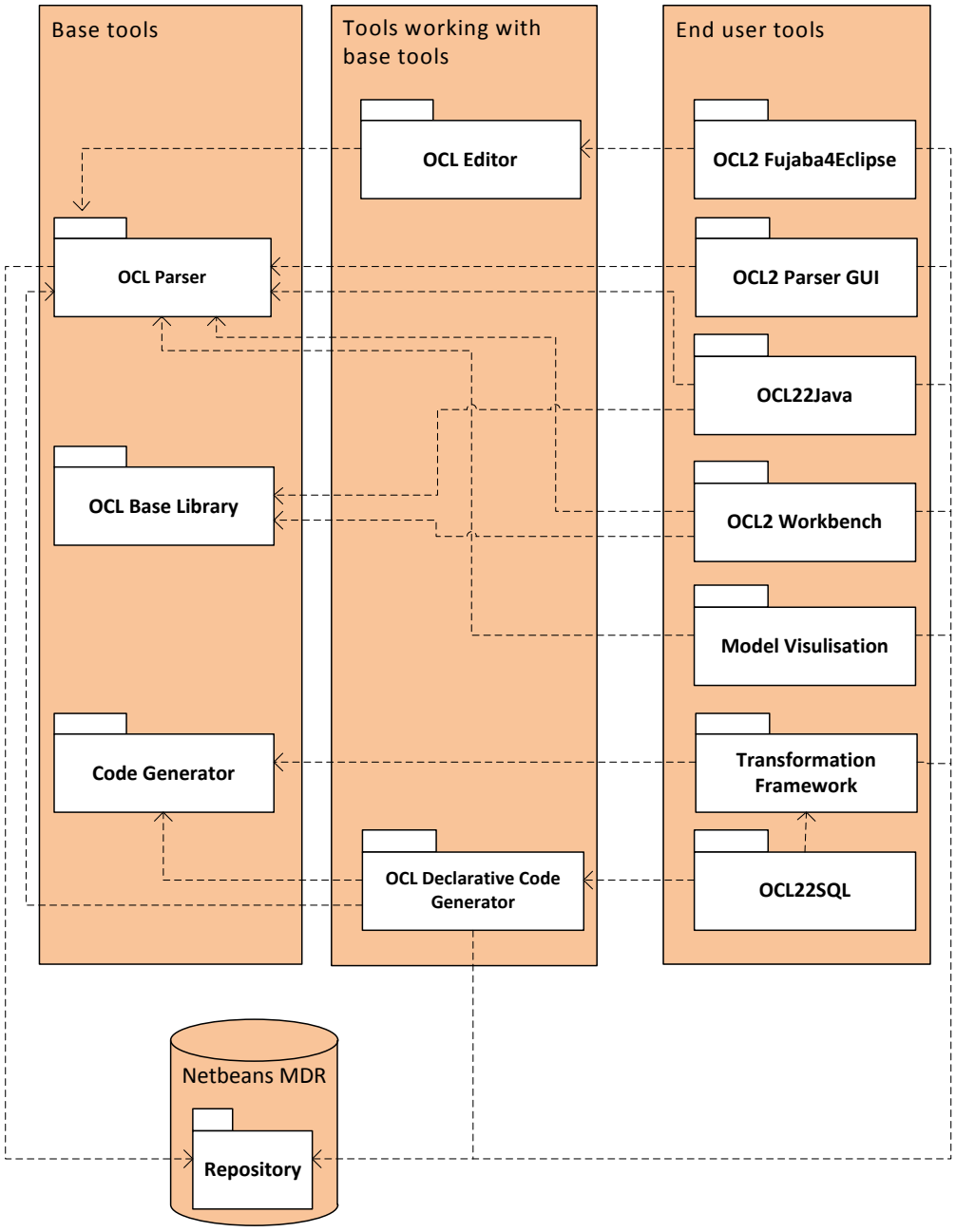

Figure 14: Structure of the Dresden OCL Toolkit [DOCL08]

Development cycle of the OCL2 Parser:
The OCL2 Parser is the only tool from the toolkit that is used for this thesis. There are four steps to use the parser according to [Kone05a]:

1. Get hold of textual OCL
2. Run them through the lexer
3. Load model
4. Run attribute evaluator

Textual constraints can be got from user input or from a file and are represented as instances of *java.lang.String*. The input is labelling as a stream of characters, called input stream. Then a syntax analysis runs through the input stream. A syntax analyzer (parser) transforms the token stream generated by the lexer into a concrete syntax tree (CST) made up of typed nodes.

The lexer is a program performing lexical analysis. The next step is to load a model. Before a model can be loaded, the model must fulfil some conditions: First, the model must match the above-mentioned textual constraints. It must contain all contextual classifiers, attributes, etc. that are used in the textual constraint file. Second, the model must conform to current parser limitations. It must not contain overloaded operations, and operations and attributes with the same name inside any namespace (especially inside any classifier). The last step is to run an attribute evaluator. The attribute evaluator transforms a concrete syntax tree into an abstract syntax tree (AST). There are evaluation rules that describe how to build the corresponding AST node from each CST node type. The AST-stream can be processed directly by other implementations or exported as an XMI document [Kone05a].

Figure 15 shows the above describes steps:

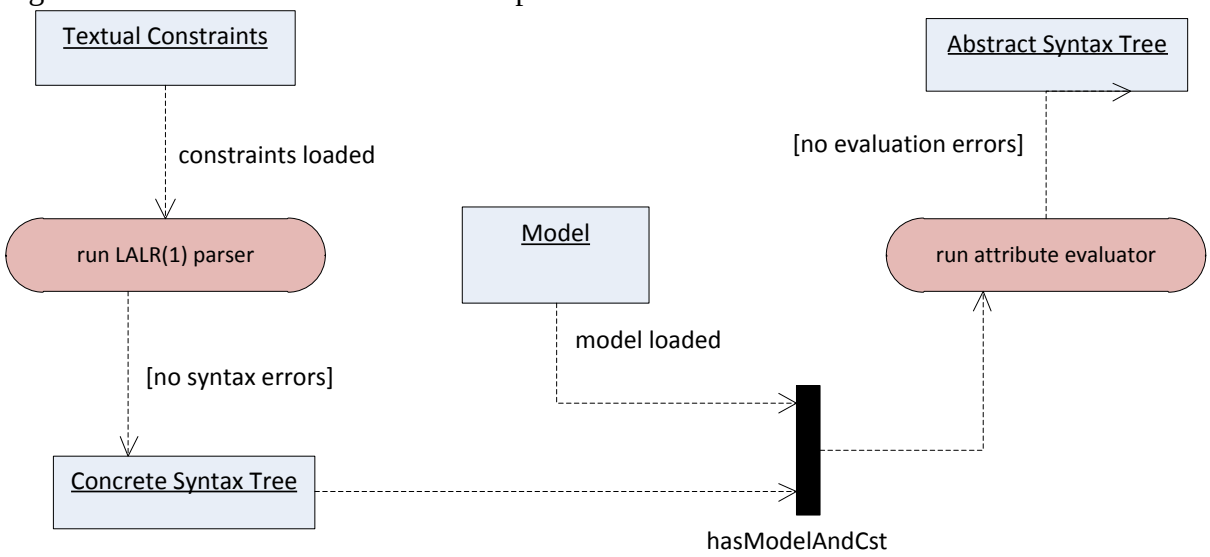

Figure 15: OCL2 Parser [Kone05b]

2.6 Model Driven Architecture (MDA)

This diploma thesis uses an MDA approach to reach its goal. This sections gives a general overview about MDA. It describes how MDA works and its advantages.
Model Driven Architecture (MDA) is strongly driven by the OMG (Object Management Group)[5]. OMG is an organization that provides vendor-independent specifications to improve interoperability and portability in software systems. MDA is a software design approach based on separating between functional and technical implementation.

MDA separates the whole model into two parts [MM03]:

- PIM (*Platform Independent Model*)
- PSM (*Platform Specific Model*)

The principle of MDA is that the user first creates a new design model in the UML-tool that supports UML profiles and constraints. Through XMI export (Extensible Markup Language Metadata Interchange) the user gets an interchangeable design-file (PIM). This model will be inserted into a MDA-generator with the favoured UML-profile. The generator transforms the design model into a new model (PSM). When the platform specific model is generated, the

[5] Object Management Group (OMG): http://www.omg.org/ (14/06/2010)

user should know the platform he/she wants to use for the implementation. The next step is to generate source code for this model. If the platform is J2EE (Java 2 Platform Enterprise Edition) for example, then the used classes and types have to be compliant to J2EE. There are two types of auto-generated sources: The first type does not need manual editing and can be used immediately. The other type gives the user only a structure of the implementation and should be edited manually.

2.6.1 Improvements through MDA

The following factors we can expect from MDA [Geig04]:

- ***Productivity***

  It is out of question that MDA can increase the productivity. However, the extent of this increase is uncertain and should be more investigated in future works.

  For the future, we can assume that there will be many tools that support the works of a developer. It will be possible that one tool can do every part of the MDA process with an integrated MDA development environment.

- ***Portability***

  Portability is achieved through the separation in PIM and PSM. Everything that is defined in PIM is portable and can be transformed in every platform we want.

- ***Interoperability***

  The PSMs generated from the same PIM may have relationships. In MDA, these are called “bridges”. PSMs cannot communicate to each other directly if they are located on different platforms. MDA ensures the interoperability between different platforms by generating not only the PSMs, but also the necessary bridges between them as well. The relationships are illustrated in Figure 16 [KWB03].

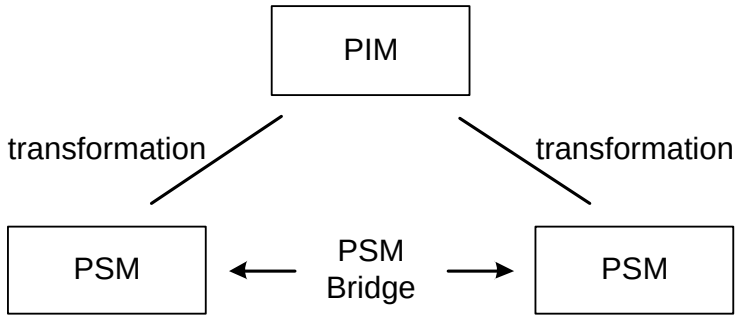

Figure 16: MDA interoperability using bridges [KWB03]

For example, two PSMs are created from one PIM, a Java (code) model and a relational database model. For an element *Flight* in the PIM, we know to which Java class and also to which table this element is transformed. Because we know every detail about the two used platforms, building a bridge between a Java object in the Java-PSM and a table in the database-PSM is easy. To retrieve an object from the database, we query the table transformed from *Flight* and instantiate the class in the other PSM with the data. To store an object, we locate the data in the Java object and store it in the “*Flight*” table.

- ***Maintenance and documentation***
  Because of the consistence of the PIM with the generated source code, the documentation of a system through PIM or PSM is easier than the traditional development process. A good documentation requires a good tool support for the specific platform. The maintenance of complex systems is also easier and more effective by existing of good documentation.

2.6.2 Basis technologies

MDA is based on these technologies [Geig04]:

- ***MOF***
  The MOF[6] (Meta Object Facility) is a standard from the OMG to define, manipulate and integrate metadata and data in a platform. A metamodel describes the correct form of models. The MOF is the basis for all other metamodels. Because of that, metamodels of UML or CWM are instances of the MOF. The syntax and the structure of metamodels are defined by the MOF, on this way all metamodels can be used as on the same way because they all have the same basis. This also makes model transformation easier.

- ***UML***
  UML is the favoured design language of MDA. This work uses UML as its modelling language. The created UML model is the entry point for all other generation steps presented in Chapter 3. For more details about UML see Section 2.3.

- ***XMI***
  XMI[7] (XML Metadata Interchange) is an XML integration framework from the OMG. Generally, a model based on MOF is a graphical model, and with XMI it is possible to transform it in a textual form. XMI allows in an easy way to use the same model in different UML tools or according to MDA to transform a model to PSM.
  This work uses XMI in different ways. First, a model exported with XMI is used to generate entity beans. Second, the same model is used to generate explicit constraint classes.

- ***CWM***
  CWM[8] (Common Warehouse Metamodel) describes metadata interchange among data warehousing, portal technologies and so on. It is similar to the UML metamodel, but it has specific metaclasses, for example to model a relational database. Warehouse-tools that import or export its model with CWM can easily change the data with each other. CWM is not used by this thesis.

2.7 AndroMDA

The open source project AndroMDA[9] is a code generation tool that can produce source code, configuration descriptors, SQL-scripts etc. from UML diagrams. The UML diagrams must be in a specific XML format, in fact AndroMDA uses the XMI format. It first takes a UML

[6] Meta Object Facility (MOF): http://www.omg.org/mof/ (14/06/2010)
[7] XML Metadata Interchange (XMI): http://www.omg.org/technology/xml/index.htm (14/06/2010)
[8] Common Warehouse Metamodel (CWM):
http://www.omg.org/technology/documents/modeling_spec_catalog.htm (14/06/2010)
[9] AndroMDA: http://www.andromda.org/index.php (14/06/2010)

model file as input and can generate source code as output in different programming languages. The programming language depends on which template file is used. The template files can be customized if necessary. Therefore, AndroMDA can produce source code in any programming language. Default AndroMDA uses the template for Java code und in particular for J2EE code.

AndroMDA uses two primary components according to [AMDA08]:

- The AndroMDA code generation engine
- Apache's Maven project builder and management system

The code is generated by the AndroMDA code generation engine. The source code template files are collected in a so called AndroMDA cartridge. There exist cartridges for Java, EJB, Spring, Hibernate, JSF and so on. The engine is not limited so the user can write his own cartridge for every programming language he/she wants. With Maven[10], the AndroMDA engine can be called from a build script. There exist already several Maven plugins so it is recommended from the AndroMDA project members to use Maven. The Apache Ant[11] tool can be used as an alternative build tool instead of Maven.

For this diploma thesis, the EJB cartridge of AndroMDA is responsible to generate entity and session beans. The generated source code from a UML-OCL model is already ready-to-use. AndroMDA is configured through a configuration document that can be called from an Apache Ant build document. Further details are described in the realization chapter, see Section 3.6.

2.7.1 Development cycle

The following describes the steps that are necessary to perform AndroMDA. This introduction should give readers the possibility to know how AndroMDA works as a standalone application. In the realization chapter, it is difficult to differentiate because AndroMDA is a part of the code generator and the development cycle is described as a whole.

The steps of the development process with AndroMDA are briefly described below:

(1) Configure the project with the project parameters for using with Apache Ant
(2) Modelling the project on a UML tool, e.g. ArgoUML
(3) Export the model by XMI export
(4) Start the AndroMDA engine with Apache Ant (source code and descriptor files will be generated on this step)
(5) Edit the generated files if necessary:
    It is important that the AndroMDA engine and all other tools use the same XMI documents. Otherwise, in case of model update a manual merging is necessary.

Each time when the model is changed, the cycle can start again from step two. The files for the business logic that must be edited by the developers manually will not be updated.

[10] Apache Maven: http://maven.apache.org/ (14/06/2010)
[11] Apache Ant: http://ant.apache.org/ (14/06/2010)

2.7.2 CASE tools

AndroMDA is compatible to lots of UML-Tools. Poseidon[12] and Magic Draw[13] are the two tools with the best AndroMDA support. Both are commercial and have a Community Edition with limited features, but only the MagicDraw Community Edition is free for developers while for using the community edition of Poseidon it is necessary to rent a subscription. Finally, this diploma thesis uses ArgoUML that is also fully supported by AndroMDA and released under the open source BSD License. The reason is that this thesis should be based on products with low costs for students and universities.

In the UML model, the so-called stereotype will be used for labelling. For example, if a class has the stereotype "Entity", the EJB-cartridge will take this class and generate the corresponding source code and descriptor files. It is possible to mask a class with different stereotypes at a time.

Features that a CASE tool should support for using with AndroMDA [AMDA08]:

- XMI
- The XMI file should contain a UML 1.4 metamodel
- Class diagrams
- Activity graphs (only necessary when using the bpm4struts cartridge)
- UML tagged values
- UML constraints

A list of UML tools that have been tested for AndroMDA compatibility can be found on the official website of AndroMDA[14].

2.8 Related work

Similar to this diploma thesis, a number of tools that support OCL are available from universities and commercial companies. Different approaches are used to generate code for constraints in several programming languages. A number of them will be presented here.

- ***Run-time constraint checking***

The work of [ASCY10] has evaluated different run-time constraint checking approaches. The authors point out that tangled handcrafted constraint code is the best choice for the use of constraint checking in production code if memory management or run-time performance is important. The code is injected to an implementation, one has to decide for each constraint the appropriate checking points and then transform the constraint to statements of the corresponding programming language. Such kind of constraints implementations are practical if it is used only once and no future changing of validation code is necessary. Otherwise, the choice of the constraint checking approach should be based on aspects such as reusability, controllability and maintainability. For such cases, aspect-oriented languages can be used like AspectJ. The disadvantage is that runtime overhead on CPU time and dynamic memory storage increases significantly which is illustrated in [ASCY10]. In contrast to their work, this

[12] Poseidon: http://www.gentleware.com/ (14/06/2010)
[13] MagicDraw: http://www.magicdraw.com/ (14/06/2010)
[14] List of CASE tools for AndroMDA: http://docs.andromda.org/case-tools.html (14/06/2010)

thesis provides a Java based code generation engine with explicit constraints classes. The generated constraint validation code is reusable and plug-and-playable managed by metadata.

- ***Framework for explicit constraints classes***

The approach with explicit constraint classes is used to encapsulate the constraints code from all other parts of code, e.g. from the code of the business logic. This approach provides more flexibility in handling integrity constraints. For example, integrity constraints can be added, removed, enabled or disabled at any time if all constraints are registered within a so called constraint repository. The approach with explicit constraint classes is issued by [FOG06, VS02]. The work of Froihofer, Osrael and Göschka [FOG06] provides the constraint checking framework for this diploma thesis.

The work of Verheecke and Van Der Straaten [VS02] has several similarities to the approach of this diploma thesis. Its prototype implementation is also based on the Dresden OCL toolkit and uses Java as the programming language. This work uses run-time handling of the constraints and requires constraint metadata for constraint configuration. It uses an explicit management of constraints and consistency threats by the middleware, while run-time handling of constraints is not an issue for Verheecke et al. Their approach is not based on a middleware such as EJB and consequently the triggering of the constraints is hard-coded during implementation through method calls to the constraint validation methods.

Due to the absence of a middleware, Verheecke et al. further describe the requirement to be able to temporarily turn constraints off and on in order to allow certain state transitions involving several methods. For example, if a state transition is made on an object by calling two setter-methods, the object might violate a certain constraint after the first setter, but be consistent again after the second setter. This diploma thesis supports this behaviour through soft constraints. It is not required to enable or disable certain constraints for special behaviour. As mentioned above a constraint repository is used that would allow to add or remove constraints during run-time dynamically, which essentially results in changing the design of running software.

The research area with declarative constraint specification, respectively with subsequent generation of validation code, is not new and has existed for a long time. Several works focus on constraint validation in the sense of Design-by-Contract (also named Programming by Contract) as introduced by Meyer for the Eiffel programming language [Meye92]. It uses pre- and post-conditions to document the change in state caused by a piece of a program.

With the introduction of OCL and its integration into the UML specification, constraints specification based on OCL became an attractive area and is still actively researched. Different approaches of generation of constraint validation code from OCL are explored. For example, Jass [BFMW01] and iContract [Kram98] inject the constraint code directly at the place where the code should be performed, e.g. within the performed method. Dresden OCL Toolkit [Wieb00] uses wrapper methods for constraint validation. Some approaches are compiler-based such as JML [LBR99].

The above mentioned code instrumentation approaches and related technologies that have influence on this diploma thesis are illustrated in Section 3.12.

- ***Transformation of OCL to SQL or stored routines***

The work of [Redl10] builds up on the OCL-to-Java constraint generator of this work presents an optimization possibility to improve the constraint validation process during run-time. The implemented OCL-to-SQL transformer operates on the database layer and is more efficient with respect to performance compared to the OCL-to-Java transformer of this work. Because the work of [Redl10] lacks support for OCL's *iterate*- and *if*-expressions, this thesis implements an OCL-to-stored-routines transformer additionally. The use of stored routines (procedures and functions) for OCL expressions is also issued by [EDC10]. Its approach uses stored procedures to address the problem with OCL iterators, while this thesis is based on stored functions. The prototype of [EDC10] only generates SQL queries of OCL expressions that can be used for further implementation in different applications. In contrast, the code generator of this work is based on applications and the generated constraints code operates as ready-to-use constraints for a special application.

3 Realization

Constraints and constraints metadata are the main focus of this diploma thesis. Constraints are based on explicit constraint classes and configured with metadata. Metadata are required to specify the running behaviour of a constraint because the explicit constraint class only contains the condition statement. Because all constraints use the same structure, it is not only tedious to write the constraints with the same code, but also error prone. A small careless mistake in the constraint code could cause wrong running behaviour of the whole system such as using the “>” quantification operator instead of “<”. Another case would be if a constraint was wrong configured with metadata of another constraint. For example, if because of wrong results one more ticket in a ticket application is sold, this may not as important as if an airplane gets a take-off signal instead of remaining in waiting position. The last error scenario is intolerable because of safety reasons. Therefore, an error management is necessary to avoid such kinds of errors. If every code has to been checked and declared as error free manually, the effort for checking is very high. Furthermore, such a small mistake could remain undiscovered even after several code checks and test evaluations.

A code generator for constraints and constraints metadata could reduce the effort for error proofing. For example, both constraints and metadata could be specified within the same model. The generated metadata are also ready-to-use without any changes. The generated explicit constraint classes have a predefined structure that is required for example by the constraint checking framework of the research project DeDiSys. The effort of a developer could be reduced to model constraints and constraints metadata in a UML model. The constraint code is also ready-to-use, but it could be modified to achieve better performance for example. However, this is much more error safe than to write the whole constraint class manually because it is only required to change the condition query statements. Generally, a CASE tool is used for modelling an application including corresponding constraints and metadata. Such a tool provides a GUI that also supports OCL. Error detection in OCL statements is quite more comfortable and precise than directly in the constraint code. The uniformity of code by a code generator is a way to reduce the effort in coding and the probability of making careless mistakes. The developer has only to concentrate on modelling and – if necessary – on optimization of constraint query statements.

This chapter presents the design approach of the code generator regarding to this thesis’ focus. At first, the MDA approach is described. At the beginning, it seems that this approach will cost more efforts than other approaches. This view is shortened because for lots of tasks such as repeated processes or reusable concepts the use of such approach would bring many advantages. Next, a description is given about the development problem during the implementation process and a possible solution for the whole generation process is presented. Afterward, the implementation is described in separate sections that include EJB generation, textual constraint generation, constraint and metadata generation. Additionally, the integration of all parts into one project is illustrated. At last, two additional works for the implemented code generator and related technologies in addition to related work in Section 2.8 are presented.

3.1 MDA approach

Before the solution of this diploma thesis will be presented, this section gives the reasoning why a Model Driven Architecture (MDA) approach is reasonable. Developments with MDA are not always good, but there are some particular reasons to use a MDA approach.

The following listed some points that are useful during the decision process [Geig04]:

- ***Repeated process***
  If a development for new projects is similar to the last one, then it will be a good idea to make a pattern for this kind of tasks. The development will be quicker and more comfortable for the designer. For example, a MDA approach can be used to generate objects from only one UML model for different programming languages.

- ***Reusability***
  For future projects, the chance that some parts will be reused is higher because of the MDA approach: Not discrete components will be reused, but the development concept.

- ***Specific architecture concepts***
  If a specific architecture concept is required, an MDA approach will be a good help for such situations. MDA hides the structure from developers. For instance, a developer can use EJB although he/she is not specialized in EJB. Because the required interfaces are generated from the model, his/her job is only to implement the business logic.

- ***Refactoring or complete new project***
  An MDA approach is good to use on a new project. However, the use of MDA on a running software project is very difficult. It is a challenge to reproduce an existing architecture to a PIM model. For new projects, the way is much easier because the PIM can be created before the implementation.

The goal of this thesis is to provide a code generator that uses the MDA approach for the code generation process. The reason is that MDA provides flexibility, uniformity and simplification. Developers should get a new way of application implementation by tool support. A developer should only need to create a UML-OCL model and specify the required metadata to each constraint. He/she should understand the basics of the based technologies and how it could be used, but it is not necessary to know how the generation process is implemented. He/she continues the implementation where the code generator ends. Generally, most parts of the code are ready-to-use and only the business logic has to be implemented manually.

3.2 Realization overview

The goal of this diploma thesis is to find out a way to solve the following problem:
On the one hand, there are many UML modelling tools. Modelling UML classes with OCL is no more a problem and should be learnable by everyone in a short period. On the other hand, the DeDiSys project provides a constraint checking framework that is able to support integrity constraints with explicit constraint files. Most solutions on the market do not support integrity constraints explicitly, the code for constraint checking is mixed with the other code (e.g. the business logic). In the sense of separation of concerns, integrity constraints and constraint checking should be treated explicitly within a system and not distributed throughout the whole system. Another big problem is due to the limitation of modelling tools that it is not possible to specify metadata for OCL constraints within a UML model. This may not be a problem for

non-distributed environments but for replicated distributed systems metadata are required for adapting. The constraint checking framework of DeDiSys already supports metadata specification to reach its goal, in fact to investigate the trade-off between availability and consistency on dependable distributed systems.

After studying the problem area, the task of this thesis can be defined as the search for a solution to deal with the gap between the modelling tool and the DeDiSys Framework. Figure 17 shows which elements are already supported by the tools.

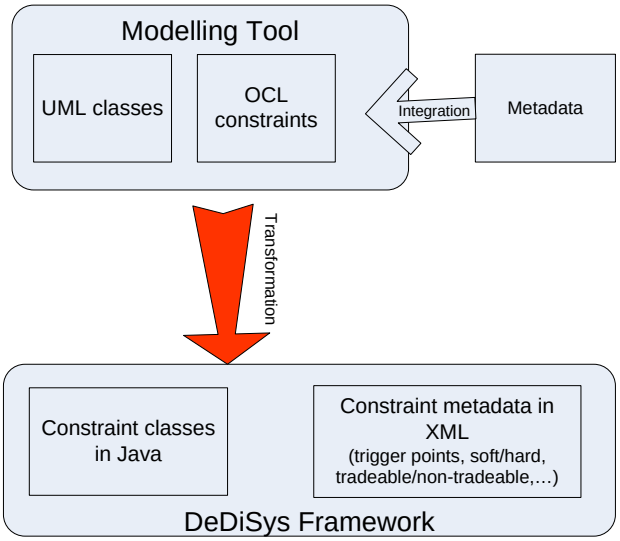

Figure 17: Problem and goal description

The solution for this problem and the main goal of the diploma thesis is to build a code generator that provides MDA support for the software development process building upon EJB and explicit integrity constraints including metadata specification. It should take a UML and OCL design model and generate entity beans, session beans, constraint implementations and constraints metadata for the constraint checking framework in EJB. The new code generator could be based on the AndroMDA framework because AndroMDA already provides the possibility to generate entity beans and session beans for a UML design model. For the constraint part, the parser of the Dresden OCL Toolkit could be used. The support of metadata specification by the modelling tool should also be implemented as a completely new part. The implementation process is presented in the following sections.

3.3 Solution of the code generation process

This section presents a solution to cope with the problem of the previous section. The illustrated elements in Figure 18 are all used by the final design approach. First, a UML-OCL model has to be modelled. This model is used to generate EJB entity beans, explicit constraint classes and metadata.

Alternatives and changes of the approach are not shown. For example, different approaches are used to create the constraints in textual format. ArgoUML is also not the first choice as the modelling tool. The reason why other CASE tools such as MagicDraw or Poseidon cannot be used are described in Section 3.4. Different approaches for metadata integration are also discussed in Section 3.5.

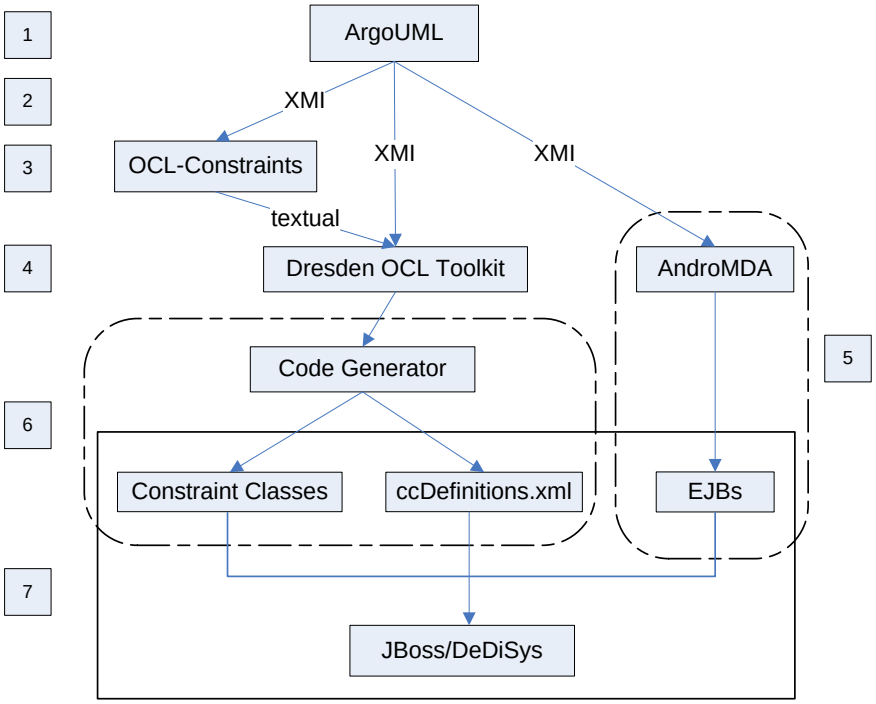

Figure 18: MDA processing steps

The code generation process has the following steps in detail and each step is marked in Figure 18 with a number:

1. The first step is to create a model that contains UML elements as well as OCL constraints and metadata. The used modelling tool is ArgoUML. For this work, only a class diagram is required. The specified classes that are used by AndroMDA contain the stereotype <<*Entity*>>. For constraint specification, ArgoUML provides an OCL editor. Metadata is implemented with an own approach that will be presented in Section 3.5.

2. The created model is exported as output in XMI file format. This document is used three times as the entry point for other processes.

3. From the exported XMI document, a new document is created which contains the OCL constraints in textual format. A detailed description and the required transformations are presented in Section 3.7.

4. Dresden OCL Toolkit transforms the exported XMI file together with the textual constraint file into an abstract syntax tree (AST) that will be used by the code generator (see Section 3.8).

5. AndroMDA is used to generate entity and session beans. For further details see Section 3.6.

6. The code generator has two tasks: the first one is to generate a constraint class for each defined OCL constraint in Java. Second, a deployment descriptor called *ccDefinitions.xml* has to be created. These two parts are described in Section 3.8 and Section 3.9.

7. The last step is to integrate the configuration document *ccDefinitions.xml*, the generated EJBs and the Java constraint classes into one project that are conform to the DeDiSys Framework that uses JBoss as its application server. Details are illustrated in Section 3.10.

3.4 Modelling

During the implementation phase, several UML-tools are chosen and evaluated. Three tools have relevance for this thesis:

- Magic Draw
- Poseidon
- ArgoUML

Magic Draw (Magic Draw UML Community Edition 9.5) works with AndroMDA very well and would be the preferred UML-tool if there were no problems in context with the Dresden OCL Toolkit. The Dresden OCL Toolkit uses a UML-profile that is not compatible to the profile of Magic Draw.

The next choice falls to ***Poseidon*** (Poseidon For UML CE 3.1) because Poseidon uses the same Meta-Object Facility (MOF) as the Dresden OCL Toolkit, in fact the NetBeans Metadata Repository (MDR). Poseidon has its origin in ArgoUML and fulfils all conditions that are essential for this thesis. Because Poseidon cannot provide the specified constraints in textual format, an own implementation is realized to perform this transformation. The idea is to extract the constraints out of the XMI model and saves it in a textual document separately. This approach is used at the first stage during the development phase of this diploma thesis. Finally, a changing of the license agreement for Poseidon necessitates the use of another tool because this thesis should be based on products with low costs for students and universities. At the end of 2006, Gentleware, the producer of Poseidon, demands a rent subscription for using the community edition of Poseidon.

Finally, ***ArgoUML*** (ArgoUML v0.24) has been chosen as the UML modelling software. ArgoUML is a Java based universal modelling language tool and released under the BSD Open Source License.

ArgoUML is written in pure Java and is available on any platform supported by Java. It supports all standard UML 1.4 diagrams and provides a GUI, in up to ten different languages such as English, German, French or Spanish. Different export functions are supported such as export the model in XMI format instead of with the default extension '*.zargo*'. Graphics can be exported single or all at once and saved in different formats such as GIF or PNG. Furthermore, ArgoUML supports code generation for Java, C++, C#, PHP4 and PHP5 and provides Java reverse engineering. Because of using a modular framework, other languages may be added in the future too.

ArgoUML also provides constraint modelling support on UML classes. The support of OCL was a requirement of this diploma thesis. With ArgoUML, the complete model that is required by the code generator can be modelled. After the modelling phase, only the model exported in XMI format is required by the code generator.

3.5 Metadata integration in CASE tool

OCL is an established useful language to specify constraints upon the UML class model. This section describes first how OCL could be extended to use metadata required for run-time explicitness.

For the use of run-time integrity constraints, it is necessary to specify metadata as described in Section 2.1.5. Metadata would allow specifying further properties about constraints. This makes it possible specifying details such as whether a constraint has to be checked during or at the end of a transaction. These details in the design models are required in order to configure correct run-time behaviour.

There is a limitation of OCL not being able to specify metadata for OCL constraints as required by this diploma thesis. The implementation of metadata in UML-OCL modelling process is not researched in the past or only partly covered by other research works.

OCL could be enhanced to support metadata in different ways that meet all demands of explicitness constraints. The enhancement should be transparent and user-friendly for a modeller with this tool.

Different options are possible for this purpose:

- Adapt and enhance the OCL syntax to allow arbitrary metadata
- Use UML class diagrams for constrained metadata with the advantage of staying OCL-compliant

After a comparison of complexity and support by existing tools, the approach is based on option two. The first option is also feasible, but not supported by existing modelling tools. For example, to validate OCL expressions a tool modification is required. As corresponding tool development was not in the focus of this diploma thesis, the option two is chosen for metadata implementation. Other options such as to encode the metadata as stereotypes were not further investigated, because these appeared not to be practicable.

The idea is to use a separate UML class diagram for each constraint. An additional specification is using the same name for OCL constraints and corresponding classes, thereby establishing the link between the constraint specified in OCL and the metadata given in the UML class diagrams. Metadata are provided by predefined attributes with specific types for constraint classes in the class diagram. Figure 19 shows an example with the flight class and its constraint *FlightTestSelect* with a metadata specification.

```
context Flight inv flightTestSelect :
self.left->size > 0 implies left -> select ( f
: Flight | f . fromAirport = 'Vienna' and f .
toAirport = 'Zurich' ) -> notEmpty
```

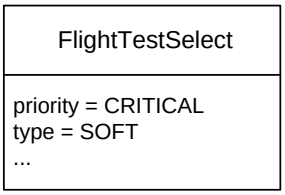

Figure 19: *FlightTestSelect*: constraint and metadata

This kind of class diagram has to be defined for every specified constraint that should be considered by the code generator. A blank class with the name of the constraint is necessary if default values should be used. The reason for this restriction is that this allows the developer

also to specify constraints that should not be used by the constraint checking framework of DeDiSys. An explicit constraint class is also generated for each of these “not relevant” constraints. For the constraint checking framework, it is only important that there is no record in the deployment descriptor *ccDefinitions.xml* for these constraints. Consequently, metadata are specified as part of the UML-OCL model that can used for the whole code generation process.

3.6 EJB generation

This section describes the generation of the entity and session beans from a UML-OCL model through an external application. This task is labelled as the step 5 in Figure 18 on page 30.

Using an automatic generation approach means that every source has to be resistant to the others. Otherwise, only a short mistake, such as using lower case instead of upper case, could cause a leak with the effect that the sources are inoperative. Consequently, the best way would be to generate the EJB sources directly from the UML-OCL model. Handwriting sources can never guarantee that they are free of careless mistakes. It would take much more time to write manual resistant codes.

The realization is quite simple for this part. There is no code implementation because the code generation engine of AndroMDA can generate all required sources. The EJB cartridge of AndroMDA is used to generate entity beans. AndroMDA identifies the classes by the used stereotypes on the UML tool. Depend on which stereotypes are used, AndroMDA generates the corresponding sources. For this thesis, the *<<Entity>>* stereotype is used to generate EJB sources for classes. *<<FinderMethod>>* and *<<CreateMethod>>* are used by methods to classify them as finder or creator methods. The *<<Unique>>* and *<<Identifier>>* stereotypes are used to declare attributes with the unique and the primary key criterion.

The first thing to do is to write an Ant target to start the AndroMDA build process. Besides including the necessary libraries, it is only necessary to point to the AndroMDA configuration document. In Listing 6, an example shows such an Ant target that is used for this work.

AndroMDA is configured through a single XML document with the default name *andromda.xml*. The required attributes and elements for this diploma thesis are described in Table 1, for further details look at the AndroMDA website [15]. A typical configuration document for this work is added to Appendix A3.
AndroMDA generates three output documents for each class that contains the stereotype *<<Entity>>*. For example, if the class with the name “*Flight*” is specified, then the following files will be created:

- *Flight.java*
- *FlightBean.java*
- *FlightLocalHome.java*

[15] AndroMDA Ant Task: http://docs.andromda.org/andromda-ant-support/andromda-ant-task/index.html (14/06/2010)

```
<target name="run.andromda" depends="…"
description="run AndroMDA">

<property name="xmi.file" value="…" />

<property name="output.dir" value="…" />
<property name="andromda.dir" value="…" />

<path id="project.class.path">
<fileset dir="${…}">
…
</fileset>
</path>

<taskdef name="andromda" classname="org.andromda.ant.task.AndroMDAGenTask"
    classpathref="project.class.path"/>
<andromda configurationUri="file: …"/>
</target>
```

Listing 6: Apache Ant target for AndroMDA

Element	Property	Description
<andromda>		Root of the configuration file
<properties>		
	modelValidation	Validate the loaded model by AndroMDA
	cartridgeFilter	Specify which cartridges should be processed, otherwise any discovered cartridges will be processed.
<repositories>		Define which repository(s) to use when processing a model
<namespaces>		Activate a cartridge or customize the properties of a plugin (cartridge, translation-library, mapping-files etc)

Table 1: AndroMDA configuration file

3.7 Textual constraint generation

The task labelled as the step 3 in Figure 18 is to generate a document in textual format that contains the OCL constraints. The constraints are specified within a UML-OCL model.

The OCL constraints are defined on a UML model. The so composite UML-OCL model can be exported as an XMI document, but it is impossible to export the constraints alone. The problem is that the Dresden OCL Toolkit needs a UML model in XML format and the OCL constraints explicit in textual format as its input files. Because of this reason, it is necessary to create a separate document that contains the constraints in the required format to cope with the Dresden OCL Toolkit.

As mentioned in Section 3.4, different UML tools are used and their exported XMI files do not have the same structure. At the time when Poseidon was used, the textual constraint file is

created through an XSLT transformation and the Ant build tool is used to call this transformation. However, the output file of ArgoUML has another structure and the first approach seems not good enough to cope with this format. A new approach through programming in Java is implemented.

3.7.1 Approach for Poseidon

The process starts with an Ant target (see Listing 7). The goal is to generate two documents from the UML-OCL model, one creates a constraints file in textual format and one generates a new UML model without OCL statements.

The transformation runs with Extensible Stylesheet Language Transformations (XLST). XLST is an XML-based language used for the transformations of XML documents into other XML documents. The following two documents are used to perform this transformation:

***xmi2ocl.xsl**:* A textual file with the OCL constraints should be created. This happens by searching the XMI file for all elements that are parts of constraints and copying these parts to a new document. If no constraints are specified, a blank document will be created. The condition for a successful process is that the constraints in the model are in the correct format that are required by the Dresden OCL2 Toolkit. Otherwise, the validation process of the Dresden OCL2 Toolkit would fail and the generation process would be stopped.

***xmi2xmi.xsl**:* The only thing to do in this transformation is to create a new file from the given one by ignoring all elements that belong to a constraint.

The two XLST transformation documents are listed in Appendix A4 and A5.

```
<target name="xslt2" depends="…"
        description="creates the ocl-files through XSLT-Transformation">

   <xslt basedir="model" destdir="…"
        force = "yes"
        extension=".ocl"
        includes ="*.xmi"
        style="xmi2ocl.xsl">
   </xslt>

   <xslt basedir="model" destdir="…"
        force = "yes"
        extension=".xmi"
        includes ="*.xmi"
        style="xmi2xmi.xsl">
   </xslt>
</target>
```

Listing 7: Apache Ant target for textual constraint generation

3.7.2 Approach for ArgoUML

This approach is the final solution to create the textual constraint document. It is written in Java and is not only a transformation process as the first approach. The reason why this approach is required is that in ArgoUML the OCL constraints are saved without their package names and only attribute names are used instead of method names that are required for the generation process. *ArgoUML v0.24* does not support the functionality to disable the constraint syntax checking function and it makes it necessary to rewrite the constraint according to a structure required by the Dresden OCL Toolkit.

This task is directly integrated in the code generator. It performs a transformation from the constraints specified in a model to the one in textual format. First, it searches the model tree for OCL constraints. If a constraint is found, the OCL expression will be selected and saved in a buffer. A modification of the OCL expressions is performed to ensure the compatibility with the Dresden OCL Toolkit. After all constraint expressions are transformed, a document in textual formal will be created by using the extension '*.ocl*'.

All necessary parts are implemented within the *TextualConstraintsGeneration* class. This class has four methods:
OCLDefinitionGenerator(): This method is the entry point for the creation process of the textual constraint file. The task is to extract all constraints from the UML-OCL model and to save it to a buffer. After the transformation of the OCL constraints to the correct output format is performed, all constraints will be written to a document in textual format.

addParenthesis(): This method verifies if the given string is a specified method of the class diagram or not. If yes, then parenthesis will be added to the given string to use it as a method.

createFile(): A new document will be created with the given strings as the destination path.

getAllClassesModel(): This method retrieves all defined classes from the given OCL model.

The UML class diagram is shown below:

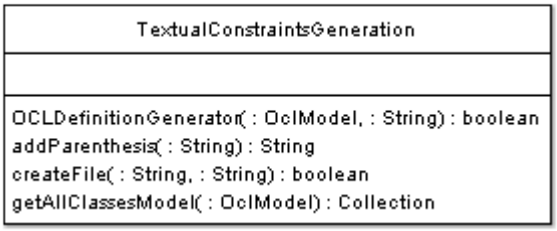

Figure 20: Class diagram: textual constraint generation

3.8 Constraint generation

This section handles the structure of the code generator. It is based on the Dresden OCL Toolkit. The used main module is the OCL Parser, described in Section 2.5.2. The implementation of the code generator, labelled as the step 4 and 6 in Figure 18 on page 30, will be described in the following.

The goal of the code generator is to generate explicit constraint classes from a UML-OCL model. The support of OCL is an important point on MDA developments in the future. OCL supports three kinds of constraints: invariants, pre- and post-conditions. OCL should be easy to learn and to understand, but code implementation needs knowledge about the underlying technologies. For different technologies, a different implementation is necessary. A code generator can support a developer during the development process. He/she only needs to create a model and the code generator would generate the basic structure of the source code in the next step. Within the scope of this diploma thesis, only Java is supported.

The code generator is not only based on other technologies, but it is also cooperation between different tools. This subsection handles the requirements that must meet for the right functionality of the code generator:

- ***Support OCL***
  OCL will be used to define the constraints. Because the Dresden OCL Toolkit is the entry point of the code generator, the constraints must be converted to a format that can be used by it.

- ***Support the Dresden OCL Toolkit***
  The Dresden OCL Toolkit is the fundamental tool to start the code generation process. It transforms the constraints from the UML-OCL model to a format that can be used for code generating. It provides the interfaces that are necessary for accessing through the UML classes and OCL constraints.

- ***Support the constraint checking framework of DeDiSys***
  The constraints that will be generated by the code generator must have the format that is defined by the constraint checking framework of the DeDiSys project. Because it should be used on the JBoss application server, the implementation must be programmed in Java. The constraint checking framework is responsible for checking the constraints during run-time. To identify the constraints, a configuration document named *ccDefinition.xml* is required. This file has the functionality to switch on/off the constraints that should be checked or to consider different properties. It is necessary to create a configuration document that is conforming to the generated Java constraint classes.

3.8.1 Structure of an explicit constraint class

The simplest approach for constraint checking in Java is to tangle the constraint checking code with other code, e.g. the code for the business logic. This kind of approach uses mostly if-statements to check certain conditions.

Another approach used by the Dresden OCL Toolkit is wrapper based constraint validation. The original method is wrapped and the constraint validation code is contained in the wrapper method. For example, the original method *methodName* would be renamed in *methodName_wrapped* and only called via the wrapper method named *methodName*. This example is shown in Listing 8.

These two approaches of constraint validation are not optimal for this work because the code is tangled with other code. Consequently, the structure of the constraints in this work is predefined by the constraint checking framework of the research project DeDiSys.

```
// - > wrapper method with the original method name
public int methodName (int i) {
        // - > constraint code for invariants
        //      and preconditions
        // -> call the original method
        int result = methodName_wrapped (i);
        // - > constraint code for invariants
        //      and postconditons
        return result;
}

// - > original wrapped class
public int methodName_ wrapped (int i) {
        // - > original method logic
        return result;
}
```

Listing 8: Wrapper-based validation

The constraint checking framework of DeDiSys uses explicit constraint classes for validation. The constraint code is contained in a validate method that is executed whenever a constraint has to be checked. Generally, constraints are specified within the context of a class for invariants or the context of a method for pre- and postconditions. When explicit classes are used for constraint checking, the code for invariants is similar or completely the same as the code for pre- and postconditions. The reason is that explicit classes contain only the validation code, but no information about the properties of a constraint. The properties are set as metadata in a separate file.

This approach requires also a trigger mechanism to ensure that the validate method is called whenever necessary. Within this work, the constraint checking framework provides a trigger mechanism configured with metadata. Listing 9 shows an example of an explicit constraint class that is similar to the generated constraints by the code generator of this work.

The simplest case for the validate method is if it contains only one if-statement. The method returns *true* or *false* back depending on whether the check is satisfied. The performance of such a constraint is quite fast and negligible to the whole running time. A more complex condition statement is if it contains different loops or calls to remote objects. The execution time of such constraints will be increased several times as illustrated in the evaluation section. The generated constraints may not be the best in performance at all time and different modifications could be performed to increase the performance. The difference in performance between generated and manually optimized constraints or between validation on the object and database layer is shown in the evaluation section.

```
package packageName;

import package.*; //several import statements

public class ConstraintName {
        public boolean validate (Object o) {
                boolean result = false;
                // - > constraint checking code
                // - >  the variable "result" will be set depending
                //      on whether the constraint is satisfied.
                return result;
        }
}
```

Listing 9: Explicit constraint class

3.8.2 Integration of the Dresden OCL Parser

The Dresden OCL Toolkit is a toolset provided by the Dresden University of Technology. It contains several modules such as an OCL Parser, different GUIs or plug-ins into the Eclipse development environment. There are different versions of this toolset and the used one for this work is the version 2.0.

The input for the Dresden OCL Parser is textual constraints that can be got from user input or from a file and are represented as instances of *java.lang.String* (labelled as the step 4 in Figure 18 on page 30). The parser transforms the input stream generated by the lexer into a concrete syntax tree (CST). The lexer is a program performing lexical analysis. After a model is loaded, the attribute evaluator performs the transformation from CST to AST (abstract syntax tree).

The task of this work is to integrate these functionalities into the constraint code generator. Figure 21 shows that the class *UML2Model* is responsible for the integration of the OCL Parser. The input documents are a UML model and an OCL file in textual format. The generation of the OCL file is described in Section 3.7. After the transformation into an AST, the output stream can be saved in a separate XMI document, but the code generator of this work uses the output stream directly for further performing steps.

The methods of *UML2Model* are illustrated in the following:
parseModel(): This method is the entry point of this class. The given argument is the path of the model. After the parsing process, a new model will be returned that will be used as the starting point for all other generating processes of this tool.

loadUMLOCLModel() and *loadModelXmi()*: These two methods have the job to initialize the model for the following parsing process. The XMI model document is provided to the string converter as the Dresden OCL Toolkit requires textual constraints from a text document. This conversion is described in Section 3.7.

loadOclFile(): The text document that contains the OCL expressions will be readout by this method.

runParser(): It starts the process to parse the constraints. A concrete syntax tree (CST) will be created. It integrates the functionalities provided by the OCL Parser of the Dresden OCL Toolkit.

generateAst(): This method uses the libraries of the Dresden OCL Toolkit. The attribute evaluator will be started and its goal is to transform the CST into the abstract syntax tree (AST). The output of the transformation is a new model that is used by the code generator of this work.

saveModel(): The created new OCL model can be saved as a document in XMI format. A model and the path of the output document must be given as parameters. This is an additional method. It is not required by the code generator, but the saved XMI document could be used for analytical reasons.

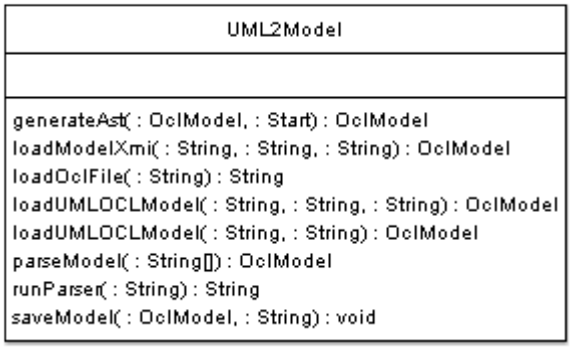

Figure 21: Class diagram: model transformation

3.8.3 Code generation architecture

This section describes the implementation of the constraint code generation process that is labelled as the step 6 in Figure 18 on page 30. The code generator works with the OCL model which is provided by the OCL Parser of the Dresden OCL Toolkit. Details about the constraints generation will be illustrated in Section 3.8.4. This section has its focus on the structure of the code generator.

Figure 22 shows the class diagram of the code generator. It contains five classes and is responsible for the generation of the explicit constraint classes. The constraints are extracted from the OCL expressions of the UML-OCL model. The generated code uses the predefined structure of the constraint checking framework of the DeDiSys project. The code generator generates one constraint class for each defined OCL expression. Each class contains only the query statement to validate the constraints, but no properties about the constraints. The properties are defined as metadata and its generation is performed with an own process.

The involved classes and their function are described briefly in the following:
ClassCodegenerator: This class is the controller class for the generation process.

CodeGenerator: This class is responsible for the generation of the explicit constraint classes. Different OCL statements and constraint types are supported by this class.

MDAConstraint: This class contains several methods to handle a given OCL constraint. With these methods, it is possible to get more information of a constraint and to convert constraints to different formats that are required for the following generation process.

JavaTypeMapping: A UML tool usually uses platform independent data types. This class has the functionality to map the given data types of a UML tool onto Java compliant data types.

CCDefinitionAffectedMethod: The task of this class is to prepare the data for the generation process of the configuration descriptor *ccDefinitions.xml*. A developer defines several properties for a constraint in a model. The properties are extracted from the model during the creating process of the explicit constraint classes. After a check is performed whether the properties are valid, this class collects all data about a constraint required by the metadata generation process.

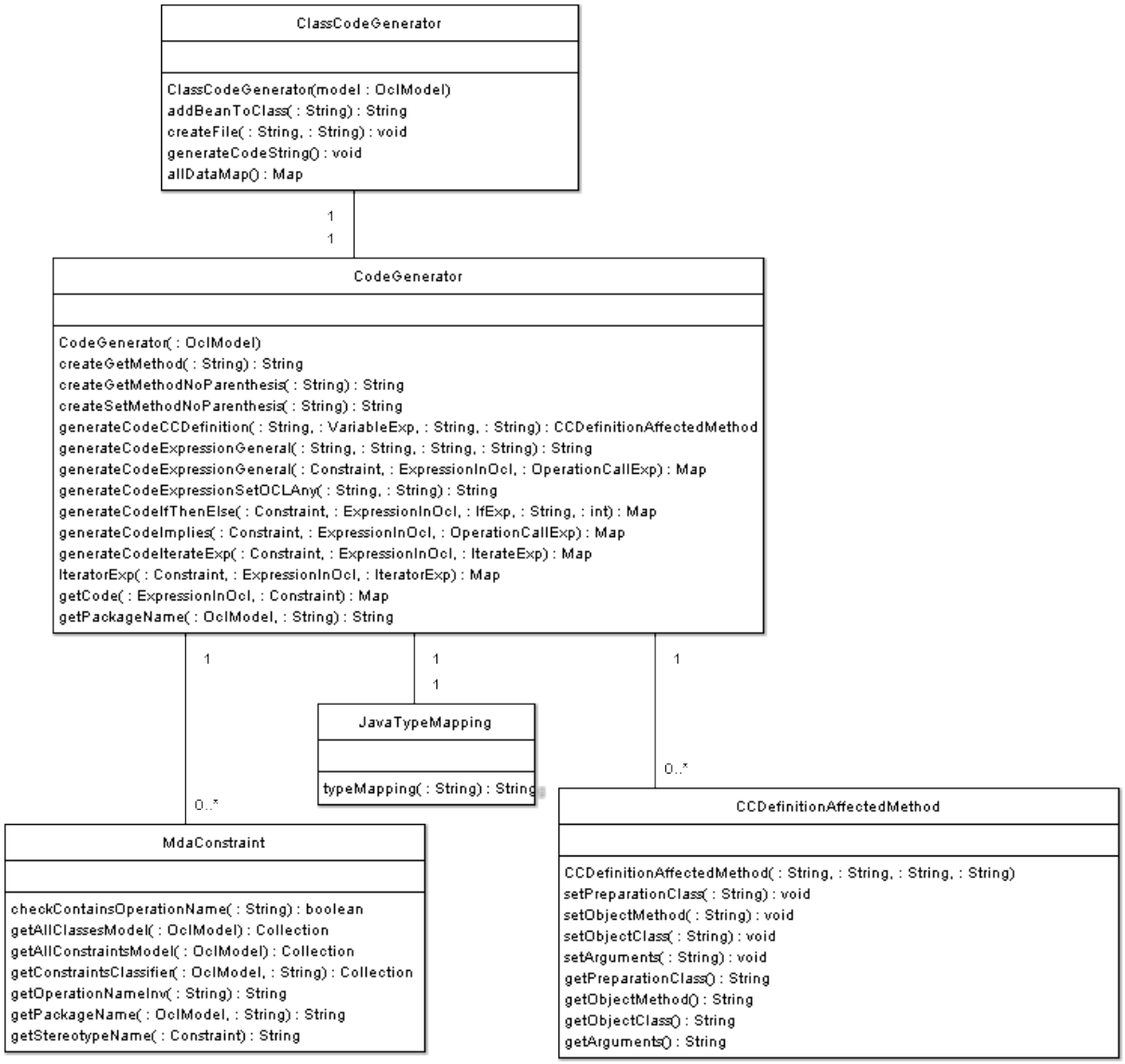

Figure 22: Class diagram: code generator

3.8.4 Java code generator

Three types of constraints must be handled during the generation process. As mentioned in the theory part, invariants, pre- and post-conditions are specified in the modelling tool in different ways.

The approach of this thesis is different from related work because of using explicit constraint classes instead of constraint code tangled with other code, e.g. the code for the business logic. The generated code must also conform to the generated code of the AndroMDA framework and to the constraint checking framework which is integrated in the context of the European research project DeDiSys. This restriction means that the Java code of the constraints uses always the same structure independent of the specified constraint type. To differentiate which type of constraints is defined, metadata are used to specify all properties of a constraint.

The code generation is managed as described in the former section by the class *ClassCodegenerator*. The functionality of this class is extracting the given OCL model prepared by the Dresden OCL Toolkit through OCL transformation into a model with tree structure. This class contains the header part of a Java class that is the same for every constraint class. After this step, the class *CodeGenerator* is responsible to generate other parts. The code here is different for every constraint and depends on the content of a constraint. Finally, the two parts are combined and the corresponding constraint class will be created. The code generator generates the Java classes for all defined constraints in one process and stores the properties of every constraint for further jobs.

Because the Dresden OCL Toolkit is a tool still in development, not all OCL mechanisms are supported. At the time of the implementation, the Dresden OCL Toolkit should support classifier contexts and all kinds of invariants. Pre- and post-conditions should also be supported but there is no warranty for completeness. For this reason, the main focus for this diploma thesis is on invariant constraints. Another reason for concentration on invariant constraints is determined by the DeDiSys project because pre- and post-conditions cannot be re-validated (in the reconciliation phase). Primitive operators (+, /, <, >, ...) are fully supported by the code generator. Other type of constraints with keywords such as “*select*”, “*forAll*” or “*iterate*” are also implemented. The feasibility of generating pre- and post-conditions is also explored by integrating demonstration constraints.

The generated Java classes have to conform to different restrictions. It must use the structure that is required by the constraint checking framework. On the other hand, it has to follow the structure of the used project. For example, a project is usually split up into several packages and a constraint could contain class objects of different packages. Therefore, it is necessary that the generated Java classes contain the correct reference path of all used class objects. The goal of this diploma thesis is to undertake a feasibility study on a code generator, but not run-time optimization. For that reason, to avoid situations where for one constraint the running time is well balanced and for another one is unacceptable, the target goal is find a compromise that may be can accept by every constraint. It is foreseeable that it is possible to generate well-balanced code for simple constraints. For complex constraints, the generated code is not well optimized with respect to performance compared to the handcrafted code by the programmer.

Listing 11 and Listing 12 shows an example for the constraint *FlightTestSelect*. The first one is the manually optimized version and the other one is generated by the code generator. The corresponding OCL statement for this constraint is illustrated in Listing 10.

```
context Flight inv flightTestSelect :
(self.allInstances->size > 0 ) implies (allInstances -> select ( f : Flight | f . fromAirport =
'Vienna' and f . toAirport = 'Zurich' ) ->notEmpty)
```

Listing 10: Constraint definition: *FlightTestSelect*

This invariant constraint does the check whether there is an object with the departure airport "*Vienna*" and the arrival airport "*Zurich*". The constraint returns "*true*" back if there is at least one object with this criterion.

The handcrafted code of *FlightTestSelect* uses a finder method to look for a matching flight in the database directly. On the other hand, for the generated constraint the select operation in this example requires to iterate over all objects in order to build up the resulting collection. It can be assumed that the database is much faster for matching purposes. The performance of the generated constraint would decrease with the number of added objects because it depends on the size of the collection and the iteration will be not aborted if a matching object is already found. The performance between validation on the object and database layer of this constraint is compared in the evaluation part, see Section 4.2.4.

```
public class FlightTestForAll extends AbstractConstraint {
   private static Log log = LogFactory.getLog(FlightTestSelect.class);
   public boolean validate(IConstraintValidationContext ctx) throws
         ConstraintUncheckableException {
      log.debug("Validating handcoded mandatory flight constraint.");
      try {
         InitialContext ic = new javax.naming.InitialContext();
         Object obj = ic.lookup("ejb/FlightBean");
         FlightLocalHome flightHome = (ejb.FlightLocalHome) obj;

         // BEGIN VALIDATION CODE
         if (flightHome.findAll().size() == 0 || flightHome.findByLocations("Vienna",
               "Zurich").size() > 0) {
            return true;
         }
         //END VALIDATION CODE

      } catch (Exception ex) {
         log.error("Error during constraint validation.", ex);
      }
      return false;
   }
```

Listing 11: *FlightTestSelect*: handwritten version

```
public class FlightTestForAll extends AbstractConstraint {
   private static Log log = LogFactory.getLog(FlightTestSelect.class);

   public boolean validate(IConstraintValidationContext ctx) throws
         ConstraintUncheckableException {
      log.debug("Validating handcoded mandatory flight constraint.");

      FlightBeanImpl flight = (FlightBeanImpl) ctx.getContextObject();
      InitialContext ic;
      FlightLocalHome flightHome =null;
      Collection foundElements = new Vector();

      try {
         ic = new javax.naming.InitialContext();
         Object obj = ic.lookup("ejb/FlightBean");
         flightHome = (ejb.FlightLocalHome) obj;
      } catch (javax.naming.NamingException ex) {
         log.debug("Problem create Initial Context",ex);
      }

      try {

         //BEGIN VALIDATION CODE
         Collection left = flightHome.findAll();
         Iterator iterElement1 = left.iterator();
         FlightLocal f = null;
         while ( iterElement1.hasNext() ) {
            f = (ejb.FlightLocal) iterElement1.next();
            if (f.getFromAirport().equals("Vienna") &&
                  f.getToAirport().equals("Zurich")) {
               foundElements.add(f);
            }
         }
         //END VALIDATION CODE

      } catch (javax.ejb.FinderException e) {
         log.debug("FinderException  ", e);
      }

      if (foundElements.isEmpty()) {
         return false;
      } else {
         return true;
      }

   }
}
```

Listing 12: *FlightTestSelect*: generated version

3.9 Generation of the metadata configuration document

This section is about the generation process to use the defined metadata within the JBoss application server in context with the constraint checking framework of the research project DeDiSys. The task is labelled as the step 6 in Figure 18 on page 30.

The metadata is defined in a UML-OCL model and this model is exported as an XMI document. This model file is the base document for all three different processing routes as illustrated in Figure 18.

The Java class *ConfigurationFileGenerator* is responsible for the generation of the document *ccDefinitions.xml*. Besides the constructor for loading the model and some helping methods, there are three essential methods for the generation process:
createCCDefinition(): This method is responsible for the organization of the generation process. The deployment descriptor *ccDefinitions.xml* contains a header part and one constraint entry for each defined constraint class.

getConsDefCode(): This method generates a record with the given data for each constraint that has an entry in the UML-OCL model. Default values are specified within this method if attributes are not set in the corresponding constraint metadata class of the UML-OCL model.

createFile(): This is the method to create the document *ccDefinitions.xml*. Two string parameters are necessary, one is the output data and the other is the destination path of the output file.

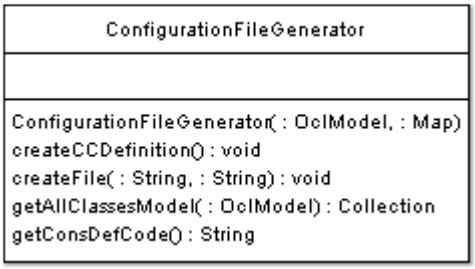

Figure 23: Class diagram of *ConfigurationFileGenerator*

All metadata are stored into the constraint deployment descriptor *ccDefinitions.xml*. For the case if some of the required properties were not defined by the developer in the constraint class diagram as described in Section 3.5, default attribute values would be used by the code generator.

The following default values are set:

type	HARD (for invariant constraints)
priority	CRITICAL
negotiation	IMMEDIATE
contextObject	Y: for invariant constraints N: for pre and postconditions
minSatisfactionDegree	SATISFIED
latestAcceptedSatisfied-Threat-Removes-IdenticalThreats	Y
intra-object	false

Table 2: Default values for *ccDefinitions.xml*

The type of constraint is extracted from the UML-OCL model directly. Three kinds of constraints are differenced: invariants, pre- and postconditions.
Preconditions are checked immediately after the invocation of a method. Postconditions must be checked upon completion of an affected method. Pre- and postconditions are clearly defined by OCL statements so that no extra information is necessary for the generation of the configuration document.

In the case of invariant constraints, no point in time is specified for validation. However, as soon as the context of an object that is constrained by the invariant is changed, a new validation must be done. Two types of invariant constraints can be distinguished, *hard* and *soft* constraints. Hard constraints have to be checked as soon as the object is updated, and must be satisfied at any time. Soft constraints can be checked at the end of the transaction that performs the update. The OCL statement does not support a distinction between soft and hard constraints. Consequently, it has to be defined by the developer manually. An own attribute for the constraint class in the class diagram is used to specify the kind of the invariant constraint. If this attribute is not specified, then the invariant constraint uses the default type "*HARD*".

Some attributes such as the *contextObject* cannot be specified by a developer. The context object depends on the context of the corresponding constraint whether a context object is required. For that reason, these kinds of metadata are determined directly by the code generator.

Listing 13 shows the structure of the constraint deployment descriptor *ccDefinitions.xml*. In this example, different metadata was set for the constraint *FlightTestSelect* in Figure 19 on page 32. This was one of the specified constraints used in the example application of the evaluation section. Attributes such as *priority* or *type* were set manually. Others such as *minSatisfactionDegree* or *negotiation* were default values.

3.10 Process integration

In the sections above, different software tools are mentioned and used by this project. Most of them can be executed with a command line interface, and some of them are implemented with a GUI application. It would take too many efforts if each tool must be executed manually. It is also too complex for a person who only wants to use the code generator and who has no interest to know how each tool works.

For reasons such as simplification and uniformity, it is necessary to have another tool for process management between the used tools of the code generation process. The choice falls to the Ant[16] build tool. It is a Java-based tool for automating software build process. It requires the Java platform and is very well suited for Java projects. Ant uses an XML based document to describe the build process and its dependencies. By default, the file named *build.xml* is used for this kind of configuration documents.

[16] Apache Ant: http://ant.apache.org/ (13.06.2010)

```
<!DOCTYPE ccDefinitions SYSTEM "cc_def_1.0.dtd">
<ccDefinitions>
<persister class="org.dedisys.ccmgmt.persistence.CCMgrDefaultThreatPersister" />
<defaultconstraintreconciliationhandler class
        ="ejb.FlightbookingConstraintReconciliationHandler" />
<minSystemSatisfactionDegree value="POSSIBLY_SATISFIED" />
<constraint name="FlightTestSelect" type="SOFT" priority="CRITICAL"
        negotiation="IMMEDIATE" contextObject="Y"
        minSatisfactionDegree="SATISFIED"
        latestAcceptedSatisfiedThreatRemovesIdenticalThreats="Y"
        intra-object="false">
   <class>Constraints.FlightTestSelect</class>
   <context-class>ejb.FlightBeanImpl</context-class>
   <expression></expression>
   <affected-methods>
    <affected-method>
     <context-preparation>
      <preparation-class>org.dedisys.ccmgmt.CalledObjectIsContextObject
              </preparation-class>
     </context-preparation>
     <objectMethod name="setToAirport">
      <objectClass>ejb.FlightBeanImpl</objectClass>
      <arguments>
       <argument>java.lang.String</argument>
      </arguments>
     </objectMethod>
    </affected-method>
    <affected-method>
     <context-preparation>
      <preparation-class>org.dedisys.ccmgmt.CalledObjectIsContextObject
              </preparation-class>
     </context-preparation>
     <objectMethod name="setFromAirport">
      <objectClass>ejb.FlightBeanImpl</objectClass>
      <arguments>
       <argument>java.lang.String</argument>
      </arguments>
     </objectMethod>
    </affected-method>
   </affected-methods>
  </constraint>
</ccDefinitions>
```

Listing 13: *ccDefinition.xml*: *FlightTestSelect* metadata

Beside the above-mentioned Ant build tool, there is another tool named XDoclet[17] that plays a role during the build process. AndroMDA generates the EJB sources but that is not fully conforming to the code that is required by the DeDiSys-Framework. XDoclet is a code generation engine that can be configured and started by the Ant build tool and its configuration document is based on XML. Its function is to transform the EJB code to a compatible format. After compiling the Java source, it will be packed together with the now conformed EJB code into a jar-archive. At the end, this archive can be used by the JBoss application server to run the application with its explicit constraints.

The Ant build process of this work has the following targets:

1. The first Ant target is used for generating explicit constraint classes for an application. In addition, the metadata are created and saved in *ccDefinition.xml*.
   All necessary dependencies such as the libraries of the Dresden OCL Toolkit are included in the same step.

2. The AndroMDA framework uses an own target for generating entity beans. Different AndroMDA packages are also appended as dependencies.

3. Compiling source code and transformation through XDoclet can be started in only one target process. If necessary, these two steps could also execute separately.

4. The last one during the build process has the aim to create a compressed EJB packet by using the files from the previous steps. Additionally, other files of the application could also be included with this EJB module into one compressed packet as a ready-to-run application for JBoss.

5. An additional target is included for deleting all generated files. If a new generation process is started, already existed old files will be not deleted, but overwritten by the new ones.

3.11 Optimization of the integrity constraint code generator

This section presents two additional works building upon the implemented code generator of this diploma thesis. The goal is to generate validation code in a more efficient way with respect to performance.

First, the work of Redlein [Redl10] uses the Java constraint code generator of this thesis as the base for his work. [Redl10] uses different techniques to improve the performance that are illustrated in Section 3.11.1. Because the generated Java validation code is too slow for complex statements as shown with the evaluation tests in chapter 4, [Redl10] implements a form of OCL to SQL transformation. For further details see Section 3.11.2.

Additionally, this work integrates an OCL to stored routines transformation building upon the OCL-to-SQL transformer of [Redl10]. This part is presented in Section 3.11.3.

The final code generator, described in Section 3.11.4, uses the best of the three types of the implemented transformers with respect to performance.

[17] XDoclet: http://xdoclet.sourceforge.net/ (14.06.2010)

3.11.1 Structure of the advanced constraint code generator

The work of Redlein [Redl10] has the goal to build a more efficient constraint code generator. For this purpose, it extends the Java code generator of this thesis with the following functionalities:

- ***Constraint normalizer***
  The task is to reduce the complexity of OCL constraints in order to generate faster constraint validation code. Therefore, the normalizer transforms the OCL constraints into semantically equivalent, but simpler counterparts that could be mapped to more efficient Java code. The implemented prototype makes use of an external library developed by [BC06].

- ***Constraint classifier***
  [Redl10] implements a classifier to decide whether an OCL constraint should be transformed to its corresponding Java code or SQL query. Three categories are introduced: *intra-instance*, *inter-instance* and *type-level*. Constraints of the category *intra-instance* are transformed to their corresponding Java constraint validation code, while the ones classified into the two other categories are handed over to the OCL-to-SQL transformer. For further information about these classifying rules, please refer to [Redl10].

- ***Constraint analyzer***
  As a major part of [Redl10], the OCL constraint analyzer has the purpose to analyze OCL expressions. First, it gathers information from the UML model to decide whether a constraint is a *size* or *value restricting constraint*. A side restricting constraint restricts the number of rows it returns, while a value restricting constraint is interested in states that the constraint needs to hold for all rows. For further implementing details about these classifying types, please refer to Section 3.11.2 and [Redl10].
  Second, the analyzer tries to split an OCL expression into several parts with the goal to transform each part independently. Currently, it is necessary to identify the top level operator because the implemented OCL-to-SQL generator can split OCL expressions only if the top level expression is *and*, *or* or *implies*. At the end, these parts are combined again.

3.11.2 Transformation of OCL to SQL

The work of Redlein [Redl10] implements an OCL-to-SQL transformer and it is able to transform simple OCL expressions to their corresponding SQL queries.

The generated Java constraint class uses the same structure as shown in Listing 9 in Section 3.8.1 on page 39. The difference is that this transformer uses an SQL query within the constraint validate method. This SQL query returns only a number of rows regarding to the performed query so that the final comparison is done by the Java code illustrated in Listing 14.

Checking the constraints directly on the database layer increases the run-time performance significantly for complex statements such as for the *AllInstances*-constraint as shown in Section 4.2.3. For further details about the reached performance improvements, please refer to [Redl10].

```
// BEGIN VALIDATION CODE
Connection con;
try {
        …….
        String sqlquery = “…..”;
        PreparedStatement stmt =  con.prepareStatement (sqlquery);
        ResultSet rs = stmt.executeQuery();
        // - > Java comparison code
        …….
} finally {
        con.close();
}
//END VALIDATION CODE
```

Listing 14: Constraint checking code of the SQL generator

The transformer of [Redl10] supports two types of OCL constraints:

- ***Size restricting OCL constraint***
  These are constraints that restrict the number of rows returned by a constraint. The top level operator of it is the OCL's *size* operator. It counts the found rows regarding to the performing query. Additionally, the returned value (sum of found rows) needs to be compared against the provided criterion of the OCL constraint. This part is done by the Java code.

- ***Value restricting OCL constraint***
  The query of a value restricting constraint returns all rows that do or not fulfil the query depends which criterion is required. Finally, the Java code checks if the number of found rows is greater than zero. For example, such an OCL constraint is one with the OCL's *forAll* operator.

Other kinds of OCL expressions are not supported by this transformer and will be generated by the Java transformer. For example, because the SQL transformer does not support *iterate*- and *if*-expressions, as a fallback a generation of OCL to pure Java is performed for these expressions.

3.11.3 Transformation of OCL to store routines

As an additional task, this work integrates an OCL to SQL store routines transformation building upon the results of the SQL transformer of [Redl10] to provide support for OCL's *iterate*- and *if*-expression on the database layer.

First, it transforms the existed generation engine of the SQL transformer to store routines so that both are able to support constraint generation at the same level. It also makes use of the transformation algorithm of [Redl10] that allows to split an OCL constraint into several parts. In contrast to the SQL transformer, it is now possible to transform parts of the OCL constraints to Java code, parts to SQL and parts to stored routines additionally.

Second, the transformer is enhanced with the support for *iterate*- and *if*-expressions. This is a major improvement with respect to performance because the *iterate*-implementation of the Java transformer is quite slow as shown in Section 4.2.5.
This transformer uses stored routines (procedures and functions) for constraint validation that are supported by MySQL[18]. In contrast to the SQL transformer, the task is to perform the whole constraint checking process on the database layer. Because the constraint validation code always return *true* or *false* depends on the result of the performed query, stored functions are used for convenient reasons. A stored function is a stored program that always returns a value, while stored procedures may return values via the *OUT* or *INOUT* variables. For more information about stored programs, please refer to [HF06].

An example for the OCL-to-stored function transformer is illustrated in Listing 15.

```
package ejb context Flight inv demoStoredFunction : (if (self.bookedSeatsEC <
self.maxSeatsEC) then self.bookedSeatsEC+1 else self.bookedSeatsEC endif) >= 4
endpackage
```

Listing 15: Constraint definition: *DemoStoredFunction*

An OCL constraint such as provided in Listing 15 results in the following stored function:

```
CREATE FUNCTION demostoredfunctionresult1 (param1 INTEGER, param2
        INTEGER, param3 VARCHAR(250))
RETURNS BOOLEAN
BEGIN
    DECLARE var_SelectQuery_Resultset INT;
    DECLARE returnResultVar INTEGER;
    BEGIN
        IF (param1 < param2) THEN
            SET returnResultVar = param1 + 1;
        ELSE
            SET returnResultVar = param1;
        END IF;
    END;

    SELECT 1 FROM FlightBean f WHERE NOT (returnResultVar >= 4) AND
            f.flightID = param3 LIMIT 1 into var_SelectQuery_Resultset;
    IF (var_SelectQuery_Resultset IS NOT NULL) THEN
        RETURN false;
    END IF;
    RETURN true;
END:
```

Listing 16: Stored function code of *DemoStoredFunction*

The stored function in Listing 16 is called by the Java code illustrated in Listing 17. The Java code also provides all required parameters to the stored function if necessary. For example, this demo constraint needs four parameters from the context object.

[18] MySQL: http://dev.mysql.com/ (19/01/2011)

```
// BEGIN VALIDATION CODE
Connection con;
try {
        CallableStatement stmt = con.prepareCall("{ ? = call
                demostoredfunctionresult1(?, ?, ?) }");
        stmt.registerOutParameter(1, Types.BOOLEAN);
        stmt.setInt(2, flight.getBookedSeatsEC());
        stmt.setInt(3, flight.getMaxSeatsEC());
        stmt.setString(4, flight.getFlightID());
        stmt.execute();
        boolean result = stmt.getBoolean(1);
} finally {
        con.close();
}
//END VALIDATION CODE
```

Listing 17: Java code of *DemoStoredFunction*

3.11.4 Combined code generator

Figure 24 illustrates how the final solution of the constraint code generator works:
First, the OCL expressions of a constraint are split into as many parts as possible. During the next step, each part is classified by the constraint classifier as described in Section 3.11.1. If this part belongs to simple constraints that can be performed in an efficient way on the object layer such as the *FlightNotEmpty*-constraint illustrated in Section 4.2.2, the Java transformer will transform it. All others are transformed to their SQL representations if possible. Every time when a problem occurs, the code generator tries to transform this part with the OCL-to-stored-routines transformer as the first alternative. If this also fails, the pure Java code transformer works as the last fallback. For example, the OCL's *isOCLkindof* operator is only supported by the Java transformer because SQL is not an object-oriented language. An exception is created when an expression could not be transformed by all three kinds of transformers. Finally, these parts are combined again.

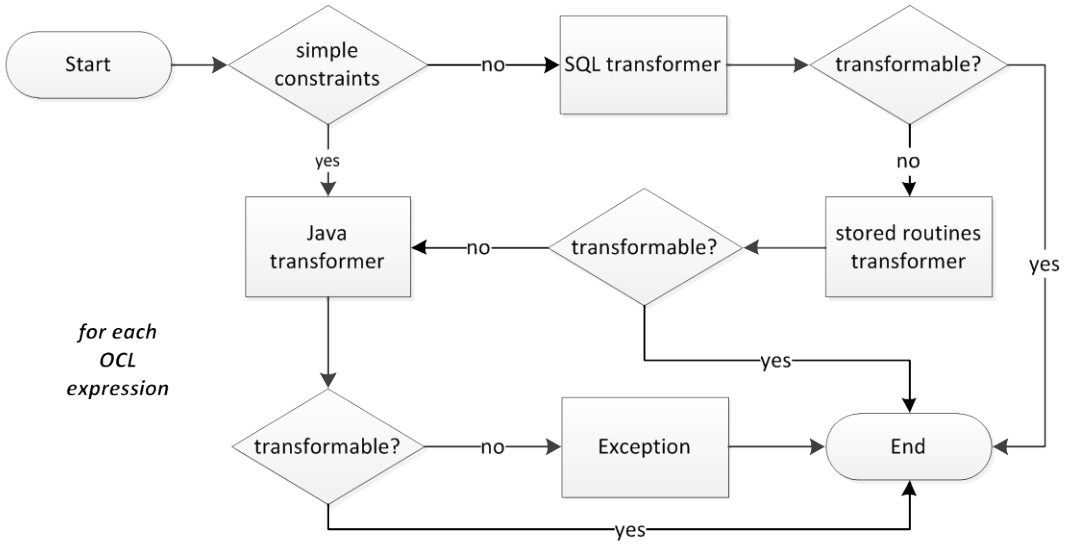

Figure 24: Flowchart of the combined code generator

3.12 Related technologies

This section gives an overview about technologies that are related to this work. First, different approaches for constraint validation are discussed. Finally, some tools with code generation support are presented.

With regard to Java, using explicit constraint classes is not the only way for implementation of constraint validation. There are two other kinds of code instrumentations, source code and byte code instrumentation. It depends on when the constraints will be instrumented. Source code instrumentation means that Java sources are instrumented with Java code before compilation. Byte code is injected into the existing byte code after the Java sources are already compiled. Compared to the approach of this work, this injection of Java code directly into the original source code has the effect that the code is highly tangled. Therefore, the code is hard to modify and maintain without the original source code and constraints metadata. The advantage is that the original source code is preserved and the instrumentation process is transparent to the developer.

For the source code instrumentation, two approaches are available: the first approach injects the constraint code directly at the place where the code should be performed, e.g. within the performed method. If several methods of possibly different objects are affected by a constraint, the same constraint code is injected at any place where constraint checking is required. Jass [BFMW01] and iContract [Kram98] are tools that use this approach. The second approach is to use wrapper methods. The original method is wrapped and the constraint validation code is contained in the wrapper method. A typical example for this approach is the Dresden OCL Toolkit [Wieb00]. In contrast to them, the approach with explicit constraint classes and constraints metadata has a much clearer code structure and is more flexible. However, the implementation of architecture for explicit integrity constraints and metadata is more complex and relatively costly.

Another approach for constraint code is for example compiler-based. A compiler-based approach is necessary if an extended grammar of the standard Java language is used. It performs a transaction from source code to byte code and integrates the constraint validation code during this transaction. As JML shows, custom compilers are also used without extensions to the standard Java language [LBR99].

A small overview and evaluation with some of the approaches and tools below can be found in [FOG07]. The following constraint generation tools are not all checked or tested with the scope of this thesis:

- ***Dresden OCL Toolkit***

  The Dresden University of Technology provides an OCL parser, written in Java that does full type checking of OCL constraints and includes code generation from OCL-to-Java and SQL. For further details see Section 2.5, respectively [Wieb00].

  In contrast to the goal of this work, Dresden OCL Toolkit enhances the existing code with the functionality of constraint validation. It uses the wrapper-based source code instrumentation. The OCL parser is the core of the Dresden OCL Toolkit. The code generation engine of this work is also based on this parser. This means that the new code generator could only support the generation of explicit integrity constraints generation to the level as it is supported by the parser of the Dresden OCL Toolkit.

- ***OCL4Java***

OCL4Java is based on the Dresden OCL Toolkit and is a Java assertion tool [O4J08]. Compared to the Dresden OCL Toolkit, OCL4Java enhances it by parsing Java5 annotations (using JDT) and providing the content, the OCL statements, to the Dresden OCL Toolkit for parsing. OCL4Java only allows code insertions at the begin and end of methods. Other characteristics that are mentioned for the Dresden OCL Toolkit are also valid for OCL4Java.

- ***AndroMDA***

AndroMDA is a code generation tool that can be used in context with all programming languages. A small description for this tool can be found in Section 2.7. AndroMDA provides different cartridges for different programming languages. The EJB cartridge is used by this work to generate entity and session beans.

AndroMDA provides translation libraries to support OCL, but no transaction from OCL to Java. Currently, two query translation libraries are supported: EJB-QL and Hibernate-QL. A translation library could be used to translate an OCL body expression into the corresponding query language. Table 3 shows a example with a transaction from OCL to Hibernate-QL and EJB-QL.

Original: OCL query
Context org::andromda::contracts::Project::findByProjectTypeStatusAfterWent-CurrentDate (type:String, status:String, wentCurrentDate:Date):Collection (Project) body : allInstances() -> select (project | project.type = type and project.status = status and project.wentCurrentDate >= wentCurrentDate)
Translation: Hibernate-QL
from org.andromda.contracts.Project as project where project.type = :type and project.status = :status and project.wentCurrentDate >= :wentCurrentDate
Translation: EJB-QL
SELECT DISTINCT OBJECT(project) FROM Project project WHERE project.type = ?1 AND project.status = ?2 AND project.wentCurrentDate >= ?3

Table 3: Example for the AndroMDA translation library [AMDA08]

- ***iContract***

iContract uses OCL in custom tags of Java comments. Three kinds of tags are introduced to specify invariants, pre- and postconditions. This tool injects code for validation directly at the place where the validation should be performed. Despite these extensions, the Java classes remain fully compliant with the “standard” Java compiler [Kram98].

- ***JML***

The Java Modeling Language (JML) is a behavioural interface specification language. The constraints are manually specified in Java comments or in separate file and they are not compatible with OCL constraints [LBR99]. It is a compiler-based approach for constraint validation. There are a variety of tools based on a custom compiler approach, which support JML annotations. The Common (formerly ISU) UML tools provide a type checker, a run-time assertion checking compiler, a tool for running applications compiled with the JML compiler, a documentation generation tool which produces Javadoc

documentation augmented with extra information from JML annotations, and tools for use with JUnit.

- ***JASS***

Jass (**J**ava with **ass**ertions) does not use the OCL syntax, but it relies on its own syntax that is similar to the Java syntax. All constraints are defined within Java comments and have an introducing keyword that defines the kind of the assertion. A class annotated with Jass assertions must be saved in a '*.jass*' file (or with extension '*.java*' if a destination directory is specified). The Jass precompiler takes this Jass file and produces a valid Java class source file. The last step is to compile this new produced file with a Java compiler and then it can be used in a Java running environment. Jass allows inserting code checking for constraints at any part of the class [BFMW01].

4 Evaluation and Results

The DeDiSys project consists of a number of work packages. One of them is the constraint checking framework and it will be used for the evaluation of this thesis. It is part of the kernel of the DeDiSys project and has been well-tested and evaluated for the delivery for the project inspectors of the European Union. The corresponding report can be found at [Kuen07].

Based on this constraint checking framework, a proof-of-concept shall be delivered to receive the goal of this thesis. Several measurements are performed to provide a relation between theory and practice. These measurements provide a basis for further discussions on the topic.

First, this chapter presents the test environment that includes the test application and a summary of the development environment. Next, different scenarios that are performed during the evaluation are illustrated in detail, ranging from the test case without constraints to the one with all constraints in one cycle. An interpretation about the evaluation results and an overview of potential enhancements are also given. Finally, some examples of real-life applications are illustrated in which the generated integrity constraints of this thesis could be used.

4.1 Test environment

The test environment and the test application for this evaluation are described in the following sections.

4.1.1 System and Evaluation environment

An overview of the development and evaluation environment that has been used for this diploma thesis is illustrated in the following. In Appendix A1, there is a list with weblinks to the used technologies.

- ***Hardware***

The used hardware was a standard personal computer (PC). The amount of hard disk space used for this project was negligible (Only some hundred megabytes in relation of some hundred gigabytes of a modern hard disk.). Detailed facts of the test computer were listed below that could be important to achieve similar evaluation data:

Processor:	AMD Athlon™ 64 S-AM2 Orleans 3200+ (2.0 GHz)
RAM:	3 GB DDR2 PC-667 CL5
Graphic card:	NVIDIA GeForce 6150 (integrated)
Mainboard:	ASUS M2NPV-VM

Table 4: Main facts of the used test computer

- ***Operating system***

The used operating system during the development and testing was Microsoft Windows XP SP3. It was not tested on other operating systems, but theoretically, it would also work on Linux systems or their derivatives. The only needed fundamental component for development and running was Java. The used version was Java Software Development Kit 6.0.

- ***Implementation environment***

For coding NetBeans 5.5 was used. The Apache Ant Tool, a Java build tool, was responsible for the compilation of the project. The DeDiSys Project was administrated with subversion[19]. For ease of use, the client version TortoiseSVN was used instead of the standard console version.

- ***UML tool***

ArgoUML v0.24 was used for modelling. The reason why ArgoUML was chosen is explained in Section 3.4.

- ***Dresden OCL Toolkit***

This tool with the version 2.0 was the basis for the code generator to generate explicit constraint classes.

- ***AndroMDA***

AndroMDA was responsible to generate to entity and session beans. Version 3.1 was used for this diploma thesis.

- ***DeDiSys framework***

The DeDiSys framework consists of a number of internal and external software packages. In the following, it is a listing of packages that were relevant during this diploma thesis:

  - ***JBoss 4.0.4***
    JBoss is a Java EE-based application server. This is required as the basis for the DeDiSys middleware.

  - ***Constraint checking framework***
    This framework is implemented by members of the DeDiSys project and is responsible for constraint checking.

  - ***MySQL***
    The MySQL server version 5.1.53 was required for database management. MySQL Query Browser and MySQL Administrator were used as GUI tools for administration.

  - ***DeDiSys middleware***
    This is the intellectual property of the DeDiSys project. It consists of a number of modules for different programming languages described in the fundamental section.

  - ***DeDiSys applications***
    A number of applications have been developed during the DeDiSys project. For this diploma thesis, only the applications that are developed on top of the EJB DeDiSys middleware are relevant. The most important one is the flight booking application that fulfils all requirements for the evaluation phase.

[19] subversion: http://subversion.tigris.org/ (14.06.2010)

4.1.2 Test-Application

For verification on the DeDiSys framework, the flight booking application was chosen as the test application. This was a simple application that fulfilled all required qualifications of this thesis. This application had several features such as booking a flight, cancelling a flight and adding passengers. For the evaluation only the function “booking a flight” was needed and all tests were performed with the healthy mode. This means the system is in a fully consistent state. The difference between the system states is described in Section 2.1.2. For further details about the model of the flight booking application see the class diagram in Figure 26.

The class diagram of the UML-OCL model had five entity classes with attributes and methods. Other classes were only helper-classes for the generation process. For the sake of clarity, the parameters of methods were set invisible. The associations between these classes were not set to show that this is not essential for the generation process. Four different kinds of OCL constraints with metadata were specified, but they are not integrated in Figure 26 and will be provided in the respective evaluation sections. This model was used by AndroMDA to generate the EJB-files and by the code generator for generating constraint and configuration files. The model did not contain every code that was needed to run the application. It was not the sense of this work to re-implement an application that already existed. For this reason, the additional required files, mainly containing the application logic, were taken from the original flight booking application of the DeDiSys project.

There were two client versions for the flight booking application, a web client and a standalone Java client. At first, the web client was used to test the principle correctness of the generated constraints. However, for the evaluation part the web client was no more suitable because of the huge number of flight objects that had to be created. For this special case, a modified Java client of the flight booking application was used. It was enhanced with a switch to enable or disable the two following transaction modes (see screenshot in Figure 25):

- insert one object per transaction
- insert all objects in one transaction

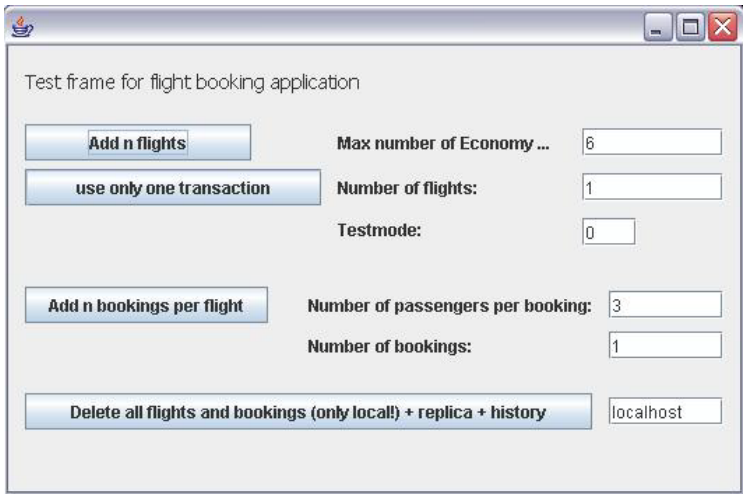

Figure 25: Test UI of the flight booking application

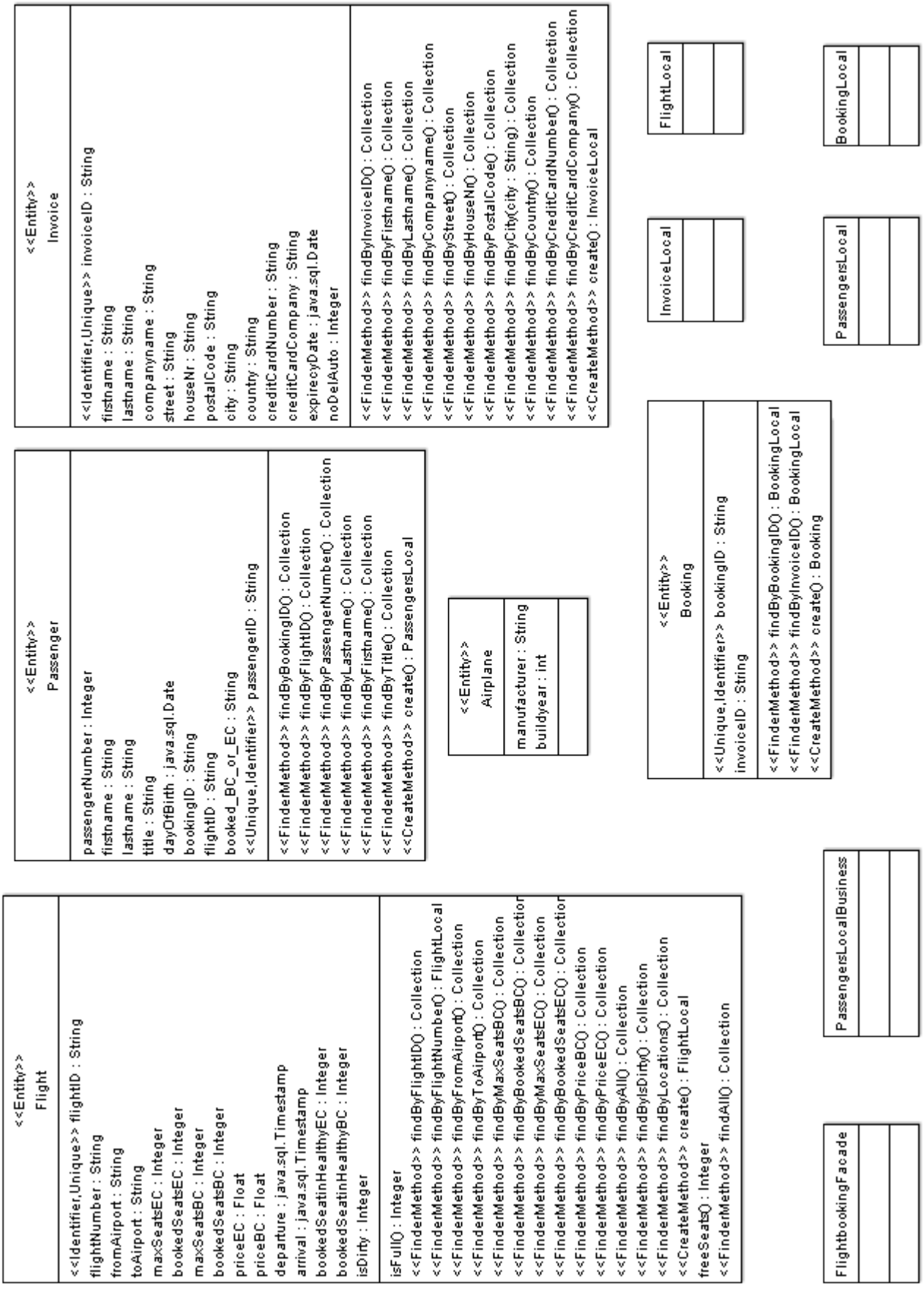

Figure 26: Class diagram of the test application

4.2 Test scenarios

Different performed test scenarios give an overview about the performance of generated constraints. They were compared to handcrafted constraints. The tests and interpretations described in the following sections shall find answers about advantages and disadvantages of the implemented code generator to establish a relation to a production system.

The generated constraints by the OCL-to-Java, -SQL and -stored routines transformers were evaluated under certain scenarios to achieve comparable results, ranging from simple attribute constraints to more complex constraints that gave room to specific optimizations. As the baseline, the evaluation started from an application without constraints.

First, all measurements were made after the initiation of the JBoss application server as the first input. The application server was restarted after each measurement. Between the shut-down and restart phase, the MySQL database of the flight booking application was dropped and re-created as a completely new database. To avoid influences from outside, during the evaluation phase all other not needed applications were closed. It was necessary to repeat the tests to reduce measurement uncertainty. The measurement uncertainty could be influenced through services of the operating system that ran in background during the test concurrently. There would be always a difference, but so small that could be negligible. At least, two same evaluations were performed for each of the six test cycles. If the difference between these two was too large, then more tests were performed until the difference was reasonably small, i. e. explainable by measurement uncertainties. The average of the two best results would be used for the test results. Finally, the switching between different parameters was investigated. The test cases and switching parameters are summarized in the following.

Adding an object or an operation in this chapter always means adding a flight object.

Test cases:

1. ***no constraints***
   This was the simplest scenario that ran without any constraints. For further details see Section 4.2.1.

2. ***FlightNotEmpty***
   This constraint checked whether an object was a null object. The tests were performed with constraints generated by all three transformers and are described in Section 4.2.2.

3. ***AllInstances (unique criterion)***
   The goal of this constraint was to validate the unique criterion of an attribute. Section 4.2.3 shows the differences between the three different transformers with respect to the performance.

4. ***Select***
   This constraint validated whether there existed an object with specific values. The tests were performed with generated constraints of all three implemented transformers. Details are presented in Section 4.2.4.

5. ***Iterate***
   This constraint, described in Section 4.2.5, simulated the functionally of an iteration with OCL through generated and handwritten constraints.

6. ***Scalability testing***
   This test case tested the time required for validation of a single constraint and is described in Section 4.2.6.

Switching parameters for the test evaluation:

- ***Generated and handwritten***

Generated constraints had reached the performance limit many times during the evaluation phase. For this reason, manually optimized constraints were used to cope with this problem. It should illustrate that there was still room for improvement. The handwritten constraints used the same structure as the generated ones. It was only necessary to change the query statements to achieve performance enhancement.

- ***Database vs. object validation***

Both transformers, the SQL transformer and the stored routines transformer, check the constraints on the database layer. Except for the *Iterate*-constraint that is not supported by the SQL transformer, the performance was always measured with all three transformers. This should show that performance improvement could also be reached on other ways instead of editing the constraints manually.

Single vs. multiple transactions

The tests were combined with the two transaction modes in each case as far as possible. It meant whether all flights were added in a single transaction or a separate transaction was used for each added flight.
This switching parameter is not invented by this thesis. It is already a function of the constraint checking framework of the DeDiSys project. This is used to demonstrate the functionally of the test cases or to show why metadata is useful.

- ***Hard vs. soft constraints***

All kinds of constraints were evaluated with two kinds of specification, hard and soft. Hard meant a hard constraint and the constraint evaluation had to be started after the insertion of each object immediately. Soft meant a soft constraint and the constraint evaluation ran only once at the end of the transaction.
This is also an additional switching parameter that already exists. It should illustrate the importance of metadata and therefore why metadata should be taken into consideration by MDA approaches.

Legend of the figures in the following section:

- soft: soft constraint
- hard: hard constraint
- [x]: one transaction per created flight
- [1]: one transaction for all created flights

4.2.1 Test scenario: No constraints

For the first test case, the application ran without constraints. Tests with different number of inserted objects were performed, in each case once in the two transaction modes.

The goal was to get a reference value for other tests. The average time for every inserted object should not make a difference if one object or some thousands are inserted concurrently.

The evaluation of this test case ran without problems as expected. As the result, we can see that the creation of one object took tenths of milliseconds shown in Table 5. Figure 27 shows that the duration was increasing with the number of objects continuously. The single transaction mode was much faster than the other one. As visible in Table 5, for example, the performance about 100 flight objects required 30 ms per flight addition if all additions were performed within a single transaction and 91 ms per operation if a separate transaction was used per added flight. This showed already that transaction handling introduced quite an overhead to the application as also visible in all other test scenarios. Another important aspect was the initialization time. Table 5 shows that the test with about 100 objects needed much more time to add an object than the others. The initialization phase gave only here a big influence on the overall duration. Other test cases were much more computing-intensive so the average effect on each object was negligible.

This test result already showed that the transaction mode could influence the evaluation time significantly. The single transaction mode should be preferred if possible. The required parameters can be set in an easy way by changing the metadata specification.

Number of created flight objects	**[x]**	**[1]**
	[average time for one object in milliseconds]	
100	91	30
200	84	25
1.000	77	19
2.000	74	18
10.000	67	15

Table 5: Test result for "*no constraints*"

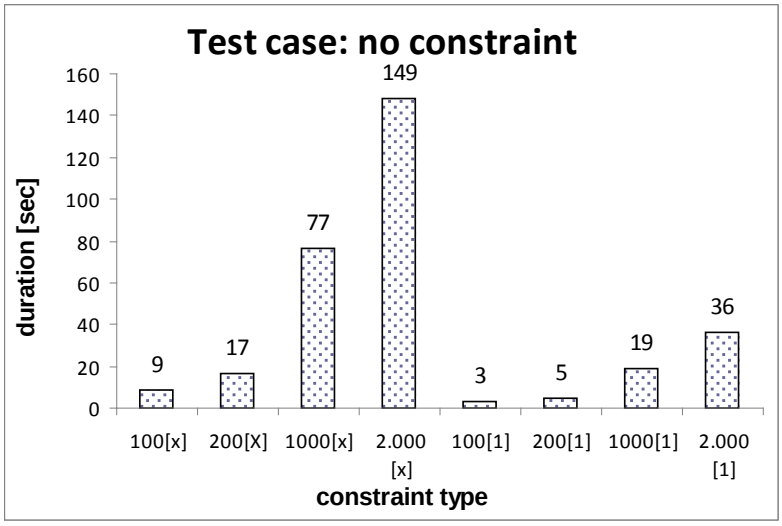

Figure 27: Test case: "*no constraints*"

4.2.2 Test scenario: FlightNotEmpty

This kind of constraints validates if an object is a null object or not. There are many other simple verifications like comparisons if an object is greater or smaller than another one. The OCL statement in Listing 18 means that the attributes flight number ("*flightNumber*"), departure airport ("*fromAirport*"), destination airport ("*toAirport*"), arrival ("*arrival*") and departure ("*departure*") time cannot be a null object.

```
context Flight inv flightNotEmpty : flightNumber -> notEmpty and fromAirport ->
notEmpty and toAirport -> notEmpty and arrival -> notEmpty and departure -> notEmpty
```

Listing 18: Constraint definition: *FlightNotEmpty*

For this test case, no handwritten constraints were used because the generated constraint is identical to a handcrafted one. Everything that was required could be specified directly within the UML-OCL model. Specifying this constraint as a hard constraint did not work as whenever the first attribute was set. The reason was that the constraint would be checked, detect that all other attributes were still *null* and therefore regard this as a constraint violation. Therefore, the *FlightNotEmpty*-constraint is a typical soft constraint.

Because this test case did only simple attribute verifications, so it should run fast. The average validation time for each object should be predictable and constant independent of the number of objects.

For this case, several tests with different numbers of flight objects were performed. Table 6 shows the average insertion time per flight addition. The test results showed that the Java version was faster than the SQL and stored routines versions. The average time fell with the increasing number of inserting flight objects due to the initialization time.

For such simple constraints, constraint validations should be always performed with the Java version to reach the best performance.

Number of created flight objects	**soft[x]**			**soft[1]**		
	[average time for one object in milliseconds]					
	Java	SQL	Stored routines	Java	SQL	Stored routines
200	95	131	137	30	32	35
500	86	124	131	25	28	29
2000	83	126	127	22	25	25

Table 6: Test result for *FlightNotEmpty*

Figure 28 shows the duration for about 200 added flight objects. The results for such simple expression were as well as the comparison to the baseline without constraints. The performance overhead of this simple constraint was rather negligible.

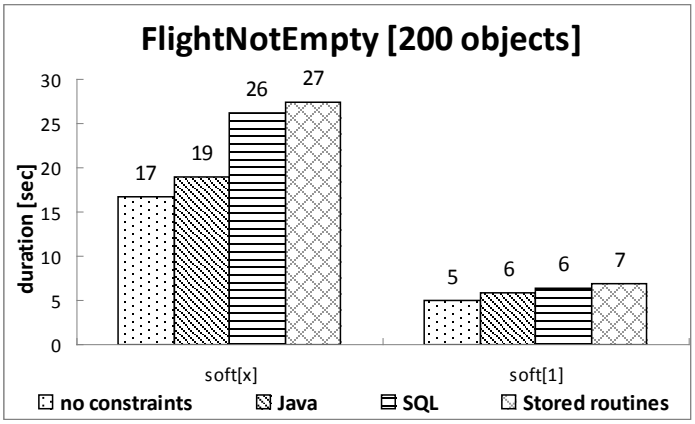

Figure 28: Test case: *FlightNotEmpty*

4.2.3 Test scenario: AllInstances

The functionality of this constraint illustrated in Listing 19 with the OCL's *forAll* operator was to verify all available flight instances whether the attribute "*flightNumber*" was unique or not. First, it checked whether the two flight instances were the same. If no, a second validation started to evaluate whether their attribute "*flightNumber*" had the same value to guarantee the unique criterion.

```
context Flight inv flightTestForAll: Flight . allInstances -> forAll ( f1, f2 | f1  <>  f2
implies f1 . flightNumber <> f2 .  flightNumber)
```

Listing 19: Constraint definition : *AllInstances*

This constraint could be normalised by the implemented constraint normalizer of [Redl10] as shown in Listing 20. The normalized version assumed that the flight numbers in the database were already unique and therefore validated only the current flight object with the unique criterion.

```
package ejb context Flight inv flightTestForAll : Flight::allInstances()->forAll(f2: Flight |
(self.flightID = f2.flightID) or (not(self.flightNumber = f2.flightNumber))) endpackage
```

Listing 20: Normalized constraint definition: *AllInstances*

This constraint was evaluated with all three kinds of transformers as usual, and because this constraint had to be validated with the context object, the hard configuration could not be performed with the same reason as described in Section 4.2.2 for the *FlightNotEmpty*-constraint.

This kind of constraints was not only complex to generate the corresponding Java code, but it was also a challenge for the computing power. The performance should depend on the number of the required transactions during an evaluation. Because of the used complex data types, the more objects were added, the more transactions for each flight addition should be needed.

The test results in Figure 29 showed that the "*soft[1]*" configuration performed similar for all constraints with about 6 seconds because the constraint validation for all flight objects ran only once at the end of a transaction. In contrast, for the "*soft[x]*" configuration the constraint

code of the database-based transformers was much better than the three Java versions with respect to performance. The difference between the validation on the object and database layer was that the Java version operated on the entity beans directly while the SQL and stored routines transformers operated with an SQL select-statement.

The monitoring of this evaluation showed that the duration for one object was monotonically increasing with the generated Java code (labelled with "*Java*" in Figure 29) as expected. At the beginning, only some milliseconds were required to add one flight object, while with the increasing number of added objects it raised to some minutes. The reason was that the more objects existed the more complex statements had to be executed for each validation. This led to the assumption that this kind of constraints could not be used on a production system due to the *O(n2)* comparisons per constraint validation in a simple, non-optimized implementation. The handwritten constraint (labelled with "*Java –handcoded*") was significantly faster and took only 105 seconds for the validation of 200 objects compared to the 1714 seconds of the generated Java constraint because the flight numbers were extracted only once into a sorted array and therefore avoided a full $O(n^2)$ comparison. The check was whether subsequent elements in the sorted array were equal so that there were no redundant comparisons. Although the handwritten version validated all objects with each other, its test results were similar to the simplified normalized counterpart (labelled as "*Java – normalized*" in Figure 29) that took 109 seconds and ensured only the unique criterion for the current flight object. As mentioned before, it assumed that the flight numbers in the database were already unique.

The SQL and stored routines transformers had enormous speed advantage compared to the Java counterparts. Because the two SQL-based transformers have the focus on speed optimization, they use only normalized versions for constraint generation. The evaluation time in Figure 29 with 27-31s for 200 added flight objects was only a bit more compared to the baseline configuration without constraints in Figure 27. Therefore, validation on the database layer should be always preferred to the one on the object layer for this kind of constraints.

Using of metadata allows a flexible change of configurations. For this case, it allowed in an easy way to change between the different transformers. It was only necessary to change the explicit constraint class, respectively the query statement of the constraint class. For using the stored routines version, the used stored functions had to be created in the database manually.

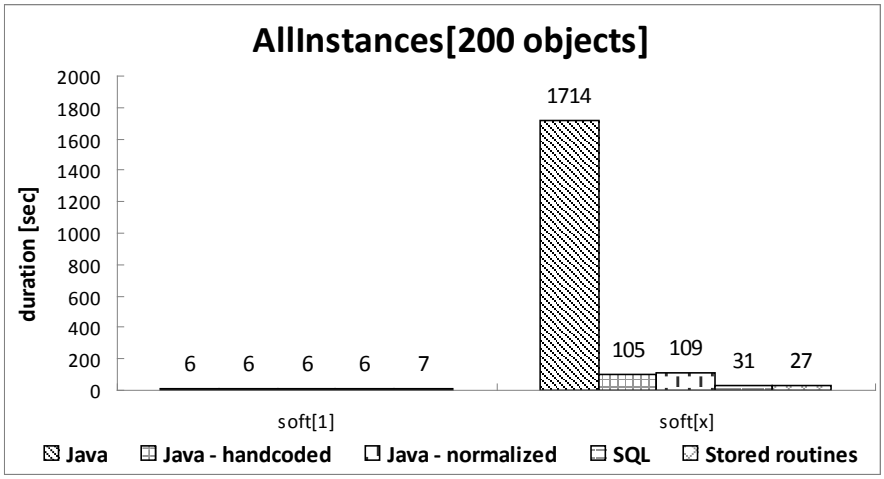

Figure 29: Test case: *AllInstances*

4.2.4 Test scenario: Select

The constraint in Listing 21 did the following task: First, a check evaluated if there existed a flight object. If yes, all flight objects were selected if their department airport was “Vienna” and the arrival airport was “Zurich”. The constraint returned back “*true*” if there existed at least one object with this selection.

The constraint would always return “*false*” when no object was available. Therefore, no transaction could be performed. For that reason, to finish the whole constraint evaluation it was necessary to have at least one object that met the requirements of this constraint. Therefore, for the test environment one extra flight object was added with the required test data with “Vienna” as the departure airport and “Zurich” as the arrival airport. It was necessary that all objects were added to achieve comparable results.

```
context Flight inv flightTestSelect :  (self . allInstances->size > 0 ) implies (allInstances ->
select ( f : Flight | f . fromAirport = 'Vienna' and f . toAirport = ‘Zurich’ ) ->notEmpty)
```

Listing 21: Constraint definition: *Select*

The used normalized constraint during the evaluation is listed in Figure 23. Because the differences were only using different simple operations that had no influence on performance, only the normalized constraint was evaluated.

```
package ejb context Flight inv flightTestSelect : ((Flight::allInstances()->size()) <= 0) or
((Flight::allInstances()->select (f: Flight |  (f.fromAirport = 'Vienna') and (f.toAirport =
'Zurich'))->size()) > 0) endpackage
```

Listing 22: Normalized constraint definition: *Select*

As the *AllInstances*-constraint, this constraint contained also complex data structures. The SQL and stored routines versions should be faster because SQL’s select-statements were much more effective than iterations in Java. The generated Java version was characterised through inflexibility because the code generator had to generate code that should fit to all kinds of constraints with the OCL’s *select* keyword.

The test results in Figure 30 showed that the difference between the generated Java version and the database-based ones was large in the case with “*hard[x]*” and “*soft[x]*”. The *select* operation of the Java constraint required to iterate over all flight objects once in order to build up the resulting collection for each transaction, while the SQL select-statement could be performed in an efficient way directly in the database. For example, for the “*hard[x]*” configuration with about 200 flight objects the performance was improved from 108 ms to about 21 ms. Because the handwritten Java constraint was optimised by using a finder method directly in the database, it could enhance the performance several times compared to the generated one. However, it was a little bit slower than the ones of the SQL/stored routines transformer due to a little transaction overhead on the object layer.

The “*hard[1]*” and “*soft[1]*” configuration performed similar for the constraints of all three transformers. The reason was that the constraint validation for all flight objects ran only once at the end of a transaction.

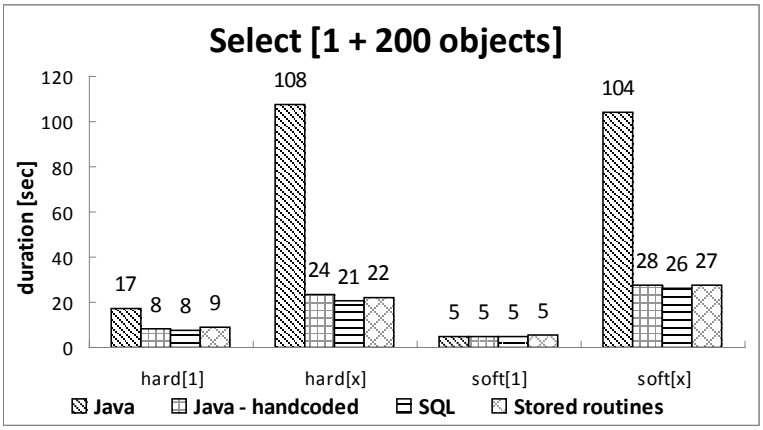

Figure 30: Test case: *Select*

4.2.5 Test scenario: Iterate

The *Iterate*-constraint did the following task: First, a check evaluated if there existed at least four flight objects or not. If yes, an iteration over all available flights started to validate if the attribute "*maxSeatsEC*" was greater than the attribute "*bookedSeatsEC*". For each object that was satisfied with this validation, the with "*0*" initialized sum (*acc*) was incremented by one. The constraint returned back "*true*" if the sum was greater than 4.
The check at the beginning was necessary for this test case. Otherwise, the constraint would always return "*false*" if there are less than four objects in the database, because the attribute *acc* would have a number smaller than 4. Therefore, at least four flight instances had to be available for a successful evaluation.

The OCL notation of this constraint is listed in Listing 23. There is no normalized version because currently this kind of constraints is not supported by the constraint normalizer.

```
context Flight inv flightTestIterate : self.allInstances -> size >= 4 implies self.allInstances
-> iterate ( v : Flight ; acc : Integer = 0 | if ( v.bookedSeatsEC < v.maxSeatsEC ) then acc
+ 1 else acc endif ) >= 4
```

Listing 23: Constraint definition: *Iterate*

This example shows the advantage of constraints that are manually written by a programmer. The optimization for the handwritten constraint was that if the sum had reached "4", the constraint would return "*true*" immediately. If the generated version was used, it would stop not until it had iterated over the last available flight instance. This increased the execution time according to the number of added objects. This constraint was evaluated only with the Java and stored routines transformers because the SQL transformer does not support the OCL's *iterate* expression.

The test results in Table 7 and in Figure 31 showed that on the one hand there was no big difference for the "*hard[1]*" and "*soft[1]*" configuration because the validation ran only once for all flight objects at the end of a transaction. On the other hand, "*hard[x]*" and "*soft[x]*" showed the efficiency of handwritten constraints. The performance of optimized constraints was significant better already with 200 objects for the Java version. The duration was only the fifth part, and the more additionally were performed the more the performance advantage was visible. For the 1000 objects test case, the performance was ten times better compared to the generated constraint. By contrast, the generated version required more and more validation

time after the insertion of a new object. As mentioned above, the iteration over all objects costs lots of time and should be avoided as much as possible. The test result also showed that the stored routines representations were always faster than the generated Java version and similar to the handwritten Java one. The handwritten counterpart of the stored routines constraint used an SQL select-operation instead of iteration over all objects. The test results showed that no optimization was required for the generated constraints of the OCL-to-stored-routines transformer because the reachable performance improvement was only tenth of milliseconds. The difference is presented in detail in Figure 35.

	hard[x]		**soft[x]**	
Number of created flight objects	*200*	*1000*	*200*	*1000*
	[average time for one object in ms]			
Java – generated	0,577	2,522	0,533	2,282
Java – handcoded	0,140	0,232	0,176	0,261
Stored routines – generated	0,111	0,104	0,145	0,137
Stored routines – handcoded	0,102	0,095	0,140	0,127

Table 7: Test result for *Iterate*

The test result showed that optimization was helpful when collections were involved. In some cases, the constraint had to be validated with loops over all elements of a collection. Through manual optimization, loops could be avoided or stopped earlier and this would save lots of execution time. Figure 31 shows that the optimization would only take effect when the number of added objects is high, otherwise the difference was negligible.

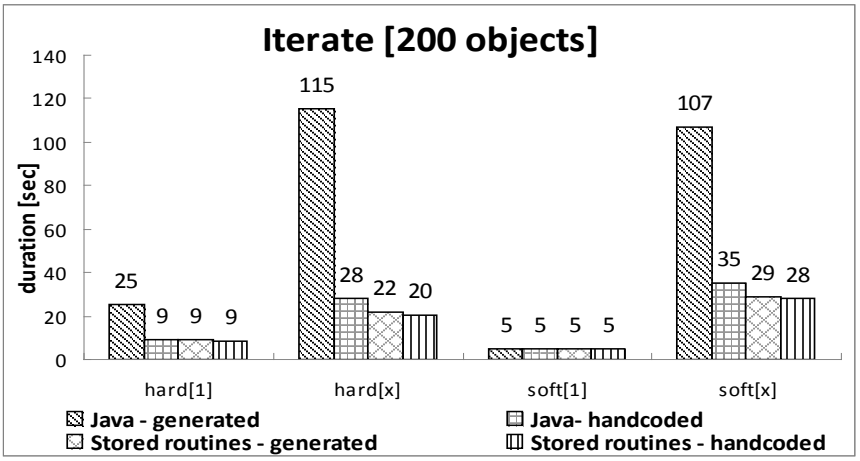

Figure 31: Test case: *Iterate*

4.2.6 Scalability testing

This scenario tested the time required for validation of a single constraint. It was performed for the *AllInstances*-, *Iterate*- and *Select*-constraint. The goal was to show the limitations of individual constraint implementations during run-time in a production system.

The test results in Figure 32 showed that the generated Java *AllInstances*-constraint (labelled as “Java - generated”) already needed 2 seconds for the validation of 50 objects and 39 seconds for 250 objects. Therefore, the validation on the object layer would not be usable in practice for this constraint. The handcoded and normalized counterparts performed much

better due to the *O(n)* comparison instead of a full *O(n²)* comparison of the generated Java version. They required only 100-500 ms for this amount of objects, 4 seconds for 2,000 objects or 54-59 seconds for 32,000 objects. The reason why the generated Java version was so slow was that almost all the time was wasted for bringing the data from the database to the object layer. For more implementing details about these three types of constraints, please refer to Section 4.2.3.
This also explained why the two database-based transformers (SQL and stored routines) always required less than 100 ms independent from the number of objects.

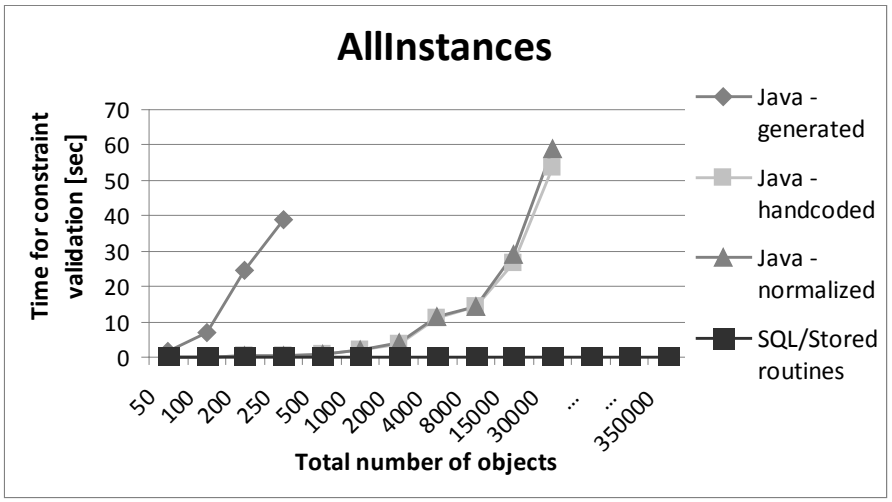

Figure 32: Scalability of constraint validation for the *AllInstances*-constraint

Figure 33 shows the limitations of the scalability testing for the *Select*-constraint. The performance for the SQL/stored routines transformer is illustrated with the same curve because the same SQL's select-statement was performed. The generated Java constraints required about 5 seconds for 2,000 flights objects or 52 seconds for the validation of 15,000 objects that is not acceptable in practise. As mentioned in Section 4.2.4, the handcrafted Java version ("Java – handcoded") had a small overhead compared to the ones of the SQL/stored routines transformer. For example, it took already 6 seconds for 250,000 objects compared to 1 second of the SQL and stored routines counterparts.

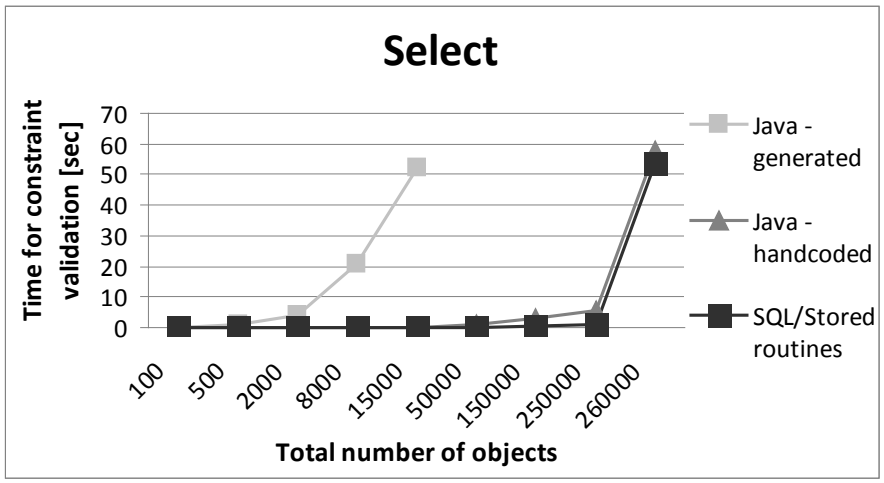

Figure 33: Scalability of constraint validation for the *Select*-constraint

Figure 34 shows the limitations of the scalability testing for the *Iterate*-constraint. The results were similar compared with the *Select*-constraint. Both constraints showed a significantly breakpoint at about 260,000 objects for the database-based transformers. It changed from about 2 seconds for 250,000 added flight objects directly to more than 50 seconds for 260,000 objects. Upon further investigation, the problem was caused by limitations of the test environment. The available physical disk was not able to handle this amount of objects at the same time and caused therefore a significant latency. Every time when the constraint validation for more than 260,000 objects was performed, the average disk read queue length increased to more than one compared to about 0,01 for less than 250,000 objects.

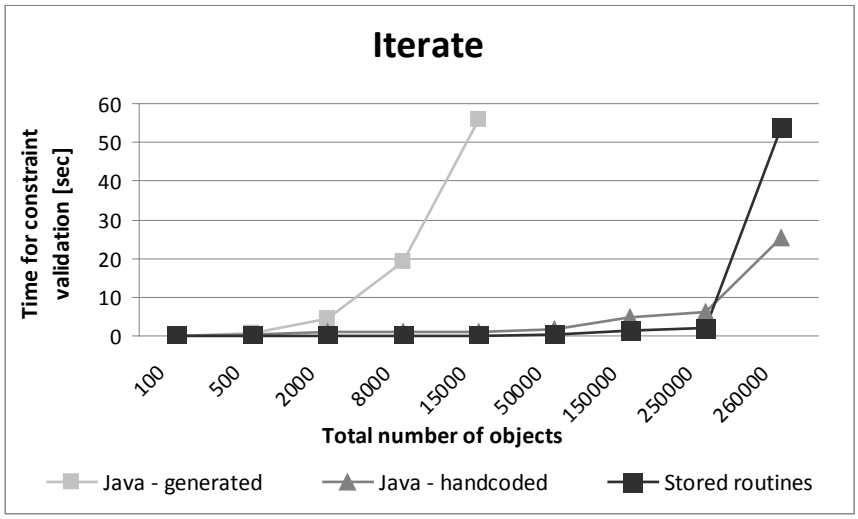

Figure 34: Scalability of constraint validation for the *Iterate*-constraint

The difference between the generated and handwritten *Iterate*-constraint of the stored routines transformer was very small and therefore their validation time is illustrated with only one curve in Figure 34 (labelled as "Stored routines"). Figure 35 shows in a detailed form that at the beginning both took 16 milliseconds for 1000 objects, while the handwritten constraint performed better for 250,000 objects with only 625 milliseconds compared to the 2219 milliseconds of the generated one.

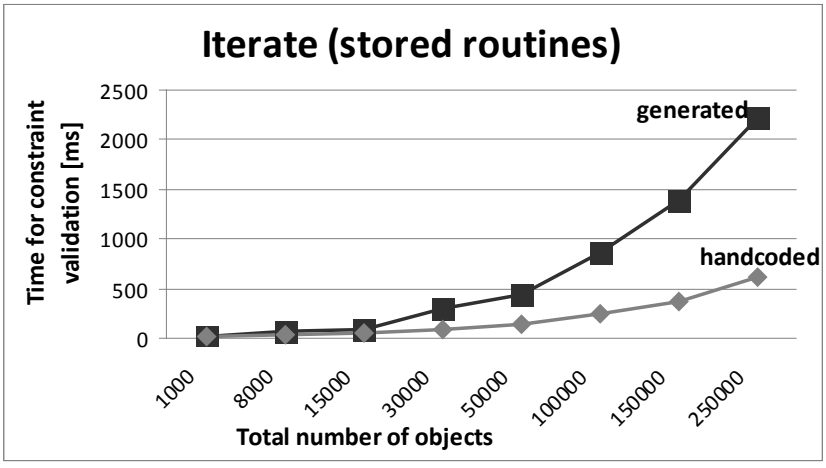

Figure 35: Scalability testing – *Iterate*-constraint (stored routines)

The results of the three evaluated constraints during this test scenario showed that the validation time required for a single constraint on the object layer was too much for using in a production system. A system usually has also other constraints that have to be validated as well and the complete validation of all constraints should not require more than some seconds otherwise it is too slow for an interactive feedback.

4.2.7 Interpretation

From the results, it is visible that the approach with constraint generation was feasible, but with performance problems in applications if complex data structures were used. Using constraint metadata allows efficient triggering of constraint validations by the middleware. The evaluation showed that soft constraints were much better in performance. The advantage is that soft constraint would only be validated once at the end of a transaction, while several modifications would cause the validation of hard constraints several times within a transaction.

The reachable performance enhancement between generated and handwritten constraints of the OCL-to-Java transformer was high. The advantage of handwritten constraints was not present in simple constraints (→ same constraint), but more and more in test cases such as *AllInstances*, *Iterate* and *Select*. The last one iterated over all entity beans in order to find the according flight with the generated constraint. The handwritten constraint was optimized by using a finder method directly in the database. Consequently, it would be meaningful to try to achieve performance benefits when complex structures exist. The generated code was useable for a general case, but every production system would have its own specification that requires special optimization. With handcrafted code, constraints could be customized for different aims. Reducing the number of access or calls to the database, especially in remote clients, could increase the performance several times depending on the number of loops or other complex statements. For simple constraints and when the number of involved objects for validation was small, the generated constraints could be already used in a production system.

Another point with high influence on performance was the database. This option was used by the OCL-to-SQL transformer or the OCL-to-stored-routines transformer. During the evaluation, the database was the best choice for constraint validation. The database had already the stored objects and could use its own mechanisms efficiently. By contrast, on the object layer a significant performance loss was caused by the high overhead and inefficient (transaction) management of a high number of entity beans, including the necessary data transfer from the database for each query. Consequently, the database should be favoured when it provided appropriate mechanisms for constraint checking. Only, when the number of objects was small and no high effort was required for manually editing, a validation on the object layer should be preferred for such constraints like *AllInstances*.

4.2.8 Potential enhancements

A major issue during the evaluation was that for some test cases the execution of generated constraint code of the OCL-to-Java transformer was too slow when loops or complex data types were used several times in the constraint validation statements. The code generator of this thesis creates code for general problems. Consequently, the improvement could be realized by generating different constraints for different applications.

The code generation process is based on several tools that can be executed with a command line interface. It would be more comfortable if a GUI application is implemented for the whole code generation process. Additionally, tool support could be provided for editing generated constraints. According to our experience gained throughout the evaluation, a manual code customization is always necessary for a special application. A tool with a user interface for constraint code editing would increase the clarity of the development work.

A list of potential enhancements is presented as future works in Section 6.2.

4.3 Examples of real-life applications

This section investigates the applicability of the generated constraints of the prototype generator in real-life applications.

The code for the flight booking application can be reused for comparable domains of ticket this application, like public transport or rental car services. It already existed for the DeDiSys middleware and is used within this thesis as the test application.

The work of [Redl10] has evaluated the Aeronautical Information Exchange Model (AIXM) in which the business rules could be transformed by the OCL-to-SQL transformer and the OCL-to-Java transformer. The OCL-to-stored-routines transformer uses the same rule base and it is also able to transform these AIXM business rules.

Other domains of interest are, for example, the automotive or telecommunication sector. The challenges in these domains are to build highly safety-critical or reliable distributed systems. In The following we provide examples for both sectors:

- ***Example in the automotive area***

The paper of [NPF08] shows that today's cars, especially the upper class cars, are more dependent on electronic units than ever. Each car contains up to 80 ECUs (Electronic Control Unit) and several bus systems. Because the embedded automotive software is highly distributed, the automotive industry increases the efforts to develop tools for automated software deployment.

[NPF08] defines a constraint model for optimizing software deployment. It contains three constraint systems: the resource constraints system (memory, power), the quality constraints system (quality functions set) and the cost constraints system (cost criteria). It addresses automated software deployment in terms of a constraint satisfaction problem (CSP) by using a model-based approach.

Although this work does not provide a CSP solver, the explicit integrity constraints is a good basis for further investigation and could be used to build an optimized CSP solver.

The results of the evaluation in this work show that business rules of [NPF08] can be partly defined with OCL expressions and transformed to source code using the implemented transformers. Most constraint types of the resource and cost constraints system are supported. However, the constraints of the quality constraints system are out of context of this thesis and therefore not yet supported. Support for this type of constraints is an interesting challenge to be solved in future works. The supported constraints cannot be always transformed by the

OCL-to-SQL or the OCL-to-stored-routines transformer because of complex OCL's *forAll*-statements, but at least the OCL-to-Java transformer as the last fallback is able to transform them.

- ***Example in the telecommunication area***

Telecommunication is a widespread field and consists of a number of different applications, like, e.g., service level agreements (SLA) to guarantee customers certain Quality of Service (QoS).

The Quality of Service Taskforce of the Open Group introduced SLAs for Application-Specific QoS for Carrier's IP-based VPN Services [QoST01] that is considered by this evaluation.

The results of the evaluation in this work show that the SLA samples can be transformed to their corresponding OCL expressions and can be evaluated with the implemented code generator. The evaluated constraints are simple and can be transformed by all three transformers. For example, a constraint can be defined in OCL to verify the packet loss parameter that should be less than 1% for all available network operating centers (NOCs). The considered SLAs are network availability, packet loss, latency, notification time and credits. In principal, the generated constraints and constraints metadata can be integrated in the applications of different telecommunication provider to ensure integrity and availability.

5 Lessons Learned

This section describes the development process of this diploma thesis. Problems, advantages and goals in context with the design and implementation phase are illustrated.

- ***Design and implementation phase***

During the research phase, many tools were still in beta phase or development so that one goal was to find a correct mix of tools to fulfil the development requirements. Because of the suspension of the community licence for the CASE tool Poseidon, the search had to start again for a new case tool and many adjustments were necessary to made for compatibility reasons.
The used strategy for implementation phase was the top-down approach, but to prepare proof of concepts the bottom-up approach was also used at the beginning. This was necessary to find out which approach could be used by this diploma thesis.

For the whole development process, especially the part with the evaluation, a detailed planning was necessary. How should it work, which part should be evaluated and how extensive should be the tests? First, a similar test was made for all test scenarios. Second, each scenario was performed with different number of objects to found if there were limitations of the test environment.

Finally, the result is to operate a fully-operational code generator. The reached results were expected. For not too complex cases, the code generator is a good and helpful tool that reduces the work of a software engineer. On complex cases, it would be depending on the kind of the application to decide if the using of such a code generator is helpful or not. The detailed reasons would be discussed within a later part of this section.

- ***Compatibility between different tools***

A major difficulty was to use different tools and some of them were still in development. Because of using only open-source tools, not all features were supported during the development phase or these would be only enhanced from time to time by the open source community. It was necessary to find an approach to solve compatibility problems between these tools.

Normally, the situation with using different tools is that from one step to another step the functionality would be more and more restricted. For example, tool A supports the functionally X, Y and Z and tool B only supports X and Y. Consequently, the tool consists of A and B would not have the functionality Z if no enhancements were made. This situation also existed in this project, the functionalities of the code generator was and is depended on other modules such as the Dresden OCL Toolkit or the determined modelling tool. The lack of good and well-commented literature or references made the work more difficult. Finding the trade-off between different features of the different used tools took the most efforts during the implementation phase.

- ***Advantages through using of MDA***

Model Driven Architecture takes and will take an important position in software creation. In the future, there will be several tools that automatically generate source code for constraint checking based on OCL constraints. They should be able to generate source code for constraints that is tangled with other code or explicit constraints code.

The influence of generating code directly from models will be more and more significant for software designers. The past is characterised through separated parts of work such as explicitly parts for the designer, the developer and the programmer. Future developments could continue the traditional procedure or make the change to a “universal designer”. The last one means that one developer would be responsible for everything as an all-rounder, from the design phase to the implementing phase. He/she would cover such a widespread area that it is impossible for one person to have knowledge about every detail. Different kinds of tool support will be needed to perform all these tasks. Old development approaches would be enhanced and new approaches would be defined.

One way would be using MDA to help a developer on his works. By providing respective tools with user-friendly user interfaces, the provided way from diagram to code is an understandable and traceable solution. Tools those are able to use the same design parts for different programming languages would be more common. The development could be directed in half-automated tools and two groups of developers would take an important part in it: the first one has a common knowledge and uses these tools for the implementation. The second one is a much smaller group who are all experts in their fields and develop the tools that are demanded by the first group.
That means there are not only developers any more who work for a customer, but also developers who only work for other developers. They have to work together to solve all possible programming tasks. The group with special skills has the task to develop modules that could be used by all-rounder in development environment with model-driven support.

The code generator from this diploma thesis supports the above-mentioned approach. Based on a UML-OCL diagram it is possible to generate code and in this specific case like for the programming language Java. This is already feasible. It is a continuous way where the developer could work with only one model from the beginning to the end, included transformations from one model to another kind of model and from model to code. This increases both the quote of the fault prevention through consistency, and allows also a simple monitoring over the whole development process. It would be easier to recognize in which state the development is and therefore a target-oriented engagement on problem cases could be started much faster.

- ***Metadata implementation in OCL***

Metadata allows a flexible use of OCL constraints. The possibility of defining further properties about constraints is required in order to configure correct run-time behaviour. Its integration was a focus in this diploma thesis. Several solutions were possible to cope with this problem. Metadata could be integrated in a separate tool or as an enhancement of OCL. The chosen way was to directly integrate it in a modelling tool that fulfilled the requirements. The using of extra classes in the class diagram for metadata makes it understandable for software designers and it allows a simple enhancement. This is even more helpful if metadata are supported by the Object Management Group officially in the future.

The feasibility of constraint-metadata specification is explored. The presented way allows the definition of metadata during the modelling phase together when the constraints are specified in OCL. All needed properties could be defined in a class that uses the same name as the constraint name. It is not necessary to use all possible properties. The type of a property is predefined and the designer needs only to set the value of a property. During the code

generation process, an error check is implemented for the case if undefined properties are used. The implementation is easy to understand and no designer should have any problems when using it. A user error-safe solution would be if the metadata integration in OCL were based on selections instead on inputs by the user. Only properties that are allowed for a specific constraint should be selectable.

- ***Code generation from UML-OCL model up to Java code***

The code generator that has been realised during this diploma thesis contains parts from other tools and other parts were newly developed. A requirement was that all software had to have open source or academic origin and had to be published under the GNU Lesser General Public License. The free availability and traceability of the source code are the main advantages. On the other side, many tools are tested on feasibility, but may be still under development. A guarantee that open source software is always fully implemented and provides 100% functionality could be never given.

Hence, all software that is developed in the context of this diploma thesis is also well tested on feasibility and implemented functions, but it is limited through the restrictions of other open source software. Software with a large function spectrum and – if possible – without any errors requires lots of efforts and this goal is often not reachable with available resources. In contrast, a useful software with well implemented functions is not only realisable with restricted resources, but also operates economically.

The code generator is able to generate EJB Java Beans und constraints from a UML-OCL diagram. The AndroMDA code generation engine generates the entity and session beans. The constraint part is based on the Dresden OCL Toolkit. Thus, constraints of type invariants are well supported, but there is a lack of support for pre- and post-conditions that are also not fully implemented by the Dresden OCL Toolkit. The generated constraints could be used without any changes. Every time when specific solutions are necessary for particular cases, customization of code is required.

The big advantage of a code generator is its large application possibilities. The developer does not need a deep knowledge about the background functions, but he/she must be able to recognise and to take action every time a failure occurs. There will be a transfer from specialist to all-rounder knowledge. Ideally, there exists an association with a standardised purpose, otherwise the costs of implementing a code generator would be much higher than its benefit. Standardised purpose means that a software packet is reusable for the same or different purposes and it can be proprietary, but must have a wide range of potential users. Only, if it would be used many times and varied on different applications, such a development will be affordable.

- ***Influences on future implementations***

As result, the code generator could be used as an independent module of DeDiSys and it is a helpful tool for the constraint checking framework. Both, small and big projects could benefit from automation generation process. The tools in the future would be more and more complex, so that generation tools could take a bigger role during the development process. This code generator is only a small part of the beginning and there should be half a dozen tools in the market in few years that are able to speed-up the development cycle. The big advantage is that also people with small knowledge in this area could benefit from this tool. However, for

development of such generation tools much more knowledge is needed because of the increasing complexity.

- ***Evaluation tests***

During the evaluation phase, three kinds of generated constraints were used, the Java, the SQL and the stored routines version. The generated constraints were compared again to the handwritten ones to show how performance improvement could be reached. The goal of the code generator was not to have the best performance for a specific application. The source code should be used on different applications, ranging from the flight booking example application to public transport ticket or addressbook application. For the first, it was a priority to achieve high consistency. For the other one, the constraints should be fit with a huge number of accesses in a short time and the availability was more important. Another problem was that from the UML-OCL model only one version could be generated for each kind of constraints. This view explained that with the actual available approaches and resources it was very difficult to generate best-performed constraints.

To cope with different requirements, handwritten constraints were used. The aim was to achieve a better performance when the complexity increased with the number of complex statements in the source code. The optimization not necessarily takes effect at the beginning, but should avoid paralysis of the whole system. This explained why the generated version was faster at the beginning of some test cases.

The evaluation results showed that many improvements are possible in the future. Constraints optimization is necessary to achieve acceptable performance. For example, implementing pattern libraries is a possibility to improve the performance. A code generator is more a half-automated tool. It could be used to generate the constructs and base level of source code, but a custom adaption is obligatory to fit the conditions on a production system. It is to evaluate the costs between the programming effort and the desired level of code generation whether a desired level is affordable or not. The cost increases not linearly, but exponentially with the complexity of the outcome.

The additional implementation with the SQL transformer of [Redl10] or the stored routines transformer of this work improves the run-time performance several times compared to the pure Java transformer. If possible, the constraints should be always validated on the database layer with respect to performance. The Java transformer should continue to exist as a fallback. For example, the OCL's *isOCLkindof* operator is not supported by the two database-based transformers because it requires support of an object-oriented language.

6 Summary & Outlook

This section provides a summary of the previous sections. Finally, future aspects are presented as there is still room for improvements.

6.1 Summary

With this work, the feasibility of model-driven support for the software development process building upon EJB on the run-time explicitness of integrity constraints is explored. Metadata are used to configure constraints. This approach allows a flexible use of explicit constraint classes. For example, metadata are used to specify details such as the priority or the type of a constraint.

This diploma thesis is part of the European research project DeDiSys and covers the implementation and evaluation of a code generator with using metadata for constraint checking in an Enterprise JavaBeans environment. The other frameworks of DeDiSys – one with .NET and another in Corba – are not targeted. The EJB framework as a middleware is based on the JBoss application server.

The first step is to model a class diagram with ArgoUML. This model implies not only UML constructs, but also constraints in OCL with its metadata. The metadata is actually not supported by UML and OCL and therefore integrated with an own approach. For metadata implementation, an own class is defined for every specified constraint. If default property values should be used, only a blank class with the name of the constraint is required.

This model will be used by AndroMDA for generating entity and session beans. These codes are ready for usage and could be integrated directly into applications such as the used flight booking application for the evaluation part. The same model will also be used for the constraint generation process. The generator for constraints uses base functionalities of the Dresden OCL Toolkit. The feasibility of generation support for all three kinds of constraints is explored through the implemented code generator. The generated constraints are explicit constraint classes as required by the constraint checking framework of the European research project DeDiSys.

The metadata are prepared by the parser of the Dresden OCL Toolkit. The generation is an own process and finally the result is written to a configuration document for the deployment descriptor.

As a result, all parts of the generation process were integrated into one project that is compatible with the above mentioned constraint checking framework. The project could be used and started in the JBoss application server.

During this diploma thesis, an evaluation has been performed. Different kinds of constraints are generated and evaluated with respect to run-time behaviour. Two kinds of specification are used during the whole evaluation process: hard and soft. Hard means a hard constraint and the constraints evaluation has to be started after the insertion of each object. Soft represents a soft constraint and the constraint evaluation ran only once at the end of the transaction. The test of robustness is determined by different number of objects in each evaluation case. Occurrence problems are determined and alternative solutions are presented. In these cases,

the used constraints are provided manually. The reason is that the code of the generator is sometimes too complex. This needs too much time during the run-time and therefore the sum of the used complex statements such as loops or remote calls should be reduced. Through optimization in form of handwritten constraints, its run-time could be improved.

Besides the OCL-to-Java transformer, an OCL-to-stored-routines transformer is implemented. The last one is an extension of the OCL-to-SQL transformer of [Redl10] that builds up on the Java Transformer of this work. The evaluation showed that this optimization was worth their effort because the OCL-to-stored-routines transformer is able to transform the OCL's *iterate* and *if* expression that is not supported by the OCL-to-SQL transformer. The central statement of the evaluation is that constraint validation on the database layer is much faster than on the object layer.

6.2 Future prospects

The proof-of-concept is demonstrated with the code generation process, but a number of tasks are left for future works. Many tools that are used during this thesis are not completed and still in development. Moreover, the code generator could be enhanced gradually with other tools when the corresponded supports are available. The following enhancements are useful not only for research reasons, but also for using in the praxis.

- ***Metadata integration in OCL***

Due to lack metadata support in modelling software, a way has been found to use metadata together with OCL. The presented solution is useable, but may not be the best one. A better way could be a direct integration in OCL. Every time, when a constraint is defined, the modeller should be able to specify the metadata for this constraint at the same time. This would make the process not only more simple, but also more user-friendly. The enhancement of OCL syntax would be restricted only to defined modelling tools that are supported by the code generators too.

- ***Performances improvement***

A big problem of code generators is generally the performance. A code generator could only create code for general problems. However, for difficult problems it is mostly too complex to fulfil the performance requirement. The run-time of the generated code by the code generator of this thesis does not increase constantly with the number of added objects. It depends on many reasons such as the number of loops or complex data types.

A solution is to validate the constraints on the database layer instead of on the object layer, for example with the generated constraints of the presented SQL or stored routines generator in Section 3.11. Another solution to improve the performances may be to generate different constraints for different purposes. For example, this could be done by purpose group allocation. During the design phase, classes in the class diagram could be allocated to a defined group such as through using predefined stereotypes. At the end, the generator could create special code for each class with one of these stereotypes. Thus, the code is optimized in performance for a certain purpose.

- ***Different programming languages***

The developed code generator is only for Java currently. Porting to other languages such as C or .NET is a task to provide a multiplatform code generator. Which language would be used

for coding should be no more relevant, the developer should only need to learn how to use the code generator directly. The scope would be too widespread for all languages, but all main programming languages should be included.

- ***Tools enhancement***

Fundamental tools such as AndroMDA and Dresden OCL Toolkit are open source projects that are developed by a community or a research institution. These tools are still in development or some functionality that is valuable for this diploma thesis would be implemented not until next versions. For this reason, it is advisable to enhance the code generator to make it state-of-art if newer versions with more functions of these two tools exist.

- ***Direct tools-integration***

The whole code generation process is a co-working of different tools. Now, it is controlled through a command-line tool and separated in many parts. A more comfortable solution for the user would be a direct integration into a CASE tool. The user no more needs to learn how to use different tools. He/she could define the model and generate code in one-step. This would increase the development process and make it faster. The users have usually different backgrounds such as project managers, designers or programmers. A uniform generation solution for everyone would be difficult to realize, but not unrealistic. By all means, a GUI integration should be always preferred over command-line solutions.

- ***UI-tools for constraint editing***

As described in the realization section, the generated source code is not always the best for all applications. They are not optimized in structure and performance. For many applications, the source code is not useable because the run-time is not acceptable. This is also true with the mentioned performance enhancement method above. A manual customization is necessary to make the code fit for a special application. However, there is no tool support during the customization process. An editing possibility directly integrated in a modelling tool or provided by an external tool with a user-friendly user interface would increase the clarity of the development work and allow the developer to optimize in a very easy clear way.
The editing tool could also integrate other functionalities such as a database with correction suggestion for different application purposes.

6.3 Conclusion

Within this thesis, the feasibility of model-driven support for integrity constraints and constraints metadata is explored. It consists of a code generator that uses a UML-OCL design model to generate entity beans, explicit constraint classes and constraints metadata. Metadata contain the properties of constraints and are introduced by the constraint checking framework of DeDiSys for adapting constraints during run-time. Because metadata are not supported by OCL and UML, this issue was solved by using UML class diagrams to encode the metadata.

The proof-of-concept is delivered through an evaluation. The evaluation results show that the approach of this diploma thesis was feasible. Additionally, handcrafted constraints and database-based transformers were provided to show how the constraint validation code could be optimized with respect to performance. Different scenarios were performed and showed that constraints metadata can improve run-time performance significantly.

References

[ASCY10] Avila, C., Sarcar, A., Cheon, Y., and Yeep, C. (2010): Runtime Constraint Checking Approaches for OCL, A Critical Comparison. In Proceedings of the 22nd International Conference on Software Engineering and Knowledge Engineering, San Francisco, CA, USA, 393-39

[AMDA08] AndroMDA Team: Reference docs 3.2. http://www.andromda.org (January 2008)

[BC06] Brambilla, M., and Cabot, J. (2006): Constraint tuning and management for web applications. In ICWE '06: Proceedings of the 6th international conference on Web engineering, New York, NY, USA, 2006. ACM 345-352

[BFMW01] Bartetzko, D., Fischer, C., Möller, M. and Wehrheim, H. (2001): Jass - Java with assertions. In K. Havel and G. Rosu, editors, Proceedings of the First Workshop on Runtime Verification, July 2001

[DeMi03] DeMichiel, L. (2003): Enterprise javabeanstm specification. Technical Report Version 2.1, Sun Microsystems, Inc. 6

[DOCL08] Dresden OCL Toolkit, DresdenOCL Team, Dresden University of Technology. http://dresden-ocl.sourceforge.net/ (January 2008)

[EDC10] Egea, M., Dania, C., and Clavel, M. (2010): MySQL4OCL: A Stored Procedure-Based MySQL Code Generator for OCL. In Proceedings of the Workshop on OCL and Textual (OCL 2010), Oslo, Norway

[Ertl07] Ertl, Dominik (2007): Evaluation of Partitionable Replication Protocol Improvements in an Enterprise JavaBeans Environment. Diploma thesis, Vienna University of Technology

[FOG06] Froihofer, L., Osrael, J., and Goeschka, K. M. (2006): Trading Integrity for Availability by Means of Explicit Runtime Constraints. In Proc. 30th Int. Conf. on Computer Software and Applications (COMPSAC'06), IEEE CS 14-17

[FOG07] Froihofer, L., Osrael, J., and Goeschka, K. M. (2007): Decoupling Constraint Validation from Business Activities to Improve Dependability in Distributed Object Systems. In Proc. 2nd Int. Conf. on Availability, Reliability, and Security (ARES'07), IEEE CS 443-450

[Froi05] Froihofer (ed.), L. (2005): FTNS system model. Technical Report D1.1.1, DeDiSys Consortium. http://www.dedisys.org/ (January 2008)

[Froi07] Froihofer, L. (2007): Middleware Support for Adaptive Dependability through Explicit Runtime Integrity Constraints. Dissertation, Vienna University of Technology

[Geig04] Geiger, R. (2004): Evaluierung von Produktivitätspotenzialen im Software-Entwicklungsprozess für Web-Anwendung. Diploma thesis, Fachhochschule Karlsruhe

[HF06] Harrison, G., and Feuerstein, S. (2006): MySQL Stored Procedure Programming - Building High-Performance Web Applications in MySQL. O'Reilly Media

[Hore06] Horehled, M. (2006): Integration of an ejb constraint consistency management framework into the jboss application server. Diploma thesis, University of applied sciences (UAS) 'FH Technikum Wien'

[Kone05a] Konermann, A. (2005): The Parser Subsystem of the Dresden OCL2 Toolkit - Design and Implementation. DresdenOCL Team, Dresden University of Technology. http://dresden-ocl.sourceforge.net/ (January 2008)

[Kone05b] Konermann, A. (2005): Status of the OCL2.0 Parser. DresdenOCL Team, Dresden University of Technology. http://dresden-ocl.sourceforge.net/publications.html (January 2008)

[Kram98] Kramer, R. (1998): iContract - The Java Design by Contract Tool. In TOOLS ’98: Proceedings of the Technology of Object-Oriented Languages and Systems, Washington, DC, USA, IEEE Computer Society 295

[Kuen07] Künig (ed.), H. (2007): FTNS/EJB system design & first prototype & test report. Technical Report D3.2.2, DeDiSys Consortium. http://www.dedisys.org/ (January 2008)

[KWB03] Kleppe, A., Warmer, J., and Bast, Wim (2003): MDA Explained: The Model Driven Architecture™: Practice and Promise. Addison-Wesley

[Meye92] Meyer, B. (1992): Applying “design by contract”. Computer 25(10), p. 40-51

[LBR99] Leavens, G. T., Baker, A.L., and Ruby, C. (1999): JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, Behavioral Specifications of Businesses and Systems, chapter 12, pages 175-188. Kluwer Academic Publishers

[MM03] Miller, J., and Mukerji, J. (2003): MDA Guide Version 1.0.1. Object Management Group, Inc. http://www.omg.org/mda/ (January 2008)

[NPF08] Nica, M., Peischl, B., and Wotawa, F. (2008): A Constraint Model for Automated Deployment of Automotive Control Software. In Proceedings of the Twentieth International Conference on Software Engineering and Knowledge Engineering, San Francisco, CA, USA, 899-904

[Ocke03] Ocke, Stefan (2003): Entwurf und Implementation eines metamodellbasierten OCL-Compilers. Diploma thesis, Dresden University of Technology

[O4J08] OCL4Java Team:
http://www.ocl4java.org/ (January 2008)

[OCL06] OMG Group (2006): Object Constraint Language - OMG Available Specification Version 2.0. Object Management Group, Inc. http://www.omg.org/ (January 2008)

[OFG06] Osrael, J., Froihofer, L., Goeschka, K. M., Beyer, S., Galdámez, P., and Muñoz-Escoí, F. (2006): A system architecture for enhanced availability of tightly coupled distributed systems. In Proc. 1st Int. Conf. on Availability, Reliability, and Security (ARES'06), IEEE CS 400-407

[QoST01] The Quality of Service Taskforce, the Open Group (2001): SLAs for Application-Specific QoS for Carrier's IP-based VPN Services
http://archive.opengroup.org/tech/qos/quarry/index.htm (Januar 2010)

[Redl10] Redlein, Andreas (2010): Optimization of an integrity constraints generator. Diploma thesis, Vienna University of Technology

[RSB05] Roman, E., Sriganesh, R. P., and Brose, G. (2005): Mastering Enterprise Java Beans. Third Edition. Wiley Publishing, Inc.

[Trus03] Truskaller, T. (2003): Data Integration into a Gene Expression Database, Master thesis, Graz University of Technology

[UML09] OMG Group (2009): Unified Modeling Language™ (UML®) - OMG Available Specification Version 2.2. Object Management Group, Inc. http://www.uml.org/ (June 2010)

[VS02] Verheecke, B., and Straeten, R. V. D. (2002): Specifying and implementing the operational use of constraints in object-oriented applications. In Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications, Sydney, Australia, 23–32

[Wieb00] Wiebicke, R. (2000): Utility Support for Checking OCL business Rules in Java Programs. Master's thesis, Dresden University of Technology

[WK99] Warmer, J., and Kleppe, A. (1999): The Object Constraint Language. Precise Modeling with UML. Addison-Wesley

Appendix

A1. Link collection

This section provides links to websites, which are relevant during this thesis. All links have been checked for validity last time on January 19th, 2011:

- Apache Ant:
  http://ant.apache.org/
- Apache Maven:
  http://maven.apache.org/
- Apache Velocity:
  http://velocity.apache.org/
- AndroMDA:
  http://www.andromda.org/index.php
- AndroMDA Ant Task:
  http://docs.andromda.org/andromda-ant-support/andromda-ant-task/index.html
- AndroMDA cartridges:
  http://docs.andromda.org/andromda-cartridges/
- Common Warehouse Metamodel (CWM):
  http://www.omg.org/technology/documents/modeling_spec_catalog.htm
- Dependable distributed systems (DeDiSys):
  http://www.dedisys.org/
- Dresden OCL Toolkit:
  http://dresden-ocl.sourceforge.net/
- IBM:
  http://www.ibm.com/
- List of CASE tools for AndroMDA:
  http://docs.andromda.org/case-tools.html
- MagicDraw:
  http://www.magicdraw.com/
- Meta Object Facility (MOF):
  http://www.omg.org/mof/
- MySQL:
  http://dev.mysql.com/
- Object Constraint Language (OCL):
  http://www.omg.org/technology/documents/modeling_spec_catalog.htm#OCL
- Object Management Group (OMG):
  http://www.omg.org/
- Poseidon:
  http://www.gentleware.com/
- Unified Modeling Language (UML):
  http://www.uml.org/#UML2.0
- subversion:
  http://subversion.tigris.org/
- XDoclet:
  http://xdoclet.sourceforge.net/
- XML Metadata Interchange (XMI):
  http://www.omg.org/technology/xml/index.htm

A2. Description: constraint deployment descriptor

The meanings of the constraint deployment descriptor's elements are as described in detail in the following tables [Hore06]:

ccDefinitions	Root element that spans all the constraint specific information for one application	
persister	Specifies the used persister for persisting all the consistency threats that have been accepted during the negotiation process. The *class* attribute with the complete path of the persister is a mandatory one.	
defaultconstraint-reconciliationhandler	Defines the default reconciliation handler to be used by the framework.	
minSystem-SatisfactionDegree	Specifies the application specific minimum satisfaction degree that is employed in case of static negotiation and lack of a constraint specific minimum satisfaction degree. It is mandatory and has the following values:	
	– VIOLATED – UNCHECKABLE – POSSIBLY_VIOLATED – POSSIBLY_SATISFIED – SATISFIED	
constraint	Each constraint tag defines one constraint with the following attributes that all are mandatory ones:	
	name	A unique name is used to identity a constraint (per deployment unit).
	type	Specifies the constraint type with one of the following values:
		– PRE – POST – HARD – SOFT – ASYNC
	priority	Specifies the importance of a constraint:
		– CRITICAL – REGULAR – RELAXABLE
	negotiation	Specifies the type of negotiation:
		– IMMEDIATE – DEFERRED
	contextObject	Specifies the necessary of a context object [Y/N]
	minSatisfaction-Degree	Specifies the current constraint's minimum satisfaction degree utilized in case of static negotiation:
		– VIOLATED

		– UNCHECKABLE – POSSIBLY_VIOLATED – POSSIBLY_SATISFIED – SATISFIED
	latestAccepted-Satisfied-Threat-Removes-IdenticalThreats	Specifies whether the latest accepted and satisfied threat removes identical threats [Y/N]
	intra-object	Specifies the kind of the constraint. Intra-object refers to attributes of one single object. [true/false]

Table 8: Constraint DTD main elements

class	Specifies the concrete name of the implementing constraint or its interface
context-class	Specifies the class name of the constraint context object
expression	Specifies a string value such as the constraint OCL expression or an empty string
affected-methods	Specifies one or more methods with *context-preparation* and method name (*objectMethod*). The *objectMethod* is defined with its *objectClass* and *arguments*.

Table 9: Constraint tag structure

A3. Configuration document: *andromda.xml*

```
<andromda>
   <properties>
      <property name="modelValidation">true</property>
      <property name="cartridgeFilter">ejb</property>
   </properties>
   <server>
      <host>localhost</host>
      <port>4446</port>
   </server>
   <repositories>
      <repository name="netBeansMDR">
         <models>
            <model>
               <uri>file:${xmi.file}</uri>
               <moduleSearchLocations>
                  <location patterns="*.xml.zip,*.xmi">
                        ${basedir}/src/andromda/andromda/xml.zips</location>
               </moduleSearchLocations>
            </model>
         </models>
      </repository>
   </repositories>
   <namespaces>
      <namespace name="default">
         <properties>
            <property name="languageMappingsUri">
                  file:${basedir}/lib_mda/Mapping/JavaMappings.xml</property>
            <property name="wrapperMappingsUri">
                  file:${basedir}/lib_mda/Mapping/JavaWrapperMappings.xml</property>
            <property name="sqlMappingsUri">
                  file:${basedir}/lib_mda/Mapping/MySQLMappings.xml</property>
            <property name="jdbcMappingsUri">
                  file:${basedir}/lib_mda/Mapping/JdbcMappings.xml</property>
         </properties>
      </namespace>
      <namespace name="ejb">
         <properties>
            <property name="entity-beans">${test.output.dir}</property>
            <property name="session-beans">${test.output.dir}</property>
            <property name="entity-impls">${test.output.dir}</property>
            <property name="session-impls">${test.output.dir}</property>
         </properties>
      </namespace>
   </namespaces>
</andromda>
```

Listing 24: AndroMDA configuration document

A4. XLST transformation: *xmi2ocl.xsl*

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
            xmlns:xs="http://www.w3.org/2001/XMLSchema"
            xmlns:UML="org.omg.xmi.namespace.UML"
                  xmlns:UML2="org.omg.xmi.namespace.UML2"
            exclude-result-prefixes="UML"
            version="1.0">

  <xsl:output indent="no" />
  <xsl:output omit-xml-declaration ="yes" />

  <xsl:template match="XMI[@xmi.version='1.2']">
    <xsl:apply-templates select=
          "XMI.content/UML:Model/UML:Namespace.ownedElement/UML:Constraint"/>
  </xsl:template>

  <xsl:template match="XMI">
    <xsl:message terminate="yes">Unknown XMI version</xsl:message>
  </xsl:template>

  <xsl:template
match="UML:Constraint/UML:Constraint.body/UML:BooleanExpression">
     <xsl:text>
     </xsl:text>
   <xsl:value-of select="@body" disable-output-escaping="yes"/>
  </xsl:template>

  <xsl:template match="text()">
    <xsl:value-of select="normalize-space(.)"/>
  </xsl:template>
</xsl:stylesheet>
```

Listing 25: Transformation document: *xmi2ocl.xsl*

A5. XLST transformation: *xmi2xmi.xsl*

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
        xmlns:UML="org.omg.xmi.namespace.UML"
        xmlns:UML2="org.omg.xmi.namespace.UML2"
        >

 <xsl:template match="*">
  <xsl:copy> <xsl:copy-of select="@*"/><xsl:apply-templates/> </xsl:copy>
 </xsl:template>

 <!-- Disable all nodes beginning with "UML:Constraint" -->
 <xsl:template match=
        "UML:Constraint[/XMI/XMI.content/UML:Model/
        UML:Namespace.ownedElement]"/>
</xsl:stylesheet>
```

Listing 26: Transformation document: *xmi2xmi.xsl*