FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Optimization of an integrity constraints generator

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Andreas Redlein, BSc.
Matrikelnummer 0326016

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Dr. Karl M. Göschka
Mitwirkung: Dr. Lorenz Froihofer

Wien, 18.04.2010

(Unterschrift Verfasser) (Unterschrift Betreuer)

Erklärung zur Verfassung der Arbeit

Andreas Redlein
Musilplatz 5
1160 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschlielich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 18. April 2010

Acknowledgements

First of all, I want to thank Lorenz Froihofer for his excellent guidance and support of my thesis. Additional thanks go to Karl M. Göschka for giving me the possibility to write this thesis.

Furthermore, I'm especially thankful to my parents and Lisi, Peter and Fredi for supporting my studies in a unique way.

Thanks also go to all my friends and all other people that supported me troughout my studies and that are not explicitly stated here.

Finally, I want to thank the R.I.C.S. EDV-GmbH for the support of my studies by offering very flexible working times.

Abstract

This thesis addresses the optimization of a Java code generator in the fields of Model-Driven Architecture (MDA) and dependable distributed systems. The generated Java code represents integrity constraint validation code, which is generated from OCL constraints that are defined on a UML model.

This Java code generator does not always produce the most efficient code with respect to its run-time performance, because of deep nested loops. For this purpose, a OCL to SQL transformer was implemented that generates SQL code validating constraints on the database layer.

Furthermore, this work introduces a transformation algorithm that allows to split OCL constraints into their incoherent parts and to transform these parts separately. Hence, it is possible to validate one constraint using Java and SQL.

Finally, an evaluation of the run-time performance of the generated Java constraint validation code shows that the performance is considerably increased.

Zusammenfassung

Diese Diplomarbeit befasst sich mit der Optimierung eines Java Code Generators in den Bereichen Model-Driven Architecture (MDA) und verlässlichen verteilten Systemen. Der generierte Java Code stellt Constraint-Validierungs Code dar, der von OCL Constraints generiert wird. Diese OCL Constraints werden innerhalb eines UML Modelles definiert.

Dieser Java Code-Generator generiert nicht immer den effizientesten Code in Bezug auf die Validierungs-Performanz wegen tiefer Verschachtelungen von Schleifen. Zu diesem Zweck wurde ein OCL to SQL Transformator implementiert, der SQL Code generiert, welcher Contraints auf Datenbankebene prüft.

Weiters präsentiert diese Arbeit einen Transformationsalgorithmus der es erlaubt OCL Constraints in deren unzusammenhängende Teile zu splitten und diese Teile getrennt voneinander zu transformieren. Dadurch kann ein Constraint mittels Java und SQL geprüft werden.

Zum Abschluss zeigt eine Evaluierung der Laufzeit-Performanz des generierten Java Codes, dass die Performanz erheblich verbessert wurde.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The development of software using a model-driven approach becomes more interesting the more complex a software module is with respect to maintainability. This thesis addresses the tuning and optimization of a Java code generator in the fields of *Model-Driven Architecture (MDA)* and *Dependable Distributed Systems*.

The main entry point to this thesis is the J2EE constraint consistency framework *Dependable Distributed Systems (DeDiSys) middleware* running within a *JBoss application server*. This middleware has its main focus on trading the availability of applications against constraint consistency in order to improve the availability of these applications. For this purpose, this middleware allows for registering application-specific constraint validation code and additional constraint meta-data. This constraint meta-data is necessary to - amongst others - define if a constraint may never be violated or if it can be temporarily relaxed to improve the availability.

As mentioned before, there previously existed a basic implementation of a Java code generator, which is referred to as *Legacy Prototype*. This prototype is responsible for generating these constraint validation classes and the constraint meta-data using a model-driven approach. For this purpose, the applications using the DeDiSys middleware are modeled as UML models and their corresponding constraints are specified directly on these models using the *Object Constraint Language (OCL)*.

This legacy prototype does not always produce the most efficient Java code to check a constraint with respect to its run-time performance, but rather produces unacceptable results in some cases. For this reason, it is necessary to take a deeper look at this prototype and therefore, this thesis has its main

focus on optimizing this prototype in order to generate faster Java code again with respect to its run-time performance.

1.1 Motivation and problem definition

As handcrafting of these Java constraint validation classes is a tedious and error prone task, this process should be improved using a model-driven approach. The idea was to model the application using a UML model, to specify the application-specific constraints using the OCL and to generate the necessary constraint validation classes as well as the constraint meta-data from it. For this purpose, the legacy prototype implemented within a prior project transforms every OCL constraint to its corresponding Java constraint validation code validating the constraint completely on the object layer. For the reason that OCL constraints themselves can become complex in a sense that often tens of thousands and more numbers of records have to be examined or deep loops have to be gone through, this transformation often generates unacceptable Java code with respect to its run-time performance.

From the implementations point of view, the DeDiSys middleware needs the applications to specify their Java constraint validation code as separate Java classes, whereas one class validates exactly one constraint. These classes need to extend the frameworks `AbstractConstraint` class and the validation code has to be placed into the inherited `validate` method. Moreover, additional constraint meta-data has to be specified for every constraint in order to improve the availability by temporarily relaxing constraints. Examples for constraint meta-data are if constraints can temporarily relaxed or if they have to hold every time. Finally, by deploying these applications to the application server, the Java constraint validation classes are registered to the DeDiSys framework using the constraint meta-data. For more information about the DeDiSys framework, please refer to [FOG07], [FGO07] and [Hor06].

For a better understanding, the problem is illustrated using two OCL constraints. First of all consider the simple OCL constraint `validBooking` illustrated in listing 1.1, which is an example for an OCL constraint that can be validated on the object layer in an efficient way.

```
context Flight inv validBooking:
self.bookedSeats <= self.maxSeats
```

Listing 1.1: Example of a simple constraint

The OCL constraint given in listing 1.1 validates the amount of booked seats not to be greater than the maximum available seats for each flight. The aforementioned legacy prototype transforms this OCL constraint to the Java constraint validation code shown in listing 1.2.

```
public boolean validate (IConstraintValidationContext ctx)
    throws ConstraintUncheckableException {
  boolean result = false;

  try {
    FlightBeanImpl flight = (FlightBeanImpl) ctx.
        getContextObject();
    result = flight.getBookedSeats() <= flight.getMaxSeats();
  }
  catch (Exception e) {
    throw new ConstraintUncheckableException(e);
  }

  return result;
}
```

Listing 1.2: Java transformation of the simple constraint

The resulting Java code generated by the legacy prototype shown in listing 1.2 is an example for efficient Java constraint validation code validating the constraint on the object layer. In contrast, due to the eventually increasing complexity of OCL constraints, this validation of constraints on the object layer can become inefficient with respect to its run-time performance.

As second constraint to illustrate the problem, consider the OCL constraint *uniqueFlightNr* provided in listing 1.3, which is an example of a more complex OCL constraint. In contrast to the previous OCL constraint `validBooking`, the transformation of this OCL constraint returns unacceptable Java constraint validation code regarding its run-time performance.

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

Listing 1.3: Example of a complex constraint

The OCL constraint given in listing 1.3, which by the way makes use of OCL's `allInstances` operator, validates the uniqueness of the flight num-

ber within all flights. The legacy prototype transforms this OCL constraint to the Java constraint validation code shown in listing 1.4.

```
public boolean validate (IConstraintValidationContext ctx)
   throws ConstraintUncheckableException {
  boolean result = false;

  try {
    FlightBeanImpl flight = (FlightBeanImpl) ctx.
       getContextObject();
    Collection left0 = flightHome.findAll();
    Iterator it_f2 = left0.iterator();
    ejb.FlightLocal f2 = null;
    ejb.FlightLocal f1 = null;

    result = true;
    while (result && it_f2.hasNext()) {
      f2 = (FlightLocal) it_f2.next();
      Iterator it_f1 = left0.iterator();
      while (result && it_f1.hasNext()) {
        f1 = (ejb.FlightLocal) it_f1.next();
        if (f1.equals(f2)) {
          result = true;
        }
        else {
          result = !(f1.getFlightNumber().equals(f2.
              getFlightNumber()));
        }
      }
    }
  }
  catch (Exception e) {
    throw new ConstraintUncheckableException(e);
  }

  return result;
}
```

Listing 1.4: Java transformation of the complex constraint

The resulting Java code of the legacy prototype shown in 1.4 is an example for Java constraint validation code where the validation of constraints on the object layer is inefficient. For every flight, this resulting Java code iterates over all flights and compares the flight numbers, which is unacceptable in terms of the run-time performance of the constraint validation process.

Hence, the main focus of this thesis is to find a way to generate more efficient Java constraint validation code, regarding the run-time performance of the constraints, when they are going to be validated.

1.2 Organization of this thesis

After the brief introduction provided in the previous section, the following chapter 2 covers the state of the art that is related to this work. This chapter is divided into the three fields *Theoretical base* that takes a deeper look at the fundamentals of this thesis, *Related work* that introduces three major projects, having their focus on similar topics and *Technical base*, which gives an overview of third party tools and libraries used within this thesis.

Afterwards, chapter 3 provides a detailed description of the prototype. As the prototype is built upon a previously existing implementation of an OCL to Java transformer, a description of this implementation is incorporated. The major sections of this chapter are the *Architectures* of both prototypes, the *Core parts* implemented within the prototype of this thesis and a sample transformation of OCL constraints.

The implemented prototype is evaluated in chapter 4. This evaluation is divided into a *Prototype evaluation* that illustrates the ability of the prototype to transform OCL constraints of different areas, a *Constraint performance evaluation*, which shows the benefit of the implemented prototype compared to the legacy prototype and moreover *Future work*, provides an outlook about further challenges and future prospects.

Finally, chapter 5 summarizes this thesis and draws the conclusion.

Chapter 2

State of the art

This chapter provides detailed information about the fundamentals to this thesis and therefore, is split into the three different categories *Theoretical base*, *Technical base* and *Related work*.

2.1 Theoretical baseline

For the purpose of a better understanding of this thesis throughout the following chapters, this section covers the necessary theoretical background.

2.1.1 Model-driven architecture

Model-Driven Architecture (MDA), which was formally specified by the Object Management Group (OMG) in the year 2000 is all about modeling an application in a platform-independent way. A more exact definition of MDA is given by the OMG as follows:

OMG's Model Driven Architecture (MDA) provides an open, vendor-neutral approach to the challenge of business and technology change. Based on OMG's established standards, the MDA separates business and application logic from underlying platform technology. [OMG]

[KWB03] distinguish in their work between three different models that build the core of MDA and furthermore the MDA development life-cycle. These model are:

1. **Platform independent model (PIM)**: The first model is a high-

level abstraction of a software system describing some business and thus, it is independent of any implementation technology.

2. **Platform specific model (PSM)**: The PSM is the second model, which is obtained from a PIM to PST transformation. There does not exist exactly one PST for each PIM, but rather there can be several PST's within each PIM. A PST is a platform specific model that includes keywords of the particular implementation technology (e.g. HomeInterface and RemoteInterface within an EJB PSM).

3. **Code**: In the final step, each PST is transformed to its corresponding code, which can be considered as a straightforward task.

Figure 2.1 taken from [KWB03], illustrates the aforementioned relationships between these models.

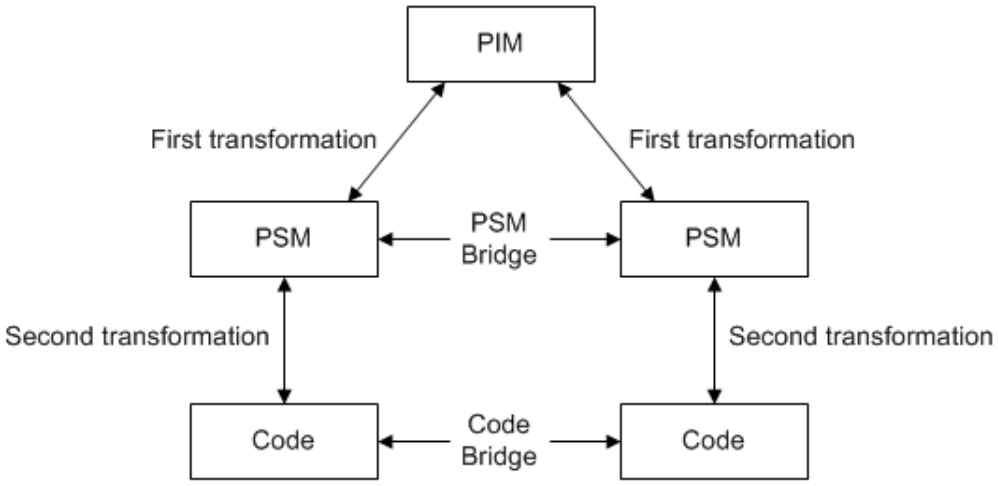


Figure 2.1: MDA development life-cycle [KWB03]

The major benefits of developing software using an MDA approach identified by [KWB03] are:

- **Productivity**: The main focus of the majority of the software developers will be shifted to the development of PIM's, whereby these developers do no longer need to struggle so much with platform-specific tasks, as a large amount of this code is already generated using the

PIM to PST transformation. Furthermore, the end users can get better code in less time, because of the moved focus of the developers and therefore, they concentrate on solving the business problems.

- **Portability**: The portability of PIM's can be derived from the fact that these models are completely platform-independent. Furthermore, portability can also be achieved for PST's, in case of using popular implementation techniques and therefore, existing PIM to PST transformations can be used.
- **Interoperability**: By using MDA, it is possible that different PST's talk to each other. To this end, MDA introduces the concept of bridges that can be generated for the PSM and the Code level.
- **Maintenance and Documentation**: The maintenance of applications is much easier, because they can be made within the PIM. Afterwards, the corresponding PST's can be regenerated easily. Moreover, as the PIM is a high-level abstraction of the software system, it fulfills the function of a high-level documentation that is needed for each software system. Nevertheless, it is necessary to write down additional documentation that can not be captured within the model.

Within this thesis the MDA approach is used to create applications that make use of the DeDiSys framework (see section 2.1.3). In the first step, these applications are created as UML models using the UML modeling tool ArgoUML. In the next step it is enriched by OCL constraint and further constraint data, before finally the Java constraint validation code is generated from the UML model.

For more information about MDA please refer to [OMG], [KWB03] and [WK03].

2.1.2 Object constraint language

Standard UML models can almost never cover all necessary information to describe an application. For the purpose to fill this gap, the Object Management Group (OMG) introduced a specification for the so called Object Constraint Language (OCL) that enriches the UML standard by integrity constraints that can be defined directly on UML models. With the use of these OCL constraints it is possible to provide an application with additional information and thus, to express more sophisticated needs of this application.

The general structure of an OCL constraint is illustrated in listing 2.1.

```
package {package}
  context {class} {type} {name}: {body}
endpackage
```

Listing 2.1: Structure of an OCL constraint

Phrases embraced by brackets are specific for each constraint and therefore have to be replaced with content corresponding to a certain constraint. Moreover, it is possible to specify more than one constraint within a package definition. All other words (words that are not denoted in brackets) are reserved words. The aforementioned constraint specific parts are:

- `package:` Defines the package that this constraint should be assigned to.
- `class:` The constraint context in which the OCL constraint is going to be evaluated. This context is the name of one class within the UML model that relates this constraint to this class.
- `type:` The constraint type, defining weather it is an precondition, a postcondition or an invariant. This value has to be one out of the following enumeration: `pre`, `post` or `inv`.
- `name:` The name of the OCL constraint. This name has to be unique throughout within a UML model.
- `body:` The constraint body that is the OCL expression that has to hold for each instance.

An example of a specific OCL constraint within a typical flight booking application is given in listing 2.2.

```
context Flight inv FlightFromBarcelonaToViennaExists:
Flight::allInstances()->select(
  f | f.fromAirport = 'BAR' AND f.toAirport = 'VIE'
)->size() > 0
```

Listing 2.2: Example of an OCL constraint

The constraint provided in listing 2.2 shows an invariant that is evaluated in the context of the class `Flight` and ensures that there always exists at least

one flight from Barcelona (BAR) to Vienna (VIE). An expression like this would not be possible to express within the UML standard.

For further information about the Object Constraint Language (OCL), especially regarding the constraint body, please refer to the [OMG].

2.1.3 DeDiSys middleware

The *DeDiSys middleware* is a middleware to support dependability of distributed systems and was developed by the Distributed Systems Group of the Information Systems Institute of the Vienna University of Technology. A short description of the project context provided by the authors is given as follows:

The major focus of DeDiSys is on optimized dependability by adaptively balancing (trading) availability and constraint consistency. System entities (objects, services) are constraint consistent if constraints (typically predicates stemming from requirements) put on them are satisfied. [Ded]

In order to trade availability and constraint consistency, two different types of applications have to be taken into account. The first one includes applications, in which consistency has to be ensured all the time, like it is in banking applications. The positive effect of these applications is that the system is always in a constraint consistent state, but also has the negative implication that the system becomes unavailable if not all constraints can be satisfied, e.g. due to node and link failures, which is referred to as degraded mode. The second type includes applications, in which availability is the ultimate ambition. The focus of these applications is the opposite of the focus of the previous type. That is to say, the application can be available even if it operates in degraded mode, but in return for this behavior it is possible to introduce constraint inconsistencies. [Ded]

The aim of DeDiSys is to investigate the optimum between the two extremes: Can some of the data and service constraints be adaptively negotiated and (temporarily) relaxed to improve the availability of the distributed system? [Ded]

The intention of figure 2.2 is to illustrates the previously described trade-off within the *DeDiSys middleware*.

From the implementation's point of view, the *DeDiSys middleware* is built upon the *JBoss Application Server* as its environment. For more information

Figure 2.2: DeDiSys trade-off [Ded]

about the interaction of *DeDiSys* and *JBoss*, please refer to [Hor06], as this is out of the focus of this thesis.

Every application using the DeDiSys middleware, needs to provide specific information to it. A detailed description about this information is covered in the upcoming sections.

Constraint validation code

Every application needs to provide a separate Java class for each constraint identified within the application. This class, which must extend DeDiSys' `AbstractConstraint` class, provides the Java code responsible for validating the constraint. For this purpose, it simply needs to return `true` in case of satisfaction of the constraint and `false` otherwise. For an example of a class validating a sample constraint, please see listing 3.16 on page 51.

Constraint definition file

Furthermore, every application needs to provide a constraint definition file (ccDefinitions.xml) that is needed by the DeDiSys middleware to register the constraints and that covers additional information for each constraint. This file has to be specified in the XML format and amongst others, it has to provide the following properties for each constraint:

- **Context object**: The name of the class this constraint is validated in.
- **Type**: The type of the constraint, whether it is a precondition, a postcondition or an invariant.
- **Minimum satisfaction degree**: The lower boundary of the satisfaction degree that defines if constraint can be relaxed throughout the validation process.

- **Affected methods**: Every call to one of these methods trigger the validation of this constraint.

For a sample constraint definition file, please see listing 3.14 on page 50.

Within this thesis, a prototype was implemented to generate exactly this data (Java constraint validation classes and DeDiSys constraint definition file) in an automated way by using a model-driven approach. In more detail, this is done by a UML model of the application with corresponding constraint data defined on it and from which this information is being generated.

An example of an application using the DeDiSys middleware could be a flight booking system. A constraint within this context could validate that each flight must not be overbooked. In other words, this constraint could be specified in a way that the number of booked seats must not be greater than the number of the maximum available seats for each flight.

This section only covers the absolute minimum information about the DeDiSys middleware that is necessary to understand how it is related to this thesis. Further information about how it works in detail is not within the focus of this thesis and can be obtained from [Ded] and from the bibliography provided in the following section.

Further reading

- [FGO07] Lorenz Froihofer, Karl M. Goeschka, and Johannes Osrael. Middleware support for adaptive dependability. In *Middleware '07: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, pages 308–327, New York, NY, USA, 2007. Springer-Verlag New York, Inc

- [FOG07] Lorenz Froihofer, Johannes Osrael, and Karl M. Goeschka. Decoupling Constraint Validation from Business Activities to Improve Dependability in Distributed Object Systems. In *ARES '07: Proceedings of the The Second International Conference on Availability, Reliability and Security*, pages 443–450, Washington, DC, USA, 2007. IEEE Computer Society

- [Hor06] Markus Horehled. Integration of an EJB Constraint Consistency Management Framework into the JBoss Application Server. Master's thesis, Technikum Vienna, May 2006

- [Fuc06] Klaus Fuchshofer. Negotiation and reconciliation of consistency threats for Enterprise JavaBeans applications. Master's thesis, Technikum Vienna, May 2006
- [Ert07] Dominik Ertl. Evaluation of Partitionable Replication Protocol Improvements in an Enterprise JavaBeans Environment. Master's thesis, Technikum Vienna, October 2007
- [Bau06] Markus Baumgartner. Adaptive Constraint-Validierung in einer verteilten Enterprise JavaBeans Umgebung. Master's thesis, Technikum Vienna, September 2006
- [Rie07] Bernhard Rieder. Balancierung von Integrität und Verfügbarkeit in verteilten Web-basierten Enterprise JavaBeans Anwendungen. Master's thesis, Technikum Vienna, April 2007

2.2 Technical baseline

The practical part of this thesis is built upon some third party tools and libraries. In order to understand the upcoming parts of this thesis, these tools are covered in this section.

2.2.1 Dresden OCL toolkit

The Dresden OCL Toolkit is all about the Object Constraint Language (OCL). OCL is part of the well-known Unified Modeling Language (UML). It extends the UML's graphical notation with the possibility of adding more formally defined textual constraints on method invocations and on class structures as a whole. [Dre]

This toolkit was developed at the Software Technology Group of the Dresden University of Technology starting in 1999 and work on it is still ongoing by students and other scientific staff. It should not be considered as a stand-alone tool, but rather as a library that can be integrated into other projects to process UML models and OCL constraints, whereas there exist some tools within this toolkit that can be used as stand-alone tools like an OCL to SQL transformer. [Dre]

Currently, there exist the following three different version of this toolkit:

1. **Dresden OCL Toolkit**: This is the first version of this toolkit that is based on the OCL 1.1 Standard and by now is out of date and therefore should not be used anymore. Furthermore, this version of the toolkit is no longer documented on their website. [Dre]

2. **Dresden OCL2 Toolkit**: This is the second release of this toolkit that is based on the OCL 2.0 Standard and furthermore based on the NetBeans Meta-Data Repository (MDR) to store all models and meta-models. [Dre]

3. **Dresden OCL2 Toolkit for Eclipse**: This is the third and most recent version of this toolkit and based on the Eclipse SDK. Furthermore this version is based on a pivot model as exchange format for models and meta models that allows to connect to any repository and meta model. [Dre]

This thesis makes use of the second release of this toolkit (*Dresden OCL2 Toolkit*) for the reasons that it

- is based on the OCL 2.0 standard,
- provides all the functionality needed within this thesis,
- is not build upon the Eclipse SDK and therefore reduces dependabilities with the benefit of an easier integration of the toolkit into the prototype implemented within this thesis and
- is used within the legacy prototype that needs to be optimized and the implemented prototype is based on.

Within the *Dresden OCL2 Toolkit* many additional tools have been developed, which are not mentioned here. For further information about these tools or more detailed information about this toolkit in general, please refer to [Dre].

2.2.2 AndroMda

Developed by a core team of 15 persons spread all over the world, *AndroMDA* is a well-established tool and is all about transforming UML models to their corresponding deployable components for different platforms like

J2EE, *Spring* or even *.NET*. There are many ready-to-use cartridges available within *AndroMDA* that focus on many common development toolkits like *Axis*, *jBPM*, *Struts* and many others. Another feature within AndroMDA is that it is possible to create own cartridges or customize existing ones, to build custom code generators. [And]

Within this thesis, *AndroMDA* is used to create the EJB's from the classes defined within the UML model of the particular application using the *DeDiSys middleware*. For this purpose, the implemented prototype uses the *EJB2* cartridge shipped along with *AndroMDA*.

2.2.3 StringTemplate

StringTemplate is a basic template engine for Java, which allows to create data from template files. A template file in this case is a file that contains any text and is refined by specific placeholders. Later on, these placeholders are replaced with data assigned to the template at the time the template is requested to be transformed. A sample template file is illustrated in listing 2.3.

```
int i = $variable1$;
String result = "$variable2$";

$if (variable3)$
System.out.println(result + " = " + i);
$endif$
```

Listing 2.3: Example of the StringTemplate "Constraint.tpl"

Within the template file provided in listing 2.3, all strings, which are enclosed by dollar signs represent placeholders. These placeholders can either be placeholders for strings assigned to the template by a corresponding Java code or placeholders for simple control structures like *if expressions*. An example of basic Java code using *StringTemplate* and corresponding to the template given in listing 2.3 is illustrated in listing 2.4.

```
StringTemplateGroup stg = new StringTemplateGroup("myConstraints
    ", "./");
StringTemplate st = stg.getInstanceOf("Constraint");
st.setAttribute("variable1", 10);
st.setAttribute("variable2", "Result");
st.setAttribute("variable3", true);
System.out.println(st.toString());
```

Listing 2.4: Example of Java code using the StringTemplate "Constraint.tpl"

The sample Java code illustrated in listing 2.4 initializes a `StringTemplateGroup` named `myConstraints` and has the template directory set to `./` as the root of all template files. Within this directory, the template file `Constraint.tpl` is requested using the `getInstanceOf` method of the `StringTemplateGroup` class. After this initialization process of the template, arbitrary values can be assigned to template by calling the method `setAttribute` of the previously created instance of the `StringTemplate` class, whereas for every assigned value a placeholder has to be defined within the template file. At the point in time, where all necessary values are assigned, the template can be transformed by calling the `toString` of the `StringTempate` class that replaces all placeholders by their corresponding assigned values and returns the transformation result as string.

The resulting file, after applying *StringTemplate*'s transformation process is shown in listing 2.5.

```
int i = 10;
String result = "Result";

System.out.println(result + " = " + i);
```

Listing 2.5: Transformation result of the StringTemplate "Constraint.tpl"

Within this thesis, *StringTemplate* is used to support the implemented Code Generator (see section 3.2.1) and moreover, to support the creation of the constraint configuration file (`ccDefinitions.xml`) necessary to deploy the constraint to the *DeDiSys* middleware. To this end, it is providing the core structure of the Java constraint validation classes and the XML definition file as template files. Furthermore, by making such a separation of the file creation and the Code Generator, the source code of the implemented prototype becomes much more readable and maintainable.

For further information regarding *StringTemplate*, please refer to [Str].

2.2.4 ArgoUml

ArgoUML is a platform independent and open source UML modeling tool developed in Java. The current version of *ArgoUML* is *0.28* and it is available in ten different languages. It supports all UML 1.4 diagrams and therefore, it

provides the core functionality to model applications, including the possibility to export these models as their corresponding XMI representation. [Arg]

Hence, these models are also usable beyond *ArgoUML* and therefore, can be processed within the implemented prototype.

Furthermore, within this tool, there exist several interesting sub-projects like an AndroMDA (see section 2.2.2) plug-in or a database modeling tool. Figure 2.3 shows the graphical user interface of ArgoUML, including a sample flight booking application that was further developed in the practical part of this thesis.

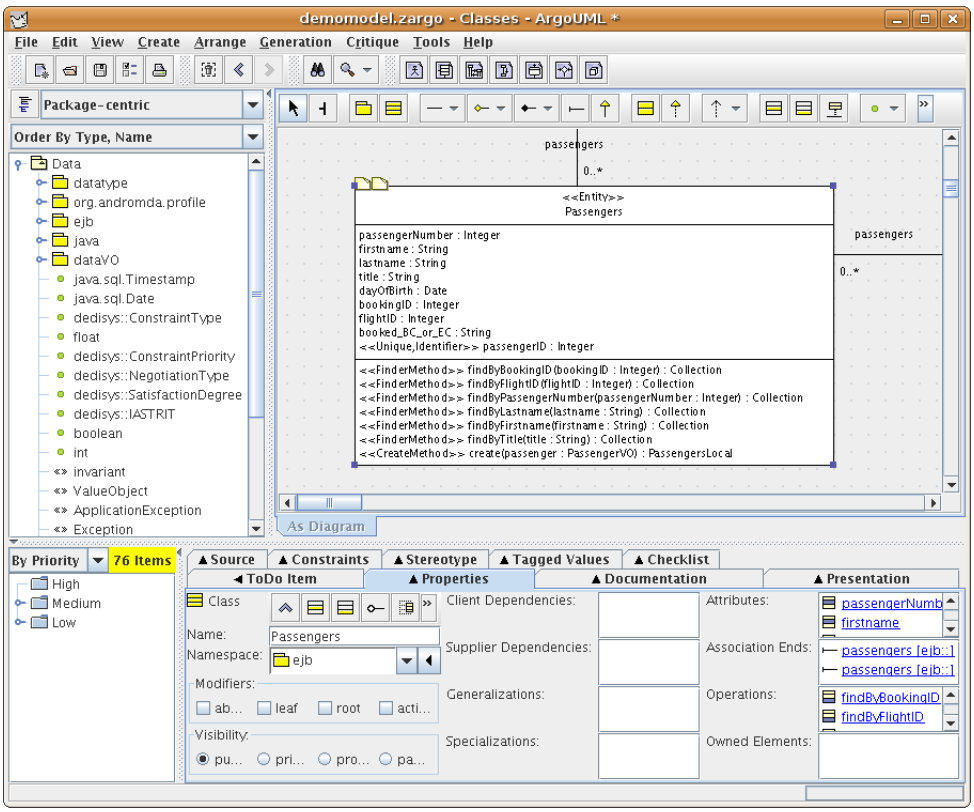


Figure 2.3: Graphical user interface of ArgoUML [Arg]

Within this thesis, *ArgoUML* is used to create the UML models of the applications that make use of the DeDiSys middleware (see section 2.1.3) and furthermore to define all OCL constraints with their corresponding constraint data, like type and degree of satisfiability of each constraint. For more information about *ArgoUML*, including a detailed feature list and the aforementioned sub-projects, please refer to [Arg].

2.3 Related work

This section introduces three major projects that have their focus on similar issues. The first project is about an OCL to SQL transformer that was carried out at the Dresden University of Technology. The second one focuses on the tuning and management of integrity constraints within web applications. Finally, a project regarding the transformation of OCL expressions to semantically equivalent but less complex ones builds the third and last part of this section.

2.3.1 Constraint tuning & management for web applications

[BC06] published a project in a similar area at the ICWE'06. This project titled *Constraint Tuning and Management for Web Applications*, has its main focus on the optimization of Integrity Constraints (IC) that are defined on the information base of web applications. A more exact definition of their work and provided by them is given as follows:

The main goal of this paper is to present a general framework to facilitate the integration of efficient integrity checking methods in web applications. This framework can be parameterized with the characteristics of a specific web application, at the purpose of offering the optimal set of techniques for implementing the verification of the imposed IC's. [BC06]

Within their implemented framework and similar to this work, they also start with a conceptual schema of the application and the IC's defined on this schema. Furthermore, they identify structural events, which can be mapped to affected methods within the *DeDiSys middleware*.

A structural event is an elementary change in the population of an entity type (i.e. a class) or relationship type (i.e. an association) such as: create object, update attribute, create relationship link, etc. Structural events are a way to define the effect of the operations appearing in the conceptual schemas. [BC06]

Moreover, within their work, they also state that many web applications offer only a small amount of data-management possibilities and therefore, not all kinds of structural events can be applied over the information base (e.g. relational database). This reduces the amount of necessary IC's within an application significantly, as the definition of IC's for data that cannot be transfered to an inconsistent state, is not necessary. Therefore, their

framework is divided into the following three steps:

1. Given the set of IC's, the framework analyses each IC to determine which kind of changes (structural event) to the information base may violate the IC.

2. Remove all potentially-violating structural events related to events that are not appearing in the model.

3. Considering the requirements and necessities of the web application, the framework recommends an implementation technique for each IC. For more information about these implementation techniques that make use of views and triggers, please refer to [BC06].

However, different to this thesis is that their framework transforms an IC as a whole to a corresponding SQL representation, which could be an SQL view, an SQL trigger or a construct that combines the two previously mentioned representations. In contrast, this thesis allows to split the IC and to transform the resulting parts independently to an SQL, to a Java or to a combined construct. At the end, these independent constructs are combined again. For more information about the splitting algorithm, please refer to section 3.

Furthermore, they do not provide an implementation of an IC to SQL transformer, but rather they only mention that the corresponding SQL representation of an IC can be obtained using already existing transformers like the like *Dresden OCL22SQL Transformer* and do not provide more information about this crucial part.

2.3.2 Transformation techniques for OCL constraints

[CT07] developed a tool to apply transformation techniques to OCL constraints as a means of reducing the complexity of these constraints. This means that these OCL constraints are transformed to semantically equivalent, but simpler ones. Moreover, the kind of how an OCL constraint is expressed is not unique, but rather an OCL constraint can be expressed in different ways, without changing the semantic of it.

The work of [CT07] is divided into two major parts, whereas the first part transforms the OCL expression into an equivalent, but simpler OCL expression and the second part tries to change the context of the OCL constraint as

a whole. Both of these parts are described in more detail in the following sections. These descriptions rely on a typical flight to passenger (on-to-many) relation, which is shown in figure 2.4.

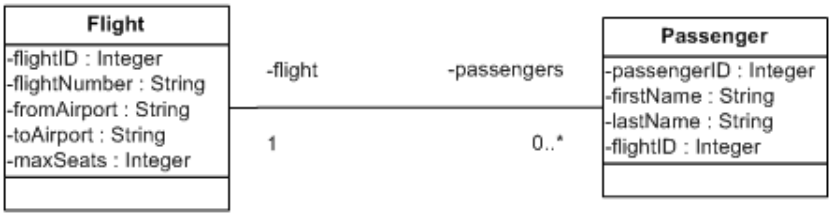


Figure 2.4: Sample UML model to illustrate the transformation techniques

Equivalences between OCL expressions

This first step of their work introduces several basic equivalences between OCL expressions, which are more efficient with respect to performance, when it comes to the validation of the constraint at run-time. These equivalences are categorized into equivalences for *boolean expressions*, *collection expressions* and *iterator expressions*. For a better understanding of this optimization step, table 2.1 provides one example of an equivalence for each category, whereas X and Y represent arbitrary OCL expressions.

Category	Equivalence
Boolean	$notX \leq Y \Longleftrightarrow X > Y$
Collection	$X \rightarrow isEmpty() \Longleftrightarrow X \rightarrow size() = 0$
Iterator	$X \rightarrow exists(Y) \Longleftrightarrow X \rightarrow select(Y) \rightarrow size() > 0$

Table 2.1: Examples of equivalences between OCL expressions

Besides these basic equivalences, this first part also tries to get rid of the use of OCL's `allInstances` operator that can occur within the body of the constraint. This is a major improvement to the performance of an OCL constraint when it is going to be validated at run-time. Furthermore, they apply rules to transform an OCL constraint to its corresponding conjunctive normal form, which completes the work done in this part. The following listing 2.6 illustrates an OCL constraint related to the UML model given in figure 2.4, which ensures that there always exists a flight from *Barcelona* to *Vienna* as long as flights are existing within the database.

```
context Flight inv FlightFromBarcelonaToViennaExists:
Flight::allInstances()->size() > 0
implies
Flight::allInstances()->exists(
  f | f.fromAirport = 'BAR' and f.toAirport = 'VIE'
)
```

Listing 2.6: Sample OCL constraint to show equivalences

Although the constraint of listing 2.6 does not look complex, there is still space left for optimization. For this purpose, listing 2.7 shows an optimized version of this OCL constraint, after applying the rules introduced within the first part of their work.

```
context Flight inv FlightFromBarcelonaToViennaExists':
Flight::allInstances()->size() <= 0
or
Flight::allInstances()->select(
  f: Flight | f.fromAirport = 'BAR' and f.toAirport = 'VIE'
)->size() > 0
```

Listing 2.7: Optimized OCL constraint showing equivalences

The outcome of this optimization is that both constraints have exactly the same semantic, but different performance when they are validated.

Changing the context type of an OCL constraint

The second and even more challenging part of their work examines the context in which a constraint is defined and how this context can be changed to another one in order to gain a better validation performance of the constraint.

As stated within their work, trying to change the context of an OCL constraint only makes sense for constraints that are defined using a single instance of the context type via the use of OCL's `self` variable. On the other side, it does not make sense for constraints that make use of OCL's `allInstances` operator, because in this case the body will stay the same independent of the chosen context type [CT07].

To better understand this optimization, listing 2.8 shows an OCL constraint also related to the UML given in figure 2.4.

```
context Flight inv minAge:
self.passengers->forAll(
  p | p.age > 10
)
```

Listing 2.8: Sample OCL constraint to show a context change

The OCL constraint illustrated in listing 2.8 ensures that every passenger is at least ten years old. This constraint can also be specified in the context of the `Passenger` class as shown in listing 2.9.

```
context Passenger inv minAge':
self.age > 10
```

Listing 2.9: Optimized OCL constraint showing a context change

Within this thesis, these transformation techniques are applied to all OCL constraints defined on the UML model, to support and ease the OCL transformation that was implemented within the prototype. Hence, it also improves the quality of the code that is resulting from the OCL transformer, with respect to performance.

The information provided in this section only gives an overview of their work, because this thesis does not focus on these transformation techniques. For detailed information about the different steps introduced within this work, please refer to [CT07]. Additional information regarding OCL constraint normalization and optimization algorithms can be found in [LLC05], [MLC06] and [LCG06].

2.3.3 Generating Query Language Code from OCL Invariants

[HWD08] published a work titled *A Framework for Generating Query Language Code from OCL Invariants* that is closely related to this thesis. Within their work they address the issue that most current approaches in *Model-Driven Software Development (MDSD)* only focus on transforming structural descriptions of software systems. For the reason, that these descriptions do not tackle all aspects of software systems, a plethora of other specification and modeling techniques exists. The OCL provides means to enrich models with detailed semantics in a formal way, which are unfortunately not preserved in current multi-staged transformation approaches. Therefore, they are lost during PIM to PSM transformation.

The Query Code Generation Framework addresses this issue by providing a general framework for mapping OCL invariants to declarative query languages and thereby enables Model-Driven Integrity Engineering. We focus on query languages, because data in business systems is mostly managed by systems that are accessible through query languages (e.g. database systems). [HWD08]

The architecture of their framework is divided into the following three modules:

- **AST model generation:** This module is responsible for reading the UML model enriched with OCL constraints and generating the corresponding AST model. Further information can be found in [LO04] and [DHK05]
- **Model transformation framework:** This module performs the transformation of the UML model to the target data schema. Within the Query Code Generation Framework, models appear on several abstraction levels. This can be *UML models* describing domain concepts platform independently, *Common Warehouse Meta-model (CWM)* describing data schemas or *Schema Facade models* providing a generic interface to these data. For this purpose, there are arbitrary model transformation necessary to mediate these levels of abstraction.
- **OCL transformation framework:** This module maps OCL invariants to equivalent sentences in declarative query languages to preserve the semantic constraints across the different abstraction levels. These expressions are used to ensure integrity rules in the platform specific data schema.

Furthermore, within their framework they present two examples namely the *OCL2SQL* and *OCL2XQuery* where they apply their introduced concepts to UML models that are enriched by OCL constraints. The UML models are mapped to platform-specific data schemas while the OCL constraints are transformed to their corresponding query language representation.

However, different to this thesis is that their Query Code Generation Framework also generates the database table structure of the UML model not needed within this thesis for the reason that an Object-Relational-Mapping library already provides this task. Furthermore, this framework creates its own primary key fields, while creating the database table structure and ignores primary key fields that are already defined within the UML model of

the application. Finally, it also creates one SQL view per OCL constraint, whereas this view is built upon the generated SQL code. Within this thesis an OCL constraint is not transformed as a whole to SQL, but rather it is possible to transform parts of the constraint (referred to as OCL expressions) to Java and other expressions to SQL.

Chapter 3

Realization

This chapter provides a detailed description of the practical part of this thesis. The first part deals with an architectural comparison of the legacy prototype and the implemented prototype. Afterwards, a deeper look at the core parts of the implemented prototype is taken. A sample application of the transformation process builds the third major part of this chapter.

3.1 Architecture

This section covers a detailed description and comparison of the architecture of the legacy prototype and the architecture of the prototype implemented within this thesis. Therefore, the first part of this section provides an overview of both architectures and shows their similarities, whereas afterwards each architecture is described itself.

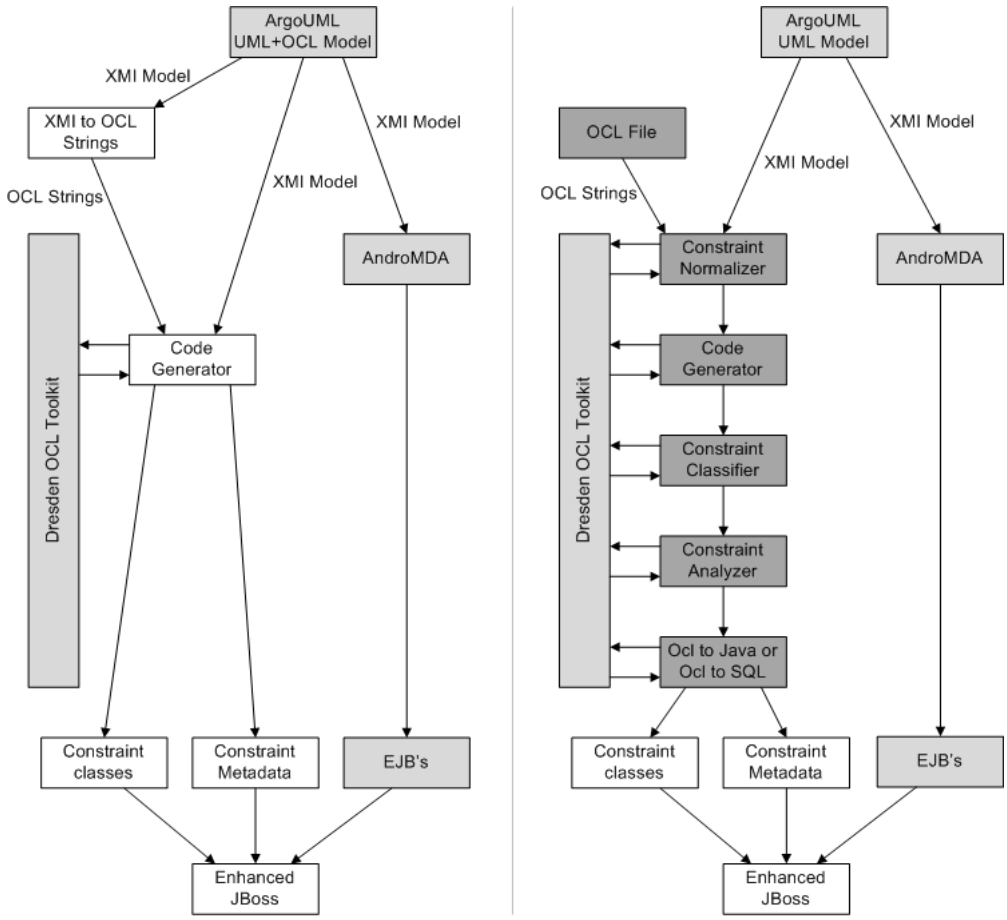
3.1.1 Architecure overview

As aforementioned, the prototype implemented within this thesis is built upon a legacy prototype. The main focus of the legacy prototype is on the transformation of OCL constraints to their corresponding Java constraint validation code and furthermore, to create a specific constraint definition file named `ccDefinitions.xml` necessary to register the Java constraint classes at the *DeDiSys middleware* (see section 2.1.3).

The implemented prototype has the same focus as the legacy prototype with the extension of optimizing the code generation step to generate more efficient

Java constraint validation code with respect to the run-time performance of the constraints. Thus, these two prototype share a base structure of their architecture.

Figure 3.1 shows both architectures, whereas the architecture given in sub-figure 3.1(a) is related to the legacy prototype and the architecture provided in sub-figure 3.1(b) belongs to the implemented prototype.



(a) Legacy prototype (b) Implemented prototype

Figure 3.1: Architectures of both prototypes

Both of the architectures given in figure 3.1 consist of different colored elements, whose meaning is as follows:

- **White:** These elements represent modules that are implemented within

the particular prototype.

- **Light grey:** These elements depict external/third-party modules that are integrated into the particular prototype.
- **Dark grey:** These elements only exist within the architecture of the implemented prototype and show the improvements made in contrast to the architecture of the legacy prototype or respectively show where improvements have been made.

Furthermore, on top of both provided architectures there is the UML model that can be considered as the entry point to the architecture. An example for such a UML model could be a flight booking system, with its necessary classes `Flight`, `Passenger`, `Booking` and others. This model that is created using ArgoUML, is enriched by OCL constraints and additional constraint meta-data that is specified directly within the UML model[1]. Constraint meta-data in this context are for example the type of the constraint, regarding precondition, postcondition and invariant, or the satisfaction degree of the constraint, defining if the constraint can be relaxed throughout degraded mode or if it has to hold at every point of time. This constraint meta-data is described in detail in section 2.1.3.

To be able to process and use the UML model in further steps, it is necessary to export this model to its corresponding XMI representation. This XMI model holds the complete UML model using the XML format that make is possible to traverse through the model in the code generation part of the prototype. The functionality to export a UML model to its corresponding XMI representation is provided within every professional UML tool.

At this point of time, there are the following two different paths to follow in each architecture:

- **Path 1:** The OCL constraint transformation that is drawn on the left hand of each architecture
- **Path 2:** The EJB generation that is drawn on the right hand side of each architecture

[1]This is the optimal place to specify the OCL constraints. Within the new architecture, the OCL constraints have to be defined in a separate text file due to restrictions within ArgoUML.

3.1.2 Legacy prototype

The upcoming explanations are based on the architecture shown in sub-figure 3.1(a) of figure 3.1.

OCL constraint transformation

This first part within this architecture transforms the OCL constraints defined within the UML model to their corresponding Java constraint validation code and generates the necessary constraint definition file called `ccDefinitions.xml`

For this purpose, the XMI representation is taken and the the OCL constraints are extracted from it and saved to an simple text file, because the *Dresden OCL Toolkit* needs these constraints in a separate file. Afterwards these files (XMI and constraint file) are handed over to the *Dresden OCL Toolkit* (see section 2.2.1), which performs the following tasks for each OCL constraint:

- Parse the OCL constraint and
- generate the AST from the CST of the OCL constraint

After the previous steps are passed successfully, which means that no syntax error has occurred while parsing the OCL constraints and the AST representation of each constraint could have been generated without an error being raised, the code generator is responsible to generate the necessary Java constraint validation code and the constraint definition file.

Generation of the Enterprise Java Beans

The second part within this architecture generates the Enterprise Java Beans (EJB's) for the classes defined within the UML model using the open source library AndroMDA (see section 2.2.2). For this purpose, the *EJB2 cartridge* of AndromMDA is used to generate the necessary interfaces and moreover, the Bean implementations.

3.1.3 Implemented prototype

The upcoming explanations are based on the architecture shown in sub-figure 3.1(b) of figure 3.1.

OCL constraint transformation

This first part within this architecture has the same focus as it has within the legacy prototype. That is, to transform the OCL constraints defined within the UML model to their corresponding Java constraint validation code and to generate the necessary constraint definition file named `ccDefinitions.xml`.

In contrast to the architecture of the legacy prototype, the OCL constraints are no longer extracted from the XMI representation of the UML model. This is because they can not be specified within the UML model any longer for the reason that *ArgoUML* does not support OCL's `allInstances` operator, which is essential for this thesis.

Furthermore, this architecture includes many improvements in comparison to the architecture of the legacy prototype that are related to the code generator. The simple code generator of the legacy prototype was extended by additional modules like the *constraint normalizer* and the *constraint classifier* aimed to generate more efficient Java constraint validation code with respect to the run-time performance.

Detailed information about each module/step of this transformation process is provided in section 3.2.

Generation of the Enterprise Java Beans

The second part within this architecture has the same responsibilities as in the legacy prototype and thus, for more information please refer to section 3.1.2.

3.2 Core parts

Within this section, the core parts responsible for transforming an OCL constraint to its Java class code are described in detail. Furthermore, this section provides detailed information about the interaction between these core parts.

3.2.1 Code generator

The first part of the implemented prototype that needs to be mentioned is the code generator. This generator, as the name already indicates, is responsible to generate a Java class for each OCL constraint defined within the UML model that validates this OCL constraint within the *DeDiSys middleware* (see section 2.1.3). For this purpose, it is supported by a separate transformer that transforms the OCL constraints to their corresponding Java code.

Furthermore, this code generator within the prototype is used as a controller class. This means that it is the base class of the prototype that provides functionality to invoke all tasks necessary to complete the whole transformation process. This starts at loading the UML model and ends up with the generation of the Java constraint classes.

For a better understanding of the tasks invoked in the code generator, figure 3.2 provides a sequence diagram that describes the work performed in more detail. Please note that within this sequence diagram a constraint can be classified into the three different categories *Intra-instance*, *Inter-instance* and *Type-level*, whereas the first category forces the OCL to Java transformation and the other two categories force the OCL to SQL transformation. For a detailed description about this classification, please refer to section 3.2.3.

As illustrated in figure 3.2 the code generator handles the tasks described in the following sections.

Initialize the transformation process

At the beginning of the transformation, the following two major configuration elements need to be initialized:

- The path to the file covering the XMI representation of the the UML model.
- The path to the file covering the OCL constraints.

This is all that needs to be defined before starting the generation process.

Load the UML model

As mentioned earlier, the UML model needs to be exported as its corresponding XMI representation that is handed over to the code generator in

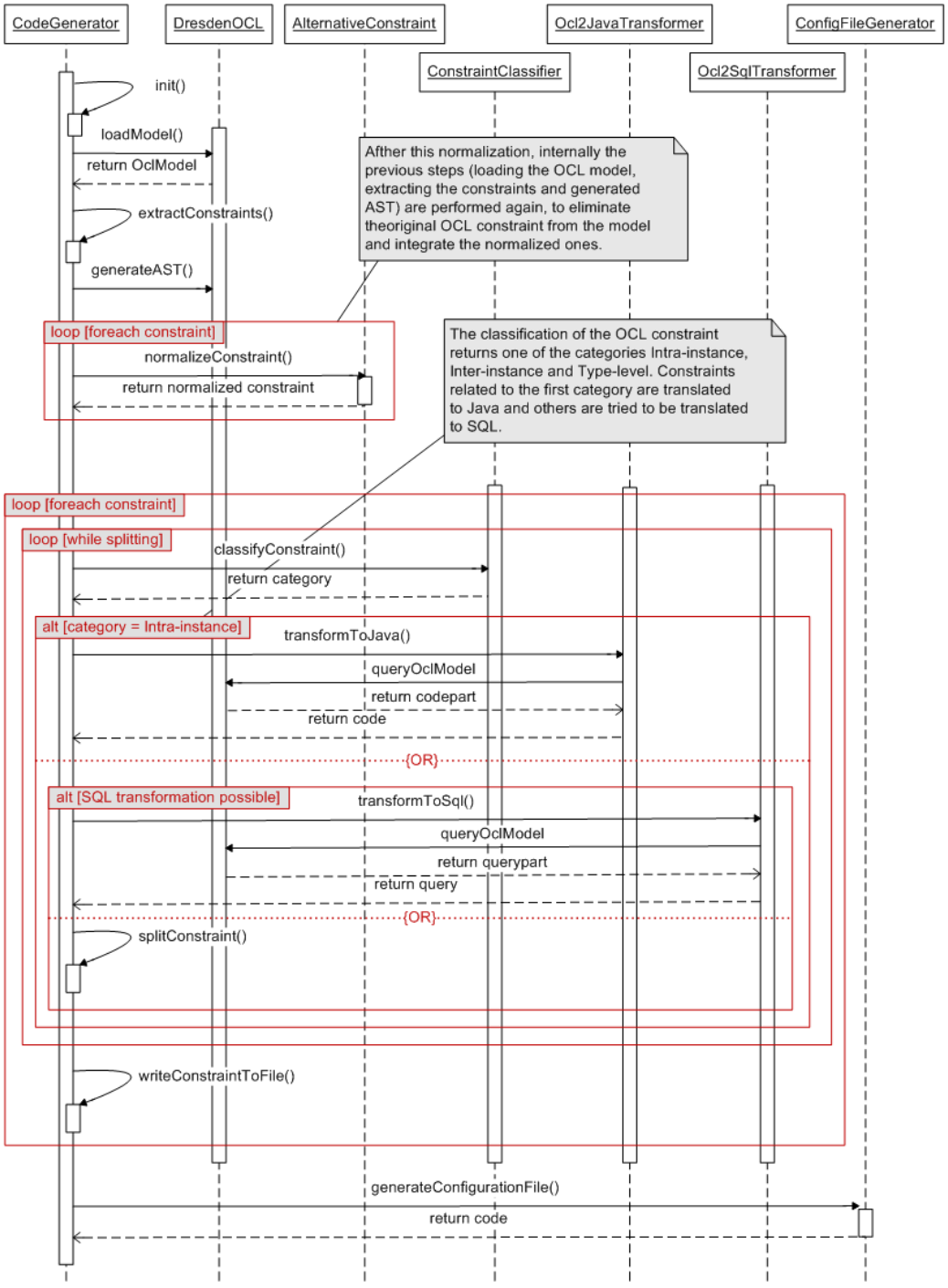


Figure 3.2: Sequence diagram of the code generator

the previous initialization step. This can be done with almost every UML modeling tool and so it is also possible with *ArgoUML* that is used within this thesis. Afterwards, the following three tasks are performed with the help of the *Dresden OCL toolkit*:

1. Load the model into the internal NetBeans meta-data repository (MDR).
2. Generate the abstract syntax tree (AST) representation from the concrete syntax tree (CST) representation of each OCL constraint.
3. Query the loaded UML model to feed application specific needs. The application specific need for this thesis are, to generate the Java constraint validation code based on the result of querying the UML model.

This process is illustrated in figure 3.3.

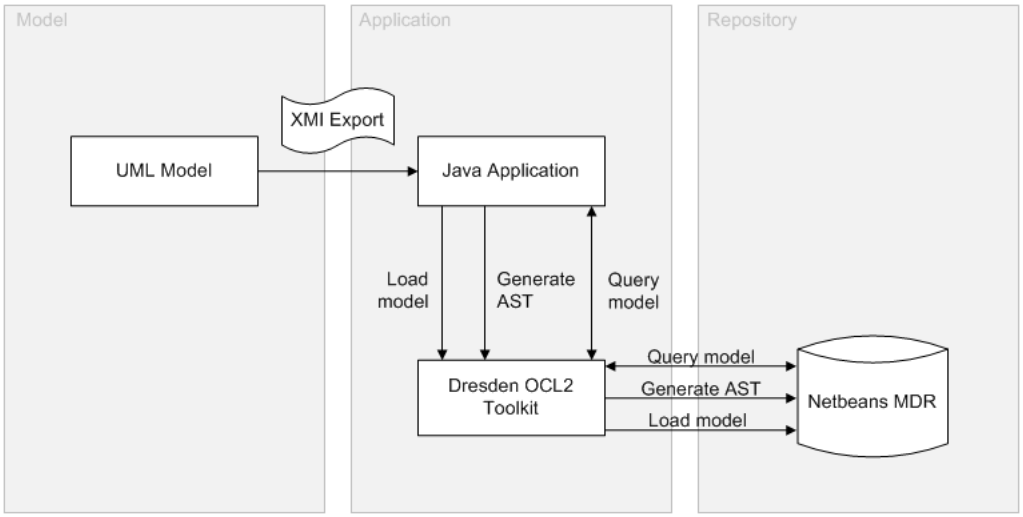


Figure 3.3: Integration of the Dresden OCL2 Toolkit

Import of OCL constraints

Normally, the OCL constraints related to a UML model are defined directly within the model, which means that they are specified within the used modeling tool, as it was done within the legacy prototype.

Within this thesis, it is not possible anymore to follow this scenario for the reason that the modeling tool *ArgoUML* is used, which has a lack of support

for OCL's `allInstances` operator. As this operator is essential for this thesis, the OCL constraints are no longer specified within the modeling tool, but rather are provided to the implemented prototype in a separate text file.

Listing 3.1 illustrates a sample file of defined constraints. Please note that every constraint needs to be surrounded by its package definition, even if they are all related to the same package.

```
# Constraint checking for valid passengers
package constraints context Passenger inv ValidPassenger: self.
    firstname <> '' and self.lastname <> '' endpackage

# Constraint checking for valid bookings
# package constraints context Flight inv ValidBookings: self.
    maxSeats >= self.passengers->size() endpackage

# Constraint checking for valid flights
package constraints context Order inv ValidFlights: Flight::
    allInstances()->forAll (f1, f2: Flight | (f1.flightID <> f2.
    flightID) implies (f1.flightNumber <> f2.flightNumber))
    endpackage
```

Listing 3.1: Definition of the OCL constraints in external text-file

Furthermore, the extraction routine allows to specify empty lines in this file as well as comments. Lines that should be interpreted as comments need to be preceded by a hash, whereas this hash needs to be the first character in this line. Hence, it is also possible to exclude OCL constraints from the transformation process, if they should not be taken into account while generating the Java constraint validation classes.

Within the legacy prototype, it was possible to declare the OCL constraint within *ArgoUML*, because it does not make use of OCL's `allInstances` operator. In contrast, it uses a dummy relation as a workaround that emulates this operator (see section 3.3.1).

Load the OCL constraints

After the OCL constraints are extracted from the text-file, they are integrated into the already loaded UML model, using the *Dresden OCL Toolkit*. Therefore, every OCL constraint needs to go through the following steps:

- Parse the OCL constraint to check it against the syntax of the OCL.

- Generate the AST representation from the CST representation of the OCL constraint.
- Integrate the OCL constraint to the loaded UML model, to be able to traverse trough it in the transformation step.

Normalize the OCL constraints

In this step, the loaded OCL constraints are transformed to a common form like a normal form. For this purpose, the implemented prototype makes use of an external library developed by [BC06]. An introduction to this library is given in section 2.3.2 on page 19.

Hence, the complexity of the handcrafted OCL constraints is reduced, which has a direct impact on the upcoming transformation process.

Transform the OCL constraints

After the previous steps are passed successfully, the code generator iterates over the OCL constraints and tries to transform each of them separately. These constraints are either transformed to their corresponding Java code or SQL query, depending on the constraint body. Therefore, the constraints need to be classified into a specific category before being transformed. For the moment, it is enough to distinguish between a *simple* OCL constraint that is transformed to its Java pendant and a *complex* OCL constraint that is transformed to its SQL representation.

The detailed description of the classification algorithm, which is not necessary to understand the transformation algorithm, is provided in following subsection 3.2.3. The transformation algorithm is illustrated using the flowchart given in figure 3.4.

Please note that the transformation algorithm illustrated in figure 3.4 does not make a distinction between an OCL constraint and an OCL expression. The term OCL expression is used in this figure, because the algorithm does not only transform the whole OCL constraints, but also parts of the OCL constraints, referred to as OCL expressions. Furthermore, within this algorithm it is assumed that every OCL expression can be transformed to its corresponding Java code, whereas it is not possible transform each OCL expression to its corresponding SQL query.

In case of that a complex OCL constraint cannot be transformed to SQL at

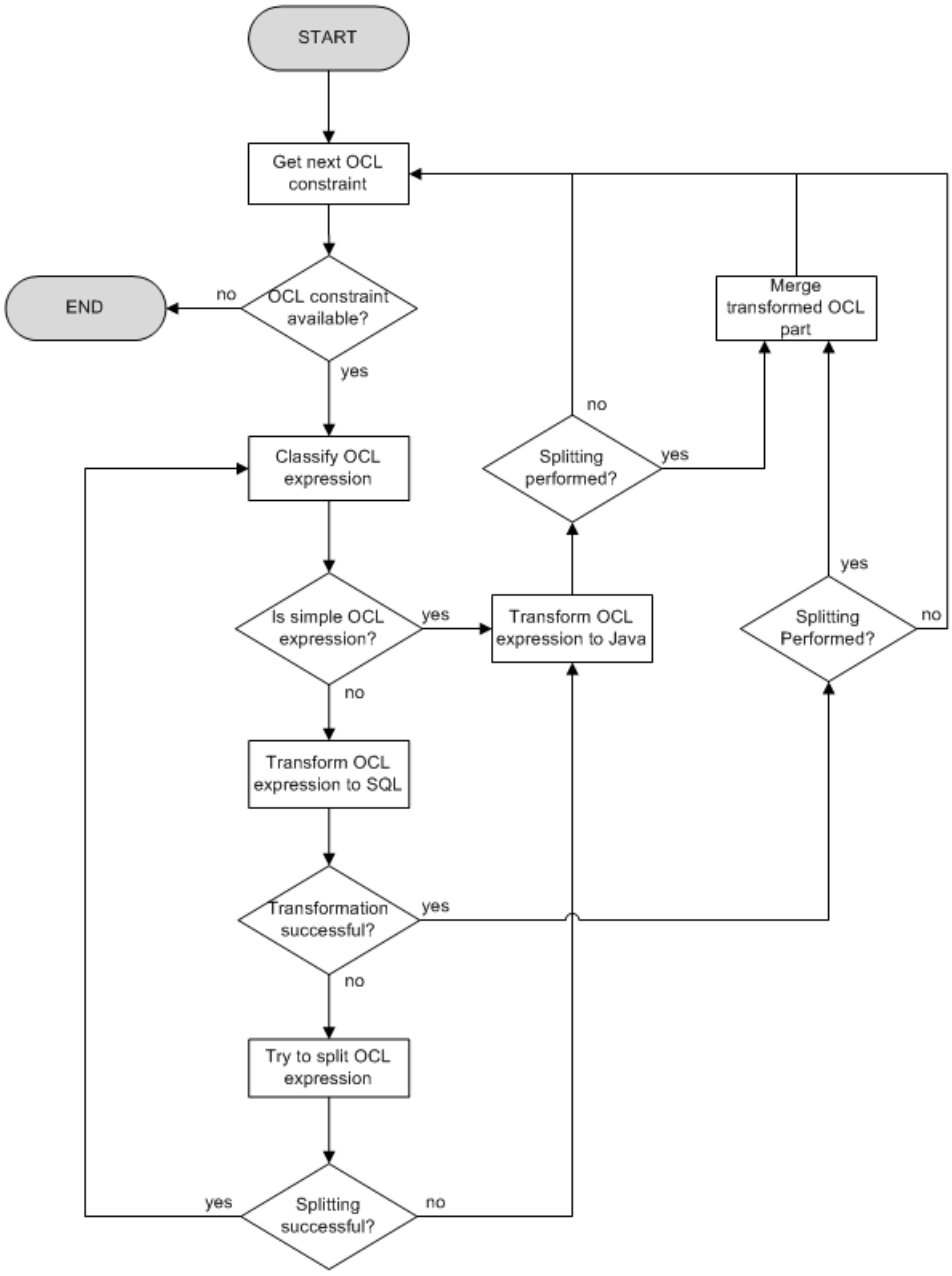


Figure 3.4: Transformation process handled by the code generator

least partly, the transformation to Java is used as the default fall-back and moreover, a warning is created. This warning states that the OCL constraint should be transformed to a semantically equivalent one if possible, to be able to transform it using the implemented OCL to SQL transformer.

The transformation algorithm starts by taking the next available OCL constraint and classifying it. In case of a *simple* OCL expression, this expression is transformed to its Java pendant. In case of a *complex* OCL expression, the code generator tries to transform this expression to the corresponding SQL pendant. If it is not possible to transform this expression to SQL as a whole [2], the algorithm tries to split this expression into two parts. Splitting of the OCL expression is allowed by the transformation algorithm if the top level operator is `and`, `or` or `implies`, and therefore this expression consists of two inner expression that can be transformed independently. Afterwards, it applies the previous steps recursively to both expression parts separately, starting at the classification of the OCL expression. After this separate transformation of the split OCL expressions, they have to be merged again at the end of the previous iteration, using the operator, the split was based on.

Using this transformation algorithm, it is possible to generate a Java class that validates parts of the OCL constraint on the object layer using procedural Java code and other parts on the database layer by setting up direct SQL queries to the database. As mentioned before, the results of these separate validation steps need to be combined again to get the result if the whole constraint has been satisfied or violated.

Generate the constraint definition file

At the end of the transformation process implemented within the prototype, the constraint definition file named `ccDefinitions.xml` is created that is necessary to deploy the generated Java constraint validation classes to the *DeDiSys middleware*, which itself runs within the *JBoss application server*.

The information the constraint definition file is built upon is gathered from the UML model. Therefore, every OCL constraint needs to have a corresponding entry within the UML model, providing this constraint specific information.

For an example of a constraint definition file, please see listing 3.14 on page 50.

[2] The OCL to SQL transformer is not yet able to transform OCL's iterate and if expressions. Furthermore, it is not able to handle complex relationships.

3.2.2 Constraint normalizer

Another major part within the transformation process is the OCL constraint normalizer. The main goal of it is to reduce the complexity of OCL constraints in order to generate faster Java constraint validation code with respect to performance. For this purpose, the OCL constraint normalizer transforms the OCL constraints into semantically equivalent, but simpler ones. Hence, the OCL to Java or OCL to SQL transformation benefits from this normalization for the reason that simple OCL constraints can be mapped to more efficient Java code.

Within the implemented prototype, an external tool that fulfills this normalization task was integrated. The tool itself was introduced by [BC06] in their paper called *Transformation techniques for OCL constraints*. A brief description of their work is given in section 2.3.2.

3.2.3 Constraint classifier

In order to decide in an automatic way, whether an OCL constraint should be transformed to its corresponding Java code or SQL query, it is necessary to classify the OCL constraint into a predefined category. This is done by an OCL constraint classifier that analyzes the body of each OCL constraint separately and classifies it into different categories. The algorithm for the classification of an OCL constraint introduced by [BC06] and implemented within this prototype, distinguishes between the three different categories *Intra-instance*, *Inter-instance* and *Type-level*. These categories are described in more detail in the following paragraphs.

For a better understanding of the classification algorithm the description of every category includes an example of an OCL constraint that is related to it. These OCL constraints are based upon the sample class diagram provided in figure 3.5.

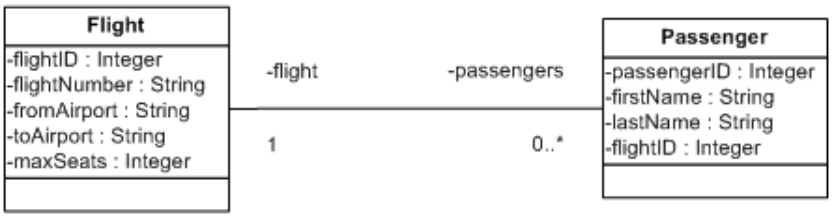


Figure 3.5: Sample UML model to illustrate the constraint classification

Category Intra-instance

OCL constraints that are restricting the values of attributes of a single object instance build this category. These constraints do not check the consistency of the whole database, but rather validate the consistency of the underlying data based on the changes made by the provided context object. Thus, these constraints are referred to as *Limited consistency check (LCC) constraints*. An example of an OCL constraint representing this category is illustrated in listing 3.2.

```
context Passenger inv ValidPassenger:
self.firstName <> '' and self.lastName <> ''
```

Listing 3.2: Example of an Intra-instance constraint

To programmatically classify a constraint into this category, the following rule has to be applied: OCL constraints defined using OCL's `self` variable and not referencing any relationship type are considered to be related to the category *Intra-instance* [BC06].

Category Inter-instance

This category contains all OCL constraints restricting the relationships between an entity and other entities, which means object instances of different entity types. As for the category *Intra-instance*, these constraints also do not check the consistency of the whole database, but rather validate the consistency of the underlying data based on the changes made by the provided context object. Hence, also these constraints are referred to as LCC constraints. An example of an OCL constraint representing this category is shown in listing 3.3.

```
context Flight inv ValidBookings:
self.passengers->size() <= self.maxSeats
```

Listing 3.3: Example of an Inter-instance constraint

To programmatically classify a constraint into this category, the following rule has to be applied: Integrity constraints defined using OCL's `self` variable and not satisfying the rule defined in the *Intra-instance* category are considered to be related to the category *Inter-instance* [BC06]. This rule was refined by stating that constraints of this category are not allowed to use OCL's

`allInstances` operator, because the construct `self.allInstances()` is valid within the *Dresden OCL Toolkit*. This construct conflicts with the definition of this category and further more, constraints using this construct are no long LCC constraints.

Category Type-level

OCL constraints that are restricting a set of objects of the same entity type build this category. Constraints of this category using OCL's `self` variable are checking the consistency of the underlying data like within the previous categories based on the changes made by the provided context object. In contrast, constraints defined without OCL's `self` are validating the consistency of the whole database. Thus, these constraints are referred to as *Full consistency check (FCC) constraints*. An example of an OCL constraint representing this category is provided in listing 3.4.

```
context Flight inv FlightFromBarcelonaToViennaExists:
Flight::allInstances()->select(
  f | f.fromAirport = 'BAR' AND f.toAirport = 'VIE'
)->size() > 0
```

Listing 3.4: Example of a Type-level constraint

To programmatically classify a constraint into this category, the following rule has to be applied: Integrity constraints not satisfying the rules defined in the *Inter-instance* and *Intra-instance* category are considered to be related to the *Type-level* category. These integrity constraints require the use of OCL's `allInstances` operator [BC06].

Within the implemented prototype, OCL constraints or OCL expressions classified into the *Intra-instance* category are transformed to their corresponding Java constraint validation code, because in that case the validation of the constraint can be done in an efficient way on the object layer, with respect to the run-time performance. All other OCL expressions are handed over to the OCL to SQL transformer, which tries to generate the corresponding SQL representation of the OCL expression.

This transformation process is described in more detail in the following sections 3.2.5 and 3.3.

3.2.4 Constraint analyzer

The purpose of the OCL constraint analyzer, as another major part of the implemented prototype is to analyze whole OCL constraints, as well as parts of the OCL constraints, referred to as OCL expressions. To this end, it traverses through the XMI representation of the UML Model of a specific application in order to extract the necessary information from it.

Examples for such information that need to be extracted are:

- **Determine type of SQL query:** If an OCL constraint should be transformed to its corresponding SQL representation, the analyzer gathers the information whether it is a *size restricting constraint* or a *value restricting constraint*. A size restricting constraint restricts the number of rows it returns. In contrast, a value restricting constraint is not interested in a specific number of rows fulfilling the constraint, but rather states that the constraint needs to hold for all rows. For further information about these constraint types, please refer to the OCL to SQL transformer that is described in detail in section 3.3.2.
- **Determine top level operator:** In order to split an OCL expression into two parts, to later on transform these parts independently, using the implemented OCL transformers, it is necessary to determine the top level operator, because the implemented code generator can split OCL expressions only if the top level expression is `and`, `or` or `implies`.

3.2.5 Constraint transformer

Another major part within the implemented prototype is the OCL transformer that transforms the defined OCL constraints to their corresponding Java constraint validation classes. For this purpose, two different OCL transformers have been implemented, whereas one transforms OCL constraints to their Java pendant and another one transforms OCL constraints to their corresponding SQL representation.

These OCL transformation is described in detail in the following section 3.3.

3.3 OCL transformation

This section covers the transformation of a single OCL constraint to its target language. At the moment, there are two target languages to which an OCL constraint can be transformed, namely *Java* and *SQL*. Both of these transformers inherit from a common abstract base class, which consists of methods that are needed in both transformers. This implementation is structured in a way that new transformers implementing support for other target languages can be integrated easily, by just extending the aforementioned base class and implement all abstract methods, defined within it. An excerpt of this base class covering the methods that need to be implemented, is illustrated in listing 3.5.

```
public abstract class ATransformer {
  ...
  public abstract CodeContainer transform(OclExpression
      expression, String resultVariableName) throws MdaException;
  protected abstract CodeContainer codeOperationCallExp(
      OperationCallExp expression, boolean assignResult) throws
      MdaException;
  protected abstract CodeContainer codeIteratorExp(IteratorExp
      expression) throws MdaException;
  protected abstract CodeContainer codeAttributeCallExp(
      AttributeCallExp expression) throws MdaException;
  protected abstract CodeContainer codeIfThenElse(IfExp
      expression) throws MdaException;
  protected abstract CodeContainer codeVariableExp(VariableExp
      expression) throws MdaException;
  protected abstract CodeContainer codeAssociationEndCallExp(
      AssociationEndCallExp expression, boolean assignResult)
      throws MdaException;
  protected abstract CodeContainer codeIterateExp(IterateExp
      expression) throws MdaException;
}
```

Listing 3.5: Base class of implemented transformers

The following sections describing the implemented transformers make no distinction between the term *OCL constraint* and *OCL expression*, because - as mentioned earlier - OCL constraints are eventually split into parts that are transformed separately. These parts are referred to as *OCL expressions*.

3.3.1 OCL to Java transformation

The first transformer that was integrated into the prototype, is an OCL to Java transformer, which validates the OCL constraints on the object layer. As there already was a version of this transformer implemented within the legacy prototype, it was not necessary to implement this transformer from scratch. In contrast, it was possible to take this transformer as base by continuously extending it.

This transformer generates the corresponding Java constraint validation code to an OCL expression. In order to achieve this, it traverses through the AST representation of the OCL expression and maps these expressions to their Java pendant. For a sample transformation, please refer to section 3.4.

Enhancements to this transformer

This OCL to Java transformer was enhanced by some essential parts that are described in more detail in the following paragraphs.

Support for relations Within the legacy prototype, it was not possible to transform OCL expressions that query over relations for the reason that this prototype had a slightly different focus as the implemented prototype. The legacy prototype was meant to give a proof of concept of an MDA approach to generate the Java constraint classes. In contrast, the implemented prototype is based on this proof of concept and focuses completely on the optimization of the run-time performance of the generated Java constraints validation code. An example for an OCL constraint that uses relations defined within the UML model is illustrated in listing 3.6.

```
context Flight inv JohnDoeExists:
self.passengers->select(
  p | p.firstName = 'John' and p.lastname = 'Doe'
)->size > 0
```

Listing 3.6: OCL constraint querying a relation

The constraint specified in listing 3.6 expresses that each `Flight` must have at least one `Passenger` named *John Doe*. For this purpose, this OCL expression starts from the a `Flight` context and queries its corresponding `Passengers` using the defined relation between these two elements.

Hence, support for this type of OCL constraints was implemented within the OCL to Java transformer.

Support for OCL's `allInstances` operator OCL' `allInstances` operator was not fully supported within the legacy prototype due to a lacking support of *ArgoUML* for this operator. To use this operator, it was necessary to define a dummy many-to-many relation for each table that was pointing to itself. Thus, it was possible to retrieve all records of a certain table. A sample table with this relation is illustrated in figure 3.6.

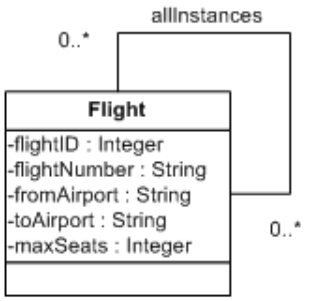


Figure 3.6: Present allInstances workaround

With this in mind, a deeper look was taken at this lacking support for the `allInstances` operator. This issue was resolved by the trade-off, to use an external file for specifying the OCL constraints, but therefore having full support for this operator. Moreover, the dummy many-to many relation could be removed, as it is no longer needed.

Support for OCL's `oclIsKindOf` operator Another operator that was not supported within the legacy prototype, is OCL's `oclIsKindOf` operator that checks, if specific instances within an OCL expression are from a specific type. Hence, support for this operator was implemented.

Optimization of constraints checking uniqueness Additionally, an optimization was introduced, which examines if an OCL constraint is checking the uniqueness of specific data within the database. For this type of constraints an handcrafted Java validation code template was created, which is parameterized with the field names that should be validated against uniqueness. An example for such a constraint that is referred to as *uniqueness constraint* is provided in listing 3.7.

```
context Flight inv flightTestForAll1:
Flight::allInstances()->forAll(f1, f2: Flight |
  f1.flightID <> f2.flightID implies
  f1.flightNumber <> f2.flightNumber
)
```

Listing 3.7: Example of a constraint checking uniqueness

This constraint assures that the `flightNumber` of the class `Flight` is unique within the database. For this purpose, this constraint states that whenever the `flightID` is unique then also the `flightNumber` has to be unique.

3.3.2 OCL to SQL transformation

The second transformer integrated into the prototype is an OCL to SQL transformer, validating the OCL constraint at the database layer. This transformer was implemented from scratch and is able to transform simple OCL expressions to their corresponding SQL queries, whereas two different types of OCL constraints have to be distinguished. These types that have to be handled in different ways are described in the following paragraphs.

Size restricting OCL constraints

OCL constraints related to the first type restrict the number of rows returned by a specific OCL constraint. This means that the top level operator of it is OCL's `size` operator. An example for such an OCL constraint is illustrated in listing 3.8.

```
context Flight inv FlightConfirmed:
self.passengers->size() > 0
```

Listing 3.8: Size restricting OCL constraint

An OCL constraint like the one provided in listing 3.8 results in the SQL query given in listing 3.9.

```
select count(p.passengerID) as total
from Flight f
inner join Passenger p on (f.flightID = p.flightID)
group by f.flightID;
```

Listing 3.9: SQL code of a size restricting OCL constraint

This resulting SQL query performs the necessary joins of the related tables and counts the found rows regarding to the performed grouping within the `group by` clause. As this query only returns the number of rows found for each group, additionally each row count needs to be compared against the provided value. This is done by the Java code illustrated in listing 3.10 that sets up the query to the database.

```
boolean result;
try {
  PreparedStatement stmt = con.prepareStatement(query);
  ResultSet rs = stmt.executeQuery();

  result = true;
  while (rs.next()) {
    if (!(rs.getInt("total") > 0)) {
      result = false;
      break;
    }
  }
}
catch (Exception e) {
  throw new ConstraintUncheckableException(e);
}
```

Listing 3.10: Java code of a size restricting OCL constraint

Value restricting OCL constraints

The second type of OCL constraints that have to be distinguished, are *value restricting OCL constraints* that are not interested in a specific number of rows fulfilling the constraint, but rather state that the constraint needs to hold for all rows. An example for such an OCL constraint is illustrated in listing 3.11.

```
context Flight inv minAge:
self.passengers->forAll(
  p | p.age > 10
)
```

Listing 3.11: Value restricting OCL constraint

An OCL constraint like the one provided in listing 3.11 results in the SQL query given in listing 3.12.

```
select 1
from Flight f
inner join Passenger p on (f.flightID = p.flightID)
where not (p.age > 10);
```

Listing 3.12: SQL code of a value restricting OCL constraint

Also this resulting SQL query performs the necessary joins of the related tables, but in contrast to *size restricting OCL constraints*, it is not necessary to count the found rows, because as aforementioned, the OCL constraint needs to hold for all rows. Thus, this query returns all rows that do not fulfill the query. This is achieved by using a `where not` clause meaning that the `where` clause is followed by the unary operator `not`. In that way, the only thing left to do for the corresponding Java code is to check, if the number of found rows is greater than zero. This is performed by the Java code illustrated in listing 3.10.

```
boolean result = false;
try {
  PreparedStatement stmt = con.prepareStatement(query);
  ResultSet rs = stmt.executeQuery();
  result = !rs.next();
}
finally {
  con.close();
}
```

Listing 3.13: Java code of a value restricting OCL constraint

3.4 Sample transformation

This section illustrates a sample transformation of OCL constraints to their corresponding Java constraint validation classes using the implemented prototype. Therefore, a distinction is made between a description of the GUI and the console version. Finally, this section shows the output that is generated as a result by the prototype.

3.4.1 Constraint transformation

After all requirements are fulfilled and the build process was configured (see appendix A), the transformation process can be started by changing the

current directory to the prototypes root directory and setting up the following commands:

```
ant clean
ant run.codegen
```

or simply:

```
ant
```

Run-level GUI

Having set the run-level configuration option (`mda.runlevel`) in the build properties file to `gui`, the prototype will show its graphical user interface that is illustrated in figure 3.7.

In the first step of the GUI provided in figure of figure 3.7 the XMI representation of the UML model, as well as the file holding the OCL constraints can be specified. Afterwards, the OCL constraints can be loaded from the specified file by a click on the button *Load OCL constraints*. Loading in this case means that a few new buttons appear on the GUI and additionally, a text-area holding the OCL constraints denoted in the specified OCL constraint file is shown. These constraints now can be freely edited within this text-area and furthermore, can be saved back to the OCL constraint file by a click on the button *Save constraints*.

As the next step, the OCL constraints have to be normalized by a click on the button *Normalize constraints*. If the normalization process was successful it appears again a new button and furthermore, a text-area holding each OCL constraint in its normalized form. If an error occurred while normalizing the OCL constraints, then the syntax of the OCL constraints should be checked and the normalization process should be started again.

Finally, the normalized OCL constraints can be transformed to their corresponding Java constraint validation classes, whereas one out of the following two different transformation options can be chosen:

- **OCL to Java**: This option tells the code generator to solely use the implemented OCL to Java transformer to transform the OCL constraints.
- **OCL to SQL and Java**: This option tells the code generator to use both implemented transformers, as described in the previous sections.

In both cases, Java constraint validation classes are generated. Hence, these

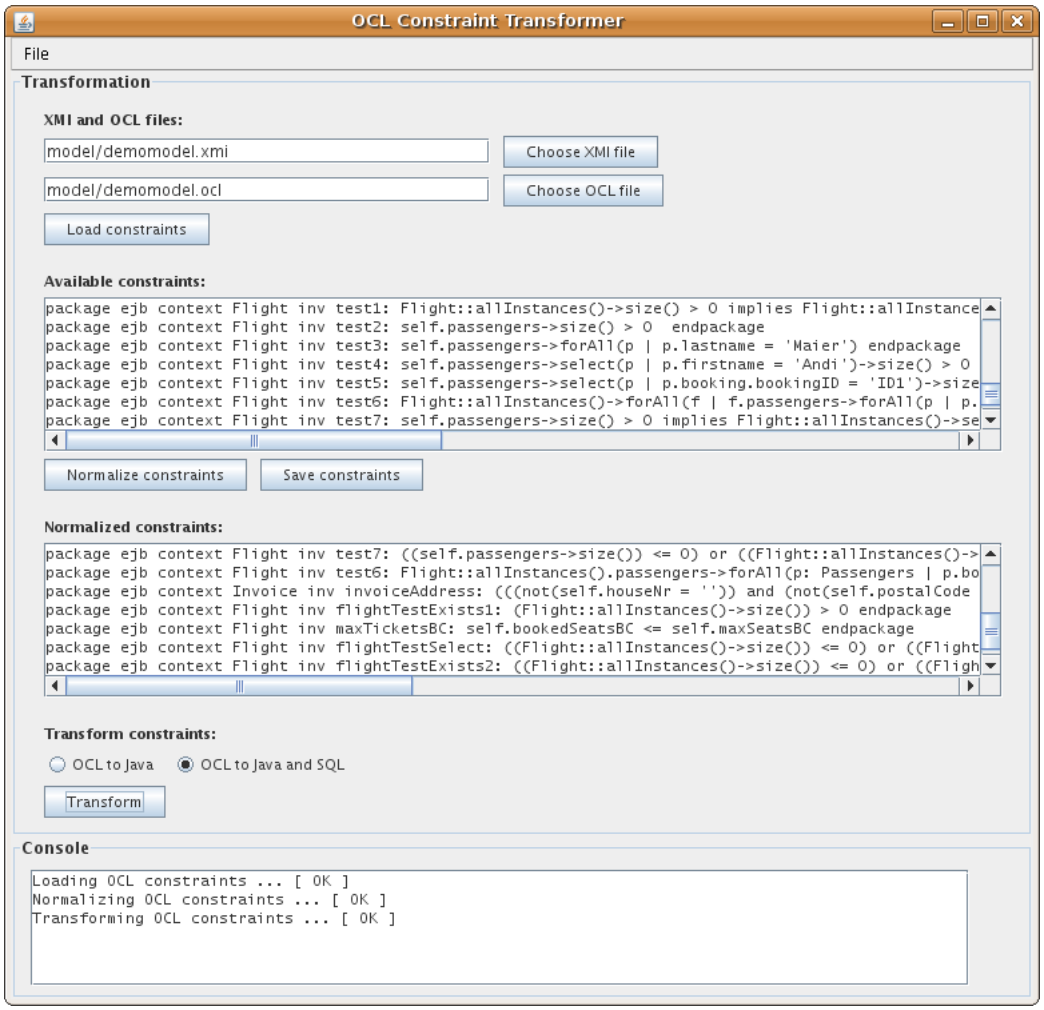


Figure 3.7: The GUI of the implemented prototype

options only define which transformer should be used to transform the OCL constraints to their corresponding Java constraint validation code. By choosing the first option, the OCL constraints are transformed solely using the OCL to Java transformer. By choosing the second option, the OCL constraints are transformed using the OCL to Java and the OCL to SQL transformer, as illustrated in figure 3.4.

The last element of figure 3.7 is the log message console at the bottom of the GUI, which protocols every action that was performed. Moreover, it prints out a state to each action whether it was successful or not.

Run-level console

In case of run-level `console` is set using the configuration option `mda.runlevel` within the build properties file, exactly the same steps as described in the previous paragraph are performed. The only restrictions are that the values, which can be set using the GUI have to be set within the properties file `mda.properties` and therefore, additional but unnecessary functionality like loading, editing and saving of OCL constraints before they are going to be transformed is not available.

For a detailed description about the requirements and the configuration of the build process, please refer to appendix A.

3.4.2 Generated output

This section illustrates some sample files for a better understanding of how the prototype works.

DeDiSys constraint definition file

Every OCL constraint that is transformed by the implemented prototype, needs to be added to a *DeDiSys* specific constraint definition file, holding additional information for each constraint. This definition file is needed to register the constraints at the constraint repository of the *DeDiSys middleware*. A sample file is provided in listing 3.14.

```
<!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="Test1" type="HARD" priority="RELAXABLE"
      negotiation="IMMEDIATE" contextObject="Y"
      minSatisfactionDegree="SATISFIED"
      latestAcceptedSatisfiedThreatRemovesIdenticalThreats="Y"
      intra-object="false">
    <class>constraints.Test1</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>

  <constraint ...>...</constraint>
</ccDefinitions>
```

Listing 3.14: Sample of a constraint definition file

For a detailed description of the elements occurring in the DeDiSys constraint definition file provided in listing 3.14, please refer to [Hor06].

Java constraint validation class

The explanations of the Java constraint validation class within this section are based on the OCL constraint given in listing 3.15.

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

Listing 3.15: Base OCL constraint for a sample transformation

Listing 3.16 shows the corresponding Java constraint validation class to the OCL constraint given in listing 3.15 that was generated by the implemented prototype.

In this case the OCL constraint is partially transformed to Java and SQL, which means that the OCL to SQL transformer was not able to transform the whole constraint and therefore, the OCL constraint was split by the top level operator `implies`. The reason why the OCL to SQL transformer is not able transform the whole constraint is that this transformer cannot handle OCL's *if expression* (see section 4.1.1).

In the next step, the part left from the OCL's `implies` operator is transformed by the OCL to SQL transformer and the right part, regarding the OCL's `implies` operator is transformed by the OCL to Java transformer.

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

  public boolean validate (IConstraintValidationContext ctx)
     throws ConstraintUncheckableException {
    boolean result = false;
```

```
try {
  // Initialize initial context
  InitialContext ic = new InitialContext();

  // Initialize context object
  FlightBeanImpl flight = (FlightBeanImpl)ctx.
      getContextObject();

  // Initialize necessary objects for validation on the
      object layer
  FlightLocalHome flightHome = (FlightLocalHome)ic.lookup("
      ejb/FlightBean");

  // Initialize necessary objects for validation on the
      database layer
  Connection con = ((DataSource)ic.lookup("java:/
      FlightbookingDS")).getConnection();

  // Validate constraint using the following transformation
      code
  boolean result2;
  try {
    PreparedStatement stmt = con.prepareStatement("SELECT
        COUNT(*) AS total FROM Flight f  ");
    ResultSet rs = stmt.executeQuery();

    result2 = true;
    while (rs.next()) {
      if (!(rs.getInt("total") >= 4)) {
        result2 = false;
        break;
      }
    }
  }
  catch (Exception e) {
    throw new ConstraintUncheckableException(e);
  }

  boolean result3 = true;
  java.lang.Integer acc = 0;
  Collection flight0 = flightHome.findAll();
  Iterator it_v = flight0.iterator();
  ejb.FlightLocal v = null;
  while (it_v.hasNext()) {
    v = (ejb.FlightLocal) it_v.next();
    if (v.getBookedSeatsEC() < v.getMaxSeatsEC()) {
      acc = acc + 1;
    }
```

```
        }
        result3 = acc >= 4;


        result = !result2 || result3;
      }
      catch (Exception e) {
        log.error("Constraint could not be checked", e);
        throw new ConstraintUncheckableException(e);
      }

      return result;
    }
}
```

Listing 3.16: Sample of a Java constraint validation class

3.5 Design and development decisions

In order to use this prototype in arbitrary scenarios, it was necessary to introduce some conventions regarding the UML model. These assumptions that are explained in the following sections, are based on the sample UML model, illustrated in figure 3.8.

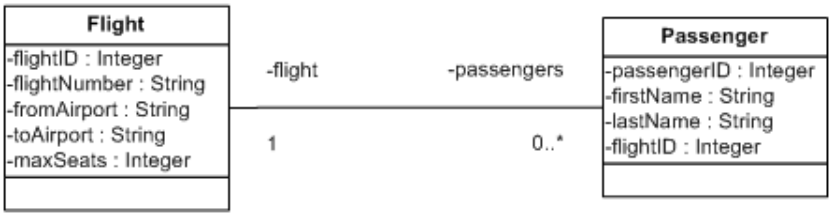


Figure 3.8: Sample UML model for illustrating introduced conventions

Primary and foreign key names

This convention is related to the OCL to SQL transformer and defines how primary key fields and foreign key fields have to be named. The primary key field name and the foreign key field names of each class are constructed by the string `ID` that is preceded by its corresponding class name in lower case. For example, the primary key of the class `Flight` results in the string `flightID`.

Hence, this convention guarantees that the OCL to SQL transformer is able to generate the table joins needed within the resulting SQL queries.

OCL constraint definition

As aforementioned, it is not possible to specify the constraints of an application directly on the UML model, due to restrictions within the UML modeling tool *ArgoUML*. Therefore, the constraint definition has to be divided into the following two parts:

- **Internal definition:** The additional constraint data (e.g. type, priority, minimum satisfaction degree, etc.) that is necessary to register the constraints within the *DeDiSys middleware* is specified within *ArgoUML*.
- **External definition:** The OCL constraints themselves have to be specified in an external text file, due to *ArgoUML's* lack of support for OCL's `allInstances` operator.

Chapter 4

Evaluation and future work

This section provides an evaluation of the prototype implemented within this thesis. To this end, it investigates the usability of the prototype within three separate scenarios and provides information regarding its ability to handle the OCL constraints of each area. Moreover, it investigates the runtime performance of the generated Java constraint validation code within the *DeDiSys middleware* and compares it to the legacy prototype, to see the benefit of this work.

Finally, this chapter provides information about some further challenges that are not covered within this thesis, but which could be interesting to be investigated and implemented in some future works.

4.1 Prototype evaluation

This section covers the aforementioned evaluation of the prototype in three different areas and provides information about the ability of the implemented prototype to transform OCL constraints of these areas. These areas are a flight booking scenario including some sample OCL constraints created by the author of this thesis, another scenario dealing with an Aeronautical Information Exchange Model (AIXM) with real-life constraints and a third scenario regarding OCL constraints of three different research papers.

4.1.1 Flightbooking application

The first area in which the implemented prototype was evaluated is a flight-booking application that already existed for the *DeDiSys middleware*. This application was developed in a work prior to this thesis and was used for the purpose of testing the *DeDiSys middleware*. The major objects of this application are given in the class diagram illustrated in figure 4.1.

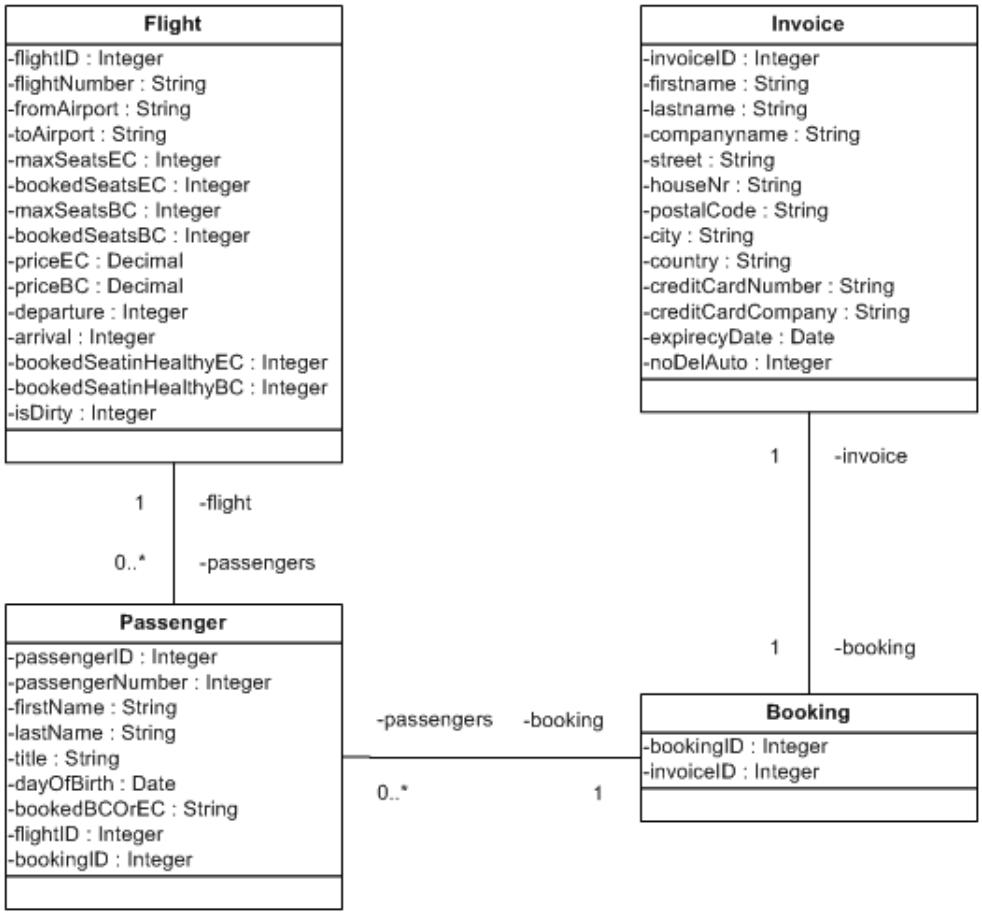


Figure 4.1: Class diagram of the flight-booking application

The major elements of the class diagram provided in 4.1 are `Flight` holding all available flights, `Passenger` containing all passengers of flights, `Invoice` representing the invoice data of a booking of a flight by a passenger and `Booking` relating an invoice to a passenger.

In order to evaluate the prototype in this area, different types of OCL constraints have been considered. In the first step an OCL constraints of the

categories *Intra-instance* have been evaluated. Afterwards, *Inter-instance* constraints have been examined. In the last step, more sophisticated OCL constraints of category *Type-level* have been evaluated. An excerpt of the evaluated OCL constraints is provided in listing 4.1.

```
# Intra-instance constraint
context Passenger inv ValidPassenger:
self.firstName <> '' and self.lastName <> ''

# Inter-instance constraint
context Flight inv ValidBookings:
self.passengers->size() <= self.maxSeats

# First Type-level constraint
context Flight inv FlightFromBarcelonaToViennaExists:
Flight::allInstances()->select(
  f | f.fromAirport = 'BAR' AND f.toAirport = 'VIE'
)->size() > 0

# Second Type-level constraint
context Flight inv OneBookingID:
Flight::allInstances()->forAll(
  f | f.passengers->forAll(
    p | p.booking.bookingID = 'ID1'
  )
)

# Third Type level constraint
context Flight inv availableFlights:
Flight::allInstances()->size() >= 4
implies
Flight::allInstances()->iterate(
  v: Flight; acc: Integer = 0 |
  if (v.bookedSeatsEC < v.maxSeatsEC) then acc + 1
  else acc
  endif
) >= 4
```

Listing 4.1: Evaluated OCL constraint types of a flight-booking application

The evaluation of the implemented prototype within this scenario showed that the prototype is able to transform the OCL constraints of all types to their corresponding Java pendant using the implemented OCL to Java transformer that generates code validation the constraints on the object layer.

In contrast, the OCL to SQL transformer is not able to handle all types of OCL constraints, due to restrictions within the transformer. These problems

arise, when specific OCL expressions occur within the OCL constraint. One solution to these problems is to restructure the OCL constraint to a semantically equivalent one, but without using the unsupported OCL expressions. If it is not possible to find a semantically equivalent OCL expression, the OCL to Java transformer acts as the default fall-back and therefore, transforms the OCL expression to its corresponding Java representation. The unsupported types of OCL expressions are discussed in the following sections.

Complex relationships

These relationships describe OCL expressions, where it is not possible to map the OCL expression to its corresponding SQL representation in a generic and straightforward way, because of necessary SQL joins that are not yet possible to generate using the implemented OCL to SQL transformer. The problem arises especially in case of that one table should be joined more than once. Support for such OCL expressions could be implemented as future work to this thesis.

Iterate expressions

The OCL to SQL transformer is not yet able to handle OCL's *iterate expression*. In order to be able to transform such OCL expressions using the OCL to SQL transformer, it is necessary, as aforementioned, to transform these expressions to semantically equivalent expressions that do not make use of OCL's iterate construct. Listing 4.2 provides an example for such an OCL constraint and additionally shows a restructured, but semantically equivalent OCL constraint that is transformable by the OCL to SQL transformer.

```
context Flight inv availableFlightsUsingIf:
Flight::allInstances()->iterate(f; i: Integer = 0 |
  if (f.bookedSeatsEC < f.maxSeatsEC) then i + 1
  else i
  endif
) >= 4

context Flight inv availableFlightsWithoutIf:
Flight::allInstances()->select(
  f | f.bookedSeatsEC < f.maxSeatsEC
)->size() >= 4
```

Listing 4.2: Removing OCL's iterate and if expression

Support for such OCL expressions could be implemented as future work to this thesis.

If expressions

With the use of OCL's *if expression* it is possible to denote more sophisticated OCL expressions. As the OCL to SQL transformer is not able to handle such expressions, the OCL constraint needs to be restructured to a simpler and for the OCL to SQL transformer transformable constraint, as mentioned earlier. An example for an OCL constraint using an *if expression*, as well as a semantically equivalent and furthermore transformable OCL expression is provided in listing 4.2. Support for such OCL expressions could be implemented as future work to this thesis.

A possible solution to the unsupported OCL expressions consisting of *Complex relationships*, *Iterate expressions* and *If expressions* could be, to extend the implemented OCL to SQL transformer in a way, to be able to generate code for *stored procedures*.

4.1.2 Aeronautical Information Exchange Model

The second area in which the implemented prototype was evaluated, is the specification of the Aeronautical Information Exchange Model (AIXM) with corresponding rules defined on it. This model was developed by the US Federal Aviation Administration (FAA) and the European Organization for the Safety of Air Navigation (EUROCONTROL) with support from the international community [AIX].

The Aeronautical Information Exchange Model (AIXM) is designed to enable the management and distribution of Aeronautical Information Services (AIS) data in digital format. [AIX]

The rules provided within the AIXM are denoted in verbal form and therefore, they can be interpreted as constraints on the model. Hence, it was necessary to transform the rules to their corresponding OCL expressions to be able to evaluate them using the implemented prototype. The evaluation of these constraints pointed out that most of the constraints are simple and straightforward to transform with the OCL to Java as well as with the OCL to SQL transformer.

Table 4.1 classifies the basic structures of the rules denoted within the AIXM,

whereas *T* denotes a table, *A* denotes an attribute, *R* denotes a related table of a table and *sizeof* counts the number of entries in a related table. For example the construct *T1.R1.A1* means attribute *A1* of the related table *R1* of table *T1*.

Nr	Structure
1	T1.A1 is specified $\Longrightarrow$ T1.A2 is mandatory
2	T1.A1 is specified $\Longrightarrow$ T1.A2 = 'abc'
3	T1.A1 is specified $\Longrightarrow$ T1.R1.A1 = 'abc'
4	T1.A1 = 'abc' $\Longrightarrow$ T1.A2 = 'def'
5	T1.A1 = 'abc' $\Longrightarrow$ T1.R1.A1 = 'def'
6	T1.A1 = 'abc' $\Longrightarrow$ sizeof(T1.R1) = X
7	T1.A1 = 'abc' $\Longrightarrow$ sizeof(T1.R1.R2) = X
8	sizeof(T1.R1) > 0
9	sizeof(T1.R1) > 0 $\Longrightarrow$ T1.A1 = 'abc'
10	sizeof(T1.R1) > 0 $\Longrightarrow$ T1.R2.A1 = 'abc'

Table 4.1: AIXM rule structures

All of these constraints can be transformed using the implemented transformer, whereas not every constraint can be transformed solely by the use of the OCL to SQL transformer, but at least the OCL to Java transformer as default fall-back is able to transform them. Furthermore, as the implemented code generator can split a constraint into smaller parts, it is possible to transform these constraints using both transformers.

4.1.3 OCL constraints of selected papers

This third area in which the implemented prototype was evaluated is about OCL expressions taken from the three different research papers [ZS06], [ZHD07] and [ZA08]. In more detail, these papers contain OCL expressions that are much more complex than the OCL constraints considered so far.

First selected paper

Firstly, the OCL constraints corresponding to the work of [ZS06] were evaluated. The main goal of this paper is given by the authors as follows:

We especially aim to model a specific subset of the dynamic composition features that can be found in dynamic object-oriented programming environ-

ments: changes to structural object-oriented features of classes or components, and the behavior changes that result from them. [ZS06]

The evaluation of this paper pointed out that the OCL constraints are built upon OCL's *oclIsKindOf* operator. As this operator is supported by the implemented prototype, OCL constraints like the ones in [ZS06] are possible to transform. In detail this means that it is possible to transform OCL's *oclIsKindOf* operator using the OCL to Java transformer, whereas it is not possible by the use of the OCL to SQL transformer.

Second selected paper

In the next step, the OCL constraints corresponding to the work of [ZHD07] were evaluated. A definition of the target of this paper is given by the authors as follows:

The main goal of the work presented in this paper is to develop and validate a novel approach to model process-driven SOAs independently of these implementation details, but in a way that allows our models to be precisely mapped to the details of particular implementations and implementation technologies. [ZHD07]

This evaluation turned out that these OCL constraints do not cover unsupported operations. This means that these types of OCL constraints can be transformed using the implemented prototype. How the transformations of these constraints look in detail, whether its is solely transformed by the OCL to Java or OCL to SQL transformer, or it needs to be split into smaller parts, depends on the specific OCL constraint.

Third selected paper

Finally, the OCL constraints according to the work of [ZA08] were evaluated. The main goal of this paper is given by them as follows:

We propose to remedy the problem of modeling architectural patterns through identifying and representing a number of 'architectural primitives' that can act as the participants in the solution that patterns convey. [ZA08]

This paper introduces OCL constraints that are based on an extension of the UML 2.0 meta-classes using the UML extension mechanism. These OCL constraints are out of the focus of this thesis and are therefore not yet supported by the implemented prototype. Support for this type of OCL constraints could be an interesting challenge to solve in the future. Hence, for more

information about these types of OCL constraints, please refer to section 4.3.

4.2 Constraint performance evaluation

This section evaluates the benefit of the prototype implemented within this thesis. For this purpose, the legacy prototype is compared with the implemented prototype, with respect to the run-time performance of the generated Java constraint validation code.

4.2.1 Testing environment

The performance evaluation has been carried out on a standalone workstation, whereas table 4.2 provides detailed information about the hardware and the software of the testing environment that was used while evaluating and comparing the performance of the generated Java constraint validation classes.

Hardware	Description
CPU	Intel Core2 Duo CPU E6750 @ 2.66GHz
Memory	2 GB
Software	**Description**
OS	Ubuntu 9.10 (karmic), Kernel Linux 2.6.31-16-generic
JRE	Sun Java(TM) Runtime Environment (JRE) 5.0
JDK	Sun Java(TM) Development Kit (JDK) 5.0
IDE	NetBeans IDE 6.5.1
Build Tool	Apache Ant 1.7.1
Database Server	MySQL 5.1.41
Application Server	JBoss 5.0.1.GA

Table 4.2: Testing environment

4.2.2 Performance evaluation

In order to measure the run-time performance of the generated Java validation classes, one constraint of each of the categories *Intra-instance*, *Inter-instance* and *Type-level* has been evaluated. Furthermore, the run-time per-

formance of each constraint was measured five times for the purpose of getting an average performance value.

Intra-instance constraint

The first evaluated OCL constraint is an LCC constraint related to the *Intra-instance* category. This OCL constraint is illustrated in listing 4.3. In order to get meaningful performance results for this OCL constraint, the database table corresponding to the `Flight` class was populated with 100, 1.000 and 10.000 dummy records. Furthermore, the OCL constraint was evaluated for each of these populations.

```
context Flight inv ValidBookings:
self.bookedSeatsBC <= self.maxSeatsBC
```

Listing 4.3: Evaluated Intra-instance constraint

The constraint provided in listing 4.3 assures that the number of booked seats is at most as high as the number of available seats. The evaluation of this OCL constraint was only performed using the implemented prototype for the reason that for *Intra-instance* constraints the legacy prototype and the implemented prototype generate identical code. The results of the run-time performance evaluation of this OCL constraint are shown in figure 4.2.

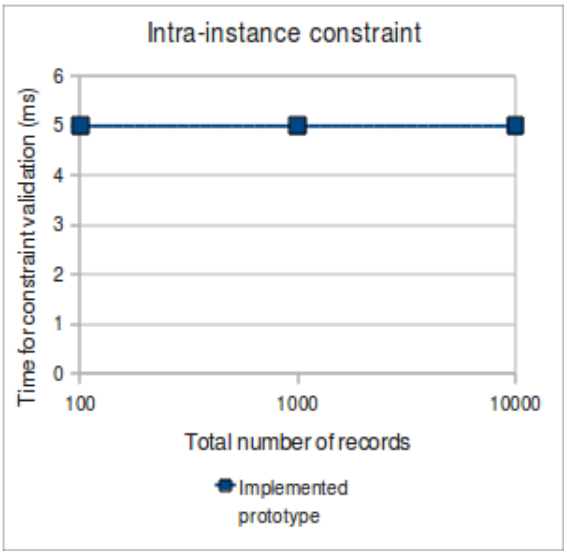


Figure 4.2: Performance results of the evaluated Intra-instance constraint

The performance results given in figure 4.2 show that the validation of *Intra-instance* constraints can be performed in an efficient way. Furthermore, these results do not significantly change for the different populations. This is an expected behavior as the validation does not directly depend on the database population.

Inter-instance constraint

The second evaluated OCL constraint is an LCC constraint related to the *Inter-instance* category. This OCL constraint is illustrated in listing 4.4. In order to get meaningful run-time performance results for this OCL constraint, the database table corresponding `Flight` class was populated with one record and the database table corresponding to the `Passenger` class was populated with 100, 1.000 and 10.000 related dummy records. Furthermore, the OCL constraint was evaluated for each of these populations.

```
context Flight inv ValidPassengers:
self.passengers->forAll(p |
  p.firstName <> '' and p.lastName <> ''
)
```

Listing 4.4: Second evaluated Inter-instance constraint

The constraint provided in listing 4.4 assures that every passenger of a flight has a valid name. This means that the name must not be empty. The evaluation of this OCL constraint was only performed using the implemented prototype for the reason that the legacy prototype is not able to handle OCL constraints using relations. The results of the run-time performance evaluation of the OCL constraint given in 4.4 are shown in figure 4.3.

The performance results given in figure 4.3 show that the validation of *Inter-instance* constraints generated using the implemented prototype can also be performed in an efficient way. This can be seen in figure 4.3 as the run-time performance is growing slow regarding the examined populations.

Type-level constraint

The third evaluated OCL constraint is a FCC constraint related to the *Type-level* category. This OCL constraint is illustrated in listing 4.5. In order to get meaningful performance results for this OCL constraint, the database table corresponding to the `Flight` class was populated with 100, 1.000 and

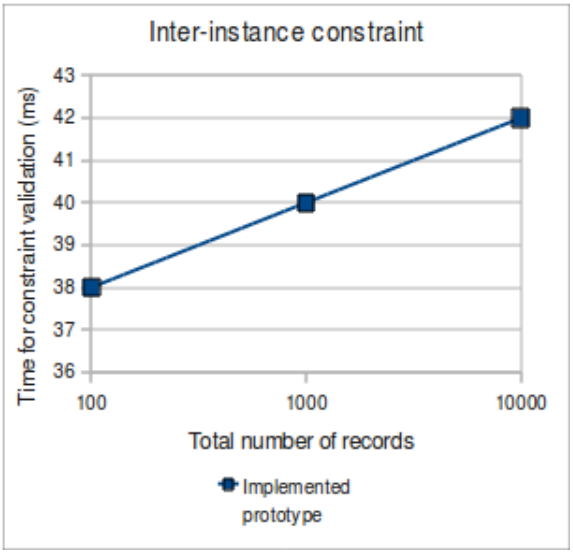


Figure 4.3: Performance results of the second evaluated Inter-instance constraint

10.000 dummy records. Furthermore, the OCL constraint was evaluated for each of these populations.

```
context Flight inv FlightFromBarcelonaToViennaExists:
Flight::allInstances()->size() > 0
implies
Flight::allInstances()->exists (
  f | f.fromAirport = 'BAR' and f.toAirport = 'VIE'
)
```

Listing 4.5: First evaluated Type-level constraint

The constraint provided in listing 4.4 assures that whenever there exist flight within the database, then there also must exist a flight from *Barcelona* to *Vienna*. The results of the run-time performance evaluation of the OCL constraint given in 4.5 are shown in figure 4.4.

The performance results provided in figure 4.4 show that regarding 100 records, the constraint validation time using the code generated by the implemented prototype is approximately 15 times faster than the code generated using the legacy prototype. Regarding 1.000 records, the constraint validation time is approximately 80 times faster and regarding 10.000 records, the constraints validation time is about 530 times faster. Hence, this leads to a huge run-time performance gain of the constraint validation process.

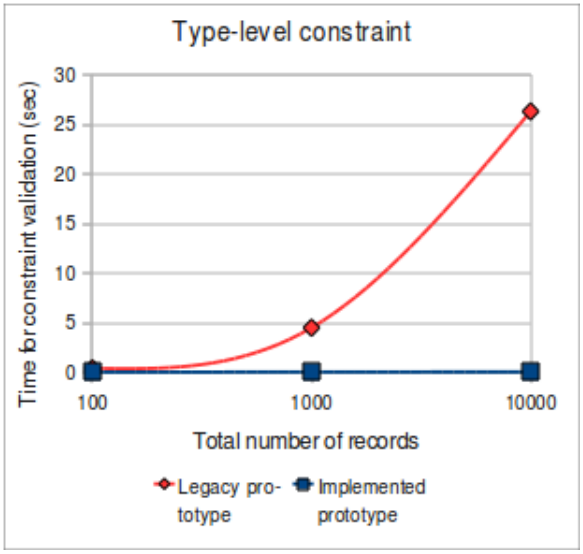


Figure 4.4: Performance results of the first evaluated Type-level constraint

Regarding the presented performance results it has to be stated that the Java constraint validation classes generated using the implemented prototype have a massive positive impact. This is especially true for for *FCC constraints* as these result can be compared directly to the results of the legacy prototype. Moreover, the results of the evaluation of the presented *LCC constraints* shows that the validation of the constraints using the code generated using the implemented prototype is also efficient. For these reasons, this work introduced a huge performance gain and therefore it was worth its effort.

4.3 Future work

Additionally to the features implemented within the prototype of this thesis, there exist a few further challenges that would be interesting to be investigated and implemented in some further works. These challenges are described in the following sections.

Tool-based restructuring of OCL constraints

The first and currently existing version of the implemented OCL to SQL transformer is, as already mentioned in previous sections, not able to trans-

form all possible OCL expressions to their corresponding SQL representations. In many cases it is possible to change the definition of not transformable OCL expressions in a way that they are transformable again.

The current implementation is able to state that specific OCL constraints should be denoted in a semantically equivalent but different way, to be able to transform them with the current OCL to SQL transformer. This could be enhanced in a future work to this thesis. A possible enhancement is, to provide a tool-based restructuring of not transformable OCL expressions in a way that the designer is free in the specification of the OCL constraints. Hence, the designer does not need to manually restructure his OCL expressions in case they are not transformable by the OCL to SQL transformer.

Change of the UML modeling tool

As aforementioned, a UML modeling tool is necessary within this work, to design the model of an application that wants to make use of the *DeDiSys middleware*. For this purpose, this thesis uses *ArgoUML*, which introduces restrictions regarding the OCL. For example, it is not possible to define OCL constraints within *ArgoUML* using OCL's `allInstances` operator. As this operator is essential for this thesis, the definition of the OCL constraints of a corresponding UML model have been divided into two parts. This means that the OCL constraints have to be specified in an external text file and the additional and *DeDiSys* specific constraint meta-data (e.g. type, priority or minimum satisfaction degree) have to be specified within the UML model.

Hence, it is necessary to evaluate some other UML modeling tools in order to use a UML modeling tool that satisfies the requirements of this work. Thus, it would be possible to define every information of the application directly within the UML model, without the need of external files holding additional information.

Enhancements of the OCL to SQL transformer

Up to now, the OCL to SQL transformer is not able to transform all possible OCL expressions. This is particularly true for OCL's `iterate` and `if` expression and furthermore this is true for complex OCL expressions as mentioned before.

Hence, the implemented OCL to SQL transformer could be enhanced with functionality that is providing support this expressions. A possible enhance-

ment could be, to extend the implemented OCL to SQL transformer in a way, to be able to generate code for *stored procedures*.

Chapter 5

Summary and conclusion

This thesis had its main focus on the optimization of an OCL to run-time constraint generator for the *DeDiSys middleware* as its J2EE constraint consistency framework. The main goal of this optimization was to generate faster Java constraint validation code with respect to the run-time performance, when the constraint are going to be validated within the *DeDiSys middleware*.

In order to achieve a better run-time performance of the constraint validation process, several optimizations have been applied to the previously existing implementation of the OCL to run-time constraint generator. These optimizations include a better support for UML models created with UML modeling tools, whereas it is no longer necessary to define a specific dummy relation for each table pointing to itself, in order to be able to use OCL's `allInstances` operator. Thus, this operator now can be freely used throughout the OCL constraint definition.

Furthermore, and build upon the previously mentioned optimization, the existing OCL to Java transformer was extended by supporting the essential `allInstances` operator of the OCL and furthermore, now is able to handle relations that can be queried by OCL constraints.

An OCL to SQL transformer builds another major improvement to the existing run-time constraint generator that transforms OCL expressions to their corresponding SQL representations. In more detail, these SQL representations are SQL queries that are directly set up to the database to increase the run-time performance of the constraint validation process.

Moreover, a transformation algorithm was implemented that allows to split an OCL constraint into several parts. This is a major improvement as now it

is possible to transform parts of the OCL constraint to their Java pendant and other parts to their SQL representation. At the end of the transformation, these parts are combined again.

Finally, an evaluation of the run-time performance of the Java constraint validation classes generated by the implemented prototype has been performed. After comparing these performance results to the performance results of the Java code generated by the legacy prototype it turned out that the introduced optimizations were worth their effort and hence, are increasing the overall performance of the *DeDiSys middleware*.

Appendix A

Prototype details

This chapter provides additional information to the implemented prototype, regarding the requirements and configuration properties.

A.1 Requirements & configuration

This section deals with the information how to setup the computer in order to run the implemented prototype. Furthermore, this chapter provides information about the Ant build process regarding the available Ant tasks, as well as the available configuration options that can be specified in a separate build configuration file.

A.1.1 System requirements

In order to run the implemented prototype, there are a few requirements that need to be fulfilled in prior. These requirements are denoted in table A.1.

Third-party tool	Version
Java Standard Development Kit (SDK)	1.5.0
Java Runtime Environment (JRE)	1.5.0
JBoss Application Server	5.0.1
Apache Ant	1.7.1
MySQL	5.1.41

Table A.1: System requirements

The version numbers of the tools given in table A.1 do not necessarily need to be as stated here, because also other version may work, but these versions have been used throughout the development of this prototype and thus, they are known to be working.

A.1.2 Build configuration

The implemented prototype can be launched using the provided Ant build file `build.xml` that is located in the root folder of project. The purpose of the following list is to describe the available Ant targets, defined within this build file.

- `clean:` Cleans the workspace by removing all generated files.
- `compile.codegen:` Compiles all Java classes of the prototype.
- `jar.codegen:` Packs the class files created by the previous target and some property files into a JAR file.
- `run.codegen:` Runs the prototype using the JAR file created by the previous target.
- `run.andromda:` Generates the EJB's of the defined UML classes using *AndroMDA*.
- `comile.app:` Compiles the application including all generated files.
- `xdoclet.app:` Generates necessary files to deploy the application to the JBoss application server.
- `overwrite.xdoclet:` Overwrites some configuration files that have been generated by the previous target with files from the flighbooking EJB module.
- `jar.app:` Builds a the deployable JAR file of the application.
- `deploy:` Deploys the JAR file to the `dedisys` server profile within the JBoss application server.
- `all.app:` Runs all previously described tasks in the given order.

Beside the Ant tasks provided in the previous list, the Ant process itself can be configured by specifiying some properties in the file `./src/mda.properties` relative to the root folder of the protoype. The available configuration options together with their corresponding default values are mentioned in the upcomming list.

- `mda.dir.out.constraints:` The directory, to which the generated Java constraint classes are saved.
  Default value: `./out/src/constraints/`
- `mda.dir.out.conf:` The directory, to which the generated DeDiSys constraint definition file (*ccDefinitions.xml*) is saved.
  Default value: `./out/src/conf/`
- `mda.dir.templates:` The directory, where the *StringTemplate* templates reside.
  Default value: `./src/org/dedisys/mda/templates/`
- `mda.file.xmi:` The name of the file that holds the XMI representation of the UML model.
  Default value: `./model/demomodel.xmi`
- `mda.file.ocl.original:` The name of the file that constains the OCL constraints.
  Default value: `./model/demomodel.ocl`
- `mda.file.ocl.normalized:` The name of the file that holds the normalized OCL constraints.
  Default value: `./model/demomodel_normalized.ocl`
- `mda.file.cc.def:` The name of the file that holds the DeDiSys constraint definitions.
  Default value: `./out/src/conf/ccDefinitions.xml`
- `mda.sql.datasource:` The JBoss datasource that should be used within the generated Java constraint validation classes to connect to the database, in case of validation on the database layer is peformed.
  Default value: `FlightbookingDS`
- `mda.transformation.type:` The type of transformation that should be performed. Possible option are 1 to use only the OCL to Java transformer or 2 to use both transformers to transform the OCL constraints.
  Default value: `2`

- `mda.runlevel:` Defines whether to launch the GUI of the prototype by setting the value of the property to `gui` or to run the console version of the prototoype by setting the value of the property to `console`. In case of runlevel `gui`, some of the previous mentioned configuration options can also be specified within the GUI.
  Default value: `gui`

Bibliography

[AIX] Aeronautical Information Exchange Model (AIXM) - A model to enable the management and distribution of Aeronautical Information Services (AIS) data in digital format. Website. `http://www.aixm.aero`.

[And] AndroMDA - An extensible generator framework that adheres to the Model Driven Architecture (MDA) paradigm. Website. `http://www.andromda.org/`.

[Arg] ArgoUML - A Java based and open source UML modeling tool supporting all standard UML 1.4 diagrams. Website. `http://argouml.tigris.org/`.

[Bau06] Markus Baumgartner. Adaptive Constraint-Validierung in einer verteilten Enterprise JavaBeans Umgebung. Master's thesis, Technikum Vienna, September 2006.

[BC06] Marco Brambilla and Jordi Cabot. Constraint tuning and management for web applications. In *ICWE '06: Proceedings of the 6th international conference on Web engineering*, pages 345–352, New York, NY, USA, 2006. ACM.

[CT07] J. Cabot and E. Teniente. Transformation techniques for OCL constraints. *Sci. Comput. Program.*, 68(3):152–168, 2007.

[Ded] Dependable Distributed Systems Middleware (DeDiSys) - A middleware for optimized dependability by adaptively balancing (trading) availability and constraint consistency. Website. `http://www.dedisys.org/`.

[DHK05] Birgit Demuth, Heinrich Hussmann, and Ansgar Konermann. Generation of an OCL 2.0 parser. In Thomas Baar, editor, *Proceedings of the MoDELS'05 Conference Workshop on Tool Support for*

OCL and Related Formalisms - Needs and Trends, Montego Bay, Jamaica, October 4, 2005, Technical Report LGL-REPORT-2005-001, pages 38–52. EPFL, 2005.

[Dre] Dresden OCL Toolkit - A software platform for OCL tool support providing tools for specification and evaluation of OCL constraints. Website. `http://dresden-ocl.sourceforge.net/`.

[Ert07] Dominik Ertl. Evaluation of Partitionable Replication Protocol Improvements in an Enterprise JavaBeans Environment. Master's thesis, Technikum Vienna, October 2007.

[FGO07] Lorenz Froihofer, Karl M. Goeschka, and Johannes Osrael. Middleware support for adaptive dependability. In *Middleware '07: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, pages 308–327, New York, NY, USA, 2007. Springer-Verlag New York, Inc.

[FOG07] Lorenz Froihofer, Johannes Osrael, and Karl M. Goeschka. Decoupling Constraint Validation from Business Activities to Improve Dependability in Distributed Object Systems. In *ARES '07: Proceedings of the The Second International Conference on Availability, Reliability and Security*, pages 443–450, Washington, DC, USA, 2007. IEEE Computer Society.

[Fuc06] Klaus Fuchshofer. Negotiation and reconciliation of consistency threats for Enterprise JavaBeans applications. Master's thesis, Technikum Vienna, May 2006.

[Hor06] Markus Horehled. Integration of an EJB Constraint Consistency Management Framework into the JBoss Application Server. Master's thesis, Technikum Vienna, May 2006.

[HWD08] Florian Heidenreich, Christian Wende, and Birgit Demuth. A Framework for Generating Query Language Code from OCL Invariants. *ECEASST*, 9, 2008.

[KWB03] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[LCG06] Tihamér Levendovszky, Hassan Charaf, and Gergely Mezei. Optimization Algorithms for OCL Constraint Evaluation in Visual Models, 2006.

[LLC05] László Lengyel, Tihamér Levendovszky, and Hassan Charaf. Normalizing OCL Constraints in UML Class Diagram-Based Metamodels - AND/OR Clauses. In *EUROCON 2005 International Conference on "Computer as a tool", Proceedings of the IEEE*, pages 579–582, Belgrade, Serbia and Montenegro, November 2005.

[LO04] Sten Loecher and Stefan Ocke. A metamodel-based ocl-compiler for uml and mof. *Electr. Notes Theor. Comput. Sci.*, 102:43–61, 2004.

[MLC06] Gergely Mezei, Tihamer Levendovszky, and Hassan Charaf. Restrictions for OCL Constraint Optimization Algorithms. *ECEASST*, 5, 2006.

[OMG] Object Management Group - International, open membership, not-for-profit computer industry consortium. Website. `http://www.omg.org/`.

[Rie07] Bernhard Rieder. Balancierung von Integrität und Verfügbarkeit in verteilten Web-basierten Enterprise JavaBeans Anwendungen. Master's thesis, Technikum Vienna, April 2007.

[Str] StringTemplate - Java template engine for generating source code, web pages, email or any other formatted text output. Website. `http://www.stringtemplate.org/`.

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[ZA08] Uwe Zdun and Paris Avgeriou. A catalog of architectural primitives for modeling architectural patterns. *Inf. Softw. Technol.*, 50(9-10):1003–1034, 2008.

[ZHD07] Uwe Zdun, Carsten Hentrich, and Schahram Dustdar. Modeling process-driven and service-oriented architectures using patterns and pattern primitives. *ACM Trans. Web*, 1(3):14, 2007.

[ZS06] Uwe Zdun and Mark Strembeck. Modeling Composition in Dynamic Programming Environments with Model Transformations. In *Software Composition*, pages 178–193, 2006.

Glossary

AIS	Aeronautical Information Services
AIXM	Aeronautical Information Exchange Model
API	Application Programming Interface
AST	Abstract Syntax Tree
CCM	Constraint Consistency Management
CORBA	Common Object Request Broker Architecture
CST	Concrete Syntax Tree
DeDiSys	Dependable Distributed System
EJB	Enterprise Java Bean
FCC	Full Consistency Check
GUI	Graphical User Interface
IC	Integrity Constraint
J2EE	Java 2 Plattform, Enterprise Edition
JAR	Java Archive
JMI	Java Metadata Interface
LCC	Limited Consistency Check
MDA	Model Driven Architecture
MDR	MetaData Repository
MDSD	Model-Driven Software Development
OCL	Object Constraint Language

OMG	Object Management Group
PIM	Platform Independent model
PST	Platform specific model
UML	Unified Modeling Language
XMI	XML Metadata Interface
XML	EXtensible Markup Language

Index