Declarative Model Constraints using the Object Constraint Language

JOHN ALMBECKER

Master’s Thesis in Computer Science (30 ECTS credits)
at the School of Computer Science and Engineering
Royal Institute of Technology year 2010
Supervisor at CSC was Stefan Arnborg
Examiner was Stefan Arnborg

TRITA-CSC-E 2010:015
ISRN-KTH/CSC/E--10/015--SE
ISSN-1653-5715

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
SE-100 44 Stockholm, Sweden

URL: www.kth.se/csc
Declarative model constraints using OCL

Abstract
A concept called Model Driven Architecture was launched in 2001 by the Object Management Group as a way of supporting Model Driven Engineering. The concept of using models to aid in the design of computer software is not new, however the use of models as a primary design artifact, and tools to automatically generate structural code from them is a newer approach.

A typical problem however has been that while models generally give a good overview of structure and flow, it has been hard to express rules and constraints within the model. Statements like “for every yellow car in database table A there should be a driver with favorite color yellow in table B” were impossible to express within the scope of the model.

This problem was tackled by the addition of the Object Constraint Language (OCL) into the most common metamodels. OCL is a declarative language used to define constraints on a model.

In this thesis I have investigated the possibility to enforce model/data correctness using OCL in real life. The investigation was done implementing an OCL framework into the platform developed by my host company LucidEra Inc. Work has been done assessing where in the LucidEra platform stack application was possible as well as how it could be implemented and made usable.

While the initial application attempts were deemed impossible within the given time-frame, the result of this work is an OCL implementation in an internal development tool.
Deklarativ modellkontroll med hjälp av OCL

Sammanfattning


Ett typiskt problem har varit att då modeller ger en bra generell överblick över strukturer och flöden har det varit svårt att uttrycka regler inom modellen. Att i modellen säga saker som ”för varje gul bil i databastabell A ska det finnas en förare med favoritfärg gul i databastabell B” har varit omöjligt.

Detta problem har tacklats genom att språket Object Constraint Language anammats i de vanligaste metamodellerna. OCL är ett deklarativt språk som används för att införa begränsningar på en modell.

Jag har undersökt möjligheten att upprätthålla korrekthet hos modeller och data genom användning av OCL. Undersökningen gjordes att implementera ett OCL-framework i en mjukvaruplattform framtagen av företaget LucidEra Inc. I mitt arbete har jag analyserat var i LucidEras plattform OCL skulle vara användbart samt hur det skulle kunna implementeras och användas.

Emedan de initiala försöken att applicera OCL verktyg bedömdes som omöjligt inom de givna tidsramarna, är resultatet av arbetet en OCL-implementation i ett internt utvecklingsverktyg.
## Contents

Chapter 1 - Introduction............................................................................................................ 1
  1.1 Background.................................................................................................................... 1
    1.1.1 Modeling ............................................................................................................. 1
    1.1.2 LucidEra ............................................................................................................. 1
  1.2 Problem.......................................................................................................................... 2
  1.3 Aim / Purpose................................................................................................................. 2
  1.4 Report outline................................................................................................................. 2

Chapter 2 - Theory.................................................................................................................... 3
  2.1 Model-driven engineering concepts............................................................................... 3
    2.1.1 A brief history......................................................................................................... 3
  2.2 Model Driven Architecture standards........................................................................... 4
    2.2.1 The Meta-Object Facility....................................................................................... 5
    2.2.2 The Common Warehouse Metamodel.................................................................... 5
    2.2.3 Platform independent and platform specific models............................................ 5
    2.2.4 Modeling maturity levels........................................................................................ 6
    2.2.5 Constraint programming and OCL......................................................................... 6
    2.2.6 OCL syntax introduction......................................................................................... 7
    2.2.7 Invariants and initial values..................................................................................... 8
    2.2.8 Queries, derived attributes and definitions........................................................... 8
    2.2.9 Pre- and postconditions......................................................................................... 8
    2.2.10 Other constructs................................................................................................... 9
  2.3 OCL implementation alternatives.................................................................................. 9
    2.3.1 Validation on user model level vs metamodel level............................................. 9
    2.3.2 When to check constraints................................................................................... 10
    2.3.3 What to do when a constraint fails....................................................................... 10

Chapter 3 - Method............................................................................................................... 12
  3.1 The LucidEra platform.................................................................................................... 12
    3.1.1 LucidDB and connectors....................................................................................... 12
    3.1.2 Mondrian.............................................................................................................. 12
    3.1.3 ALS....................................................................................................................... 13
    3.1.4 LEAP.................................................................................................................... 13
Chapter 1 - Introduction

In this chapter a short introduction to the problems faced at LucidEra will be given as well as the scope of what was chosen to be included in the thesis.

1.1 Background

1.1.1 Modeling

Software modeling; the practice of making sketches and diagrams of program flows, data structures, object hierarchies, etc. for use as software implementation blueprints is old. There are today a number of standards for creating such blueprints and their usage is standard practice in bigger software development projects.

There are different ways of treating models. In some cases models are only used as a quick way of sketching the outlines of a project and are not to be taken too literally. In other cases great effort is made to keep models accurate and up to date. This second methodology of using models as the primary design objects is called Model-driven Engineering or MDE.

If done with a good level of detail these models can be used by tools to automatically generate parts of, or sometimes even full, implementations of the object being modeled.

1.1.2 LucidEra

LucidEra delivers business intelligence products based on a software as a service model. The term “business intelligence” refers to the gathering and analysis of organizational data in order to produce information which helps gaining deeper insight into the organization, aiding better decision making.

Software as a service refers to a model where the running of the software is hosted and provided as a service to the customer via the Internet. Instead of buying installation licenses for the software, customers subscribe to the usage of the software.

Customers let LucidEra access their various business data sources. The data is then downloaded daily to a LucidEra server. When downloaded it is aggregated, processed and analyzed. Using a web-browser customers access reports based on their data, reports can be created or edited in real time by using an interactive drag-and-drop user interface.

In order to deliver reports an infrastructure has been created consisting of two major software building blocks. Firstly there is a platform dealing with underlying responsibilities such as databases, runtime services and external interfaces like the web interface and connectors to external data sources, secondly there are applications running on top of this platform, defining how customer data should be transformed, analyzed, merged, etc.

Since business intelligence involves dealing with large amounts of data and metadata from various fields such as sales, finance and marketing, careful planning has to be made as to how certain data relates to other data and how data is to be aggregated. To
alleviate the problem of organizing and storing these large amounts of meta-data, industry standard models and model driven engineering has been used in several areas of the platform.

1.2 Problem

In the past LucidEra has, on several occasions, bumped into very real problems where model and data corruption have caused serious trouble including loss of work, as, for example, when 6 person-weeks worth of work was lost when the use of an in-house development tool was discovered to corrupt data. In this instance the problem also caused a block of further development while the tool was fixed and the damages assessed.

A problem which is faced is that there is no way to, in a general manner, make declarative checks of the models and the model instance validity and correctness.

Seeing how problems had arisen in the handling of data and metadata a couple of areas of interest had been identified where enforcing model and data correctness would be beneficial. One way of doing this would be through a constraint language called Object Constraint Language (or OCL) which is part of the UML modeling standard. Both OCL and UML will be introduced in greater detail in chapter 2.

1.3 Aim / Purpose

The goal of this thesis is to evaluate how, where, and to what extent usage of the Object Constraint Language can be implemented into the LucidEra platform to enforce model correctness. When a suitable point of application has been found an implementation will be done as a proof of concept.

1.4 Report outline

I will start in chapter two by giving a quick background and overview on the theory necessary to understand the following sections, explaining model driven engineering and the usefulness of OCL.

After that, chapter three will give a description of the LucidEra platform together with a description of how the problem was approached and what was examined.

In chapter four the results and outcomes of the research is presented. Finally, in chapter five I present the conclusions drawn from these results.
Chapter 2 - Theory

In this chapter we will look into the background theory needed to understand subsequent chapters. Regrettably this chapter will introduce a confusing amount of abbreviations. When confusion strikes, a list of terms and abbreviations can be found in Appendix 1.

2.1 Model-driven engineering concepts

Wikipedia defines model-driven engineering as

“... model-driven engineering refers to a range of development approaches that are based on the use of software modeling as a primary form of expression.”

2.1.1 A brief history

Computer software engineering has since its childhood been striving towards abstraction. As the size and complexity of software grows, abstracting the engineer from underlying structures and domain specific knowledge is necessary. Abstractions are also made since using available abstraction layers speeds up development time as the same code does not have to be reinvented over and over again. Going from machine code to assembly to higher and higher level languages all adds to the level of abstraction.

Computer Aided Software Engineering (CASE) is a term generally used to refer to methods that let computers assist in developing software. One branch of CASE-tools are tools geared towards letting the user create graphical representations of information flows, systems, programs or data structures which are then partially or fully generated by the tool.

This approach was introduced during the 1980s. In these early stages however, the CASE approach suffered from several problems such as lacking maturity in operating systems, causing difficulties and high complexity in the development of CASE code generation tools as well as other problems which made it an item of interest in academic circles, but not widely practiced.

Therefore early software creation CASE tools were used mainly to help developers create models representing the software architecture. These models were then used when hand coding out the actual implementation.

In the mid to late 1990s a multitude of different object-oriented modeling standards had seen the light. This was seen as threatening the pace of adoption to object-oriented technology. One of the leading companies in the market, Rational Software Corporation, launched an initiative to create a non-proprietary modeling language to replace all others. The result was published in 1997 under the name Unified Modeling Language (or UML) which is currently at version 2.0.

---

1 http://en.wikipedia.org/wiki/Model-driven_engineering
2.2 Model Driven Architecture standards

A framework called Model Driven Architecture (MDA) was launched in 2001 by the Object Management Group (OMG) as a way of supporting Model Driven Engineering.

Within this framework a model is a structure for describing objects. When talking about modeling in MDA Frankel (2003) defines a view with four different modeling layers.

<table>
<thead>
<tr>
<th>M3</th>
<th>Meta-Metamodel</th>
<th>MOF, Ecore..</th>
</tr>
</thead>
<tbody>
<tr>
<td>M2</td>
<td>Metamodel</td>
<td>UML, CWM..</td>
</tr>
<tr>
<td>M1</td>
<td>Model</td>
<td>E.g. Database table layout</td>
</tr>
<tr>
<td>M0</td>
<td>System/Instance</td>
<td>E.g. Database data</td>
</tr>
</tbody>
</table>

Table 1: The four modeling layers proposed by OMG in MDA

- **M0 - System/Instance**: Model instance refers to an object adhering to the specifications provided by the model.
  
  *Example - If we were to model a database containing users, this would be an actual instance of the database containing entries conforming to the table layout.*

- **M1 - Models**: A model is a framework for describing instances
  
  *Example – In the above example this model would correspond to a database model describing all the tables and columns required for the user database.*

- **M2 - Metamodels**: A metamodel is a model describing the
  
  *Example – The metamodel in this case would be a model containing the concepts of columns and rows as objects*

- **M3 - Meta-metamodels**: Are models of the modeling language used in the lower layers. Oftentimes these meta-metamodels are designed to conform to themselves, thus creating a closed architecture.
  
  *Example – The meta-metamodel would in our example be a model describing the modeling language and have concepts such as classes, data and relationships.*

To achieve the goal of saving substantial effort by using models to generate code structure, code artifacts, or even full software implementations including language logic, the MDE approach suggests using combinations of special models called Domain Specific Language (DSL) models together with transformation tools.

A DSL model is a metamodel specifying the language and application structure within an organization or a domain. As opposed to the CASE tools previously used where vendors created metamodels which tried to model all possible scenarios and domains, DSL models and editors are usually designed within the organization and allows for a closer following of the organizations development environment, easing the difficulty to learn the tools as well as allowing for easier, more extensive and more accurate code generation, resulting in less hand written code.
2.2.1 The Meta-Object Facility

The *Meta-object Facility* (MOF) was designed around UML as OMG needed a metamodeling architecture to define UML. MOF is designed around the four layer-architecture and the MOF meta-meta model conforms to itself, making it a closed architecture.

MOF has in turn been split into two variants:

- CMOF or Complete MOF
- EMOF or Essential MOF

Where EMOF is a minimal subset of CMOF using only the most basic class diagram concepts.

2.2.2 The Common Warehouse Metamodel

The *Common Warehouse Metamodel* (CWM) is a specification for modeling metadata in a data warehousing environment designed by MOF with contributions from several companies such as IBM, Oracle and Unisys. The idea was to develop a standardized format for exchanging metadata between tools from different vendors.

In CWM it is possible to use objects which describe when data was created and from were it originated. CWM instances are stored using the *XML Metadata Interchange* format (XMI).

2.2.3 Platform independent and platform specific models

MDA introduces the concept of having a *Platform Independent Model* (PIM) in which an application is modeled with a high level of abstraction. This model is in no way connected to final implementation technology or implementation platform.

The PIM is then transformed into a *Platform Specific Model* (PSM) which holds all platform specific information. From this PSM the end result generated.

The advantage of this approach is that the same PIM can be used to generate code for multiple different platforms, granted that conversion tools from the PIM to the various PSMs exists, thus facilitating testing of different platforms and minimizing the tailoring of an application to a specific platform.

Warmer/Kleppe (2003) states that since the PSM fit its technology closely, automated conversion from PSM to code is straightforward. The big news about MDA is the development of tools that do automated transformations from PIMs to PSMs.

Consider for example a UML model containing a database layout, this is our PIM. Using our transformation tools we now convert this to a SQL PSM which in turn can be used to generate the final SQL code. Now suppose we want to try out a different DB using QUEL instead of SQL. All we would have to do is use our original PIM with a new set of transformations to create a QUEL PIM and from this generate QUEL code. The idea is that all these conversions should be fully automated, minimizing human interaction.
2.2.4 Modeling maturity levels

In order to identify how models are used in software development, Kleppe (2003) identifies 5 different levels of model maturity.

- **Level 0: No Specification**: the specification of software is not written down. It is kept in the minds of the developers
- **Level 1: Textual Specification**: the software is specified by a natural language text (be it English or Chinese or something else), written down in one or more documents
- **Level 2: Text with Models**: a textual specification is enhanced with several models to show some of the main structures of the system
- **Level 3: Models with Text**: the specification of software is written down in one or more models. In addition to these models, natural language text is used to explain details, the background, and the motivation of the models, but the core of the specification lies in the models.
- **Level 4: Precise Models**: the specification of the software is written down in one or more models. Natural language can still be used to explain the background and motivation of the models, but it takes on the same role as comments in source code.
- **Level 5: Models only**: the models are precise and detailed enough to allow complete code generation. The code generators at this level have become as trustworthy as compilers, therefore no developer needs to even look at the generated code.

Within MDA it is desired to strive towards level 5. However level 5 has, according to Kleppe (2003) never been reached and is currently only to hold as an ideal. At level four Kleppe states it is possible to start reaping the benefits of automated code generation at a fairly detailed level.

2.2.5 Constraint programming and OCL

A typical problem with modeling has been that while models generally give a good overview of structures and flows, it has been hard to express rules and constraints within the model. Furthermore, models have a tendency to lend themselves to interpretation. This leads to ambiguities and unclarity. Statements like “for every yellow car in database table A there should be a driver with favorite color yellow in table B” were impossible to express.

For this very reason model constraint languages where developed and the currently most prominent language is called Object Constraint Language and is part of the UML modeling standard.

OCL is a strongly typed declarative language with a mathematical foundation. The mathematical background based on use of predicate logic vouches for not leaving implications open for interpretation. OCL does not, however, use a strict mathematical notation, but rather a slight adaptation towards natural languages which makes it more accessible to the average user.
Let us illustrate with an example.

In Illustration 1 a small part of an insurance company structure is shown. From the diagram we can get an idea of what is being modeled; people own cars, cars can have several different types of insurances, in order to get insurance presumptive policyholders must make payments. From the references between the objects we can understand the basics of this structure but the diagram still leaves a lot of information hanging which is up for interpretation. We can easily point out a couple of things which are not explicitly stated, such as

- Insurance start time should be before end time
- VIN-numbers should be unique
- Coverage for damage to a car can not be higher than the value of the car

There might be other domain specific constraints that could be hard to fit into a model and which are not as apparent, such as

- Only the owner of a car can acquire insurance for the car
- In order to acquire insurance a person has to be over 16 years of age
- Insurance is only valid if all its payments have been made

This is where OCL comes into the picture. Originally OCL was meant to provide only the possibility to add constraints to UML diagrams. Contradicting the name other types of expressions have been included over time, such as queries and derived values to mention a couple.

### 2.2.6 OCL syntax introduction

OCL has a number of basic data types and operations such as integers, real numbers, strings, some basic string operators such substring and concatenation, basic arithmetic operators and boolean operators.

There are also a number of collection types built in which support ordered and unordered collections with and without duplicates. The collection types are

- **Bag**: Unordered collection with duplicates
- **Set**: Unordered collection without duplicates
- **Sequence**: Ordered collection with duplicates
- **OrderedSet**: Ordered collection without duplicates

These collection types can be converted using operators such as `MyBag->asSequence()`
Furthermore there are a number of operations working on collections such as append(obj), includes(obj), sum(), size(), first() etc. All of which are fairly self-explanatory.

All OCL expressions starts by stating the context in which they are to be validated. Say that we want to target the Person object, we would then start our OCL statement with context Person.

Let's clarify by looking at a couple of examples of how we can use OCL expressions to add detail to our insurance model from Illustration 1.

### 2.2.7 Invariants and initial values

Invariants provides a way of stating rules that should be upheld. The invariant should be true for an object during its entire lifetime. In our case we might want to write a rule to make sure that insurance start times are before their endtimes.

```ocq
context Insurance
inv: startDate<endDate
```

Let's add another invariant to make sure that vehicle identification numbers are unique.

```ocq
context Car
inv: Car::allInstances()->isUnique(VIN)
```

Now let's say we want to make sure that every time a new payment is issued it is initialized as unpaid. We can define initial values in OCL by using the `init` keyword.

```ocq
context Payment::paid
init: false
```

### 2.2.8 Queries, derived attributes and definitions

Query operations do not change the state of the system but simply queries the model and returns a value or a set of values. In the example above we could write a function that given a person returns all his ensured cars by doing the following

```ocq
context Person::getInsuredCars(): Set(Car)
body: insurances.car->asSet()
```

Derived attributes are attributes that are derived from other attributes, as an example we could derive a persons full name by concatenating his first and second name.

```ocq
context Person::fullName : String
derive: firstName.concat(' ').concat(lastName)
```

Usually most elements in a model would be introduced in the UML diagram. It is however possible to define attributes and operations from OCL. Say that we wanted to introduce the property `totalPayments` in the Person class.

```ocq
context: person
def: totalPayments : Real = payments.select(paid = true).sum()
```
2.2.9 Pre- and postconditions

Pre- and postconditions are constraints which allow for more detailed modeling of an operation by stating what conditions have to be satisfied before the operation is started and what conditions has to hold true once the operation is finished. Lets say that before a customer is allowed to acquire insurance we want to make sure they own the car to be insured.

\[
\text{context Insurance::aquireInsurance(car : Car, policyHolder : Person) }
\]
\[
\text{pre: car.owner = policyHolder}
\]

Lets add one more precondition for acquiring insurance stating that the aspiring policyholder is over 16 years of age.

\[
\text{context Person::aquireInsurance(car : Car, policyHolder : Person) }
\]
\[
\text{pre: policyHolder.age > 16}
\]

When writing postconditions it is possible to refer the condition before execution of the operation. Lets make sure that the customers age is updated correctly when we recalculate it on their birthday.

\[
\text{context Person::birthdayHappens() }
\]
\[
\text{post: age = age@pre + 1}
\]

2.2.10 Other constructs

Some constructs are unaccounted for such as \texttt{let} which gives the possibility to make a definition only valid in the scope of the current OCL-statement and \texttt{self} which is the equivalent of “this” in Java and lets an OCL-statement refer to the object of it’s context. Mostly OCL is fairly easy to read and in later examples examples will be given when necessary.
2.3 OCL implementation alternatives

2.3.1 Validation on user model level vs metamodel level

As explained in section 2.1.1 models in MDA exists in several “layers”.

- **M3**: The meta metamodel is the model normally used to defined the modeling language we choose (e.g. MOF)
- **M2**: The metamodel is the model we use to create our own models (e.g. UML)
- **M1**: The model is an abstraction of an object/process/etc. (e.g. a model of car insurance as given above)
- **M0**: Application model instance, an actual instance of our model (e.g. a number of instantiated objects in the above model)

Adding model correctness validation can be of use both on the metamodel level as well as on the model level. It is easy to see the good in making sure the instance of a model conforming to predefined rules. But it can also be of interest to make sure that the model itself conforms to certain rules.

As an example, we might want to make sure that we keep a clean namespace in our model. Thus we could create an OCL rule stating that every type of object in the model has to start with a certain prefix.

As can be seen in *Illustration 2* these OCL constraints would be defined on different levels and be applied to the underlaying model level. We therefore have to separate constraints used to ensure correctness of a model instantiation and constraints used to ensure correctness of the model itself.

This also implies different requirements on our validation. As for the model we would expect this to be static, which means one validation when initializing it would be enough to ensure it’s correctness. The model instantiation however is likely to be updated continuously and would thus require some efficient way of continuous validation. This leads us on to the next topic.

2.3.2 When to check constraints

We have stated that our constraints should be upheld at all times. One way of ensuring this would be to recheck invariants every time an object is updated. This could be optimized to only checking the invariants that relate to the current object as well as variants in other objects that depend on the current object.
Depending on the system this approach can prove too expensive. What it comes down to is weighing how expensive the desired invariants are to check and how severe an unnoticed violation would be.

As a rule preconditions should be checked on every call to an operation. But this can of course also be subject to runtime performance requirements.

Acknowledging to Warmer/Kleppe (2003) it is generally more important to do precondition checks if the system is a component in a bigger system with unknown components than if the system is closed off and well defined.

Furthermore Warmer/Kleppe states that practice has shown precondition checking to be much more important than postcondition checking.

### 2.3.3 What to do when a constraint fails

Once it has been decided how often to check invariants it's time to state how a violated constraint should be handled.

The easiest way of handling a violation would be to simply print an error message every time it happens. Say in the case of a database where a certain amount of violations is no major problem. In this situation some way of simply notifying the database administrator so that he can clean out the problems every now and then could do the trick. In general, when violations are not critical, as in a system where model integrity is not vital or when the system is in a development environment, a simple printout on every violation might suffice.

Another approach would be to roll back the change that caused the violation. Say that we have a user interactively editing the instance. In this case, when the user violates a rule a simple rollback of the violating change and a warning message might be the most suitable solution. This however requires validation to be done on every update. It also requires that information about what update was done is stored on every update so that the change can be rolled back.

When validation is not done continuously but instance integrity is still vital, another solution could be to halt the system and notify a system administrator as to what has failed.

A final example could be an instance of a system that handles some shorter amount of user interaction. In this case, if the instance causes a violation it might be sufficient to simply reset the model to a healthy starting state and let the user start over.
Chapter 3 - Method

Firstly we will take a look at the LucidEra platform. After that we will take a closer look at the problems at hand. Finally I will account for my findings.

3.1 The LucidEra platform

The LucidEra products are on demand applications. These applications run on a platform developed and hosted by LucidEra into which customer data is loaded for processing and analysis.

In the LucidEra platform a majority of the modeling is done using a set of standards designed and promoted by the Object Management Group consortium. These standards also declare support for modeling using model constraint languages.

3.1.1 LucidDB and connectors

The LucidEra database is built on open source tools. LucidDB is a relational database management system (RDBMS) built upon tools from the Eigenbase project\(^4\). The Eigenbase project provides an extensible platform for online analytics processing.

The Eigenbase project consists of two parts. A bottom part called Fennel which is a C++ implementation of the lower level tasks and a top part named Farrago which provides support for standard RDBMS tasks such as SQL parsing and execution, connection handling as well providing JAVA plugin extension support.

In order to get customer and test data into the data warehouse a number of data connectors have been developed. The external connectors are data providers implementing the Farrago SQL/MED interface. This provides the ability to query data on foreign servers issuing normal SQL commands as the data appears as foreign tables.

\(^4\) http://eigenbase.sourceforge.net/
3.1.2 Mondrian
Queries to the warehouse are issued using Multidimensional Expressions (MDX). MDX is a query language for OLAP-databases simplifying querying a database with multidimensional data sources.

Since the supported language in LucidDB is SQL this means that queries issued in MDX has to be translated along the way. Mondrian\(^5\) is an open source OLAP server providing access to LucidDB by translating MDX queries into SQL statements.

3.1.3 ALS
Application Lifecycle Services (ALS) handles the application metadata repository and provides entry points to this repository for various other packages that need to access this metadata.

This means that in order to interact with the application all tools go through ALS. ALS supports services like application template validation, ensuring that the application template conforms to LAM.

3.1.4 LEAP
LucidEra Application Processor (LEAP) is a process flow engine for running server jobs. Included in LEAP is also a scheduler for scheduling of automatic execution of jobs.

3.1.5 ClearView
ClearView is the web interface that users interact with in order to create reports. Using web technologies such as Ajax ClearView provides an interactive way to let the user design custom reports using drag and drop techniques. ClearView also provides access to certain administration and management tasks such as setting up a company's profile and application settings as well as user management.

\(^5\) http://mondrian.pentaho.org/
3.2 MDE at LucidEra

3.2.1 FEM and LAM

The **Farrago Extension Metamodel** (FEM)\(^6\) is a model created by the Farrago team which extends upon CWM by adding support for newer SQL-standards than CWM does. Also the FEM-model is designed to be further extended by systems that extends Farrago.

The **LucidEra Application Model** (LAM) extends upon FEM to create a format in which to model the LucidEra applications. The LAM-extensions are at large dividable into four groups.

- Application specific elements to describe details such as application version, application build number and concepts such as deployed instances, external bindings etc.
- Rdbms specific elements bringing up SQL views, managed tables, external tables etc.
- Workflow import enabling the usage of JBPM processflows defined in XML
- Online analytics processing (OLAP) elements defining concepts such as measures, data dimensions and cubes etc.

3.2.2 LucidEra Applications

LucidEra has a number of products such as **Lead analysis**, **Pipeline analysis** and **Sales analysis**. These products are referred to as applications and run on top of the LucidEra platform stack.

The applications specify operations to extract data from the client data sources, transformations to get the data into a usable format and loads to get the data in its final form into the LucidDB database.

All these procedures are modeled using LAM and stored as XMI-objects which are loaded and resides in the metadata repository.

3.2.3 Workbench

The Workbench is an internal tool for application template development and server management.

\(^6\) http://pub.eigenbase.org/wiki/FEM
The workbench is built on top of the Eclipse Rich Client Platform (RCP)\(^7\) which is a Java-based platform for building applications. Essentially everything in RCP is a plug-in extending a fairly small core set of functionality.

Workbench is such a plug-in providing the possibility to connect to a LucidEra server. Once connected Workbench provides features like loading/deployment of applications and creating/modifying/exporting applications.

### 3.3 Assessing suitable extension points for OCL

#### 3.3.1 Primary suggestions

When starting the project two desired starting points had already been identified.

The first point would be to apply declarative constraint checking on the FEM instantiations residing within the LucidDB MDR.

The second suggested starting point would be the LAM models residing within the ALS MDR. Integrating OCL here would allow adding constraints to prevent corruption to application instantiations in the MDR.

In the past problems with maintaining this correctness had arisen in both LucidDB and the ALS MDR. In some cases this even caused severe loss of development effort as in the case when several weeks worth of work was lost due to a bug causing corruption of an application template under development, rendering the work useless and application implementation had to be redone.

#### 3.3.2 Secondary suggestion

Yet another place where OCL was deemed to be useful was identified within the Workbench. One of the application object types that the Workbench provides is the option to edit is process flows.

Process flows define the flow of actions and events within an application and are executed as steps, each step defining the transition to another until the end state of the process flow has been reached.

Execution of process flows are done by the a JBoss Business Process Management (jBPM) engine. Process flows are defined in the jBPM Process Definition Language (jPDL).

---

As can be seen in Illustration 3 the existing editor for process flow content was simply a text editor with XML syntax highlighting. Plans were presented to make this editor into a fully graphical flow editor with drag and drop and object palettes and a flow canvas where the flows were to be represented as graphs with actions as nodes and transitions as edges. As a part of this effort, the question was raised, whether or not a number of the constraints imposed on the user within such an editor could be simplified by implementing support for OCL.

3.4 Assessing suitable OCL frameworks

A number of OCL tools were examined for suitability before a choice was made. In case of the jPDL process flow editor the choice was made easy seeing that the Workbench is developed as an Eclipse RCP plugin and that the Eclipse Modeling Framework already included OCL support. However in case of FEM the sheet was blank and a number of different toolkits had to be examined.

3.4.1 Framework requirements

Since both the FEM and LAM are custom made meta models extending the CWM the chosen framework would have to be applicable to CWM. In order to provide real time constraint checking the framework would also have to be able to interact with the Netbeans MDR.

3.4.2 OCLE

OCLE is a CASE toolset created and maintained by the Computer Science Research Laboratory at Babes-Bolyai University. This tool set features an intuitive GUI and can operate on a metamodel definition as well as on a user model. A JAVA code generator is included which generates full class structures and methods skeletons with runtime OCL constraint validation attached. Furthermore the OCLE project did not seem to have an activity since July 2005.

3.4.3 OSLO

OSLO (Open Source Library for OCL) is based on the Kent OCL library which was originally developed by the Computing Laboratory at the university of Kent. The OSLO toolkit works only on metamodel level. At the time of research the OSLO-project had not been updated since November 2005.

3.4.4 The Dresden toolkit

The Dresden OCL Toolkit is an open source, Java implementation of the OCL framework created and maintained by the Software Engineering group at Technische Universität Dresden. The development was started in 1999 and today the toolkit consists of a software platform containing a number of tools to validate OCL syntax, parse OCL and to generate and infuse OCL code into existing Java code.

3.4.5 OCL in Eclipse

In the open source Eclipse Modeling Framework (EMF) the metamodeling standard is called Ecore, and is very closely linked to EMOF. Models in EMF can e.g. be
specified using XML, UML or annotated Java and imported into EMF. Given a model specification EMF can generate Java classes for the model, along with adapter classes allowing for command line based editing of the model and a stripped down editor utilizing the adapters.

Within the scope of the Eclipse Modeling Framework resides a number of validation plug-ins. These validation plug-ins has out of the box support for Java and OCL as validation languages. Within EMF there is also support for generating simple, tree based, graphical Ecore model editors.
Chapter 4 - Results

4.1 Applying OCL
The approach was to start targeting the primary OCL insertion points. As this proved difficult the secondary insertion point was located and implemented.

4.1.1 Farrago metamodel and LAM validation
The primary insertion targets, that had been identified before the project start, were assessed. A four step approach plan was created.

1. Create Farrago instance export and evaluate an OCL rule on it using a tool and the Farrago metamodel
2. Make an integration with the Farrago MDR and do the validation against it
3. Evaluate interesting use cases for OCL on the Farrago metamodel
4. Evaluate interesting use cases for OCL on the LucidEra application model

As the catalog export from Farrago used the FEM metamodel the selected OCL framework would have to support some way of evaluating OCL on a custom metamodel. A look at OCLE revealed that the tool is closely coupled with the UML-metamodel and would not be usable. The OSLO project could also be discarded as it was limited to UML/Ecore metamodels.

The most promising approach seemed to be using the Dresden Toolkit since its implementation used the Netbeans MDR, the same metadata repository that Farrago uses. This might ease integration. At the same time the Dresden project seemed to be the one with the most activity and the most recent changes.

A promising first step towards getting a single rule to validate against exports of the Farrago metamodel and the Farrago catalog was made as the the Farrago metamodel was loaded into the Dresden toolkit workbench together with a Farrago catalog export and an OCL invariant was successfully executed.

The OCL provided was however written in a semi parsed format called OCL script. Further investigation revealed that this was an old solution put in place to facilitate testing of the repository during development of the OCL parser. In an email correspondence, the Dresden development team confirmed that OCL script was the only accepted OCL input unless the model was defined in plain MOF, and that OCL script was only an arcane testing language.

The Farrago Metamodel is created according to the procedure in Illustration 4 and thus incorporates the OMG Common Warehouse Metamodel (CWM).

Further correspondence revealed the preferred way around the problem which would be to use a pivot model to transform Farrago meta model into a MOF metamodel, which was deemed to be outside the scope of the thesis project.
This effectively put a stop to the rest of the steps as they all relied on this approach to work out.

**4.1.2 The Workbench process flow editor**

A first outline of the graphical process flow editor made use of the fact that jPDL is modeled using Ecore as a metamodel. Since this was the case a straight forward path was to use the Eclipse *Graphical Modeling Framework* (GEF)\(^8\) in order to create the editor. GEF is part of EMF and provides ways to use the Ecore-definition of jPDL to automatically generate large parts of the code needed to create a graphical editor.

The generated editor would however need a lot of hands on coding as many of the limitations in jPDL are not explicitly modeled in its model, such as that there can only be one start-node in each process flow. Also LucidEra had a number of further restriction on what a process flow is allowed to contain, e.g. application process flows should not allow fork-nodes.

A project plan was made.

1. Get OCL evaluation up and running on jPDL instances
2. Draft up OCL use cases
3. Implement proof of concept

**Step 1 – get it running**

Since the graphical process flow editor was still in development at the time of research start, a temporary editor was needed. EMF has powerful features for creating simple model editors using a metamodel. So using the jPDL metamodel schema a very basic editor was generated.

The Eclipse OCL SDK contained a number of validation examples. As an initial step one of these examples, an OCL console example, could with some minor changes be connected to the sample editor. This enabled validating OCL on jPDL instances statement by statement.

After some additional changes a couple of OCL validation examples were modified to work on the same sample graphical editor. Now the full power of OCL was accessible at a fairly cheap implementation cost.

---

\(^8\) [http://www.eclipse.org/modeling/gmf/](http://www.eclipse.org/modeling/gmf/)
This meant that OCL expressions could be collected in one place. An invariant would be stated in XML. As can be seen in Code example 1 the OCL context is set by the target class tag. The mode property can be set to either Batch or Live. In case the constraint is defined as live the triggering event can be defined. The actual invariant is put in a CDATA block. Code example 1 defines a constraint stating that all process states must have unique names.

```xml
<constraint
  lang="OCL"
  severity="WARNING"
  mode="Live"
  name="Live mode constraint 3"
  id="LiveExample3"
  statusCode="103">
  <description>Unique process names rule</description>
  <message>Process names must be unique "{0}"</message>
  <target class="ProcessDefinitionType">
    <event name="Set">
      <feature name="Name"/>
    </event>
  </target>
  <![CDATA[
    processState->forAll(p1, p2 | p1.name=p2.name implies p1=p2)
  ]]>)
</constraint>
```

*Code example 1: Example of OCL invariant in Eclipse*

**Step 2 – use cases**

There are a number of different things to consider when dealing with validation when the model instance is changed directly by the user. Firstly we have to consider when an invariant is to be checked. Some invariants lends themselves well to real time checking. For example, if we do not allow a node to define a transition to itself, a warning message could be popped up straight away if such a transition is created. However other constraints might not be as suited to check in real time. Say that we don’t allow the existence of unreachable nodes. If we checked for this in real time the user would be warned every time a new node was added to the canvas, before the user had a chance to add an incoming transition to it. This would just be a major annoyance to the user. So this might be a rule that would be suitable to check in a bigger batch validation, say before saving or when the user invokes it.

A second parameter to consider is what to do when an invariant fails. In real time checking we could choose to simply pop up a user warning. We could also choose to make a rollback of the latest edit that caused the inconsistency. When making a batch validation before saving a third option would be not to allow the user to save the document.

The chosen approach was to identify all failing constraints that are “active” modeling errors, i.e. errors that the user actively introduces, such as attaching incoming transitions to the start node. In these cases we chose to show an error message before rolling back the action.
The user was given the ability to manually invoke a batch constraint check which triggers a check of all invariants. If any constraints should fail an appropriate error message is given explaining what constraint was broken.

Furthermore trying to save the current process flow also invokes a full batch validation. If the validation fails the user is given an error message and is not allowed to save the process flow.

A number of constraints were identified and then categorized as seen in Table 2.

<table>
<thead>
<tr>
<th>Restriction</th>
<th>Action</th>
<th>Type of evaluation</th>
<th>OCL statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes are not allowed to have transitions to themselves</td>
<td>Warning + rollback</td>
<td>Live + batch</td>
<td>context processDefinition inv: processState-&gt;forAll(p</td>
</tr>
<tr>
<td>Start nodes cannot have incoming transitions</td>
<td>Warning + rollback</td>
<td>Live + batch</td>
<td>context processDefinition inv: processState-&gt;forAll(p</td>
</tr>
<tr>
<td>End nodes cannot have outgoing transitions</td>
<td>Warning + rollback</td>
<td>Live + batch</td>
<td>context processDefinition inv: endState.transition-&gt;size()=0</td>
</tr>
<tr>
<td>There has to be exactly one start/end node</td>
<td>Warning</td>
<td>Batch</td>
<td>context processDefinition inv: endState-&gt;size() = 1</td>
</tr>
<tr>
<td>Node names must be unique</td>
<td>Warning + rollback</td>
<td>Live + batch</td>
<td>context processDefinition inv: processState-&gt;forAll(p1, p2</td>
</tr>
<tr>
<td>Fork nodes are not allowed</td>
<td>Warning + rollback</td>
<td>Live + batch</td>
<td>context processDefinition inv: fork-&gt;size()=0</td>
</tr>
<tr>
<td>For every node containing a SET command there has to be a corresponding GET command node</td>
<td>Warning</td>
<td>Batch</td>
<td>context processDefinition inv: self.node-&gt;forAll( n1</td>
</tr>
<tr>
<td>Exception handling must be present in all process flows</td>
<td>Warning</td>
<td>Batch</td>
<td>context processDefinition inv: not exceptionHandler-&gt;first().oclIsUndefined()</td>
</tr>
<tr>
<td>Maximum call size depth should be controllable</td>
<td>Warning</td>
<td>Batch</td>
<td>context processDefinition inv: processState-&gt;size() &lt; %maxDepth</td>
</tr>
</tbody>
</table>

Table 2: Process flow constraints
Step 3 - Implementing proof of concept

The final process flow editor is a graphical editor created using Eclipse Graphical Editing Framework (GEF)\(^9\) and EMF. This means that the original use of the OCL example plugins used in the generated editor could be reused without any major rework.

Additional benefits

Since OCL can be used for more than just adding validation constraints, the usefulness of using OCL as a query language was examined. Using the same API as the OCL terminal mentioned above, queries can be evaluated and model objects returned.

This enables us to do on the fly searches in the model on arbitrary object properties. Something that could potentially become quite useful. Say for example that we want to grab all process state nodes with a certain property and do something to them. Using Java this would require for us to start digging through the model to get a hold of these node objects. The same thing can easily be done issuing a single OCL statement.

Chapter 5 - Conclusions

In this thesis I have looked into the possibilities to add declarative model correctness checks using OCL to an existing platform. The main goal has not been to do a full implementation to be used in production but rather to assess what routes would be possible and what advantages can be obtained at a low cost, and where possible create a proof of concept.

5.1 OCL, theory vs. practice

It is easy to agree on that the thought of an easy way to add declarative checks to all models that might be part of ones current software is an appealing idea. The question is at what cost this idea comes. Is the implementation worth the effort?

5.1.1 Ease of implementation

At a first glance LucidEra seems like the perfect candidate for application of OCL. The company does a lot of data handling and transformation making heavy use of models for data and metadata based on OMG standards.

However the choice of of modeling languages was eventually the deal breaker. This goes to show that in order to get model integrity checking with OCL for free, or at least for cheap currently means sticking to the most common modeling languages.

As in the case of the process flow editor where the metamodel was supported by existing tools, implementation could be made at fairly low effort.

5.1.2 Usefulness

There is no question that LucidEra can benefit by adding OCL support in their platform stack.

As in the example of the Workbench and the Process Flow Editor, the OCL validation can be considered low hanging fruit, and the EMF hierarchy makes for a clean and easily separable implementation, where OCL constraints can be kept separate from other code in a clean and extensible manner.

The OCL constraints also provide easy to read one-line rules that would be tedious to implement in program code.
Bibliography

Introduction to OCL including full language reference.

ISBN 032119442X
Overview of MDA concepts.

ISBN 0471319201
An in-depth guide of how to use MDA in software development.

(Accessed on Dec 28, 2007)
Master thesis at Dresden University of Technology.

(Accessed on Dec 28, 2007)
Presentation background material from an MDA workshop.

ISBN 0596008430
Overview of BPM with explanation of common BPM design patterns.

(Accessed on Dec 28, 2007)
Workshop material from the MoDELS conference of 2006 in Genova, Italy.

[8] *Farrago Extension Metamodel (FEM)*
http://farrago.sourceforge.net/fem/index.html
(Accessed on Dec 28, 2007)
An overview of the Farrago Extension Metamodel

http://www.cs.wustl.edu/~schmidt/GEI.pdf
(Accessed on Dec 28, 2007)
Article giving brief history and introduction to MDE

[10] *Object Constraint Language Environment (OCLE)*
http://lci.cs.ubbcluj.ro/ocle/
(Accessed on Dec 28, 2007)
Home of the OCLE UML CASE Tool
http://dresden-ocl.sourceforge.net/  
(Accessed on Dec 28, 2007)  
Home of the Dresden OCL toolkit

[12] Open Source Library for OCL (OSLO)  
http://oslo-project.berlios.de/  
(Accessed on Dec 28, 2007)  
Home of the OSLO project

(Accessed on Dec 28, 2007)  
Short usage description for the OSLO OCL tool.

http://www.cs.kent.ac.uk/projects/ocl/  
(Accessed on Dec 28, 2007)  
Home of the Kent OCL project
Appendix 1 – Glossary

CASE (Computer Aided Software Engineering)
Tools that help a software engineer develop and maintain software. (Introduced in chapter 2.1.1)

CWM (Common Warehouse Metamodel)
CWM is a specification for modeling data warehousing objects. CWM is designed by OMG and is level M2. (Introduced in chapter 2.2.2)

DSL (Domain Specific Language)
Programming or specification language tailored to a specific problem domain. (Introduced in chapter 2.2)

EMF (Eclipse Modeling Framework)
An open source modeling framework based on Eclipse. EMF enables code generation for creating tools based on model data.

Ecore
Ecore is EMF’s metamodel. It is a subset of MOF geared towards being a generic object-oriented metamodel.

FEM (Farrago Extension Metamodel)
FEM is a metamodel used to describe how Farrago metadata is modeled. FEM extends CWM and is designed to in turn be further extended by systems which extends Farrago. (Introduced in chapter 3.2.1)

jBPM (JBoss Business Process Management)
A JBoss workflow engine implemented in Java. Can execute processes described in jPDL.

JMI (Java Metadata Interface)
JMI defines creation, access, lookup and exchange of metadata in the Java programming language.

jPDL (jBPM Process Definition Language)
XML based representation of JBoss process definitions.

LAM (LucidEra Application Model)
The model defining the applications used on the LucidEra platform. LAM Extends CWM and FEM. (Introduced in chapter 3.2.1)

M0-M3 (modeling layers)
This refers to the layers describing instances, models, metamodels and meta-metamodels. See Table 1 on page 3 for more details. (Introduced in chapter 2.2)

MDA (Model Driven Architecture)
An approach to software design launched by OMG. It describes the usage of different layer models and translation tools for generating runnable code from a platform independent model. (Introduced in chapter 2.2)

MDE (Model Driven Engineering)
An engineering approach in which models are considered the primary design objects. (Introduced in chapter 1.1.1)

MDR (MetaData repository)
A metadata repository is, as the name suggests a storage facility for data about data. Its purpose is to provide consistent and reliable means of access to data.

MOF (MetaObject Facility)
MOF is a metamodel designed to be in the M3 layer. MOF was designed when OMG needed a metamodel architecture to define UML. (Introduced in chapter 2.2.1)

OCL (Object Constraint Language)
A declarative language for adding rules within a MOF model like UML. (Introduced in chapter 1.2)

OMG (Object Management Group)
A consortium founded in 1989 aiming to developing standards for modeling and model-based standards. (Introduced in chapter 2.2)

PIM (Platform Independent Model)
A model independent of platform specific details used to generate PSMs for the various platforms needed. (Introduced in chapter 2.2.3)

PSM (Platform Specific Model)
A model used to generate code for a specific platform. (Introduced in chapter 2.2.3)

RCP (Rich Client Platform)
A set of components that enables a programmer to quickly build fully featured GUI-based applications. In this case I will exclusively refer to the Eclipse RCP. (Introduced in chapter 3.2.3)

UML (Unified Modeling Language)
A general purpose metamodeling language for creating models. UML includes a graphic notation allowing design using diagrams. (Introduced in chapter 2.1.1)

XMI (XML Metadata Interchange)
An OMG standard for exchanging metadata information using XML. (Introduced in chapter 2.2.2)