Harvesting existing software systems for MDA-based reengineering

Thijs Reus
January 10, 2006
Harvesting existing software systems for MDA-based reengineering

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

SOFTWARE ENGINEERING

by

Thijs Reus
Born in Alkmaar, The Netherlands
Cover picture: Julien Dupré (1851 – 1910)

*Harvest Time*

Oil on canvas (26 x 32 inches)
Signed
Provenance
Private collection, New York

Note: the order of the company listing is solely chronological with respect to this project
Harvesting existing software systems for MDA-based reengineering

Abstract

Software systems degrade over time: they are affected by a phenomenon called aging. It makes using and changing the software hard, risky and expensive. Therefore migration projects are initiated, for example reengineering projects to revitalize degraded systems. OMG’s MDA provides means to prevent, or at least slow down, software aging. It focuses on models rather than code, which is merely a derivate of the models. MDA makes reengineering projects more attractive by preventing the necessity of reengineering again in the near future. Currently MDA-based reengineering gains more attention both in industry and academics.

This thesis describes the process and results of the Master project in which we aimed to provide means and a process for MDA-based reengineering. A flexible workbench solution has been developed for reverse engineering textual sources into MDA-ready UML models, a process which we call harvesting. The commercial, state of the art MDA-tool ArcStyler has been used as target environment for harvested UML models. To improve reusability of harvesting components, a generic model is used as an intermediate step in the harvesting process, which makes investments in more comprehensive harvesting assets economically attractive. It is a first attempt to provide an initial generic intermediate model, for which evolution during usage in different projects is foreseen.

The workbench has been tested on a 178 KLOC PL/SQL production system in a case study. The textual sources have been harvested using the generic intermediate model, based on which UML class and collaboration diagrams have been created to give insight in the structure and to some extent in the behavior of the system. Although the current prototype implementation suffers from scalability issues, parallel harvesting has proved to be an effective countermeasure. As a showcase, the harvesters have been adapted to another PL/SQL system, of which 33 KLOC have successfully been harvested to show adaptability of the solution.

The generated UML models have been used for documentation and code generation following the MDA philosophy, currently resulting in code without behavior implementation.

Thesis committee:

Chair: Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor: Ir. H.J.A.M. Geers, Faculty EEMCS, TU Delft
Company supervisor: J. Rommel, Diplom.-Phys., Interactive Objects GmbH
Company supervisor: L.M. van de Geest, De Amersfoortse Verzekeringen
Committee member: Dr. K.G. Langendoen, Faculty EEMCS, TU Delft
This thesis is the tangible result of my Master project as a conclusion of my study Computer Science at the University of Technology, Delft. During this project I could combine science and business, which is a very challenging combination. It allowed me to use the knowledge and experience from university courses in a business context, where in the end the solution has to work. A theoretical exercise on how to solve a problem does in fact not yet solve the problem.

It was a unique opportunity to do this project in cooperation two different companies. The one is a high-tech front-runner in the model-driven development area; the other is a specialized insurance company for ensuring income, part of a large, international financial institute. The one has its head-quarters at the border of the Black Forest in Germany; the other has its office in Amersfoort, The Netherlands. Both are internationally active, both are outstanding in their area of expertise.

The first part of the project I performed at Interactive Objects GmbH in Freiburg, Germany. It is a highly motivating environment where new, state of the art technology is continuously being born. This environment, set by inspiring colleagues, allowed me to have a wonderful time in Freiburg. I hereby wish to thank Simon, who is always enthusiastic and has always directly a clear opinion and does not hesitate to share it, and Jens, who is the ‘guy who is always right’ and thinks well before giving a statement, for being such great supervisors. They both helped me when it was needed, inspired me with ideas and provided valuable feedback at any stage in the project. Besides that, always ready for some fun or a barbeque in the weekend. I also want to thank my other colleagues (just too many too mention here) for having such a great time.

The second part of the project I performed at De Amersfoortse Verzekeringen in Amersfoort, The Netherlands. At De Amersfoortse it all began by sending Lex an email that I was looking for an interesting project as a student. Coincidentally he was about to have a project about harvesting with a company from Freiburg, so I could fortunately join in. Here I got to know Interactive Objects and before I knew it, the project became the step-stone for my Master project. I am very grateful to Lex, the man with a vision about business IT architecture and new, promising technologies and a great network, for supporting me throughout both projects, providing all the help I could possibly want and involving me in many different areas of business IT. Without Lex, this project would have never been possible at all. Of course, I also want to thank all my other colleagues for providing such a great environment.

I am very grateful to both Interactive Objects and De Amersfoortse Verzekeringen for providing the possibility to do this project, without any bureaucracy or alike; it was really a unique opportunity and an excellent cooperation. I wish to thank Hans and Arie, from the university, for being open-minded about this project from the beginning and being very supportive to the end. Writing a thesis is not an easy task, but Hans supported me greatly with valuable suggestions and remarks.

Special thanks go to my parents Vincent and Ria for their always enthusiastic support, and to my girlfriend Marleen for always being there for me.

THIJS REUS
Delft, The Netherlands
January 10, 2006
# Table of Contents

**Preface** ............................................................................................................................................ 1

**Table of Contents** .......................................................................................................................... 3

**List of Figures** ............................................................................................................................... 7

**List of Tables** .................................................................................................................................. 9

**Introduction** ................................................................................................................................... 11

1. **Context & Motivation** .................................................................................................................. 13
   1.1 Thesis Structure .......................................................................................................................... 14
   1.2 Figures ........................................................................................................................................ 14
   1.3 Terminology .............................................................................................................................. 15
      1.3.1 Reengineering .................................................................................................................. 15
      1.3.2 Models ............................................................................................................................... 16
      1.3.3 Parsing ............................................................................................................................... 16
   1.4 Conceptual Approach .................................................................................................................. 17

2. **Technological Context** .................................................................................................................. 19
   2.1 MDA – Model Driven Architecture ............................................................................................ 20
      2.1.1 ArcStyler ......................................................................................................................... 20
      2.1.2 M2M-Transformations ..................................................................................................... 21
   2.2 ADM – Architecture Driven Modernization ............................................................................. 22
   2.3 Meta-modeling ........................................................................................................................... 23
      2.3.1 MOF – Meta Object Facility ............................................................................................. 23
      2.3.2 JMI – Java™ Meta-data Interface ..................................................................................... 24
   2.4 Reverse Engineering Process ..................................................................................................... 24
      2.4.1 Methods of Extraction ....................................................................................................... 25

3. **Industrial Partners** ....................................................................................................................... 27
   3.1 Interactive Objects GmbH .......................................................................................................... 28
   3.2 De Amersfoortse Verzekeringen ................................................................................................. 28
   3.3 VDA Informatiebeheersing ........................................................................................................ 29

4. **Research Problem** ......................................................................................................................... 30
   4.1 Problem Definition ..................................................................................................................... 30
   4.2 Research Questions ..................................................................................................................... 31
      4.2.1 Develop Generic Harvesting Framework ........................................................................... 31
      4.2.2 Reengineer Software System ............................................................................................ 31

5. **Generic Harvesting Framework** .................................................................................................. 33
   5.1 Research Problem ....................................................................................................................... 34
   5.2 Requirements .............................................................................................................................. 34
   5.3 Non-functional Requirements ...................................................................................................... 36
   5.4 Testing the Solution ..................................................................................................................... 36
6 Harvesting Workbench ................................................................. 26
   6.1 Approach .................................................................................. 26
   6.2 Architecture ............................................................................ 27
   6.3 Harvester Generator ................................................................. 27
      6.3.1 Parser Generator ............................................................... 28
      6.3.2 Repository Population Strategies ..................................... 33
      6.3.3 Repository Population Algorithm ................................. 35
      6.3.4 Test Results .................................................................................. 38
   6.4 Repository Generator ............................................................. 39
      6.4.1 Grammar to MOF Mapping .............................................. 40
      6.4.2 Test Results ................................................................................. 46
   6.5 Code Generation ......................................................................... 46
   6.6 Conclusion .................................................................................. 47

7 GenericAST ..................................................................................... 49
   7.1 Approach .................................................................................. 49
   7.2 Rationale ................................................................................... 50
      7.2.1 Reuse ...................................................................................... 50
      7.2.2 UML Meta-Model ................................................................. 51
      7.2.3 Representation ........................................................................ 51
      7.2.4 Cost ........................................................................................ 51
   7.3 Architecture ................................................................................ 52
   7.4 Meta-Model.................................................................................. 53
      7.4.1 Structural Part ........................................................................ 53
      7.4.2 Behavioral Part .......................................................................... 55
   7.5 Frameworks & Tools .................................................................. 56
      7.5.1 Analysis Framework ............................................................. 56
      7.5.2 Annotation Framework ....................................................... 57
      7.5.3 Model Element Classification ............................................ 59
      7.5.4 Graphical User Interface ..................................................... 61
      7.5.5 Model Merger ........................................................................... 63
   7.6 Test Results ................................................................................ 64
      7.6.1 Structural Part ........................................................................ 64
      7.6.2 Behavioral Part ......................................................................... 65
   7.7 Risk Assessment .......................................................................... 66
      7.7.1 Model Size .............................................................................. 66
      7.7.2 Transformation Performance ............................................ 68
      7.7.3 Genericity Mismatch ............................................................ 71
   7.8 Conclusion .................................................................................... 72

8 Conclusion ....................................................................................... 74

9 Future Work .................................................................................... 75
   9.1 Harvesting Workbench ............................................................. 75
      9.1.1 Support Island Grammars ................................................... 75
      9.1.2 Grammatica Improvements ............................................... 75
      9.1.3 Model Filtering During Parsing ......................................... 76
   9.2 GenericAST ................................................................................ 76
   9.3 Total Process ................................................................................. 77
      9.3.1 Model Driven Harvesting ................................................... 77
      9.3.2 Refactoring Support ........................................................... 78
      9.3.3 Lazy Harvesting ...................................................................... 80
LIST OF FIGURES

Figure 1-1 Existing system decision matrix [JL 91] ................................................................. 3
Figure 1-2 High-level harvesting process ................................................................................. 4
Figure 1-3 Figure legend for processes and architectures ......................................................... 5
Figure 1-4 Terminology – reengineering .................................................................................. 6
Figure 1-5 Terminology – models ............................................................................................ 6
Figure 1-6 Terminology – parsing ............................................................................................. 7
Figure 1-7 Example AST ............................................................................................................. 7
Figure 1-8 From text to UML model – how to harvest? .............................................................. 7
Figure 1-9 From text to UML model – using a parser ................................................................. 8
Figure 1-10 From text to UML model – using a generated MOF meta-model ......................... 8
Figure 1-11 From text to UML model – using a generic intermediate model ......................... 9
Figure 1-12 Artifacts controlling the transformation processes ................................................. 9
Figure 2-1 MDA supports multiple target platforms ............................................................... 11
Figure 2-2 ArcStyler architectural IDE ................................................................................... 12
Figure 2-3 Models and meta-models ....................................................................................... 13
Figure 2-4 Meta-models and meta-meta-models ..................................................................... 14
Figure 2-5 General architecture for reverse engineering tools [Moo 02] ................................... 15
Figure 4-1 Gap between existing systems and MDA-based forward engineering .......... 19
Figure 5-1 Gap between existing systems and MDA-based forward engineering .......... 23
Figure 6-1 Harvesting workbench overview ........................................................................... 26
Figure 6-2 Harvesting workbench usage overview ................................................................. 27
Figure 6-3 Harvester generator overview ................................................................................. 28
Figure 6-4 Grammatica internal AST structure ....................................................................... 30
Figure 6-5 Grammatica internal AST structure for example production ............................... 30
Figure 6-6 Example AST walk call sequence ........................................................................ 35
Figure 6-7 Corresponding AST ............................................................................................... 35
Figure 6-8 Harvested model from Haebop PL/SQL source code (in UML Profile for EJB) .. 39
Figure 6-9 Repository generator overview .............................................................................. 39
Figure 6-10 Normal meta-model style class overview (in UML Profile for MOF) .......... 42
Figure 6-11 Mapping two production alternatives to MOF (in UML Profile for MOF) ........ 44
Figure 6-12 Mapping a named element to MOF (in UML Profile for MOF) ....................... 44
Figure 6-13 Mapping multiplicities to MOF (in UML Profile for MOF) ............................... 44
Figure 6-14 Compact meta-model style class overview (in UML) ......................................... 45
Figure 6-15 Part of generated meta-model (in UML Profile for MOF) ................................. 46
LIST OF TABLES

Table 2-1 MOF 4-layer meta-data architecture ................................................................. 14
Table 6-1 Test results for different population strategies .................................................. 34
Table 6-2 Named population strategy compared to normal population strategy ............. 35
Table 6-3 Valid match example ....................................................................................... 38
Table 6-4 Another valid match example ......................................................................... 38
Table 6-5 Repository contents for meta-model styles and population strategies .......... 40
Table 6-6 MOF 4-layer meta-data architecture with EBNF example ............................. 41
Table 6-7 Mapping EBNF constructs to MOF constructs ................................................. 43
Table 6-8 Test results for different meta-model styles ...................................................... 45
Table 6-9 Compact meta-model style compared to normal meta-model style ............... 46
Table 6-10 Code generation ratio for different grammar sizes ....................................... 47
Table 9-1 Refactoring comparison .................................................................................. 79
Table 9-2 Lazy harvesting compared to traditional harvesting ....................................... 80
Table 12-1 Haebop harvester measurements ................................................................. 95
Table 13-1 Haebop system size ...................................................................................... 99
Table 13-2 Haebop package procedure complexities ...................................................... 100
Table 13-3 Haebop complexity overview ...................................................................... 100
Table 16-1 ADM compared to GenericAST .................................................................. 114
Table B-1 Parser generator comparison ....................................................................... 136
Table E-1 Interviews and official meetings with Haebop stakeholders ......................... 148
Table E-2 Documents about Haebop ............................................................................. 148
1 **Context & Motivation**

Companies rely on software systems for successful execution of their business processes. We can think of systems for customer administration, product administration, process software, etc. In many cases the systems have been developed and tailored for (or even completely by) the company that uses them. They are the result of many years of development, repairs and changes.

Software systems are affected by a phenomenon called software-aging [Par 94], which is the degradation of software over time. Degraded systems suffer from the following shortcomings [BLW 97]:

- Systems lack documentation, which makes system comprehension hard
- Systems run in old environment, which is slow, expensive to maintain, and hard to understand by current IT-staff
- Systems are hard to modify thus expensive

Aging makes responding to changing business processes and requirements hard, risky and expensive. Improving cost effectiveness, quality and flexibility is an important business incentive for initiating migration projects [LDP 99]. Four options for acting on degraded systems can be chosen: discarding, maintaining, enhancing, and reengineering. The appropriate choice depends on both the business value and the changeability of the system, as shown in figure 1-1 [JL 91]. For example, a system that resists change and has high business value should be reengineered to lower the risks associated with degraded software.

![Figure 1-1 Existing system decision matrix (JL 91)](image)

This Master project will provide tools and experience to reverse engineer software systems to an MDA target context. Reverse engineering is helpful in all but the discard scenario. It provides information on the current operational state of the system and may even provide a basis for forward engineering. We will focus on reengineering because it imposes the most exhaustive requirements for the extracted models, should they be usable for MDA-based forward engineering.

Our solution will support answering specific questions about an existing software system by using its source code as input. In several steps, the source code will be converted to a model which can be analyzed to discover the answer. Examples of possible questions are ‘How does the data structure look like?’, ‘Which modules depend on which other modules?’, and ‘Where is this functional unit used?’. The answer to the question should of course be present in the source code; our challenge is to get it out. Figure 1-2 shows the anticipated high-level harvesting process. Section 1.4 elaborates the harvesting process of textual input to UML models in more detail.
MDA is an initiative to counter the problems with degraded software systems that were mentioned before. It focuses software engineering on models rather than on source code: models become the main software assets from which source code is derived. Using the MDA approach reengineering is attractive on the long term, because it enables prevention of software system degradation.

We have developed a toolset to extract models from existing source code and present it in ArcStyler, a commercial MDA-tool. The toolset has been developed at and in cooperation with Interactive Objects GmbH. We have tested the toolset in a case study on a production system at De Amersfoortse Verzekeringen. The system has been developed by VDA in Oracle PL/SQL\textsuperscript{1}. It is an example of an ‘ideal’ software system for reverse engineering in this project, because it is documented, clearly structured and recent. Above all, the developers are still present for sharing knowledge about the system and the environment. Therefore we could concentrate on the value of the toolset instead of obscure degraded software issues.

1.1 Thesis Structure

The first part of the thesis gives an introduction to the subject and gives it a theoretical context. The used technologies are described in chapter 2. The industrial partners are introduced in chapter 3. The research questions and goals are elaborated in chapter 4.

In part two, ‘Generic Harvesting Framework’, we describe the toolset. First the research problem is specified and the requirements for the solution are defined in chapter 5. In chapter 6 the source code extraction phase is discussed. In chapter 7 we introduce a generic intermediate model for transformation and analysis reuse. Chapter 8 and 9 finish this part with the conclusions and future work regarding the generic harvesting framework.

The case study is described in the third part ‘Case Study’. In chapter 10 the case study is introduced by stating the research problem, showing our approach and explaining why the used system was chosen. Chapter 11 gives a background on the used system and what information has been used for harvesting. Development of harvesters is discussed in chapter 12 with both approach and experiences. A glimpse of the harvested results is shown in chapter 13 in both numbers and diagrams. Chapter 14 and 15 conclude the part with conclusions and future work discussion.

The thesis is concluded with the part ‘Conclusions’. We discuss related work in chapter 16 and show our contributions in chapter 17. The conclusions are summarized in chapter 18, in chapter 19 we briefly discuss and reflect our solution. Finally the future work is summarized in chapter 20.

\textsuperscript{1} http://www.oracle.com/technology/tech/pl_sql/index.html
1.2 Figures
Several figure styles are used throughout this thesis. The chosen style depends on the purpose of the figure. The UML style is determined by the ArcStyler UML tool. Figure 1-3 shows the elements used in several process and architecture figures. Names in the diagram elements are usually self explanatory and arrows are usually specified with a short description.

Figure 1-3 Figure legend for processes and architectures

1.3 Terminology
This section gives a brief overview of the main terms that are used in this thesis. It gives the terms a more specific meaning. We refer to the glossary in appendix A at page 133 for a formal definition of the terms.

1.3.1 Reengineering
In this project we use textual source code of existing software systems as input artifacts. A special group of software systems are legacy (information) systems. A legacy information system ‘can be defined as any information system that significantly resists modification and evolution’ [BLW 99a]. Degraded software systems, among other things, fit this description. ‘Existing (software) system’ is used in this thesis, because our solution is not restricted solely to legacy systems.

‘Reverse engineering to an MDA target context’ is denoted by the term ‘harvesting’. It is the process of extracting valuable information from source artifacts to produce a representation of the source at a higher level of abstraction. Usually a UML model is used, ready for MDA-based forward engineering. In the ArcStyler documentation MDA Harvesting [Arc 05a] and MDA Enabling [Arc 05b] are used in a similar way as harvesting.

‘Initial model’ is used for denoting an extracted model that has abstracted from source system specifics; for example a UML model that can be used for further analysis.

A ‘[UML] modeling style’ defines what subset of UML models is valid and defines specific semantic meaning to complying models.

‘Restructuring’ or ‘refactoring’ aims to improve internal system quality without changing its external behavior. It reduces complexity and increases maintainability: it revitalizes the degraded system.

‘Reengineering’ is the process of reverse engineering, optional restructuring and forward engineering. Reengineering is a form of maintenance, because ‘intentions of software maintenance are to perfect the system, to adapt the system to changes in the runtime environment and to correct the system’ [CHK 01].

Figure 1-4 shows the reengineering concepts in relation with each other. It shows the process of extracting models, improving models and generating target code.
1.3.2 Models

A meta-model defines the structure of a model, because it is a model of a model. Besides that, a model can be verified with a meta-model, which determines whether a model is an instance of a meta-model. Models can be serialized to text, which is another representation of the same information (called a serialized representation). Parsing will recreate a model from a serialized representation.

A repository is a data storage implementation complying with a meta-model; it stores models of the respective meta-model. We define ‘population [of a repository]’ as the process of filling a repository with model elements, when a model is being created.

A meta-model can be mapped to another meta-model, by relating corresponding model elements. The mapping defines transformations between models of the two meta-models. Figure 1-5 shows the model concepts in relation with each other.

1.3.3 Parsing

A parser converts textual input artifacts to an abstract syntax tree (AST). An AST is a model of which the textual input is a serialized representation; its meta-model is defined as a grammar. A grammar consists of tokens and productions, which represent the terminals and non-terminals of an AST. Figure 1-6 shows the concepts regarding parsing in relation with
each other and compares them with the model concepts. Figure 1-7 shows a simple example of an AST with productions and tokens.

![terminology_parsing](image1)

![example_AST](image2)

### 1.4 Conceptual Approach

The main problem covered in this thesis regards automatic generation of UML models from textual input. The conceptual approach is illustrated in this section by increasingly elaborating the harvesting process from text to UML models.

Figure 1-8 shows the problem context: how can a UML model be generated from existing textual input? Textual source code is a serialized representation of a certain source specific model, which should be transformed into a target specific UML model. The UML meta-model defines the structure of a UML model. This overview suggests the need to parse the textual source code into a model with a source specific meta-model.

![from_text_to.uml](image3)

Figure 1-9 shows the first step in the harvesting process: creating a model of the textual source code, which can be harvested to a UML model. The source specific model is called an AST, the structure of which is defined by a grammar. A parser can be generated from a grammar, which automatically builds an AST from textual input.

The grammar is specified in EBNF\(^2\), whereas UML is specified in MOF\(^3\). As will be shown in section 2.3, it is convenient if both meta-models are defined in the same meta-meta-model: it

---

\(^2\) Extended Backus Nauer Form – a formalism to specify textual structures in a grammar  
\(^3\) Meta Object Facility – a meta-model definition standard by the Object Management Group
enables usage of a generic M2M\textsuperscript{4}-transformation engine to execute M2M-transformations. It implies the need for a meta-model similar to the grammar, defined in MOF.

Figure 1-9 From text to UML model – using a parser

Figure 1-10 adds the second step to the harvesting process: the creation of a meta-model in MOF, which is similar to the grammar. It defines a source specific repository which is populated with contents of an AST. It requires a mapping between the EBNF and MOF meta-meta-model, which is covered in section 6.4.1. This step can be done completely automatic.

The structure of both source and target is now defined using the same meta-meta-model (which is MOF). This enables usage of a generic M2M-transformation engine to specify the mapping between a specific AST structure and UML.

A transformation from a specific AST to UML could now be specified directly, but it would make transformation reuse difficult: each specific AST structure would require a different M2M-transformation, because a transformation maps a source structure onto a target structure. This implies a benefit for having a generic intermediate model which comes in between the source specific and target specific models.

Figure 1-10 From text to UML model – using a generated MOF meta-model

Figure 1-11 shows the final step in the harvesting process: using a generic intermediate model between the source specific and target specific models. The generic intermediate model (which we call GenericAST) provides a single meta-model for representing structure and behavior, independent of source and target. The GenericAST is covered in chapter 7; it enables reuse of transformations and analyses.

\textsuperscript{4} M2M-transformation: model to model transformation
Each AST structure should be mapped to the GenericAST meta-model. Because the GenericAST meta-model and UML meta-model do not change for different harvesting projects, a mapping from GenericAST to UML needs to be defined only once.

Figure 1-11 From text to UML model – using a generic intermediate model

The presented approach supports generation of UML models from textual source code in several transformations. Usage of harvested UML models is supported by existing UML and MDA environments, for example for documentation or MDA-based forward engineering. Therefore the main problem covered in this project is how to harvest textual input to UML models.

The source specific models are processed completely automatic, once a grammar is available. We will show that the gaps between source specific and generic models, and between generic and target specific models require more complex mappings to be bridged. These mappings are currently developed by hand.

Figure 1-12 shows the artifacts that control the transformation processes from textual source code to UML models. The grammar and first M2M-transformation are input specific, whereas the second M2M-transformation is generic. Although many grammars are available, for specific structures one may not be existent. If a grammar is not available it must be developed by hand. Both M2M-transformations must be developed manually, because it requires reasoning about similar concepts in source and target meta-models.

Figure 1-12 Artifacts controlling the transformation processes
2  TECHNOLOGICAL CONTEXT

This chapter describes the techniques that have been used during the project. It gives a brief introduction to MDA, ADM, MOF and parsing technologies, which form the basis of the solution implementation that has been developed. This knowledge is required for understanding the next chapters of the thesis.

We do not provide a complete course on the discussed technologies. We introduce them briefly and leave further investigation to the interested reader, who can use the references as a starting point for further exploration.

2.1  MDA – Model Driven Architecture

Model Driven Architecture (MDA) is an initiative to prevent the degradation of software systems. It is the current flagship of the Object Management Group (OMG). According to OMG’s MDA Executive Overview:

“MDA provides an open, vendor-neutral approach to the challenge of interoperability, building upon and leveraging the value of OMG’s established modeling standards: Unified Modeling Language (UML); Meta-Object Facility (MOF); and Common Warehouse Metamodel (CWM). Platform-independent Application descriptions built using these modeling standards can be realized using any major open or proprietary platform, including CORBA, Java, .NET, XMI/XML, and Web-based platforms.” [OMG 01]

We use an MDA target context for harvesting because it provides a standardized environment for the harvesting results. It enables reengineering a system using state of the art technology for model and code generation. Above all, it enables us to prevent degradation of a (reengineered) system, making reengineering economically more attractive.

In the MDA Guide, we read:

“The Model-Driven Architecture starts with the well-known and long established idea of separating the specification of the operation of a system from the details of the way that system uses the capabilities of its platform.

MDA provides an approach for, and enables tools to be provided for:

- specifying a system independently of the platform that supports it,
- specifying platforms,
- choosing a particular platform for the system, and
- transforming the system specification into one for a particular platform.

The three primary goals of MDA are portability, interoperability and reusability through architectural separation of concerns.” [OMG 03]

MDA separates the system specification in models to different levels of abstraction. A model with lower level of abstraction (thus more specific) can be (partly) generated from the model of one abstraction layer higher. MDA defines the following abstraction levels (from high to low level of abstraction):

- **CIM or Computation Independent Model**
  The business model or domain model of the system. Does not contain any architectural or implementation details but focuses solely on the functional requirements of the system.
• **PIM or Platform Independent Model**
  High-level architectural view of the system. Has no implementation specific details, but may incorporate architectural styles.

• **PSM or Platform Specific Model**
  Architectural, design or code view of the system. Has implementation specific details, for example J2EE settings or implementation logic.

Automatic transformations between models on different levels of abstraction provide a consistent view of the system at every level of abstraction. Additional details on lower abstraction levels may need to be specified manually.

MDA does not put restrictions on which models are used for the CIM, PIM or PSM. One possibility is UML, a modeling standard maintained by the OMG. To increase quality, information value and automation capabilities of UML models in ArcStyler, a modeling style is used to restrict which models are valid [Arc 05b]. A modeling style defines verifiable model constraints and semantics for valid models, which is important for code generation, for example.

The different abstraction levels allow us to specify a software system on the right level of abstraction. For example, when the implementation platform changes (specified in a platform specific model) the business process does not. A new platform specific model can be automatically generated from the same platform independent model. Reusing the PIM and the CIM is illustrated in figure 2-1.

![Figure 2-1 MDA supports multiple target platforms](image)

Using MDA makes reengineering economically more attractive because it enables prevention of degradation of a reengineered system. The models are the main software assets, which form a significant part of the documentation. Because MDA forces the models to be up-to-date with the implementation, a significant part of the documentation is also up-to-date. It also provides a standardized environment for our harvesting results, based on open standards.

### 2.1.1 ArcStyler

ArcStyler is a state of the art MDA-tool from Interactive Objects GmbH. It is an extendable platform for MDA-based software architecting and engineering. It integrates, among other things, a UML modeling environment with a collection of model transformations, which can generate models or textual output based on UML models. ArcStyler uses a cartridge as a
collection of related M2M and M2T transformations. They use the Carat® architecture, which stands for Cartridge Architecture [Arc 05b]. This architecture applies MDA to the development of M2M and M2T transformations: ArcStyler is built with ArcStyler.

Figure 2-2 shows the ArcStyler GUI with on the left side the current UML model tree and on the right side a UML class diagram, which is a view on (part of) the model. Given the right cartridge, code can be generated for the loaded UML model. What code can be generated depends on the used cartridge, for example plain Java, J2EE for IBM WebSphere application server, or C#. For more information we refer to the ArcStyler website⁶ and to the documentation that comes with ArcStyler.

2.1.2 M2M-Transformations

Currently MDA specifies models on different levels of abstraction (CIM, PIM and PSM) but does not describe yet how to transform one model to another [GLR 02]. These transformations are called M2M-transformations. This kind of transformations is to be standardized by the OMG MOF2.0 QVT-specification⁷. ArcStyler (version 5) contains a development release of a M2M-transformation engine, called AIM, which we have used extensively in the project. It is an initial implementation similar to QVT.

2.1.2.1 AIM – Atomistic Information Mapping

ArcStyler (version 5) contains a development release of a M2M-transformation engine, called AIM. It allows us to conveniently define a M2M-transformation by specifying transformation rules. It is similar to pattern matching: AIM generates a target model with a certain structure and contents based on a set of rules and source models [Arc 05c].

Using AIM is the preferred way of writing M2M-transformations to developing a custom Java implementation. The graphical view provides us with a clear overview of the transformation

---

⁵ M2T transformation: model to text transformation

⁶ http://www.arcstyler.com

⁷ Query/View/Transformation – part of the MOF2.0 specification, which is currently being finalized
and the transformation engine takes care of a lot of bookkeeping, for example in what order to create the target model elements. AIM uses the Jython\(^8\) scripting language to define each transformation rule. It provides a flexible programming environment for specifying details for each transformation rule.

We measure the size of an actual transformation with the number of links. A link is the creation of a target model element, in relation with the source model elements. It is caused by the combination of a transformation rule and a source model element. A transformation rule may cause many links, but a link always has been caused by a single transformation rule.

### 2.2 ADM – Architecture Driven Modernization

In 2003 the OMG started the Architecture Driven Modernization Task Force\(^9\). It aims at extending MDA practices and standards towards existing systems. MDA uses a top-down approach for developing new systems: from models to code. ADM works bottom-up by extracting architectural models from existing systems. The extracted models can be used for MDA-based forward engineering. It is similar to reverse engineering to an MDA target context.

It is out of scope for this project to comply with the ADM specifications. The first of seven specifications was due June 2005, the seventh specification is due after 2009. We refer to section 16.1.1 about related work at page 113 for more information on ADM.

### 2.3 Meta-modeling

A meta-model defines the structure of a model: it is a model of a model. For example, the UML meta-model defines the structure of UML models. It defines that a class can have a name, attributes and associations to other classes. With an explicitly defined meta-model it can be verified whether a model complies with a meta-model. Besides that, an explicit meta-model makes models understandable by and interchangeable between people and tools; UML is a famous example of this in the software engineering practice.

When we specify an M2M-transformation, we essentially define a mapping between two meta-models\(^10\): we specify which elements in the source meta-model map to which elements in the target meta-model. For example, we could specify that a database table maps to a UML class. When applied to a complete source model (e.g., a database), each table in the source model results in a class in the target model. Figure 2-3 shows models in relation with a meta-model and M2M-transformations.

![Figure 2-3 Models and meta-models](image)

When defining a mapping between two meta-models, it is practical if the meta-models are specified in a similar or even identical meta-meta-model. It allows a M2M-transformation to

---

\(^8\) [http://www.jython.org](http://www.jython.org)

\(^9\) [http://adm.omg.org](http://adm.omg.org)

\(^10\) Although we can influence the M2M-transformation process with model information, as opposed to a pure meta-model mapping in which only meta-model information is used.
use any meta-model for both source and target models. The meta-meta-model we use is MOF, as specified by the OMG. It is an open standard with broad support in industry.

![Figure 2-4 Meta-models and meta-meta-models](image)

**2.3.1 MOF – Meta Object Facility**

MOF specifies a 4-layer meta-data architecture\(^\text{11}\) [OMG 02]. It is a standardized formalism to define meta-models, models that define the structure of other models. MOF itself is specified as a MOF meta-model. A repository is an implementation of a meta-model, which provides storage for models of the respective meta-model.

Because we use this open standard for describing and storing our models, they are available to any MOF-capable tool, for example ArcStyler. MOF provides so-called reflective functionality to discover a meta-model at runtime. This is an important advantage over a proprietary format, which must be hard-coded into the programs that use it.

Table 2-1 shows the 4 meta-data layers including an example for how UML and a database fit into this architecture.

**Table 2-1 MOF 4-layer meta-data architecture**

<table>
<thead>
<tr>
<th>Layer</th>
<th>Purpose</th>
<th>UML example</th>
<th>Database example</th>
</tr>
</thead>
<tbody>
<tr>
<td>M3</td>
<td>Meta-meta-model</td>
<td>MOF meta-model</td>
<td>MOF meta-model</td>
</tr>
<tr>
<td>M2</td>
<td>Meta-models</td>
<td>UML meta-model</td>
<td>Relational database model</td>
</tr>
<tr>
<td>M1</td>
<td>Models</td>
<td>UML models (e.g. class diagram)</td>
<td>Database schema</td>
</tr>
<tr>
<td>M0</td>
<td>Data, instances</td>
<td>Program, objects</td>
<td>Database records</td>
</tr>
</tbody>
</table>

Each layer contains models with a structure described in the layer above. Or, each layer defines how models directly below it should look like. An M3 meta-meta-model defines how meta-models at M2 are described. An M2 meta-model defines how M1 models are described. An M1 model defines the structure of M0 data.

For the database example, the concepts ‘table’, ‘column’ and ‘relation’ are at M2. They describe the structure of a relational database. On M1 the structure of a particular database is defined, for example a table ‘Car’ with a relation to the table ‘Person’. Notice that these concepts are instances of the M2 model (table, relation). On M0 there are database tables with records, for example a car ‘Mercedes SLK’ with a relation to person ‘Thijs Reus’.

\(^\text{11}\) We refer to the MOF 1.4 specification, section 2.2.2, for why the word ‘architecture’ is used
2.3.2 JMI – Java™ Meta-data Interface

JMI defines a Java mapping for MOF [JSR 02]: it defines how MOF constructs are represented in Java. Because the repositories built with ArcStyler are JMI compliant, they are accessible from any Java program, for example ArcStyler. Just as MOF, JMI has reflective functionality to discover a meta-model at runtime. This is a great advantage over hard-coding it in to the programs that use it, which would be necessary with a proprietary data storage.

2.4 Reverse Engineering Process

Reverse engineering tools generally have a common architecture, which is shown in figure 2-5 [Moo 02].

![Figure 2-5 General architecture for reverse engineering tools [Moo 02]](image)

In the reverse engineering process, relevant information is extracted from the source artifacts and stored into a repository. Relevance is determined by the question that is to be answered or the problem that is to be solved. Then, several abstraction phases may derive extra information based on present information, or remove present information that is considered irrelevant. Finally the contents of the repository are presented to the user, for example in diagrams.

2.4.1 Methods of Extraction

Reverse engineering extracts relevant information from textual source code into a structured model. This implies a need for a suitable way of textual extraction, which recognizes relevant information and separates it from irrelevant information. Here several textual extraction methods are described, which form the basis of our harvesting solution. We do not claim to have an exhaustive list of extraction methods.

2.4.1.1 Scanning

Scanning is the simplest form of extraction. The input artifacts are scanned for the occurrence of a certain text fragment. When it is found the scanner derives the required information. All non-conforming input can easily be ignored. Scanners are written by hand, making them flexible yet only suitable for simple extraction complexity.

2.4.1.2 Lexical Analysis

Lexical analysis converts a stream of characters from the input artifacts into tokens. A token is a meaningful series of characters, for example a keyword or an identifier. A token has a class and a representation, e.g. class ‘number’ and representation ‘107’.

A lexical analyzer can be built by hand, but good generators exist. A generator can build an efficient lexical analyzer from a token specification [GBJ 00].

2.4.1.3 Syntactic Analysis

Syntactical analysis is done by a parser, which generates an abstract syntax tree (AST) from a stream of tokens. The parser knows what order of tokens is valid, which is usually specified in a grammar.
A parser can be built by hand, but can also be generated based on a grammar. The grammar defines the syntactic structure of the intended input and thus which AST’s are valid [GBJ 00]. There exist many parser classes, each with a certain strength. The strength of a parser class is indicated by the size of set of languages it can parse. Examples of parser classes are LL(k), LR(k), LALR(k) and GLR. For more information on grammars and parser classes we refer to [GJ 90].

2.4.1.3.1 Full Grammars
In general Extended Backus-Naur Form (EBNF) is used to describe full grammars. It specifies the basic constructs to describe all possible AST structures. A full grammar specifies the complete structure of the input. If the parser encounters a construct that is not described by the grammar, it results in an error. This is preferable behavior in most situations because it indicates an error in the input. Full grammars result in complete, accurate parsers, which are not flexible or robust [Moo 02].

2.4.1.3.2 Island Grammars
Island grammars, as described in [Moo 02], provide more flexible grammar development. An island grammar allows us to specify the constructs of interest (the islands) in detail, while specifying the rest (the water) with few rules. It basically tells the parser to ignore anything but the constructs of interest. Island grammars combine flexibility and robustness with the accuracy and completeness of full grammars [Moo 02].

Building practical island parsers requires a strong parser class, the class of Generalized LR (GLR). In theory it is possible to use any parser class, but in practice only GLR works fine. GLR allows ambiguity in the grammar and infinite look-ahead, which simplify specifying island grammars.
3 INDUSTRIAL PARTNERS

The three companies that took part in the project are introduced briefly below. The information has been taken from their websites.

3.1 Interactive Objects GmbH
Interactive Objects GmbH, founded in 1990, has built a solid reputation as front-runner in the field of IT architecture by delivering award-winning architectural platforms and services to global industry leaders. Interactive Objects builds ArcStyler, a state-of-the-art MDA-tool, and implements it successfully at many international customers. Interactive Objects actively participates in OMG standardization processes, sharing knowledge and experience in real-world MDA.

In 2000 Interactive Objects introduced the ArcStyler architectural platform. ArcStyler is an Architectural IDE which supports MDA as defined by the OMG. It supports UML-driven development including complete four-tier J2EE/EJB systems, .NET systems, custom infrastructure systems and Web Services for leading application servers. Not only does ArcStyler boost a rapidly growing list of customers, it is also the winner of numerous national and international awards.

3.2 De Amersfoortse Verzekeringen
De Amersfoortse Verzekeringen is a Dutch insurance company of medium size. The Amersfoortse Verzekeringen is specialized in insuring income. It offers a complete set of insurances for employers, employees and private people. The three main areas are Disability, Healthcare and Pension. De Amersfoortse is part of Fortis N.V., a large bank-insurance company, active worldwide in many countries. Fortis ranks in the 20 largest European financial institutions.

3.3 VDA Informatiebeheersing
VDA delivers unique software solutions for media companies and corporate service companies. Located in Hilversum, their core activity is developing and maintaining process-focused administrative software. VDA has built the Haebop system that was used in the case study. They have supported us with knowledge and experience to enable successful harvesting.

12 www.interactive-objects.com
13 www.amersfoortse.nl
14 www.vda.nl
4 RESEARCH PROBLEM

This chapter introduces the research problem. It defines the main research questions and goals of the project.

4.1 Problem Definition

Software systems in general and legacy systems specifically are expensive to maintain [HW 02]. It is estimated that between 40% and 75% of the total cost of a software system is spent on maintenance [Gui 83]. Newer software engineering methods, such as MDA, promise lower maintenance costs, among other things. Newer platforms, such as J2EE, promise improved cost effectiveness, scalability and security. Green-field engineering (i.e. development of new applications) can easily benefit from these promises, but what about existing systems? To benefit from these promises for existing systems, they must be migrated to the new environment, preferably using new software engineering methods.

After migration the new systems should (at least) have improved maintainability [JL 91]. This results in cost and risk reduction during the operational lifetime of the system. Cost reduction, together with improved quality and flexibility, is an important business incentive for initiating migration projects [LDP 99]. It enables companies to respond quicker, better and more cost-effective to market changes [Par 94].

Much knowledge is stored in the source code of a system. It contains the actual implementation of business rules and workflows; it may also contain information about the rationale, in the form of comments and documentation throughout the source code. In a reengineering project, one wants to recycle as much of this knowledge as possible. Not only does it save manual work, it also helps to comprehend the system. It may even be the single source of information because system documentation is often out of date and unreliable [Par 94], if it exists at all.

Reusing existing information should at least achieve a cost reduction at project level. This limits applicability of reengineering to systems with a certain intrinsic quality. Obscured information cannot efficiently be extracted. If extracting usable models from an existing system takes more effort than creating them manually, green-field engineering should be done instead of reengineering.

The research questions aim to find an optimal way of extracting and reusing existing information in a reengineering project. ‘Optimal’ should compromise technical, economical and timing aspects: an optimal way is a solution that can be developed, tested and evaluated during this project. It should reuse as much existing information as needed and benefit the reengineering project as much as possible.

4.2 Research Questions

The main research questions that we address are the following:

1. How can we efficiently and effectively extract information from a software system to an MDA target context?

2. What is an optimal way of reusing information on a software system in an MDA-based reengineering project?

Closely related to the research questions are the goals of the project. These are:

1. Develop a generic harvesting framework at Interactive Objects GmbH
2. Reengineer a software system at De Amersfoortse Verzekeringen using the generic harvesting framework and MDA

The goals are elaborated in sections 4.2.1 and 4.2.2.

### 4.2.1 Develop Generic Harvesting Framework

Interactive Objects has created a state of the art MDA-tool, ArcStyler. With this tool customers develop new systems, following the MDA philosophy. During development, several models are designed and code is generated from these models. Models are the main software assets: code is a derivate of the models. This allows the new systems to remain maintainable and understandable by preventing the software degradation phenomenon.

However, many customers do not start from scratch: they have systems in operation, which they do not want to discard in order to benefit from MDA. They want to reuse knowledge and investments in their systems, for example by extracting models from these systems. This is where harvesting may prove valuable. Figure 4-1 shows an illustration of the gap between existing systems and MDA-based forward engineering.

![Figure 4-1 Gap between existing systems and MDA-based forward engineering](image)

The first goal of the project is to bridge the gap by providing means for harvesting UML models from existing source code. These UML models provide the basis for MDA-based forward engineering. To support this process, we have developed a generic harvesting framework.

### 4.2.2 Reengineer Software System

De Amersfoortse Verzekeringen has many software systems. These systems are affected by the software aging phenomenon: the systems have business and economic value but do not comply with the current architectural standards. Therefore De Amersfoortse Verzekeringen is developing a standard migration plan to migrate existing systems to new architectural standards, possibly using new software engineering methods.

A possible migration solution is reengineering based on the generic harvesting framework and MDA. The second goal of this project is to reengineer an existing system using these concepts. We will reengineer parts of the Haebop system, which is an offering application for collective insurances. It is a production system developed in PL/SQL by VDA.

This goal supports two sub-goals. First, it evaluates the generic harvesting framework developed at Interactive Objects. Second, it assesses the feasibility to reengineer existing systems at De Amersfoortse Verzekeringen using an MDA target context. It shows the (absence of) benefits of reengineering compared to redevelopment from scratch.
5 \hspace{1em} \textbf{INTRODUCTION}

This part of the thesis covers the work done at Interactive Objects GmbH in Freiburg, Germany. We start with a description of the research problem. Then the two main components of the solution are discussed: the harvesting workbench and the GenericAST. Finally, future work on this part is discussed. The technological context as described in chapter 2 at page 10 is required knowledge to understand the contents of this part of the thesis.

The basis for the generic harvesting framework is the harvesting workbench. It provides a flexible toolset that can be used for developing and running harvesters. A harvester reads existing source code and generates an initial model of it, for example in UML.

The GenericAST provides a single meta-model in which a systems structure and behavior can be described. The representation is abstract yet semantically rich; it is independent of the source and target environment. The GenericAST forms an intermediate layer between source oriented and target oriented models. It enables reuse of transformations and analyses in multiple harvesting projects.

5.1 \hspace{1em} \textbf{Research Problem}

MDA offers a new way of developing software systems. It focuses on models rather than source code and allows prevention of degradation of software. UML models of the system are not only part of the documentation, but also form the input for generating parts of the target source code.

Currently, MDA is not able to handle existing systems: it does not define how to generate models from existing source code. There is a gap between the existing systems and the MDA target context, which is illustrated in figure 5-1. We aim to bridge this gap by providing a generic harvesting framework which can be used in many harvesting projects.

![Figure 5-1 Gap between existing systems and MDA-based forward engineering](image)

We consider textual source code of which the structure can be described in a grammar; we will not focus on other information sources. Grammars for many programming languages can be found on the internet, which can be used in harvesting projects. If a grammar is not available, it must be developed based on the existing source code and programming language documentation.

5.2 \hspace{1em} \textbf{Requirements}

The generic harvesting framework must provide means to bridge the gap. The following requirements and goals have been used while working on the solution:
• **Generate initial UML models**
  The aim is to create an initial UML representation of the existing system, based on its source code. UML provides a standardized target model and allows using the full potential of MDA for reengineering the harvested system.

  o **Describe source code structure**
    Grammars for many programming languages can be found on the internet. If a grammar is not available, it needs to be developed to describe the structure of the source code. It allows harvesting source code with a similar structure with minimal extra effort.

  o **Incremental grammar development**
    It should be possible to incrementally develop the grammar in a flexible way, because a full grammar might not always be available. A light-weight solution is preferred, which only describes the structure of constructs that appear in the existing sources. Focus should be on the relevant parts of the source code while ignoring irrelevant parts, or at least dealing with them quickly.

  o **Use MOF-compliant meta-model**
    The source code structure must be transformed to a MOF-compliant meta-model. It allows using models of the source code in any MOF-capable environment.

  o **Map source and target constructs**
    It should be possible to define a mapping between source constructs and target constructs. It allows applying the harvester to any source code that complies with the structure defined in the grammar.

  o **Focus on structural elements**
    Initial focus is on the structural elements of the source system because that can be described in UML\textsuperscript{15} models. A separate meta-model is needed to represent behavioral elements.

  o **Access to original source code**
    Access to the original source code in every generated model (when required) is important. It helps us in understanding the harvested results by providing a reference to the original source code. It can be as documentation or as a reference.

• **Perform analysis**
  It should be possible to perform analysis on the existing system, e.g. identify dependencies between concepts. Analyses may provide additional information that could not be derived by a regular M2M-transformation.

  o **Suitable target model**
    Analyses should have a suitable target model for the analysis results.

  o **Information available in ArcStyler and Java**
    The source code representation must be available in both ArcStyler and Java, such that we can define our analyses in the AIM M2M-transformation engine and in a custom Java program.

\textsuperscript{15} We consider UML 1.4 which is supported by ArcStyler
5.3 Non-functional Requirements

A number of non-functional requirements have been used while working on the solution as a set of guidelines. The solution may not comply with all, but it should not make it impossible to realize them later on.

1. **Genericity**
   The solution should be as generic as possible, such that it is reusable for systems with different structures. It needs to be easily adaptable to a wide range of source structures.

2. **Ease of use**
   No extensive trainings should be required for using the solution. Complexity should be hidden as much as possible.

3. **Correctness**
   Our solution may not show unexpected behavior. It may not result in false positives (each harvested model element must be traceable in the existing source code) and must be complete (no relevant information may be lost in the harvesting process).

4. **Understandable**
   Every component must be clearly defined. Our solution should not contain 'magic'.

5. **Abstract from source system specific constructs**
   Improves reusability of solution components. The sooner we abstract from source specific constructs, the more solution components can be reused in different projects.

6. **Portable**
   The solution should not rely on certain operating systems or environments, except on ArcStyler. It must be as portable as ArcStyler to maximize applicability.

5.4 Testing the Solution

The most important goal is that the solution should at least work for this project. This has been emphasized many times during the project. The second part of the project consists of a case-study on PL/SQL, for which we need a working solution. Genericity is a less important goal although it should not be avoided.

An incremental development approach has been chosen in which small feature additions have been developed and tested. Each increment has been tested with PL/SQL and generalizations have been applied wherever possible. The used test-set is a subset of PL/SQL source code from Haebop. It consisted of 31,000 lines of code and covered the following constructs:

- **Table definitions**
  Defines the data storage structure. They contain database tables with columns that have a type, and possibly documentation and constraints.

- **Stored Procedures**
  Contains behavior that can be invoked from a trigger or another stored procedure.

- **Packages**
  Contains a collection of stored procedures.

- **Triggers**
  Contains checks and behavior that is applied upon certain events on the data storage.

For testing individual components other source code has also been used, for example additional PL/SQL test source code from the Oracle website and artificial source code for performance and feature testing.

We performed all tests on a computer with an AMD Athlon processor at 1.68 GHz with 1 GB of RAM.
6 Harvesting Workbench

The harvesting workbench is used for solving the first part of the research problem by extracting valuable information from existing source code. From this information initial models of the existing source code are generated. This chapter describes the architecture of the workbench, the anticipated process and several implementation details.

6.1 Approach

We bridge the gap between existing source code and initial models using a workbench architecture. A ‘workbench’ is an open collection of tools which are easy to integrate and replace [BCK 03], without unnecessarily affecting each other. It provides a flexible toolset in which components can be replaced when required.

The solution includes all steps necessary to generate initial models from source code of which the structure can be described by a grammar. Certain source code structures may not easily be described in a grammar, for example column-based structures. For these source structures other methods of extraction should be considered for generating initial models.

Figure 6-1 shows a high-level overview of the harvesting workbench.

![Harvesting Workbench Overview](image)

The workbench consists of two components: a harvester generator and a repository generator. Both take as input the same source specification, which is a grammar that describes the structure of the existing sources. The generators generate a specific harvesting infrastructure which is tailored for the source code that is described by the grammar.

Generators are used to hide as much complexity as possible. Using generators a source structure can be specified concisely in a grammar, and all required logic to build a harvester can be generated. It allows us to quickly develop consistent harvesters, because generators make no mistakes, as opposed to manual development.

The repository generator generates a tailored data storage facility for storing the results of the harvester. The repository is JMI compliant, allowing any MOF- or JMI-capable tool to access the repository. Initial models are generated from the contents of the repository. It requires a

---

16 In a column-based language, the meaning of a text-fragment may vary by the position on the line. It appeared to be cumbersome to define a grammar for this structure during testing with RPG-3, an old IBM language which is column-based. We refer to [http://www.itjungle.com/tfh/tfh062705-story01.html](http://www.itjungle.com/tfh/tfh062705-story01.html) for more information on RPG.
specific M2M-transformation with the repository contents as input, and the initial models as output.

The harvester generator generates a tailored harvester. A harvester can read the existing sources and populate the generated repository. A harvester is the first in a series of steps that filter and transform the existing sources into the target models.

We tested the complete workbench process with PL/SQL source code from Haebop. Several artificial grammars and test files have been used to test harvesters with large input.

6.2 Architecture

The architecture of the harvesting workbench imposes a process for using it, which is shown in figure 6-2. It is an elaboration of the high-level overview from figure 6-1 by specifying the harvester generator and repository generator into more detailed components.

When using the workbench, first the structure of the existing sources must be described in a grammar. The grammar is fed to the parser generator to generate a harvester. Then the grammar is fed to the grammar wrapper, which is the entry point of the repository generator. The grammar wrapper presents the grammar in a convenient way to the M2M-transformation by transforming the textual grammar to a convenient tree structure. A generic M2M-transformation generates a repository model in UML, after which a repository implementation is generated using ArcStylers Carat.MOF functionality.

When the harvester and repository have been generated and built, the existing sources are fed to the harvester, which parses the source code and populates the repository. The harvester can populate the repository because it knows the meta-model, which has been generated from the same grammar as the harvester itself.

The contents of the repository are used to generate initial models, for example using a M2M-transformation in AIM.

All manual work in the workbench is concentrated in two places: the grammar and the M2M-transformation to initial models. All other components in the workbench are either generic or generated. This makes the workbench easy to use and harvesters easy to debug.

6.3 Harvester Generator

The harvester generator is based on a modified parser generator. This section describes which parser generator has been chosen, what modifications have been made and how the generated
The harvester populates the repository. Figure 6-3 shows a more detailed view of the harvester generator.

![Harvester generator overview](image)

Figure 6-3 Harvester generator overview

The harvester generator generates harvester source code, which consists of standard parser source code and additional harvester functionality. The harvester generator also provides general functionality which is required to run harvesters.

### 6.3.1 Parser Generator

The parser generator in the workbench allows specifying the structure of given existing sources and generating a parser that transforms the existing sources into a convenient tree structure. This supports quick harvester development. We refer to section 6.5 at page 46 for more information on actual code generation based on a grammar.

The Grammatica parser generator has been selected after comparing several parser generators. We refer to appendix B at page 136 for the comparison table. A parser generator was needed that is easy in use, freely available and supports incremental grammar development. Above all it should be portable. This restricted our choice to parser generators written in Java and generating Java source code.

The following arguments support the choice of Grammatica:

- Takes EBNF grammars as input
- LL(\(k\)) with flexible \(k\), allowing for easy grammar development
- Has been written completely in Java
- Generates parsers in pretty-printed Java source code
- Open source which enables modifications
- Supports custom code generation, e.g. to populate generated MOF-repository
- Transforms grammar to match a simple tree representation

Possible drawbacks of the Grammatica parser generator at this moment are:

- Not many grammars available for common languages
- Not widely used and tested
- Not LALR(1) or GLR, which are by definition stronger parser classes [GJ 90]
The drawbacks were considered less important than the advantages for this project. After initial testing with partial PL/SQL grammars on production PL/SQL source code, Grammatica has been considered adequate for developing PL/SQL grammars.

Island grammar support was preferred for the workbench, but absence of a suitable GLR parser generator prevented this attempt. To enable island grammars in the future, a workbench architecture has been chosen to allow for interchanging the parser generator when required, while minimizing impact on other workbench components.

A parser generator for which a PL/SQL grammar already existed was preferred, because it would save time developing it manually. Only one PL/SQL grammar was found, for the Antlr parser generator. During initial testing the grammar appeared to be invalid because it could not parse PL/SQL source code from the Oracle website. No attempt has been made to debug the grammar; instead it was decided to manually develop partial PL/SQL grammars.

The parser generator has been modified to improve support for the harvesting process. The modified Grammatica is backward compatible with the original version, such that existing Grammatica grammars can be used in the workbench. Below is an overview of the modifications, which can all be activated when needed. The modifications are briefly elaborated in sections 6.3.1.1 to 6.3.1.5.

- **Treatment of artificial productions was altered**
  To support correct MOF-repository population.

- **Production element naming was introduced**
  To allow for a more semantic structure of the MOF-repository by using meaningful names. It provides an initial filter mechanism by populating only named AST elements into the MOF-repository.

- **MOF-repository population code generation was introduced**
  To allow 100% harvester code generation.

- **Standard harvester framework was introduced**
  To support quick harvester development.

- **Meta-information concept was introduced**
  To allow for additional information in the MOF-repository. It includes references to the original source code and inclusion of source code fragments.

- **Convenience features were introduced**
  To support the harvester development process.

### 6.3.1.1 Treatment of Artificial Productions
The treatment of artificial productions has been changed to create the mapping from an AST to a MOF-repository. Grammatica uses artificial productions to map a grammar to an internal AST structure. Figure 6-4 shows how Grammatica stores its AST internally, as a simplified version of the object structure.

A token is a terminal and therefore trivial. It contains a token-id, which represent the class of the token, and a textual representation.

A production is a non-terminal. It contains the production-id, which represents the type of production from the grammar. A production has a collection of alternatives. They are separated by the EBNF operator ‘|’ in the grammar. Each alternative has an ordered list of

---

17 http://www.antlr.org/grammar/list
18 http://www.antlr.org
19 http://www.oracle.com/technology/sample_code/tech/pl_sql/htdocs/tabdemo.txt
elements with type ‘Node’, which can be tokens and/or productions. Each element has a multiplicity equivalent to the optional EBNF operators ‘?’ , ‘*’ and ‘+’ in the grammar. Artificial productions may be introduced while generating an internal representation of the grammar. This is needed to map a valid grammar to the internal AST structure when it contains compound elements. For example, consider the following production in EBNF:

$$A \rightarrow (B \mid C)? D E$$

The production has a compound element ‘(B | C)’ followed by two single elements ‘D’ and ‘E’. We cannot assign a multiplicity to ‘(B | C)’ directly, because multiplicities can only be assigned to single productions and tokens. The grammar must be internally transformed into:

$$A \rightarrow A' D E$$
$$A' \rightarrow B \mid C$$

This grammar can be mapped to the internal AST structure. A’ is an artificial production which represents the alternatives ‘B’ and ‘C’. Figure 6-5 shows the internal representation of AST structure.

**Figure 6-4 Grammatica internal AST structure**

**Figure 6-5 Grammatica internal AST structure for example production**

Grammatica hides artificial productions, which is convenient for the parser developer, who does not need to know about the presence of artificial productions in his AST.

Artificial productions must be explicitly present for populating the generated MOF-repository, because it is similar to the internal AST structure. Therefore the Grammatica code generation has been changed to support making artificial productions explicit. It affects the process for creating the AST during parsing and the AST analyzers. Both must consider artificial productions instead of hiding them.

### 6.3.1.2 Production Element Naming

Support for naming production elements has been introduced to the original Grammatica grammar (or meta-grammar). This is a form of grammar annotation to provide support for more semantic repositories. Naming is optional to ensure backward compatibility with existing Grammatica grammars. Besides, not all elements should be named, but only the ones that are of interest. An unnamed production in EBNF looks like this:

$$A \rightarrow C \mid D C (E F)?$$

The type and position of an element give an indication to its meaning, but it is not explicitly present in the grammar. For example, what if the element ‘D’ is called a ‘prefix’ and the
optional compound element ‘(E F)’ is called a ‘postfix’? In the generated repository ‘D’ would be ‘the first item’, and ‘(E F)’ would be ‘the third item’; names which do not resemble the semantics of ‘prefix’ or ‘postfix’. This is semantic information that could not explicitly be defined in the grammar. In the modified parser generator the following production can be used to achieve such naming:

\[ A \rightarrow C \mid :prefix:D \mid C :postfix:(E F)? \]

This results in the element ‘D’ being named ‘prefix’ in the generated repository. The compound element ‘(E F)’ will be transformed to an artificial production ‘A’’. The internal representation looks like:

\[ A \rightarrow C \mid :prefix:D \mid C :postfix:A'? \]
\[ A' \rightarrow E \mid F \]

The main benefit is that each element in the generated repository can be accessed in a semantic rather than syntactic way. Instead of using ‘the \( n \)th element’ in a M2M-transformation, the element with name ‘postfix’ is used. Of course this really pays off with larger productions, because it is easy to mistakenly use ‘the 28th item’ instead of ‘the 29th item’, which is a hard to find mistake.

Elements that are not named are given artificial names by the parser generator. Artificial names are always ‘item’ post-fixed with the element index in the production alternative.

### 6.3.1.3 Code Generation for MOF Population

The Grammatica code generation has been extended to generate the repository population logic. This is generated into a separate class which can be used by the parser. Normally Grammatica generates the following functionality for a grammar:

- **Constants class**
  Contains the mapping between the names and numerical id’s of tokens and productions.

- **Tokenizer class**
  Sets up the tokenizer with tokens as defined in the grammar.

- **Parser class**
  Sets up the parser with productions as defined in the grammar.

- **Analyzer class**
  Base class containing a tree walker for the AST. A custom analyzer can be created by subclassing this base class and overriding relevant methods.

The code generation has been extended such that the following functionality is also generated for a grammar:

- **Populator class**
  AST-analyzer that populates a generated MOF-repository.

- **Verifier class**
  AST-analyzer that verifies an AST and a populated MOF-repository with the grammar.

- **Parser-Factory class**
  Interface for the standard harvester framework.

These modifications to Grammatica allow us to generate all necessary code for harvesting and populating a MOF-repository. Complete populating harvesters can be built with no additional manual coding, allowing for quick harvester development.
6.3.1.4 Standard Harvester Framework

A standard harvester framework has been developed to enable fast harvester development. The framework contains standard functionality that uses generated harvester components. It allows creation of harvesters with no additional manual coding, as long as the provided behavior is sufficient.

Normally generated harvester components are integrated in an application using a pipe-and-filter architecture: each component needs to be set up, given the right input and the output should go somewhere. The generated components are a collection of independent tools suited for the particular task of tokenizing, parsing, analyzing and populating.

To minimize the effort to develop a separate application for each given grammar, a framework has been created with two main components:

- **Factory interface**
  Provides a generic interface for instantiating specific harvester components. It abstracts specific implementations.

- **Default main class**
  Uses generated harvester components. It implements the standard harvester functionality. It is independent of any specific harvester because it uses the factory interface.

The framework provides the following functionality, among other:

- **Save and load MOF-repository contents**
  Standard MOF-repository functionality is used to save the repository contents to or read them from an XMI-representation. It allows saving the models and loading them later on.

- **Generate statistics for an AST and/or a populated MOF-repository**
  Generates statistics about an AST and/or a populated MOF-repository. This provides a first insight in the size of a model, by looking at the occurrence of every element.

- **Store MOF-repository meta-model**
  Standard MOF-repository functionality is used to save the repository meta-model to an XMI-representation. This meta-model can be used for writing M2M-transformations that use the MOF-repository as input.

The framework allows quickly developing harvesters with standard functionality without additional manual coding.

6.3.1.5 Meta-Information

The concept of meta-information has been introduced in the modified parser generator. With meta-information a parser can decorate the AST with runtime information. This can be useful in some cases, for example to attach a fragment of the original source code to an AST node. Currently the following information can be added to a particular AST node:

- **Time information**
  Date, time or timestamp of parsing.

- **File information**
  Name, path or extension of the file that is being parsed.

- **Parser information**
  Version, description or timestamp of the parser.

- **Block-text**
  Source code fragment.
The file information and block-text provide native support for reference to the existing sources, either by a link or a text fragment. The block-text includes all characters from the source input stream while parsing the given production, including discarded tokens which the parser itself never sees.

Decoration is controlled by switches in the grammar.

6.3.1.6 Convenience Features
Several convenience features have been added to the parser generator, which help developing harvesters. These are:

- **Case-insensitive tokens**
  Both grammar-wide and per-token setting. It allows us to indicate if a token should be considered case sensitive or not. Without this feature, all combinations of uppercase and lowercase characters would have to be specified explicitly in the token declaration in the grammar.

- **Implicit token declarations**
  Allow token declarations directly in productions instead of specifying them in a separate token declaration part in the grammar. Without this feature each token would have to be explicitly declared in the token declaration part of the grammar, giving overhead during grammar development.

- **Stronger regular expression tokens**
  Use the Java5 regular expression library instead of Grammaticas native regular expression library. This feature enables a more comprehensive way to specify regular expression tokens in the grammar.

- **Special token directives**
  Allow for island grammar-alike behavior. These directives tell the parser to skip all input until a certain token has been found. Then parsing continues, having skipped a complete fragment of the input which was considered to be irrelevant.

These features have proven to be convenient while developing harvesters and allow for quick development of compact grammars. These features will not be elaborated more extensively because it would be too detailed for this thesis; they have been well documented in the generic harvesting workbench documentation.

6.3.2 Repository Population Strategies
Several repository population strategies have been implemented to save resources by filtering information as soon as possible. A population strategy decides which elements from the grammar are populated into the MOF-repository. Information that is not populated into the repository is discarded during harvesting. It is a first filter on the input data of the harvesting process and tailors the MOF-repository meta-model: only constructs that can be populated are present in the meta-model. The following strategies have been implemented. They can be activated in the grammar.

- **Populate all**
  All elements of an AST are populated into the repository.

- **Populate named**
  All named elements of an AST are populated into the repository. Only relevant information is populated, indicated by tags in the grammar.

- **Populate productions**
  All productions of an AST are populated into the repository.

- **Populate tokens**
  All tokens of an AST are populated into the repository.
Strategies can be combined: the combination ‘named and productions’ results in all production elements and all named token elements to be populated.

The ‘named’ strategy is generally the most useful because it results in a meta-model containing only the elements we consider relevant, with meaningful names\textsuperscript{20}. As a direct consequence, the resulting models are relatively small, especially when combined with the compact meta-model style\textsuperscript{21}. It is currently the most semantic repository meta-model that can be generated from a grammar.

Population information for a given element is located in the grammar: it is used as the single point of definition. It enables to generate only the model elements in the meta-model that can be populated and improves the workbench performance. Both the meta-model and the model are smaller in size, as the test below shows for some initial measurements.

The workbench has been tested with a simplified PL/SQL grammar for table creations, which contains information about tables, comments on columns and constraints. The test source files have 1200 lines of PL/SQL in total. Table 6-1 shows collected data for the ‘full’ and ‘named’ population strategy with both meta-model styles: normal and compact. We refer to section 6.4.1 at page 40 for more information on meta-model styles. The data is not statistically significant but gives a nice indication of the differences of the population strategies.

Table 6-1 Test results for different population strategies

<table>
<thead>
<tr>
<th>Time (seconds)</th>
<th>Full population</th>
<th>Named population</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Normal</td>
<td>Compact</td>
</tr>
<tr>
<td>Initialize parser</td>
<td>0.050</td>
<td>0.040</td>
</tr>
<tr>
<td>23 productions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>39 tokens</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parse</td>
<td>0.681</td>
<td>0.752</td>
</tr>
<tr>
<td>(nr. productions)</td>
<td>(2520)</td>
<td>(2520)</td>
</tr>
<tr>
<td>(nr. tokens)</td>
<td>(5706)</td>
<td>(5706)</td>
</tr>
<tr>
<td>Analyze</td>
<td>1.482</td>
<td>0.520</td>
</tr>
<tr>
<td>Both AST and MOF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write repository to file</td>
<td>8.563</td>
<td>2.613</td>
</tr>
<tr>
<td>(file size, XMI)</td>
<td>(3972 Kb)</td>
<td>(1399 Kb)</td>
</tr>
<tr>
<td>Write meta-model to file</td>
<td>0.560</td>
<td>0.561</td>
</tr>
<tr>
<td>(file size, XMI)</td>
<td>(273 Kb)</td>
<td>(226 Kb)</td>
</tr>
</tbody>
</table>

Table 6-2 compares the ‘full’ and ‘named’ population strategy for both meta-model styles. The value of the ‘named’ population strategy is evident, even though parse time has increased. The numbers shown in table 6-2 are rough indications for the data from table 6-1 and therefore rounded; it is again not statistically significant.

\textsuperscript{20} We refer to section 6.3.1.2 at page 30 for more information about production element naming

\textsuperscript{21} We refer to section 6.4.1 at page 40 for more information about meta-model styles
<table>
<thead>
<tr>
<th>Property</th>
<th>Normal</th>
<th>Compact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse time</td>
<td>-60 %</td>
<td>-6 %</td>
</tr>
<tr>
<td>Number of productions while populating</td>
<td>30 %</td>
<td>30 %</td>
</tr>
<tr>
<td>Number of tokens while populating</td>
<td>80 %</td>
<td>80 %</td>
</tr>
<tr>
<td>Analysis time</td>
<td>80 %</td>
<td>70 %</td>
</tr>
<tr>
<td>Model size (XML)</td>
<td>70 %</td>
<td>50 %</td>
</tr>
<tr>
<td>Model save time (XML)</td>
<td>70 %</td>
<td>50 %</td>
</tr>
<tr>
<td>Meta-model size (XML)</td>
<td>40 %</td>
<td>60 %</td>
</tr>
<tr>
<td>Meta-model save time (XML)</td>
<td>40 %</td>
<td>50 %</td>
</tr>
</tbody>
</table>

### 6.3.3 Repository Population Algorithm

The repository population algorithm has been implemented as a standard Grammatica AST-analyzer, which is completely generated during harvester generation. An AST-analyzer provides a convenient input of data while populating the repository; it is a top-down, left-to-right, depth-first tree walker. When a parser builds an internal AST\(^\text{22}\), several methods are called for nodes that are created; this is standard functionality provided by Grammatica.

- When a node is encountered, the ‘enterNode’ method is called for this node.
- When a node is finished, the ‘exitNode’ method is called for this node.

A possible method invocation sequence may look like figure 6-6, for which the AST is shown in figure 6-7. Every box denotes a method invocation life-time. Dashed lines connect method invocations that belong to the same node.

The analyzer behavior implies we need to keep track of history while populating the repository: when a production is encountered, it is not yet known which alternative should be chosen. That can only be decided based on the list of child nodes, which must completely fit into the alternative.

---

\(^{22}\) Or when an already existing AST is analyzed
All direct child nodes that have been encountered should be stored in a list of encountered
nodes. This list is needed to decide which of the production alternatives should used, which
can only be done after all child nodes have been encountered. To decide which alternative to
use, a matching algorithm has been developed, which is described in section 6.3.3.1.

For tokens the following happens when populating the MOF-repository:

- **When the parser finds a token node, it invokes ‘enterNode’ for the token.**
  The corresponding model element in the MOF-repository is created and stored in the
  list of encountered nodes. A token is a terminal and thus does not contain any
  children. When using the compact meta-model style only the textual content is stored
  in the list and the token type in a separate list. The type information is needed for
  production alternative matching.

- **When ‘exitNode’ is invoked for a token, there is no action.**
  All has been done at the ‘enterNode’ method.

For productions the following happens when populating the MOF-repository:

- **When the parser knows a production node will be encountered, it invokes
  ‘enterNode’ for the production.**
  All direct child nodes for this production must be recorded; therefore the current list
  of encountered nodes is saved and an empty one is created. It is not yet known which
  production alternative will be encountered, because that will be decided based on the
  contents of the list of encountered nodes.

- **When the parser finished all child nodes, it invokes ‘exitNode’ for the
  production.**
  The list of encountered nodes contains all direct child nodes of this production. It is
  matched with one of the production alternatives where the alternative must consume
  all nodes in the list. The production alternative model element is created and
  populated with all items from the list, respecting the population strategy, the correct
  order and multiplicity. Finally the saved list of encountered nodes is restored and the
  new production alternative is stored on it for use by its future owner.

The population code for each token and production is limited and can easily be generated
from the grammar. The only non-trivial part of the algorithm is the matching algorithm, for
which no code is generated because it is general functionality and therefore contained in the
super-class of the populator.

### 6.3.3.1 Matching Algorithm

The population algorithm needs to decide which production alternative completely matches
the list of encountered nodes. A matching algorithm is needed to solve a non-trivial problem,
which is illustrated below. The decision for an alternative must be made using solely the
following information:

- a list of \( k \) nodes that have been encountered and are direct children of this production
- a set of \( m \) alternatives of the given production, with information about production
elements and multiplicity

Apart from deciding which alternative to choose, it must also be determined what children
should be populated into what production element. This may seem trivial, but it is not, as
shown in the following production ‘A’.

\[
A \rightarrow B \ B^* \ B \ C
\]

It is a legal production in LL(k). It says ‘at least two B’s followed by one C’. Matching a list
of encountered nodes of at least two B’s and one C will decide for the first (and in this case
only) alternative, because the alternative can store all children. It is found that the first and the last B should be stored in a separate element, all other B’s go in the element ‘B*’. Possible lists of encountered nodes are:

- B B C
- B B B C
- B B B B C
- Etc.

For each possible list the right alternative must be decided and the node-counts for each element must be determined. For example, in the last case we have node-counts 1 B, 2 B’s, 1 B and 1 C. This process should be done in the general case for many alternatives with many elements.

The matching algorithm, which has been implemented for this, does the following:

Given

- LEN(k) : a list of k typed encountered nodes
- PA(m) : a production with m production alternatives
- EL(n) : a production alternative sequence of n typed elements
  - Each element has a multiplicity of 1, 0..1, 0..* or 1..

Find out

- For which m all items of LEN(k) fit into the production alternative PA(m)
- For each i determine the exact number of nodes that go into EL(i)

It assumes the longest match for the element with the highest index wins. Consider the following production:

\[ A \rightarrow B? B^+ \]

It says: at least one ‘B’. Suppose the list of encountered nodes contains 10 B’s. The algorithm finds the longest match for B+ first, which is 10. It finds that it is a legal match because the first B is optional. Thus the (valid) match 0 B’s 10 B’s is returned.

This assumption has been made because of the nature of the list which was used in an earlier version of the algorithm (a stack). For the current algorithm the choice is arbitrary whether to start with the first or the last node. It has been left similar to the initial version for reasons of simplicity. We refer to appendix C at page 139 for a pseudo-code representation of the complete algorithm.

The matching algorithm is actually stronger than LL(k): it can match more sequences than that an LL(k) parser can produce. For example, the matching algorithm can match the following production:

\[ A \rightarrow B \, B? \, B^* \]

An LL(k) parser generator detects it as an ambiguous production. It makes a grammar illegal in LL(k), although it actually says ‘at least one B’. The matching algorithm will successfully match a list of n B’s as shown in table 6-3. The table ordering is top-down, just as the algorithm.

---

23 Which is illegal in LL(k); we use it to show the example
The algorithm puts as many B’s as possible in the last element ‘B*’, while still having a valid number of B’s left for the other elements. Other matches are possible because it is an ambiguous production. Table 6-4 shows an example of a valid match that is not returned by the algorithm.

Table 6-4 Another valid match example

<table>
<thead>
<tr>
<th>Element</th>
<th>Number of B’s with in total n B’s input</th>
</tr>
</thead>
<tbody>
<tr>
<td>B*</td>
<td>n – 2</td>
</tr>
<tr>
<td>B?</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
</tr>
</tbody>
</table>

Another option for this matching algorithm is to consult the parser for which production alternative to choose. It has been decided not to use this approach because it implies a stronger dependency of the population algorithm with the Grammatica parser generator, thereby making parser generator replacement harder. Besides that, it implied changing the parser generator to provide the required information. Extending the population algorithm was preferred to modifying the parser generator.

6.3.4 Test Results

The workbench has been tested continuously during development to ensure correct functionality and assess performance. A selection of interesting test results is briefly described in this section.

The capability of handling larger input has been tested by using a PL/SQL file from the Oracle website\(^2\). It has been multiplied 67 times to get >40,000 lines of PL/SQL. A grammar for this file had already been developed in the second week of the project while assessing Grammatica. Some special constructs have been used to mimic island grammar behavior to skip some parts of the source code. Parsing the source file and populating the MOF-repository took around 25 seconds. The AST contained >34,000 production and >36,000 token nodes. It shows the workbench can handle larger input without running into memory or performance problems.

The population algorithm has been tested for different grammars. One of the grammars in combination with a specific test file resulted in a single production alternative with 10,000 nodes. Parsing and populating took around 2 seconds: it shows the population algorithm can match production alternatives with many nodes.

The full harvesting process has been tested with sample PL/SQL source code from Haebop: from existing source to initial UML models. For each type of source file a grammar has been developed and a harvester has been generated. At that time the GenericAST did not yet exist, thus UML models were used as initial models. Figure 6-8 shows a sample result from harvesting table definitions from the Haebop sample source code.

---

\(^2\) http://www.oracle.com/technology/sample_code/tech/pl_sql/htdocs/tabdemo.txt
Figure 6-8 Harvested model from Haebop PL/SQL source code (in UML Profile for EJB)

It shows each table as a class with typed attributes. Each attribute contains the documentation as specified in the PL/SQL source code. The primary keys have been identified and ‘tagged’ in the UML model, following the EJB modeling style.

6.4 Repository Generator

The repository generator generates a tailored MOF-repository from a grammar. Figure 6-9 shows a more detailed view on the repository generator. This section describes the repository generator architecture and the used algorithms.

Figure 6-9 Repository generator overview

The repository generator uses a grammar as input. The grammar wrapper presents it in a convenient way to AIM, which is used to execute a generic M2M-transformation to generate a repository model. The repository implementation is generated in three steps:

- Generate repository model using UML Profile for MOF
- Generate repository implementation Java code using ArcStylers Carat.MOF functionality
• Build the repository implementation by compiling generated Java code

The result is a repository implementation that is both MOF and JMI compliant. It can store the very first harvesting results.

6.4.1 Grammar to MOF Mapping

The repository generator maps a grammar to a MOF compliant meta-model. The mapping, which is described here, is an improved version of the one described in [AP 03] which produces a rather naïve repository structure. We call the mapping EBNF→MOF because the grammar is specified in EBNF and the repository in MOF.

A meta-model style defines the general structure of the repository. Two different meta-model styles have been implemented: normal and compact. The normal meta-model style produces a repository similar to the AST and has model elements for productions and tokens. The compact meta-model style produces a simplified and more semantic repository: tokens have been replaced by their textual representation in the productions. We describe both meta-model styles in the following sections.

The population strategy can be chosen independently from the meta-model style. We refer to section 6.3.2 at page 33 for more information on population strategies. The repository consequences for both concepts are summarized in the list below.

- **Meta-model style**
  Defines the general structure of the repository model. It defines the mapping of the grammar to the meta-model and decides whether tokens are represented as separate objects or as plain text.

- **Population strategy**
  Defines which elements are populated in the repository by the harvester. The decision is taken for each single token, production, production alternative and element. The population strategy decides if a corresponding model element should be generated in the repository model.

Each combination of population strategy and meta-model style results in a tailored repository. The meta-model style decides if the meta-model contains tokens or not; the population strategy decides which tokens and/or productions are populated. As an optimization, only model elements that can be populated are included in the meta-model. Table 6-5 shows the contents of the repository for some combinations. The options are all, some or no model elements for both tokens and productions. For example, the compact meta-model style with named population strategy results in a repository with no token model elements and some production model elements.

<table>
<thead>
<tr>
<th>Tokens and/or Productions</th>
<th>Population Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Meta-model style</strong></td>
<td><strong>Population Strategy</strong></td>
</tr>
<tr>
<td>Full</td>
<td>Named</td>
</tr>
<tr>
<td>Normal</td>
<td>Some T. + Some P.</td>
</tr>
<tr>
<td>Compact</td>
<td>Some P.</td>
</tr>
</tbody>
</table>

The compact meta-model style has been developed for reasons that appeared during testing of the normal meta-model style:

- **Resource saving**
  A full-blown repository quickly gets very large in size and has negative impact on
performance. The size of the repository should be reduced, possibly at cost of some information.

- **More semantic representation**
  The repository should improve semantics compared to the AST. The concept of tokens is only important during parsing. For example, in a model element ‘Table’ it is interesting to know its name, instead of the token that contains the name, which only adds another model element. Removing the concept of tokens simplifies developing M2M-transformations for generating the initial models.

Resources are saved by not storing token elements but only strings: tests show that this saving can be more than 50%. A more semantic representation of the source code is achieved by skipping tokens in element representations: it simplifies using the models in M2M-transformations. A traditional element access would have looked like:

```
Root \rightarrow table_definition \rightarrow table_name_token \rightarrow text_contents
```

With the compact meta-model style this would look like:

```
Root \rightarrow table_definition \rightarrow table_name_text
```

It completely hides the notion of tokens and stores all information in the production alternative.

### 6.4.1.1 EBNF and MOF Relation

EBNF and MOF are both meta-meta-models: they describe how meta-models are defined. This is a plausible statement as EBNF is compared to the 4-layer meta-data architecture of MOF. We refer to section 2.3.1 at page 14 for more information on MOF and the 4-layer meta-data architecture.

EBFN is a meta-meta-model (on M3) which describes how we should describe grammars (on M2). A grammar is a meta-model which describes source code in a populated AST (on M1). The source code may describe a running program (on M0). It is shown in table 6-6.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Purpose</th>
<th>EBNF example</th>
<th>UML example</th>
</tr>
</thead>
<tbody>
<tr>
<td>M3</td>
<td>Meta-meta-model</td>
<td>EBNF</td>
<td>MOF</td>
</tr>
<tr>
<td>M2</td>
<td>Meta-models</td>
<td>Grammar</td>
<td>UML</td>
</tr>
<tr>
<td>M1</td>
<td>Models</td>
<td>Populated AST</td>
<td>UML model</td>
</tr>
<tr>
<td>M0</td>
<td>Data, instances</td>
<td>Program</td>
<td>Program, Object</td>
</tr>
</tbody>
</table>

It could be argued that an M4 layer must exist as well because we can describe EBNF in MOF. This discussion is out of scope for this thesis. In general, MOF is self-describing; by this recursion there exist an infinite number of M-layers.

The choice for M-layers for EBNF is consistent with our intentions to generate UML models from populated AST’s: both the populated AST and the UML models reside on the same layer (M1). It does not mean that it is illegal to cross layer-boundaries when generating (meta-) models, but remaining on the same layer is the most logical and therefore preferred way.

The observations above have the following consequences for the workbench process. It is shown for each M-layer:

- **M3**
  The EBNF meta-meta-model must be mapped to the MOF meta-meta-model. An EBNF\rightarrow MOF transformation should be developed to implement the mapping.
• **M2**
  For each grammar a corresponding meta-model should be generated in MOF.

• **M1**
  For each source file a repository should be populated with the extracted information.

• **M0**
  No action.

The EBNF meta-meta-model is actually the ‘grammar of the grammar’ or meta-grammar, used to generate the parser generator itself. It defines a particular implementation of EBNF for a particular parser generator. This has the following practical consequences for the harvesting workbench:

• **Parser generator**
  For each parser generator in the workbench the meta-grammar must be mapped to the MOF meta-meta-model. An EBNF to MOF transformation should be developed for each used parser generator (on M3) to implement the mapping.

• **Grammar**
  For each grammar a meta-model (on M2) should be generated in MOF.

• **Source file**
  For each source file a model (on M1) should be generated.

Changing the parser generator has implications for the repository generator: another transformation must be developed to achieve the EBNF to MOF mapping.

### 6.4.1.2 EBNF to MOF: Normal Meta-Model Style

An EBNF to MOF mapping has been implemented using the normal meta-model style, which generates a MOF meta-model similar to the AST structure: both productions and tokens are represented in the meta-model. It uses the base classes shown in figure 6-10. These classes are present for each generated meta-model.

![Diagram](image-url)

**Figure 6-10** Normal meta-model style class overview (in UML Profile for MOF)
‘Node’ is the base class for all classes in the meta-model. It defines line attributes for referencing the original source code. ‘Token’ defines column and text attributes for referencing the original source code. ‘Production’ does not add any attributes, but is included to distinguish production and token classes.

Each token in the grammar inherits from ‘Token’ and does not define any other attributes or operations. Its name starts with ‘T_’ and ends with the token name in the grammar.

Each production in the grammar inherits from ‘Production’ and does not define any other attributes or operations. Its name starts with ‘P_’ and ends with the production name in the grammar. It serves as a single base class for all related production alternatives, which are generated as separate classes.

Each production alternative inherits from the corresponding production class and defines attributes for all populated elements in the alternative. The types of the attributes are either a specific token or a specific production. We call this strong attribute typing, opposed to weak attribute typing where all attributes are of type ‘Node’. The class name starts with ‘ALT_’, followed by the production name from the grammar and ends with the alternative number.

Each production alternative element has a name that is defined in the grammar or generated by the harvester generator. We refer to section 6.3.1.2 at page 30 for more information on naming of production elements.

Strong attribute typing provides a more semantic meta-model. It simplifies developing transformations to initial target models because the meta-model is easier to understand. Besides that, it enforces a smaller set of valid models therefore requiring fewer checks and asserts in the transformations that use the models.

Table 6-7 shows the mapping from EBNF constructs to MOF constructs.

<table>
<thead>
<tr>
<th>EBNF construct</th>
<th>MOF construct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terminal (i.e. token)</td>
<td>Concrete class inherits from Token-class</td>
</tr>
<tr>
<td>Non-Terminal (i.e. production)</td>
<td>Abstract class inherits from Production-class</td>
</tr>
<tr>
<td>Alternative (or-clause)</td>
<td>Concrete class inherits from production</td>
</tr>
<tr>
<td>Sequence of Elements</td>
<td>Named list of attributes</td>
</tr>
<tr>
<td>Multiplicity (?, * or + clause)</td>
<td>Attribute multiplicity</td>
</tr>
<tr>
<td>Optional (0 or 1-clause)</td>
<td>0..1</td>
</tr>
<tr>
<td>Required (1 clause)</td>
<td>1</td>
</tr>
<tr>
<td>Many (0 or many-clause)</td>
<td>0..* (ordered)</td>
</tr>
<tr>
<td>At least 1 (1 or many-clause)</td>
<td>1..* (ordered)</td>
</tr>
</tbody>
</table>

Figure 6-11, figure 6-12 and figure 6-13 show the EBNF→MOF mapping for some simple EBNF constructs. The EBNF constructs are shown in the text box on the left side, the corresponding MOF constructs are shown as UML model elements on the right side.

The mapping results in a repository model that is similar to the AST structure. When using production element naming the repository gains semantics over syntax. For example an element ‘fieldName’ is more informative than ‘the third element’. It enables quick transformation development in a semantic rather than syntactic way.

The mapping to MOF produces attributes with types instead of composite associations for the containment relationships. The reasons are performance and simplicity. Attributes with types
are a light-weight version of composite associations for the Carat.MOF functionality of ArcStyler. It causes a performance increase and clearer meta-model diagrams.

![Figure 6-11 Mapping two production alternatives to MOF (in UML Profile for MOF)](image1)

![Figure 6-12 Mapping a named element to MOF (in UML Profile for MOF)](image2)

![Figure 6-13 Mapping multiplicities to MOF (in UML Profile for MOF)](image3)

6.4.1.3 **EBNF\(\rightarrow\)MOF: Compact Meta-Model Style**

An EBNF\(\rightarrow\)MOF mapping has been implemented using the compact meta-model style, which reduces the size of meta-models and models compared to the normal meta-model style. Obviously it comes at cost of less information. The meta-model structure is shown in figure 6-14.

The main difference with the normal meta-model style is that there are no tokens anymore. Each token class is replaced by its textual representation: many resources are saved by not storing additional information about each specific token. Also, no token classes are generated in the meta-model.

Tests show a significant size reduction for meta-models and models. It has been tested with a simplified PL/SQL grammar for table definitions. The test source files have 1200 lines of PL/SQL in total. Table 6-8 shows collected data for the ‘full’ and ‘named’ population strategy for both meta-model styles. These numbers are statistically insignificant but give a nice indication of the possible savings.
Table 6-8 Test results for different meta-model styles

<table>
<thead>
<tr>
<th>Time (s)</th>
<th>Normal meta-model style</th>
<th>Compact meta-model style</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Action</strong></td>
<td><strong>Full</strong></td>
<td><strong>Named</strong></td>
</tr>
<tr>
<td>Initialize parser</td>
<td>0.050</td>
<td>0.050</td>
</tr>
<tr>
<td>23 productions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>39 tokens</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parse</td>
<td>0.681</td>
<td>1.091</td>
</tr>
<tr>
<td>(nr. Productions)</td>
<td>(2520)</td>
<td>(1694)</td>
</tr>
<tr>
<td>(nr. Tokens)</td>
<td>(5706)</td>
<td>(1078)</td>
</tr>
<tr>
<td>Analyze</td>
<td>1.482</td>
<td>0.261</td>
</tr>
<tr>
<td>Both AST and MOF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write the repository to file (file size, XMI)</td>
<td>8.563</td>
<td>2.433</td>
</tr>
<tr>
<td></td>
<td>(3972 Kb)</td>
<td>(1135 Kb)</td>
</tr>
<tr>
<td>Write the meta-model file (file size, XMI)</td>
<td>0.560</td>
<td>0.341</td>
</tr>
<tr>
<td></td>
<td>(273 Kb)</td>
<td>(152 Kb)</td>
</tr>
</tbody>
</table>

Table 6-9 compares both meta-model styles. The value of the compact meta-model style is evident, even though parse time may have increased for full population. The numbers are calculated from the data in table 6-8, which are not statistically significant and therefore the numbers in table 6-9 are rounded.
### Table 6-9 Compact meta-model style compared to normal meta-model style

<table>
<thead>
<tr>
<th>Property</th>
<th>Full</th>
<th>Named</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse time</td>
<td>-10 %</td>
<td>20 %</td>
</tr>
<tr>
<td>Analysis time</td>
<td>60 %</td>
<td>40 %</td>
</tr>
<tr>
<td>Model size (XML)</td>
<td>60 %</td>
<td>40 %</td>
</tr>
<tr>
<td>Model save time (XML)</td>
<td>70 %</td>
<td>50 %</td>
</tr>
<tr>
<td>Meta-model size (XML)</td>
<td>20 %</td>
<td>30 %</td>
</tr>
<tr>
<td>Meta-model save time (XML)</td>
<td>0 %</td>
<td>30 %</td>
</tr>
</tbody>
</table>

### 6.4.2 Test Results

The workbench has been tested continuously during development. Several grammars have been used to test all possible EBNF-constructs during repository generation.

A test showing benefits of the compact meta-model style has already been shown in the previous section. It shows that the compact meta-model style is an improvement in space and time performance at cost of token information.

An example repository model is shown in figure 6-15. For simplicity only a subset of the meta-model is shown, which has been generated using compact meta-model style and named population strategy.

![Figure 6-15 Part of generated meta-model (in UML Profile for MOF)](image)

For example, the class ‘P_plsql’ contains meta-information, implemented as attributes starting with ‘MI_’. We refer to section 6.3.1.5 at page 32 for more information on meta-information.

Due to the compact meta-model style all token elements in the production alternatives have the type ‘string’ instead of a specific token class.

### 6.5 Code Generation

The harvesting workbench extensively uses code generation, which minimizes the amount of manual work required to build a working harvester. Table 6-10 shows several tests regarding the code generation compared to the grammar size. All numbers are lines of code including comments, etc., and are not statistically significant. They give an indication of the conciseness...
of the grammar compared to the functionality. The generated lines of code for the repository are estimated based on the file size because of the large number of generated files. The normalized values are shown for easier comparison.

All grammars used implicit token declarations, named population strategy and the compact meta-model style, which results in the most concise grammar with the most concise results.

Table 6-10 Code generation ratio for different grammar sizes

<table>
<thead>
<tr>
<th>Lines of code</th>
<th>Harvester Generator</th>
<th>Repository Generator (Estimated)</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grammar</td>
<td>Default Parser</td>
<td>Extra Harvester</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>1 777</td>
<td>2 513</td>
<td>4 290</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>33</td>
<td>57</td>
</tr>
<tr>
<td>107</td>
<td>3 330</td>
<td>4 320</td>
<td>7 650</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>40</td>
<td>71</td>
</tr>
<tr>
<td>203</td>
<td>7 296</td>
<td>12 864</td>
<td>20 160</td>
</tr>
<tr>
<td></td>
<td>36</td>
<td>63</td>
<td>99</td>
</tr>
</tbody>
</table>

The normalized amount of generated code increases for larger grammars. It is caused by the fact that a grammar contains a fairly fixed amount of initialization lines and token definitions. Extra lines in the grammar generally imply extra productions, which cause much code to be generated.

6.6 Conclusion

The harvesting workbench provides infrastructure and tools for the first phase of the harvesting process, satisfying a number of requirements. We refer to section 5.2 at page 23 for more information on the requirements. It supports the harvesting process to:

- Create an initial UML representation of an existing system. Tests have proven this.
  - The source structure can be defined in a grammar.
  - The grammar can be developed incrementally using special grammar constructs.
  - A MOF-repository can be generated and populated with initial harvesting results.
  - A mapping between source and target constructs can be specified in two phases: the grammar and the transformation to initial models.
  - Structural elements can be harvested. There are no restrictions on what elements can currently be harvested, other than the meta-model of the initial models.
  - References or fragments of the original source code can be attached to the generated models. It is supported in the harvester and the generated repository.

This part has not covered the analysis requirement, which will be covered by the GenericAST in chapter 7 at page 49.

Below the concerns with the non-functional requirements are described. We refer to chapter 5.3 at page 25 for more information about these requirements.

1. Genericity
   Input structure can be described concisely, for which input specific harvesting
functionality is generated. The standard harvester functionality framework provides common harvesting functionality.

2. **Ease of use**
   Manual effort has been concentrated to the grammar and the transformation to initial models. A single point of access to the workbench has been provided by a script, which controls the complete workbench process.

3. **Correctness**
   A standard solution has been used for generating input specific functionality: parser generators are widely used in industry and academics. It is a well defined, tested and proven technology. During development the harvesting workbench has been tested thoroughly in many different configurations and all apparent problems have been repaired. However, we cannot guarantee 100% reliability. The Grammatica parser generator is not widely used, such as Antlr\textsuperscript{25}, which may have negative impact on reliability.

4. **Understandable**
   A harvesting process has been defined which is easy to understand, because each step clearly adds value to the process. The harvester generator generates a harvester based on the grammar. The harvester can read the source code and populate a repository, which has been generated using the same grammar. Finally the contents of the repository are transformed to suitable initial models.

5. **Abstract from source system specific constructs**
   The harvester is the first step of abstracting from source specific constructs. It provides a first filter by allowing grammar annotations, which can filter information and replace syntax by semantics. The transformation to initial models provides further abstractions: it can generate UML models or GenericAST models.

6. **Portable**
   A purely Java based solution has been used for the harvester generator. It has been written in Java and generates Java. The repository generator depends on ArcStyler and produces Java-based MOF-repositories. Therefore, the whole workbench is as portable as ArcStyler.

\textsuperscript{25} [http://www.antlr.org](http://www.antlr.org)
In a harvesting process (as shown in figure 6-2 at page 27) all M2M-transformations must be developed by hand. Reusing M2M-transformations from previous harvesting projects could save much effort. Generally reuse is quite hard because the source code structure and thus the grammar differ. A M2M-transformation maps two meta-models and if one meta-model changes, the M2M-transformation must be changed as well.

The concept of a generic intermediate model has been investigated to simplify reuse. We call the generic intermediate model ‘GenericAST’ (short for Generic Abstract Syntax Tree). This chapter describes the concepts and initial implementation regarding the GenericAST. The GenericAST comes with a set of tools that support its usage, which are also described in this chapter.

The GenericAST is a generic meta-model in which information about software systems can be stored independent of the source language. The disadvantage of an extra step in the harvesting process brings several advantages over the straight-to-the-target-model approach. The main benefit is reuse of transformations and analyses.

The current implementation of the GenericAST is a first attempt to provide a generic meta-model. It is not a finished or true generic solution. That would have been too ambitious for the available time frame. An evolution of the meta-model based on usage experience in harvesting projects is foreseen, far beyond our project.

The GenericAST is an optional step in the harvesting process. It is an attempt to make harvesting economically more attractive by allowing transformation and analysis reuse. For every harvesting project it must be decided if the GenericAST is usable, both technically and economically.

The GenericAST is a concept that is used by the ADM\textsuperscript{26} taskforce of the OMG. We refer to section 16.1.1 at page 113 for more information on ADM.

## 7.1 Approach

An incremental development approach has been used for the GenericAST meta-model, which consists of the following parts:

- **Structural part**
  Represents structure of the system, for example database tables.

- **Behavioral part**
  Represents behavioral logic of the system, for example procedure contents.

First the structural part of the GenericAST has been developed and tested. Then an extension has been made by including the behavioral part. In parallel several frameworks and tools have been developed that support usage of GenericAST models.

During development, the focus has been put on representing procedural and object-oriented languages (PL/SQL, C, Java, etc.). The GenericAST has continuously been tested using Haebop PL/SQL source code. The tests covered transformations to a GenericAST model and transformations from the GenericAST model to initial UML target models. The structural part has also been tested thoroughly with Java because it did not take much effort to implement a reflection-based Java harvester. Testing with other languages is out of scope for this project, however.

\textsuperscript{26} Architecture Driven Modernization
OMG’s ADM taskforce is developing a similar generic meta-model. No attempt has been made to make the GenericAST ADM-compliant, because ADM is not even close to a standard. Initial ADM submissions have been used for knowledge and inspiration. They are not completely implemented because the submitted meta-models are work-in-progress and not all parts have been considered very useful for this project.

7.2 Rationale

The main reason for adding the GenericAST to the harvesting process is transformation and analysis reuse. It enables investments in more complex transformations and analyses because they are not specific to one project. The UML meta-model has not been used for the GenericAST because it is unsuitable for representing certain constructs, such as behavior.27

The GenericAST is a sensible step when considering how the harvested information is represented during the harvesting process. In every step syntax is replaced by semantics and model filtering is applied. The GenericAST provides an intermediate model, in between source-oriented and target-oriented models. It comes at cost of extra storage and time requirements, however.

Usage of the GenericAST requires a match between source models, a GenericAST model and target models. Sensible transformations must exist to benefit from the GenericAST.

7.2.1 Reuse

The GenericAST enables transformation and analysis reuse. The benefit is illustrated with the following example. Suppose we have N different source structures which are specified by a grammar, and M different kinds of target models we want to generate. Without the GenericAST we need for every source structure a transformation to every target model: N \* M transformations in total.

When the GenericAST is included in the process we need a transformation for every source structure to the GenericAST, and a transformation from the GenericAST to every target model. This results in N + M transformations. For small values of N and M this already pays off. It is illustrated in figure 7-1 where we need 6 transformations instead of 9 for N = M = 3.

![Figure 7-1 Harvesting process with GenericAST](image)

Generic transformations can be developed that transform GenericAST models to respective target models. A library of harvesting assets can be built up, with best practices, human skills and ready-to-go transformations and analyses. With reusability investment in developing more complex transformations and analyses becomes more attractive. It increases the return on investment of harvesting projects.

27 The UML 1.4 meta-model has been used in this project.
7.2.2 **UML Meta-Model**

The UML meta-model has not been used for implementing the GenericAST because it cannot represent all information efficiently and effectively. For example, UML lacks expressiveness to store behavior constructs. It could be represented in tagged-values but this is not a practical solution. Therefore effort has been put in developing a custom GenericAST meta-model.

Future versions of the GenericAST may converge to the UML meta-model using action semantics. Action semantics provide representation for behavior. A reason not to converge to the UML meta-model is that it is specially designed for object-oriented models. The GenericAST does not enforce usage of the object-oriented paradigm, because many languages are not object-oriented by nature. For example, PL/SQL is not object-oriented but procedural.

7.2.3 **Representation**

Different phases in the harvesting workbench have different representations of information, as illustrated in figure 7-2. At the source code phase the representation is a sequence of lines of text. The representation in the AST during parsing and population is a tree. The representation in the generated MOF-repository is a filtered and named tree. The filtered and named tree has more explicit semantics than the textual source code.

The GenericAST contains a set of concepts with properties organized in a tree. The concepts may be derived from different subtrees in the generated MOF-repository. The semantic value of the representation is increased with each transformation.

![Figure 7-2 Representation in different harvesting phases](image)

7.2.4 **Cost**

The price we pay for the genericity and increased semantic representation is performance. The GenericAST models will be larger than the original source code. Semantics have been made explicit that were implicit in the source code. The following has been done in the harvesting process:

Let $G$ be the grammar which specifies language (subset) $L$

Let $S$ be source code in the language (subset) $L$

Then

1. Generated MOF model = $S \times G$

2. GenericAST model $\leftarrow$ transform(Generated MOF model) = transform($S \times G$)

The first statement says that the knowledge in the grammar has been ‘exploded’ over the source code; the second says that the GenericAST model is solely gained by a transformation.

---

28 We consider UML1.4 here because that version is currently supported by ArcStyler.
of the source code exploded with the grammar\textsuperscript{29}. It is trivial that making implicit knowledge explicit costs performance.

7.3 Architecture

The GenericAST is a generic meta-model that comes with supporting tools and frameworks. This section describes the GenericAST in relation with its tools and frameworks. Figure 7-3 shows it in a high-level overview.

![GenericAST with frameworks and tools](image)

Figure 7-3 GenericAST with frameworks and tools

The ‘Core’ consists of the GenericAST repository which implements the meta-model.

The ‘General Frameworks & Tools’ consists of the following elements:

- **Analysis framework**
  Supports developing GenericAST analyzers in Java. Analyzers collect information from GenericAST models by tree walking. The models may not be changed.

- **Processor framework**
  Supports writing GenericAST model processors in Java. A model processor is an analyzer that may change the model. The processor framework extends the analysis framework.

- **GUI**
  Provides a single point of access for GenericAST models. It supports the complete GenericAST model life-cycle (creation, view, modification, deletion). The current GUI implementation integrates all GenericAST tools and frameworks.

- **Annotation framework**
  Provides means to store manual model annotations externally. It allows reapplying annotations when the GenericAST model has been regenerated.

- **Various tools**
  Provide convenience functionality during development and testing.

The ‘Specializations & Implementations’ consists of the following elements:

- **Implementations**
  Several analyzers and processors have been implemented.

\textsuperscript{29} The GenericAST model is not a subset because it contains other model elements than the generated MOF model.
Classification module
Specialization of the annotation framework. It allows us to classify GenericAST model elements, which is discussed in section 7.5.3.

Sections 7.4 and 7.5 discuss the meta-model and various frameworks.

7.4 Meta-Model
This section gives a high-level overview of the GenericAST meta-model. The meta-model is not described in detail because it is too large and therefore out of scope for this thesis. We focus on the general ideas and concepts that have been used and refer to appendix D at page 141 for a more elaborate view on the GenericAST meta-model.

The GenericAST has been developed as generic as possible while ensuring it would work for PL/SQL. Attempting to build a perfect solution would have been naive; rather an initial pragmatic solution has been developed, which can be further developed as more usage experience is gained.

Usability of the GenericAST was very important: the more different ways the models can be accessed, the better. Therefore several design decisions have been made that may seem illogical or resource intensive at first. For example, every association between model elements is bi-directional. It simplifies querying the models from different view-points, although it costs extra resources. Having bi-directional associations we can find out, for example, what type a certain attribute has (e.g. integer) from an attribute viewpoint. From the type viewpoint we can find out every attribute that has the type. At cost of extra storage time performance and improved usability are gained.

The GenericAST supports storing custom information in its model elements by using the tagged-value concept, inspired by the UML meta-model. A tagged-value contains a textual name and a textual value. Any amount of tagged-values can be attached to any model element, making it a simple but powerful extension mechanism. It enables to store information that was not anticipated during GenericAST development.

References to the original source code are important throughout the whole harvesting process. The GenericAST supports it by having a separate model element for storing source reference information. This model element can be attached to any other model element. It is a preferable solution over storing the information in a tagged-value because it is more explicit, making it easier to be used.

7.4.1 Structural Part
The structural part of the GenericAST meta-model provides representations for the structure of a system; it does not represent behavior. This section describes the high-level overview of the structural part of the GenericAST meta-model. It has been subdivided in several packages as shown in figure 7-4, which shows the packages and dependencies between the packages.

The Core package contains the base elements of the meta-model. It defines super-classes used in the other packages with general features, such as names, visibility setting, tagged-values and references to the original source code.

The Container package provides model elements to contain other model elements as their dedicated purpose. It can represent Java packages as containers, for example.

The Entity package contains model elements that are similar to a Java class or a database table. It is a logical unit with features, such as attributes or operations.

The Feature package contains model elements such as an attribute (a named element with a type and a value) or an operation (a named element with behavior). An operation can have parameters.
The Constraint package provides constraint model elements that can be attached to an attribute (e.g. a foreign key constraint) or on a parameter (e.g. pre- and post-conditions).

The Type package contains a type hierarchy. Attributes, parameters and operations can have a type. There are a number of primitive types (such as integer and string), and entities can also be a type.

The Trigger package contains trigger model elements as found in databases. It allows for behavior on certain events (such as creation of an entity-instance). The actual behavior is not represented in the structural part.

Figure 7-5 and figure 7-6 show examples for representing Java and PL/SQL constructs in a GenericAST model. In figure 7-5 we see that packages are represented by containers and classes by entities. Both features have an association to a specific type model element. In figure 7-6 we see that a database table is mapped to an entity and each column is mapped to a structural feature. Although the syntax for specifying Java and PL/SQL is different, similar constructs from both languages can be represented using the GenericAST meta-model.
7.4.2 Behavioral Part

The behavioral part of the GenericAST meta-model is an extension to the structural part. Figure 7-7 shows the package structure for the behavioral part in combination with the structural part, including the dependencies between the packages.

The behavior packages all start with ‘behavior_’. They extend the level of detail of certain model elements in the structural part. Every model element that can contain behavior (operations, triggers) has a reference to the container of behavioral model elements (which is a Block).

Figure 7-7 GenericAST behavioral packages added to structural packages (in UML)

The behavior Core package contains the base classes for behavioral model elements. It defines a Block and the abstract statement. A Block is a statement that contains other statements as its sole purpose. Each statement can be enriched with a context, which is a collection of known named elements with a type and a value.

The behavior Statement package contains further specifications of statements. A dedicated model element has been created for each general programming language construct. Examples are a loop statement or a conditional statement.

---

30 Although we do not claim to be complete here
The behavior Expression package contains expression representations. A broad range of expressions can be represented at different detail levels. Expressions can be represented as a single string or be split up into sub-expressions. This package uses the behavior Operator package, which is not shown in the figure. It defines operators that are used in expressions, such as ‘plus’ or ‘minus’.

The behavior Property package contains property analysis representations. Several model elements can be enriched with properties. For example, the statement ‘x := 1’, which puts the value ‘1’ into the variable ‘x’, can be enriched with the property ‘writes variable x’. It is intended to support more complicated analyses on GenericAST models. This package has not been tested yet due to the lower priority, but a base has been provided for future extension.

Figure 7-8 shows a simplified example of representing Java behavior in a GenericAST model. The if-statement is represented by a ConditionalStatement. It has a condition, which is a ConditionalExpression with a LookupExpression to lookup the value of ‘a’, which is used as the condition. The ConditionalStatement has two Blocks: one in case the condition is true; another in case the condition is false. Both contain similar behavior: a method call. The method call is a LookupExpression (to lookup the method) in a hierarchy of a CallStatement, a CallExpression, and an ExpressionStatement. This complex structure has been chosen to be able to represent both expression and statement method calls. In the C-language a method call is an expression (indicated by the operator ‘()’); in COBOL it is a statement (the ‘call’ statement). Although the syntax differs for many languages, similar constructs can be represented in a single model with the GenericAST meta-model.

![Figure 7-8 GenericAST mapping for Java example (in UML)](image)

7.5 Frameworks & Tools

In this section various frameworks and tools that come with the GenericAST are described. We refer to section 7.3 at page 52 for a high-level overview of the GenericAST and associated frameworks and tools.

7.5.1 Analysis Framework

An analysis framework has been developed for analyzing GenericAST models. It simplifies implementing model analyzers by providing general functionality. Figure 7-9 shows the usage of the analysis framework on GenericAST models.

Analyzers are tree walkers that walk the contents of a GenericAST model. They may accumulate information, but may not alter the repository contents. Processors are intended to change the repository contents and extend analyzers. The analysis framework consists of base-classes that can be used, which define by default idle behavior for each model element type. Several methods may be overridden to implement custom actions, for example to print the model element contents on screen.
7.5.2 Annotation Framework

An annotation framework has been developed for annotating GenericAST models. In a harvesting project generated GenericAST models may need to be enriched with additional information. For example, to indicate whether a model element has a technical purpose or is part of the business logic. Subsequent steps in the harvesting process can be influenced by this information. It supports to enrich the models with human assessment. We call the process of manual model enrichment ‘annotation’.

It may happen that the final GenericAST models are generated after multiple iterations during the harvesting project. Starting with low detail models, selected details that are thought to be relevant can be added in several iterations. Of course manual annotations should not be redone manually in each iteration: repetitive work is boring and human effort is costly.

With the annotation framework GenericAST models can be annotated and the annotations can be stored in a separate repository. The annotations can be reapplied automatically when the GenericAST model is regenerated: we call this process ‘injection’. The framework stores annotations in such a way that the injector can identify model elements in the model and optionally re-inject the annotation.

Figure 7-10 shows the process of the annotation framework. The framework has been integrated in the GenericAST GUI to provide a convenient interface for the user.

Annotations cannot be applied to every GenericAST model element, caused by the choice of how to identify a specific model element. For each model element the combination of the following properties are chosen as being identifying:

- Name of the element
- Type of the element
- Feature name in the owning element

These properties are used in combination with the complete containment path of the model element as identification. The containment path is the list of model elements from the repository root to the specific model element over composite associations. Figure 7-11 shows an example: the containment path for model element ‘C’ is ‘A’-‘B’-‘C’.
It implies that only named elements can be annotated if their containment path consists solely of named elements. In practical situations this is sufficient because the following model elements will be interesting for manual annotation:

- Behavioral features and alike, such as triggers
- Entities which contain behavioral features
- Containers which contain entities

All are named elements and are usually contained by other named elements, and can thus be annotated. Even statements can be annotated because a statement is a named element. However, manually annotating statements is already questionable: statements are quite detailed and a GenericAST model may contain a huge amount of them.

Figure 7-12 shows the composite structure of the annotation framework, which specifies how annotations are stored. An identically structured XML format has been created for storing annotations on disk; a dedicated MOF-repository would have been too heavy-weight for this purpose.

Currently an annotation can be:

- Creation of a new model element with attributes
• Setting the value of an attribute of an existing model element

An annotation contains a set of properties that specify the values of the attributes. The set of primary keys defines which of the attributes compose the primary key of the model element in the repository. It prevents duplication of annotations when doing multiple re-injections in one model.

A specialization of annotations is classifications, which we cover in section 7.5.3.

### 7.5.3 Model Element Classification

Classification is the process of classifying GenericAST model elements into the following categories:

- Pure business logic
- Blend
- Pure technical logic

Semantics of the source code are used to decide on its category. Classification is a specialization of annotation; therefore the annotation framework is used for implementing classifications. Classification uses a floating scale, as shown in figure 7-13.

![Figure 7-13 Model element categories for classification](image)

The purely technical logic is generally abstracted away by MDA: it can be generated by the code generation process. The purely business logic generally contains the interesting information, such as calculations. It may be transformed into a target language by using a pretty-printer for the desired language. The blend, with a mixture of both, should be manually transformed into the target model because it is hard to automate.

Classification can be done by various agents:

- **Harvesting engineer**
  This will lead to good classifications: an engineer can reason about the semantics of a code fragment and can use associated source code comments.

- **Model analyzer**
  This will lead to an initial classification guess, based on a set of inference rules regarding the model element in its context. For example, when many IO-statements are found in a procedure body, it is probably a technical procedure. We expect a poor classification quality, because an analyzer cannot reason about the semantics of the source code, not to mention reason about surrounding comments.

- **Artificial neural network**
  This allows classification prediction using neural network functionality. The network must be trained with examples that have been manually classified to gain predictive power. Finally, the results must be verified to assess the classification quality.

Classifications are stored in a special tagged-value with the classified model element, using a scale from 0 (purely technical) to 100 (purely business). When classifying a model element the annotation framework is used to manage the associated annotation. Classifications can be
recovered if the annotations repository has been lost: classifications can be harvested from a given GenericAST model because it is known how to identify them. Figure 7-14 shows the extended process.

**Figure 7-14 GenericAST classification usage overview**

Classification recovery is an extra safeguard and provides a round-trip classification process. When having either the classified repository or the classification repository, the other can be derived from it. This functionality could be extended to other annotations as well, as long as the annotations can be identified.

Having a classified GenericAST model, the classifications should be used. We discuss two options for classification usage in sections 7.5.3.1 and 7.5.3.2.

7.5.3.1 Model Filtering During Parsing

Multiple iterations may be used in a harvesting process, where each iteration increases the level of detail but still limits the model size by filtering for ‘interesting’ elements. Figure 7-15 shows this process. Several items in the figure have been omitted for simplicity: the first iteration may include UML models for visualization; the second iteration may include a classified GenericAST model.

**Figure 7-15 Model filtering during parsing in iterative harvesting process**

Model filtering during parsing requires filtering support in the parser. An implementation could not be provided because of timing constraints on this project.

7.5.3.2 Model Filtering During Target Model Creation

Different target models can be generated using classified GenericAST models, resulting in different views on our GenericAST models. Figure 7-16 shows the process.
The classifications control the generation of the target models. Using the classifications, the M2M-transformation may decide for example:

- Whether target model elements are created
- In what location target model elements are created
- What target model elements are created

### 7.5.4 Graphical User Interface

A dedicated graphical user interface has been implemented for the GenericAST, providing a single point of access for viewing and manipulating GenericAST models. It has been developed to provide better interaction with the models than hacking rough XMI files. The GUI integrates all tools and frameworks around the GenericAST and provides complete model life-cycle functionality.

It is a rudimentary first version with essential functionality to (among other):

- **View GenericAST models**
  For example as shown in figure 7-17, which shows a composite view of the loaded GenericAST model in a tree structure. The model elements are shown with their composite associations. The right side shows properties and actions for the selected model element.

- **Modify GenericAST models**
  For example as shown in figure 7-18, which shows the creation of a new model element in the repository. The set of model elements we can choose from is restricted by the valid types for the composite association.

- **Access frameworks and tools**
  For example as shown in figure 7-19, which shows we can run model analyzers on parts of a loaded GenericAST model. The results are shown on the lower-right part of the interface.

The GUI has been designed to be easily adaptable for other meta-models, using an interface to abstract repository implementation details. Several features are specific for the GenericAST, such as the source reference detail view, however.

The GUI can be easily accessed from ArcStyler, giving it the same look & feel.
Figure 7-17 Composite view of GenericAST model

Figure 7-18 New element in GenericAST model respects valid types
7.5.5 Model Merger

A rudimentary model merger has been developed and integrated into the GenericAST GUI. It duplicates a given source model element (including the complete composite structure) into a target location. It requires the GenericAST meta-model for considering the names of the elements.

The model merger uses a relatively simple algorithm. When the following conditions are true:

- a model element is about to be created in the target context
- the model element has a name
- the name is valid (i.e. the name has at least one character)

Then it does the following check:

- If in the target context a model element exists with the same name and the same type, that model element is used instead of creating a new one

In all other cases it creates a new model element in the target context.

The model merger can be used for merging different GenericAST models. It counteracts the scalability issues when using high-detail models for large input by allowing splitting up the source code in multiple parts and harvesting them in parallel. Finally the resulting GenericAST models are merged into a single target GenericAST model. This workaround allows harvesting larger amounts of source code using high-detail GenericAST models. We refer to section 7.7.2 at page 68 for more information on GenericAST performance issues for larger input.
7.6 Test Results

The GenericAST has been tested continuously during development. In this section we first discuss test results on the structural part of the GenericAST, which have been performed with both PL/SQL and Java. Then test results on the behavioral part are discussed, which have been performed only with PL/SQL.

7.6.1 Structural Part

The structural part of the GenericAST has been tested with both PL/SQL source code and Java structures. Input for the tests were sample PL/SQL source code from Haebop and generated Java structures by a reflection-based Java harvester. Figure 7-20 and figure 7-21 show some initial test results, which are UML models generated from GenericAST models. No visual representation of GenericAST models is available other than that offered by the GenericAST GUI, which is not as clear as the UML models shown in figure 7-20 and figure 7-21.

Figure 7-20 looks very similar to the resulting UML model in the workbench tests shown in figure 6-8 at page 39. This is intended because the same source files were used; only the GenericAST model has been introduced in the harvesting process. A non-similar model would have indicated a serious error.

Figure 7-20 GenericAST structural test result for PL/SQL (in UML Profile for EJB)

Figure 7-21 shows a part of the harvested structure of the package ‘java.util’. For simplicity all attributes and operations have been hidden. The GenericAST model was generated by running the reflection based Java harvester and contained 71 packages with 523 classes and interfaces, with 1370 attributes and 3979 operations with 3692 parameters. This gives an indication of the size of both the GenericAST and the target UML model.

Figure 7-21 GenericAST structural test result for Java (in UML)
7.6.2 Behavioral Part

The behavioral part of the GenericAST has only been tested with PL/SQL source code from Haebop. The used test source code had 2100 lines and contained package declarations with procedures. Figure 7-22 and figure 7-23 show example UML models that have been generated from the GenericAST models. Target UML models are shown because they give the best overview of the results.

In figure 7-22 a dependency for each ‘DatabaseIOStatement’ in the GenericAST model is shown, which represents an action on the database (e.g. an update of a record). On the left, classes with operations represent PL/SQL packages with procedures. Fictional classes on the right represent the kind of database statement. A dependency indicates which procedure may execute what database statement. For example, only two operations may update the database whereas many may open, close and read. No procedure writes to or deletes from the database. The documentation of each dependency contains the exact statement that is performed.

Figure 7-22 GenericAST behavioral result for PL/SQL (in UML)

Figure 7-23 is a diagram with direct call dependencies, in which a UML state chart diagram has been used for visualization. Each state represents a PL/SQL procedure and each arrow means ‘source may call target zero or more times’. For example, procedure ‘p_bereken_ex5_variant’ may call operation ‘p_bepaal_verzekerd_bedrag’ zero or more times.

The test results show that behavioral information can be represented and used to produce different views on the models. The two examples provide a convenient overview of information that is otherwise hard to find by browsing the source code.
7.7 Risk Assessment

During design and testing potential risks regarding the GenericAST appeared, regarding model size, transformation performance, and genericity mismatch. Each risk is elaborated in the following sections, together with options to minimize or deal with the consequences.

7.7.1 Model Size

The more information is stored in a GenericAST model, the bigger it gets. This may look trivial but has significant performance consequences. Tests show that when using the structural part of the GenericAST the spatial increase compared to the original source code is already significant. However, as soon as behavioral details are added even small original source code results in (very) large GenericAST models.

Figure 7-24 gives an indication of the model size in different harvesting phases. The first phase is the original source code, which is plain PL/SQL. The second phase is the generated MOF-repository which stores a subset of the AST. The third phase is an initial GenericAST model. The fourth phase is an optimized GenericAST model, which is currently only applicable for expression-level representations. Model size is measured in lines of PL/SQL or XMI. The values are normalized with respect to the PL/SQL phase. The data is not statistically significant.

The ‘structural’ test consisted of 1280 lines of PL/SQL code with table definitions, having a file size of 47 KB. In this test only structural concepts are represented in the models. Both behavioral tests consisted of 2172 lines of PL/SQL code with package definitions, having a file size of 93 KB. The ‘behavioral-1’ test represents both structural and behavioral concepts until the statement-level, whereas the ‘behavioral-2’ test also represents each expression in detail.
Figure 7-24 Normalized model size during harvesting – lines of code

The expression representation optimization strongly reduces the model size. It would have been preferable to do this optimization directly in initial GenericAST model but it appeared to be unfeasible. The optimization restructures the expression representation without changing semantics.

Figure 7-25 shows the uncompressed file size during harvesting, which appears to be quite similar to the lines of code graph. Compression will lead to significantly smaller files because it normally achieves at least 90% compression for XMI-contents.

Figure 7-24 and figure 7-25 show that for every line of PL/SQL code around 26 lines of XMI are needed to represent the same information at the expression-level in an optimized GenericAST model. This is already an improvement to the 108 lines of XMI in the generated MOF-model, but still significantly more than plain original source code.
Ways to minimize the model size risk are:

- **Use file compression**
  This reduces disk space requirements and load/save time.

- **Store only required information**
  Adapt harvesting to the information that is needed: only include relevant details.

- **Use model optimizations**
  The sooner the model is optimized, the more efficient the whole process is.

An optimal harvesting strategy uses models that contain exactly the information that is needed. For each harvesting step it must be carefully planned what level of detail is required and therefore sufficient. When considering large amounts of source code focus should be on the high-level structure; small details should only be used on smaller scales, because then models are of reasonable size and transformations are doable in reasonable time.

### 7.7.2 Transformation Performance

The M2M-transformation that generates the GenericAST model is very time-consuming. For the ‘behavioral-2’ part the transformation execution time was around 20 minutes (for 70.000 AIM-links\(^{31}\)). Figure 7-26 shows a rough indication of the M2M-transformation performance. No statistically significant data has been collected because it is out of scope for this project.

Three different transformations have been tested with four different source sizes. Each test has been performed only once because the intention was to get an indication rather than a hard claim on performance or complexity. The repository loading time has been measured and subtracted from the transformation time. The used computer was equipped with an AMD Athlon processor at 1.68 GHz with 1 GB of RAM.

Trend-lines are included to give an indication on the time complexity of the different transformations. The three transformations are of different complexity, because they differ in the amount and complexity of required transformation specifications.

![Figure 7-26 M2M-transformation execution time for different model sizes](image)

It appears that for each transformation, the execution time is polynomial in the number of links. Formulated otherwise: M2M execution time is \(O(n^{0.515})\), where \(n\) is the number of links. But still, even if the execution time is quadratic in the input size (and not exponential), it has serious performance impact on larger input.

\(^{31}\) We refer to chapter 2.1.2.1 at page 12 for more information on AIM-links
Figure 7-27 shows the relation between the source size and the number of AIM-links for each transformation. It appears to be roughly linear, although it is not statistically significant. Formulated otherwise: the number of links is $O(n)$, where $n$ is the number of lines PL/SQL source code.

Figure 7-27 Relation between lines of PL/SQL and M2M-transformation size

Figure 7-28 shows the execution time of a single AIM-link for the different transformations and model-sizes. The figure indicates two things:

1. Time to execute a single AIM-link increases with a growing source model size.
2. Execution time grow-rate increases with the complexity of the transformation.

Considering that larger transformations have more links, the execution time grows super-linear because we have more links that each takes longer to execute. Figure 7-26 suggests that the increase is not exponential but polynomial.

Figure 7-28 Execution time per M2M-transformation link

The complexity indication of the M2M-transformations suggests that the approach is quite scalable. But even for a couple thousand lines of PL/SQL code it can take hours to generate a high-detail GenericAST model. The performance problem can be in various components of the transformation, as illustrated in figure 7-29, which shows the M2M-transformation process.
In each component of the process improvements could be applied, which are listed below.

- **AIM**
  Optimize the AIM component which implements the M2M-transformation engine.

- **MOF-repository**
  Optimize the MOF-repository, of which the transformation uses at least two instances. This can be done on two levels:
  
  - **Meta-model**
    Improve the used meta-models to reduce the transformation effort.
  
  - **Implementation**
    Improve the implementation of the used repositories such that they would perform better in time and space. This can be done in two ways:
    
    - **Carat.MOF**
      Improve the MOF-cartridge such that it generates more efficient repositories.
    
    - **Manual improvements**
      Manually change the generated implementations to apply performance improvements. The following options exist:
        
        - **Convenience methods**
          Implement utility methods for model elements that simplify common tasks. They may perform short-cuts to model elements and apply caching mechanisms for speed. The methods are part of the meta-model and do not break the regeneration capability of the implementation.
        
        - **Change implementation**
          Change the repository implementation to improve the performance. This is not recommended because it breaks the regeneration capability of the implementation and therefore hinders evolution of the meta-model.

- **Transformation**
  Improve the transformation to prevent the usage of Jython code to specify the M2M-transformation. Jython code is interpreted and therefore slower than precompiled code. It can be prevented by:
Generic Harvesting Framework

- **GenericAST functionality**
  Add utility methods to the GenericAST for frequently used functions, filters and queries.

- **Generated MOF functionality**
  Enrich the generated MOF-repository with utility methods. Frequently used operations could be performed quickly (i.e. precompiled). Methods should be included in the meta-model to preserve them when regenerating the implementation.

- **Java transformation**
  Discard AIM and create custom Java transformations, which implies creating a transformation framework for the generated MOF repositories. The M2M-transformations are hand-coded and executed efficiently. It may impair overview for larger transformations, for which AIM has a convenient user interface. The speed-improvement may be considerable, however.

- **Faster equipment**
  Trivial.

Below some practical ways to deal with the transformation performance consequences are listed.

- **Optimize transformation**
  Do not use constructs that slow down the transformation, such as large amounts of Jython-code or expensive operations. The less time is spent for each link the faster the transformation is. Profiling would help to gather this information; it is not known if this is feasible, however.

- **Small models**
  Smaller models lead to smaller and faster transformations.

- **Parallel harvesting**
  Split a large input into small parts and harvest them in parallel. Several small optimized GenericAST models can then be merged into one big GenericAST model.

- **Apply transformations overnight**
  Do large transformation on a separate computer, such that it can run for hours (or days) without being needed for something else.

- **Use powerful computers**
  Faster computers with more memory tend to execute transformations in shorter time.

Especially the parallel harvesting is promising method of dealing with the performance risk. It allows harvesting larger pieces of source code in shorter time.

### 7.7.3 Genericity Mismatch

The genericity (or expressiveness) of the GenericAST could be inadequate for a given harvesting project. It means that the GenericAST cannot represent constructs that appear in the projects source code. This risk is hard to identify and predict; it will show up during individual harvesting projects.

Possible ways of minimizing or dealing with the consequences are:

- **Provide extension points in meta-model**
  Currently each element can be extended by tagged-values, which is a light-weight, pragmatic extension mechanism. This will not suffice for all situations, but it is a start.

- **Extend GenericAST meta-model**
  Evolving the meta-model requires extensive testing in both the meta-model and tools,
because it is a heavy-weight extension mechanism. It has impact on compatibility with existing models and transformations.

The genericity (or expressiveness) of the GenericAST could be too large. It happens when a semantic construct can be represented in more than one way. If this is true it is harder to create generic transformations and analyses on GenericAST models, because there could be two semantically equal models that do not result in the same target model and/or analysis result. This risk is hard to identify and predict.

Possible ways of minimizing the consequences are:

- **Thorough meta-model review**
  For each meta-model element make sure why it exists and what can be represented with it. It should not be able to represent the same construct with any other element.

- **Definition of well-formed model guidelines**
  Identified ambiguities must be resolved by making one option ‘preferred’ and the other options ‘illegal’, which can be done with a guideline. Any model that violates a guideline is not a valid GenericAST model. It may be enforced by providing a model-checker which detects and reports violations.

### 7.8 Conclusion

The GenericAST is a generic meta-model for describing structure and behavior independent of a source language. It comes with a number of frameworks and tools to support its usage. The meta-model is not finished: evolution is foreseen as usage experience is gained in harvesting projects.

A number of requirements have been satisfied with our solution. We refer to section 5.2 at page 23 for more information on the requirements. The GenericAST supports to:

- Create initial UML representation of the source system. Tests have shown initial UML models can be generated.
  - Access all information from a MOF compliant meta-model. The GenericAST has been implemented as a MOF-repository.
  - Define a mapping between source and target constructs. A mapping can be defined by a M2M-transformation to a GenericAST model and to a target model. The GenericAST meta-model can be considered as a mapping itself too, because it is an intermediate step between source oriented and target oriented models.
  - Represent structural elements and behavioral elements. We have focused on both structural and behavioral part of the GenericAST, although the structural part has been tested more thoroughly.
  - Access the original source code either as link or as fragment. The GenericAST natively supports it by a special model element.

- Perform analysis. An analysis framework is provided for quick development of analyzers on GenericAST models.
  - Analyzers have no restrictions on the chosen target model.
  - Analysis results are available in ArcStyler and Java, which is trivial by using MOF and JMI.

Below the concerns with the non-functional requirements are described. We refer to section 5.3 at page 25 for more information about these requirements.

1. **Genericity**
   The GenericAST increases genericity by allowing transformation and analysis reuse:
it is a generic intermediate step in the harvesting process. Efficiency has been increased because the GenericAST is optional: it should only be used if it is expected to be profitable.

2. **Ease of use**
   The GenericAST decreases ease of use by including an extra meta-model in the harvesting process; it increases ease of use by allowing transformation and analysis reuse. The ease of use has been increased by providing the meta-model with frameworks and tools that support its usage.

3. **Correctness**
   The GenericAST increases the risk of incorrectness by introducing an extra step in the harvesting process. This must be monitored closely during harvesting projects.

4. **Understandable**
   The concept and rationale of the GenericAST are easy to understand: it is a generic meta-model for representing structural and behavioral information. The representation is independent of a specific source language to allow for transformation and analysis reuse.

5. **Abstract from source system specific constructs**
   Trivial by the generic nature of the GenericAST meta-model.

6. **Portable**
   A purely Java based solution has been used for implementing the GenericAST, frameworks and tools. Therefore, it is as portable as ArcStyler.

The GenericAST comes with a number of risks. Among these are large model sizes, transformation performances and too large or too small genericity. These risks need to be monitored and minimized during usage of the GenericAST.
8 Conclusion

The generic harvesting framework provides tools and a process to bridge the gap between existing source code and initial UML models. The harvesting workbench can generate source specific harvesters and data storage, which help to efficiently harvest textual source code. The GenericAST is a source language independent intermediate model, which allows building up harvesting assets that can be reused in multiple harvesting projects.

A multi-phase implementation of the harvesting process has been chosen, where each phase raises the level of abstraction, filters the data and approaches the target model. The workbench allows describing a mapping between source and target constructs in a very concise way and for ease of use manual work has been concentrated in two places:

- Grammar, in which the source code structure is described and annotated.
- M2M-transformation from generated repository to initial target models, where the relevant source constructs are mapped to semantically equivalent target constructs.

These specifications are the instructions for the workbench: it generates a specific harvester or an initial target model based on the given input data.

An initial target model could be a GenericAST model. In a GenericAST model most source specific constructs have been abstracted: the source system is now represented in a generic structure. The main benefit is transformation and analysis reuse in multiple harvesting projects. It allows investing effort in building more complex analyses and transformations, thereby preventing repetitive manual work. The GenericAST meta-model is not finished: evolution is foreseen, as usage experience is gained in harvesting projects.

The GenericAST comes with several frameworks and tools that support its usage:

- **Analysis framework**
  Supports developing analyzers for GenericAST models. It provides infrastructure and convenience methods with default behavior.

- **Annotation framework**
  Supports to externally store manual annotations in GenericAST models. These annotations can be reapplied automatically when a model is regenerated.

- **GUI**
  Provides a single point of interaction for GenericAST models. It supports the complete model life-cycle and integrates all frameworks and tools. It allows for different views on the models.

The workbench extensively supports references to the original source code:

- Grammar provides tags for including fragments of the original source code.
- Generated MOF-repository reflects the structure in the grammar; therefore it includes the tagged fragments of original source code.
- GenericAST contains a special source reference object, which can be attached to GenericAST model elements.

With the harvesting framework, we believe we can efficiently and effectively support the reverse engineering process to an MDA target context.
9 Future Work

The generic harvesting framework is not a finished toolset: many improvements are possible and preferred. In this chapter our ideas for future work on the solution are summarized. The chapter has been subdivided in a part for the workbench, the GenericAST and the total harvesting process.

9.1 Harvesting Workbench

The harvesting workbench is the part of the framework that converts existing source code to initial models, which can be GenericAST models, UML models, etc. This section describes the future work that can be done with respect to the workbench.

9.1.1 Support Island Grammars

Using island grammars enables fast and incremental harvester development, because it focuses on specifying constructs of interest in the source code structure. Integrating a GLR parser generator in the workbench is one way to support island grammars.

Another potentially interesting technique is using a so-called expression grammar parser generator, which is an alternative way of specifying source code structure. It integrates lexical and syntactical definitions and uses prioritization for ambiguity resolving [For 04], necessary when dealing with island grammars. A big advantage is that it generates linear time parsers, although it may use the complete input to resolve the AST, which has negative impact on memory requirements. An available Java implementation of such a parser generator is Rats.

Changing the parser generator has quite an impact on the first part of the workbench. It has at least the following consequences:

- Adapt the repository generator to support the new parser generator
- Develop grammar annotations to allow for
  - Named production elements
  - Meta-information
- Develop a generator for generating a MOF-populator
- Adapt standard harvester functionality

9.1.2 Grammatica Improvements

The Grammatica parser generator can be improved with new features to make it more powerful and/or usable. Integrating these new features has less impact than integrating a GLR parser generator in the workbench. Examples of new features are:

- **Improve meta-information facility**
  Support to parameterize the meta-information tags and define custom meta-information tags.

- **Allow for grammar modularization and reuse**
  Enables to reuse commonly used grammar parts.

• **Allow for pre-processing input source code**
  Support to strip off certain lines or columns of text, for which we had a business case during testing with a column-based language.

• **Support for advanced tokenizer**
  It could support tokenizer modes and more possibilities for restrictions on tokens.

• **Detection of faulty annotations in a grammars**
  A correct LL(k) grammar can produce unexpected population behavior for faulty annotations in combination with certain population strategies. The parser generator should be able to detect this, report a warning and advice how to fix it.

• **Allow for attachment of ‘ignored’ tokens**
  It allows for example to once define the notion of a comment, which can then be added to every production node in the AST without extensive specification.

### 9.1.3 Model Filtering During Parsing

More complex parse-time filtering can be implemented to enable efficient iterative harvesting. During the harvesting process elements of interest and/or elements of no interest can be identified, which can be used to apply model filtering during parsing, to filter out irrelevant information. This is required to counteract performance issues while dealing with larger input source code.

Although filtering can be done during multiple harvesting phases, here we focus on parse-time model filtering. Filtering should be done in an automated way: given a filter specification, it should be done at minimal extra manual effort.

Figure 9-1 shows an example of a harvesting process that applies filtering during parse time. Each harvesting iteration results in more specific filter information that may limit the model size. Several items in the figure have been omitted for simplicity: the first iteration may include UML models for visualization; the second iteration may include a classified GenericAST model.

![Model filtering during parsing in iterative harvesting process](image)

**Figure 9-1 Model filtering during parsing in iterative harvesting process**

### 9.2 GenericAST

The GenericAST currently has a research status: it is not fully tested or complete. Many improvements can be made which are strongly related to the risks regarding the GenericAST as described in section 7.7 at page 66; therefore we address the issues only briefly in this section.
First of all, the GenericAST should be tested on more different languages to show the
genericity of the meta-model and identify possible improvements. A very interesting test case
would be to merge different source languages into a single GenericAST model and migrate
them to a single target environment.

The implementation of the process around the GenericAST models can be improved,
especially regarding performance. For that, a detailed performance analysis should be done to
identify where performance bottlenecks are. Next, these performance bottlenecks should be
solved or minimized to efficiently deal with larger input.

Certain aspects of the GenericAST have not been tested yet. For example, the ‘Property’
package, which is intended for data flow and control flow analysis, has not been tested; and
the usage of scopes has only been marginally tested. It is interesting to find out how much and
what quality of information can be derived by analyzing GenericAST models.

Finally, more complex mappings to UML and other models can be developed to allow for
more different and more useful views on the models.

9.3 Total Process
Here we describe possible improvements of the total harvesting process, which is shown
in figure 9-2. It covers model driven harvesting, refactoring during harvesting and a lazy
harvesting process.

![Harvesting process including forward engineering](image)

**Figure 9-2 Harvesting process including forward engineering**

9.3.1 Model Driven Harvesting
The ‘user interface’ for the harvesting framework could be improved, because currently a
textual command line is used to control the first phases of the harvesting process. In later
phases ArcStyler is used to run the M2M-transformations, which is already an improvement
to the purely textual interface of the first phases.

A harvesting modeling style could be developed in ArcStyler. A modeling style defines the
semantics of a complying model. It should enable to model specific harvesting projects and
let ArcStyler generate harvesting assets. Using the ArcStyler build-tool, the process can be
executed and the generated harvesters can be controlled. We could call it Model Driven
Harvesting or MDH, which is the harvesting framework supported by ArcStyler’s MDA
capabilities.

More complex harvesting projects can be managed with MDH, because it provides a
convenient high-level overview that can also be used for communicative purposes. Figure 9-3
shows an example of a fictional harvesting project overview.
9.3.2 Refactoring Support

Refactoring is the step in the reengineering process where the systems architecture and design are improved. It should result in systems that are easier to maintain because they are implemented less complex. Refactorings can be applied in various reengineering phases, but the sooner we refactor the more phases benefit from it. However, the later we refactor the more generic tools can used to help in the refactoring process. The list below shows when refactoring could be done.

- **Existing source code**
  Refactor if adequate tools and staff are available. The transformation processes that will be applied during harvesting must be known, to indicate what consequences the refactorings have in the target models.

- **Generated MOF-repository model**
  Do not refactor here because these models are hard to edit: there are no specific viewing or editing tools available. It lacks the richer semantics which are introduced while transforming to the GenericAST model.

- **GenericAST model**
  Refactor here if there are adequate tools. It would be good to refactor here because it will increase the value of the target models, as long as whole model can be updated such that it remains consistent, including behavior. The annotation framework could be adapted towards a refactoring framework for saving refactoring information.

- **UML model**
  There is a tool for modifying UML models, but it may prove hard to keep the behavioral information consistent. The more is refactored here, the less information from the GenericAST models can be used while generating target source code, unless a feedback mechanism is provided to keep them consistent.

- **Target code**
  Refactoring could be done here, for example using dedicated refactoring tools. The UML models must be kept consistent, however, because the UML models are the basis for the application. According to the MDA philosophy the target source code is mostly a derivate of the models.

The most realistic places to refactor are the GenericAST models, the UML-models or both. Each option has its pros and cons, which are briefly listed in table 9-1.
Table 9-1 Refactoring comparison

<table>
<thead>
<tr>
<th>Refactoring</th>
<th>Advantage</th>
<th>Disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>GenericAST</td>
<td>Right time (increases UML model value)</td>
<td>No available tools</td>
</tr>
<tr>
<td></td>
<td>Full representation (behavioral consistency)</td>
<td>Need a suitable refactoring framework for regeneration capability</td>
</tr>
<tr>
<td>UML</td>
<td>Right level of abstraction (structural, architectural)</td>
<td>Decreases GenericAST model value</td>
</tr>
<tr>
<td></td>
<td>Available tools (ArcStyler)</td>
<td>Need a suitable refactoring framework for regeneration capability</td>
</tr>
</tbody>
</table>

Model regeneration capability can be supported by using a refactoring framework similar to the GenericAST annotation framework. Single refactorings can be seen as simple transformations, such as rename, move, add and remove; they can be stored and reused in subsequent iterations of the harvesting process.

Multiple harvesting iterations with refactorings are possible under the following conditions:

1. All applied refactorings can be described in an adequate transformation for the selected harvesting phase. It must be possible to apply these transformations on newly generated models and it must be known which transformations lead to the desired changes in the target models.

2. The refactoring must keep the resulting models in a consistent state, including behavior which may be affected by the refactoring. It may also affect different models that refer to the refactored model.

Figure 9-4 and figure 9-5 show two variants of the concept, which is a standard harvesting process with refactoring on UML models with a feedback mechanism to earlier harvesting phases.

Figure 9-4 Applying refactoring during initial GenericAST model generation

Figure 9-5 Applying refactoring after initial GenericAST model generation

We prefer the variant in figure 9-5 because it allows the first part of the process to remain stable. The whole process remains repeatable with the addition of one model-processor that
handles all refactorings on the GenericAST level. Specific harvesting projects may require specific refactoring capabilities; for that, they may use the generic refactoring functionality if applicable as a basis.

### 9.3.3 Lazy Harvesting

Ideally the harvesting process is completely demand-based: nothing is actually harvested until it is requested. This is ideal because any amount of source code can be harvested to any amount of detail, while keeping performance very high. It is a pull-based approach.

Of course costly operations remain costly, but they will not be executed unless it is required. This is in contrast with a big-bang approach of pushing everything through the harvesting process, not knowing if it will ever be used at all. Careful planning in advance (what to harvest, to what target to harvest) is necessary to optimize a push-based harvesting approach.

In the current architecture of the harvesting process the lazy approach is not possible, because it is completely push-based: from phase \( n \) results are pushed into phase \( n+1 \). Currently the level of detail can be specified at three different transformations (grammar, M2M-transformation to a GenericAST model, M2M-transformation to a UML model) but it holds for the complete input.

In the envisioned architecture, the GenericAST is populated on demand. It results in a GenericAST model in which, some source constructs are fully specified and represented, some are partially specified and represented, and some are not represented at all.

Lazy harvesting requires a very coherent set of tools, because they need to cooperate closely and ‘know’ where to find the requested information. Otherwise, it is still required to process all input to find the requested information. This approach violates the workbench architecture that has been defined for this project, because in the workbench architecture different tools are loosely coupled and can easily be replaced by other (similar) tools.

Figure 9-6 shows a fictional pull architecture with current harvesting components. We start on the right side in the figure: using ArcStyler an action is performed on an initial UML model, which is an initial view of a minimal set of model elements from the existing sources. This action will lead to a series of pull-queries to phases closer to the existing sources; and eventually the required information will flow from left to right, where it will end up in the result of our action.

![Figure 9-6 Lazy harvesting process](image)

**Table 9-2 Lazy harvesting compared to traditional harvesting**

<table>
<thead>
<tr>
<th></th>
<th>Push architecture</th>
<th>Pull architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>Left to right</td>
<td>Right to left</td>
</tr>
<tr>
<td>Data</td>
<td>Left to right</td>
<td>Left to right</td>
</tr>
<tr>
<td>Initiative</td>
<td>Harvesting process</td>
<td>User</td>
</tr>
</tbody>
</table>
CASE STUDY
10 INTRODUCTION

This part covers the work done at De Amersfoortse Verzekeringen in Amersfoort, The Netherlands. The structure is roughly the same as the part on the generic harvesting framework. First we give a problem definition that sets the stage and defines the goals. Then, a problem analysis follows, which takes a closer look at sub-problems that may be encountered when solving the problem in the case study. Next the results of our solution are discussed, followed by the conclusions and future work discussion.

10.1 Research Problem

De Amersfoortse Verzekeringen has identified that it has several existing software systems, which are at the end of their lifecycle. Combined with the necessity to reduce the heterogeneity of the current IT systems, options for migration projects are gaining interest. The generic harvesting workbench may prove to be helpful in this kind of projects.

To assess the applicability of the generic harvesting framework in a legacy migration project, a case study has been conducted on a production system at De Amersfoortse Verzekeringen. The main question to be answered was: “In what way is the generic harvesting framework able to assist migration projects at De Amersfoortse Verzekeringen?” The case study had the following goals:

- Assess the feasibility of harvesting a large application using the generic harvesting framework
- Assess the usability of the harvested models in the MDA-tool ArcStyler
  - For documentation purposes
  - For forward engineering using MDA in the Fortis Software Development Organization

The first goal is a first full-scale real-world test of the generic harvesting framework. It needs to prove that it can handle large amounts of input, with a reasonable performance both in time and resources. It tests both the harvesting workbench and the GenericAST.

The second goal is to assess the quality of the harvested models, which may be used for documentation and reengineering purposes. The models must be complete (e.g. may not have omissions of information) and true (e.g. must not contain facts that do not exist in the source code nor can be derived from it), and should be answering the questions that have been set in the beginning.

To be able to use the models for code generation in ArcStyler, they must comply with a certain modeling style. A modeling style defines what subset of UML models is valid for use by the code generator, as shown in figure 10-1. There is no gap if the harvested models completely comply with the modeling style. The size of a gap is determined by the effort to transform the harvested models to models that comply with the modeling style, either manually and/or automatically. This gap should be assessed, especially the effort to bridge it.

A similar gap may exist for the code that is generated and the code that the Fortis Software Development Organization can use in their development process. This gap should also be assessed, including how to bridge it. Figure 10-2 shows the two gaps in the process of harvesting, code generation and the Fortis Software Development Organization.
10.2 Approach

The system used as input for harvesting is called Haebop, which will be discussed in more detail in chapter 11. It supports the offering process for collective insurances, for which the system is of mission critical importance. The system has been developed in Oracle PL/SQL by an external IT service provider, VDA. It consists of approximately 178 KLOC\textsuperscript{33}.

The workbench assessment has been done on a proof-of-concept basis, which means experience is tracked as the project is carried out. The steps that have been taken fit nicely in the Symphony software architecture reconstruction process [DHK 04] which describes a high-level, problem-driven software architecture recovery process. In the case study, first the problem has been made clear: what problems are we trying to solve with harvesting? To solve the problem, required information and information that can reasonably be harvested from the source code have been identified. The harvesting process itself consisted of multiple iterations of information extraction, knowledge inference and information interpretation.

To construct the harvesters for the case study, the PL/SQL harvesters which have been used for testing the workbench, have been extended. The M2M-transformations have been adapted and in several iterations the complete input source code has been harvested, from PL/SQL to GenericAST models, and then to UML models. Analyses have been carried out for both GenericAST and UML models, for example complexity measurements and dependency analyses.

Usability assessment of the harvested models has been done in cooperation with an independent third party to make the assessment more reliable. The Dutch MDA-authority Jos Warmer, from Ordina\textsuperscript{34}, has given valuable help in the process of adapting the harvested models.

---

\textsuperscript{33} 1 KLOC = 1 kilo lines of code = 1.000 lines of code

\textsuperscript{34} A Dutch/Belgian service provider on IT and management consulting: http://www.ordina.nl
models to conform to the Fortis Software Development Organization guidelines. Both the harvesting process and the harvested models have been assessed, as well as the gaps from figure 10-2. Assessment of generated code gap has been formally done in another project, parallel to this case study. It was not yet completed during writing of this thesis.

The initial plan was to use the harvested models to partly reengineer Haebop. This appeared to be unfeasible in the given time frame, although target code has been generated and an implementation strategy has been developed. This activity has also been transferred to another project.

10.3 Why Haebop?
The main reasons for choosing Haebop as test system for harvesting were the following:

- **Large business value**
  Haebop has large business value for De Amersfoortse Verzekeringen. It has been built and is maintained by an external party, VDA. De Amersfoortse Verzekeringen wants to internalize knowledge about the system to reduce the risk of an external dependency.

- **Relatively few interfaces with other systems**
  Haebop has only few interfaces with other systems, which makes the system relatively easy to reengineer.

- **Clear system boundary**
  Haebop has a clear system boundary, which means it is easy to identify an element as part of Haebop. For other mainframe applications that could have been used, this identification was expected to be more complex and arbitrary.

- **Modular structure**
  Haebop consists of one base module and approximately ten independent product modules. This allows for a scalable harvesting approach: a small, easy module can be started with and while time allows, larger and more complex modules can be harvested.

- **Relatively easy development environment**
  The Oracle environment (including the PL/SQL language) is a relatively easy environment compared to other environments, such as Adabas/Natural, which is also used at De Amersfoortse Verzekeringen mainframe. This makes harvesting easier.

- **Available developers**
  Haebop is relatively young: development started only in 1998, which means key developers are still available for information exchange and feedback.

- **Available documentation**
  Haebop has been documented functional and technical, which gives a good insight in the system before harvesting. Of course, the claims in the documentation should be verified with the results after harvesting.

- **Good programming style has been used**
  Specific coding guidelines and naming conventions have been used during Haebop development. This makes the system easier to harvest and understand; for example, the name of a table refers to the module it is part of.

In short, Haebop allows us to really focus on the harvesting and reengineering process, instead of dealing of all kinds of difficulties found in other systems and environments. It makes Haebop an almost ‘ideal’ candidate, which is just good for this project. We have no intention to show that ‘the most difficult system in the world’ can be harvested; instead we want to test the generic harvesting framework in combination with an MDA reengineering project.
11 **HAEBOP**

This chapter discusses the system that was used as input for harvesting in the case study. It provides information about the business process it supports, the anticipated architecture, the environment and the input used for harvesting. The information has been distilled from several sources, both documentation and stakeholders, which is listed in appendix E at page 148.

### 11.1 Supported Business Process

Haebop supports a business process at De Amersfoortse Verzekeringen on collective insurances. De Amersfoortse Verzekeringen has a product called ‘Amersfoortse Personeels Plan’ (Amersfoortse Employee Benefits Plan), short APP, which is an integrated solution for insurances related to employee benefits. A high level view of the APP offering process is shown in figure 11-1. Haebop is an acronym for ‘Het Amersfoortse Employee Benefits Offerte Pakket’, translated into English: ‘The Amersfoortse Employee Benefits Offering System’.

![Figure 11-1 APP offering process](image)

A client always communicates with an intermediary insurance expert, denoted as ‘intermediary’. This intermediary collects information from the client and transfers it to De Amersfoortse Verzekeringen. De Amersfoortse Verzekeringen calculates an offer and returns it to the intermediary, who asks the client if he agrees. The client can either reject (and may request another offer) or commit the offer. In case of acceptance the client issues a request for insurance via the intermediary, who transfers the request to the insurance administration of De Amersfoortse Verzekeringen.

The offering part of the business process is handled by Haebop.

### 11.2 Haebop and its Environment

Haebop operates in a moderately complex environment: it has interactions with six other systems, which is shown in figure 11-2.
The process begins when an offer request enters ‘Process message’, which loads the contents of the request into the so-called ‘Voorportaal’ (buffer table). Voorportaal is a temporary storage location for offer requests. Haebop users at De Amersfoortse Verzekeringen check offer requests in the Voorportaal for completeness and correctness through a User Interface. When they decide the request is complete and correct, it is loaded into Haebop. There an offer is calculated based on the request. The offers can be loaded by the mainframe, which directly accesses the Haebop database. Offers are periodically loaded into a Data warehouse system for analysis. When errors occur at the ‘Process Message’ or ‘Voorportaal’, they are logged into a logging system. Haebop users can detect these errors and take appropriate actions.

Haebop has interactions with the following systems:

1. **Voorportaal (i.e. buffer table)**
   Voorportaal collects requests for offer from the intermediary, which are temporarily stored until inspected by Haebop users. Correct and complete requests are transferred from Voorportaal into Haebop.

2. **User Interface**
   Users of Haebop can check the requests in the Voorportaal, approve calculated offers and manage the system (inspect/change product parameters) through a proprietary user interface built by VDA.

3. **Data Warehouse**
   Data warehouse analyzes offers for business intelligence purposes (for example to answer questions like: ‘How many offers where accepted by the client and ‘migrated’ to an insurance policy?’).

4. **Process Offer via DocHandling**
   An approved, calculated offer is transferred to the DocHandling system for post-processing, in order to be sent to the clients.

5. **Local printing**
   Calculated offers can be printed locally. This functionality is hardly used anymore, because all post-processing is handled by the DocHandling system.

6. **Mainframe**
   The mainframe accesses Haebop database tables directly to retrieve information on calculated offers.

These interactions will not be taken into account in the case study, because we do not aim at completely reengineering Haebop. They should be taken into account in a real migration project, because then the new system must finally replace the old one. The interactions should.
be adapted or removed before the business process is optimally supported again: there cannot be open ends left.

11.3 Architecture

The anticipated architecture of Haebop has been determined before commencing harvesting. It may give suggestions for a specific harvesting approach; and the anticipated architecture can be verified from harvesting results. Haebop is structured as a set of independent modules which each represents an insurance product. Each module depends on a central module that manages data and behavior common to all other modules. The anticipated architecture is visualized in figure 11-3, which shows the core module (‘Aanvraag Variant’, English: request variant) in the middle and the other modules around it (four are shown, in reality more than ten exist). The arrows indicate dependencies between the different modules.

![Haebop anticipated architecture](image)

**Figure 11-3 Haebop anticipated architecture**

This information has been used in the harvesting process to subdivide the harvested model elements into different sections, corresponding to the modules. It immediately introduces the first question to be answered by harvesting: “Is this anticipated architecture truly represented in the source code?” Once the harvested model elements are placed into respective modules, inter-modular dependencies can be calculated and should result in a figure similar to figure 11-3. Any other dependencies indicate an architectural mismatch, which should either be explained or refactored.

11.4 Harvesting Information

This section discusses pre-existing knowledge about Haebop that has been used during harvesting. The more information of this kind available, the higher quality the initial harvested models have, because the harvesting process can be refined according to this information.

Haebop has been developed as an Oracle application, which uses the standard functions of the Oracle9i database environment, including the language PL/SQL. PL/SQL is Oracle’s extension to SQL, which allows for procedural language constructs, such as loops, conditional statements and procedures mixed with plain SQL.

---


11.4.1 Constructs in Haebop

Each module of Haebop consists of several constructs that are suitable for harvesting. These constructs are listed below, grouped by data structure and behavior. For each construct the source code size in KLOC is shown; in total Haebop consists of approximately 178 KLOC.

**Data structure**

- **Table definitions (47 KLOC)**
  This defines the persistent data storage, as specified in plain SQL. According to the anticipated architecture there are approximately 60 tables. Harvesting table definitions gives insight in the data structure of Haebop.

- **Constraints (23 KLOC)**
  This defines constraints over tables, such as required fields. It gives information on extra restrictions in the data structure.

- **Triggers (28 KLOC)**
  This specifies actions just before or just after an insert, update, deletion of a record in a table. The action can be a check (for specific constraints that could not be defined directly on the table) or an action on another table (for example, keeping a change log of a table).

**Behavior**

- **Packages (73 KLOC)**
  A PL/SQL package contains a collection of related procedures and functions. Usually there is one package per module; there is one module however that has five packages (due to the complexity and size of the insurance product), according to the anticipated architecture.

- **Stored Procedures/Functions (7 KLOC)**
  Procedures are a collection of statements that perform a certain action. A procedure may not return a value, whereas a function may return a value. Both may have parameters that may either be input, output or both.

11.4.2 Haebop Naming Conventions

This section describes a selection of relevant naming conventions that have been used in Haebop. Names of concepts that can be harvested include information about the module they are part of. Using this information, concepts can be placed in the right module automatically. Unfortunately, two separate naming conventions have been used during development and maintenance of Haebop. The old naming convention has never been replaced with the new one, which might complicate the harvesting process.

**Tables**

Table names are prefixed with ‘T_’ and post-fixed with 2 digits representing the module. Table names are descriptive (i.e. they have sensible names). In the old naming convention table names have no prefix, but have the 2 digit post-fix.

**Fields**

Field names are prefixed with a short table alias (preferably 3 characters) and an underscore (‘_’). Field names should be descriptive (e.g. ‘abc_firstname’ instead of ‘abc_32$53’). The name of the primary key of a table is always ‘<table-alias>_SEQ’. In the old naming convention, the name of the primary key is always ‘<table-alias>_SEQNR’.

**Constraints**

Constraint names are prefixed with a specific constraint type indicator (‘PK_’ for primary key, ‘UK_’ for unique key, ‘FK_’ for foreign key, ‘CK_’ for check) followed by an optional
table name (two in the case of a foreign key constraint) and an optional field name that the constraint refers to.

**Triggers**

Trigger names are ‘TR_<table-name>’ followed by ‘B’ or ‘A’ (for ‘before’ or ‘after’), followed by ‘I’ and/or ‘U’ and/or ‘D’ (for ‘insert’, ‘update’, ‘delete’).

In the old naming convention, identical characters are used. A trigger name is ‘T’ followed by either ‘B’ or ‘A’, followed by either ‘I’, ‘U’ or ‘D’, followed by a table name.

**Packages**

Package names consist of the system name, followed by a ‘$’, followed by the package name. In the old naming convention, package names are ‘PK_<package-name>’.

**Procedures**

Stored procedure names are ‘SP_<procedure-name>’. Package procedure names are ‘PP_<procedure-name>’. Private package procedure names are ‘PPP_<procedure-name>’.

The old naming convention does not specify (private) package procedure names.

**Functions**

Stored function names are ‘SF_<function-name>’. Package function names are ‘PF_<function-name>’. Private package function names are ‘PPF_<function-name>’. In the old naming convention stored function names are ‘FU_<function-name>’. The old naming convention does not specify (private) package function names.

**Parameters**

Parameter names are ‘P’ followed by either ‘I’, ‘O’ or ‘IO’ (for ‘input’, ‘output’, ‘input/output’), followed by the parameter name.
12 HARVESTER DEVELOPMENT

This chapter describes the process of developing harvesters for Haebop. Here the word ‘harvester’ denotes both the grammar and the M2M-transformation to the initial GenericAST models. Together they describe transformations from the PL/SQL source code to a GenericAST model, for which analyses and transformations to UML models are available.

12.1 Approach

The goal for harvesting was set clear to start with: try to gain insight in the structure and behavior of Haebop. Being a vague goal it allowed exploring the applicability of the current harvesting implementation: what information can be harvested from an existing system? The focus has been put on the structural part of Haebop (what modules, what dependencies) and behavioral analysis (how complex is the business logic, what dependencies).

Custom grammars for Haebop needed to be developed based on language documentation, because no working PL/SQL grammar was available. Four grammars have been developed for Haebop; one for each construct found in Haebop: table definitions (including constraints), triggers, stored procedures and packages. The grammars have been set up in a modular way, such that recurring grammar parts (such as expressions) have been developed only once, after which it could be reused, just as the corresponding M2M-transformation part. This modularization took some effort to organize and plan, but the benefits in reusability have been significant (a simple matter of copy-paste instead of manually redeveloping similar or identical constructs).

Based on the grammars, M2M-transformations were developed for transforming the harvesting results to GenericAST models. In these transformations, Haebop specific information was used to improve the quality of the initial models. Naming conventions provided information in which functional module the unit belonged; therefore the units could be grouped into a specific container (representing a certain module) automatically.

The GenericAST has been used as an intermediate model in the harvesting process. It enabled transformation reuse and merging results of several smaller harvesting efforts using ready-to-go GenericAST functionality. Harvesting Haebop in parallel was the only feasible way to harvest to the expression level, otherwise the transformations would have been too resource intensive (both in time and memory).

The source code was split up in several parts and harvested in parallel. The intermediate results were merged to finally two main GenericAST models (one with all packages, the other with tables, triggers and stored procedures). From these GenericAST models, UML models have been created. Additional dependency analysis was performed and a transformation was done to prepare the models for code generation using ArcStyler code generation functionality.

Harvester development and harvesting Haebop has been done on two computers (Intel Pentium4 3 GHz processor, 1 GB RAM) in parallel.

12.1.1 Anticipated Process

Harvesting all Haebop sources at once was unfeasible, because the M2M-transformations do not scale very well, caused by the M2M-transformation engine implementation. A suitable countermeasure is splitting up the process in multiple smaller processes and merging the results. The split occurs both in input (different harvesters for different constructs, harvesting only part of the input at once) and in output (harvesting towards multiple models, instead of

37 http://www.unix.org.ua/orelly/oracle
one big model). A harvesting process as shown in figure 12-1 has been used for harvesting Haebop.

The input has been split up according to the different constructs: for each construct a separate harvester has been developed. This kept the grammars (and thus M2M-transformations) relatively small and concentrated to a specific construct, instead of being a general PL/SQL grammar. During harvesting, even the source code has been split up into multiple parts, to be harvested in parallel. For example, the trigger source code has been split up in 7 different parts, was harvested and merged into one GenericAST model; packages have been split up in 17 different parts. The split up is plausible because harvesting behavioral information results in large M2M-transformations, which run with super-linear time performance (see section 7.7.2 at page 68 for more information on M2M-transformation performance issues). This implies harvesting two small parts separately is faster than harvesting one big part at once.

The output has been split up into a UML model for the packages and a UML model for the tables, triggers and stored procedures. The UML model for packages is based on a GenericAST model with mainly business logic on which analyses can be done; the UML model shows the structure and analysis results. The other UML model contains both data structure and a small amount of business logic (in triggers and stored procedures), to test if both data structure and business logic can be merged into one valid model.

Based on the initial UML models, which can be used for documentation, additional analyses have been performed (which is not shown in the figure for simplicity). For example, aggregate dependencies are calculated and added to the model. An aggregate dependency is a derived dependency between two model elements, which is created if a dependency between children of both model elements exists. This analysis aggregates dependencies on high-detail levels to a lower-detail level.

The initial UML models were not directly usable for code generation by ArcStyler; they had to be adapted first. Although some adaptations could have been prevented during harvesting, other adaptations were part of a necessary refactoring process to map PL/SQL constructs to the target platform. The adapted models have been successfully used for code generation using ArcStyler code generation functionality.

Figure 12-1 Anticipated harvesting process

12.1.2 Grammar Modularization

Four grammars have been developed for Haebop to keep the grammars relatively small and focused on a specific construct: tables, triggers, stored procedures and packages. However, many PL/SQL language elements occur in multiple grammars. For example, stored procedures occur in the stored procedures grammar, but also in the packages grammar,
because a package is a collection of stored procedures. The split up that has been used for harvesting Haebop is shown in figure 12-2.

The harvesting workbench does not support grammar modularization; it has been achieved by carefully copying and pasting grammar parts. The main benefit is that besides faster grammars development, related M2M-transformations can be partly reused. The M2M-transformation from generated MOF to GenericAST closely follows the structure of the generated MOF repository. Thus, once a M2M-transformation part has been defined that takes a stored procedure as input, it can be used in both the stored procedure M2M-transformation and the packages M2M-transformation.

![Figure 12-2 Grammar modularization for Haebop](image)

12.2 Experience

The modular development of the grammar (and thus of the M2M-transformations) proved useful to be able to reuse parts of both the grammars and the M2M-transformations.

It took approximately one working day to develop (and test) four working grammars, it took approximately four working days to develop (and test) four M2M-transformations from generated MOF repository to GenericAST.

Harvesting Haebop in parallel proved to be very useful, even the only feasible option to harvest such an amount of source code. The merge-functionality was rudimentary, but adequate, because the merged models contained exactly the right information: it was a clear merge without any cross references. This is caused by the fact that the models that are merged are independent of each other, which means there are no references in between. These could not have been resolved then, nor during harvesting, or during merging. A better M2M-transformation engine will solve many current problems regarding performance, but still more sophisticated merge functionality might be necessary to harvest very large systems.

It took approximately five working days on two computers to harvest all of Haebop to maximum detail level. After model inspection it appeared that the harvesting process needed to be fine-tuned. In total three iterations were needed before the harvested models were assessed to be adequate for further analysis and processing.

Using the GenericAST as an intermediate step proved to be a very useful decision. Although not every construct could be optimally represented, it allowed developing harvesters very quickly. It was not necessary to develop transformations for associations, call dependencies and type lookups for each harvester; that was located in the generic transformation from the GenericAST to UML. It did occur that the generic transformation needed to be adapted, for
example if there were procedures with an identical name (but with a different number of parameters). Concluding, it was not truly generic, but mostly generic for our purpose.

It took approximately one working day to optimize the generated GenericAST models, merge them and transform them to UML using the generic M2M-transformation.

Code generation based on generated UML models was possible after several small modifications of the model, such that they complied with the correct modeling style as required by the ArcStyler code generation functionality. For example, associations must have an explicit multiplicity on both ends and documentation may not contain comment delimiters (such as ‘*/’ which denotes end of comment in Java). The generated code provided no method implementations for triggers and stored procedures, because this information was not present in the UML models such that it could be used for code generation purposes.

Using behavioral information in the GenericAST proved to be difficult to use in code generation. It was out of scope to develop a code generator that uses behavioral information (either from the GenericAST model, or from the UML model) to provide method implementations. But in general, it might not be the best migration strategy to keep every line of code. Harvested models generally are platform specific models\(^{38}\), because they represent the actual implementation of a system.

Models harvested from Haebop contain database specific constructs, such as triggers. Most triggers in Haebop contain auditing functionality that keeps track of operations on the database. In a migration project, these triggers can be removed if auditing functionality is generated by MDA, which is a much better way of handling this kind of general functionality. For other triggers it must be figured out what purpose they have and how that best fits into the target environment. The process of removing platform specific implementations can be seen as abstracting from a platform specific model (PSM) to a platform independent model (PIM). Figure 12-3 illustrates this in a reengineering process (note that it does not include abstraction to a computation independent model). Abstraction to a PIM is useful if source and target platforms differ.

\[\text{Figure 12-3 Reverse and forward engineering using PSM and PIM}\]

For constraints something similar holds. Many constraints are simply not-null constraints or value-checks (e.g. invariants), which either can be put in the model as an attribute annotation or as a constraint on an attribute; for some constraints this might not be possible however.

\(^{38}\) See section 2.1 at page 10 for more information on platform specific and platform independent models and how they are used in MDA
12.2.1 Harvester Measurements

Apart from harvesting Haebop source code to generate target models, performance measurements have been done. The measurements give an indication on the performance of the harvesters (which in this case only includes parsing and population of the generated MOF repository). Each measurement has been carried out ten times to gain statistical significance. Table 12-1 shows a summary of measurements, numbers have been rounded for simplicity.

Table 12-1 Haebop harvester measurements

<table>
<thead>
<tr>
<th>Property</th>
<th>Unit</th>
<th>Table</th>
<th>Trigger</th>
<th>Stored Procedures</th>
<th>Packages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grammar size</td>
<td>Lines of EBNF</td>
<td>190</td>
<td>240</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>Source coverage</td>
<td>KLOC PL/SQL</td>
<td>70</td>
<td>28</td>
<td>7</td>
<td>73</td>
</tr>
<tr>
<td>Generated harvester/repository code</td>
<td>KLOC Java</td>
<td>73</td>
<td>112</td>
<td>159</td>
<td>122</td>
</tr>
<tr>
<td>Parse time</td>
<td>Sec</td>
<td>6</td>
<td>13</td>
<td>3</td>
<td>33</td>
</tr>
<tr>
<td>Population time</td>
<td>Sec</td>
<td>9</td>
<td>42</td>
<td>7</td>
<td>144</td>
</tr>
<tr>
<td>Compressed XMI file size</td>
<td>KB</td>
<td>2580</td>
<td>15600</td>
<td>1800</td>
<td>41000</td>
</tr>
<tr>
<td>XMI save time</td>
<td>Sec</td>
<td>117</td>
<td>827</td>
<td>103</td>
<td>2660</td>
</tr>
</tbody>
</table>

It is clear that parsing and populating a repository is much faster than saving repository contents to (compressed) XMI. When using a populated repository in a M2M-transformation to generate a GenericAST model, it is more efficient to parse and populate the repository in memory, than to separately parse, populate, save and load the generated repository to and from disk.

It is remarkable that the amount of generated Java code for a harvester and generated MOF repository can differ much for two equally sized grammars. This is possible because the complexity per line of grammar can differ, resulting in different amounts of Java code that is generated.

12.2.2 Best Practices

This section summarizes several best practices that appeared during execution of the case study.

Have a clear picture of what to harvest

It is important to have a specific question to answer or problem to solve before harvesting, and to know how to find the answer or solution: if you don’t know where you’re going, any road will take you there. In other words, constructs of interest that appear in the input must be specified, such that a harvester can recognize them. For example, if for a database system a question is ‘what is the data structure?’ then constructs of interest are table definitions, field definitions and relations between the tables.

---

39 African American proverb
Know the anticipated target models in detail

Having good knowledge on the target models that need to be harvested increases usability of the harvested models. For example, if the harvested models will be used for code generation and the code generator requires that associations have names then association should be given names during harvesting, which might not be necessary when generating models solely for documentation. This results in models that can be directly used for their purpose.

Make use of coding guidelines and naming conventions

The more system-specific information is used during harvesting, the better quality the initial models have. For example, if a table name contains a number that indicates the module it is part of, it can be used to automatically relocate the table to the right module. It improves the usability of the initial models.

Keep grammars small and focused

Smaller grammars are easier to maintain than bigger grammars. When a complete grammar is not available for a harvesting project, a minimal grammar should be developed. A minimal grammar has exactly the right information to describe the anticipated input, but not a complete language. This approach has been taken while harvesting Haebop.

Modularize grammars

Modular grammars improve reusability of commonly used grammar parts, such as statement and expression definitions. In the case study, four different grammars were composed from several grammar parts. Not only does this save work while developing grammars, it also allows reuse of several M2M-transformation parts, which correspond with the grammar parts. This has successfully done in the case study.

Design grammar towards a target meta-model

Grammars should be developed with a certain target meta-model in mind, including how to map the grammar to the target meta-model. This improves reusability of M2M-transformation parts. For example, if in a target meta-model statements are all contained by a certain container, this should be reflected in the grammar. This could be done by having a production ‘statement_container’ which contains other statements and acts the entry-point for statements in the grammar. This creates an extra node in the AST that can be mapped to the statement container model element in the target meta-model.

Avoid usage of XMI to store model contents

Persistence of harvesting results using XMI should be avoided, because it is a slow mechanism. Instead of saving and loading every time to and from XMI, as much as possible should be done without XMI. In the case study, parsing and populating a repository took in the order of seconds, while streaming to XMI took in the order of minutes. Therefore, it is more efficient to parse the input every time the models are needed for M2M-transformations. Because generation of a GenericAST model takes more time than saving and loading XMI, it should be used to save a generated GenericAST model.

Optimize M2M-transformations

M2M-transformations should be optimized everywhere possible. In the case study, M2M-transformations were the longest operations, which took in the order of days to complete. Simple optimizations could improve the performance, for example by doing calculations once, pass on the results to child rules where filtering takes place. In the case study, calculations where done at the same time as filtering, requiring each calculation to be executed multiple times instead of once.
12.2.3 Shortcomings

This section describes shortcomings that were found in the implementation of the harvesting framework.

Weak tokenizer

The current tokenizer in the harvester is not suitable for many purposes. It lacks tokenizer mode capabilities, as well as scalable case insensitive tokens. Currently, the longest feasible case insensitive token is about 20 characters long, whereas it should be as long as needed. Tokenizer modes could have been useful in several cases, for example to harvest input in the form of ‘key = value’ which requires to effectively tokenize the value.

Slow M2M-transformation engine

The current M2M-transformation engine (AIM) is not very scalable. When transforming approximately 4 KLOC PL/SQL to a high-detail GenericAST model, it takes several hours to complete, using approximately 800 MB of RAM. This requires the ability to split up the input and harvest parallel to harvest larger input.

Limited expressiveness of the GenericAST

The current GenericAST meta-model lacks several representation capabilities, such as the possibility to represent details on database IO statements and dependencies between model elements.

Limited information retrieval capability for the GenericAST

The current harvesting environment lacks the possibility to query GenericAST models other than to develop a M2M-transformation or a custom GenericAST analyzer. In general a MOF-based query environment could prove to be helpful to gather information from GenericAST models.

12.2.4 Adaptability

The Haebop harvesters were adapted for another PL/SQL application of about the same size as Haebop. The application was ‘Amesys’ which consisted of approximately 200 KLOC PL/SQL. Because it was a showcase to test the adaptability of the harvesters, only 33 KLOC of it was harvested (which were all tables, triggers and stored procedures). The grammars and transformations were copied and extended to maximize reusability of existing functionality, which took approximately two hours to adapt, including the M2M-transformations. After three days of harvesting on a single computer the target UML models had been created and presented to the ‘Amesys’ project team.

The harvested models were similar to the models of Haebop, because the same process has been followed and the same analyses have been performed.

12.2.5 Model Quality Assessment

A measure for model quality is the completeness and trueness of a model, based on harvested input. To monitor this measure statistics have been collected for each intermediate model in the harvesting process. For example, if there are 60 table definitions in the input, there should be 60 model elements in the generated UML model that represent the tables. It has been ensured that the statistics were identical for consecutive harvesting steps. An arbitrary number of model elements in the initial UML models have been manually verified for completeness and trueness in the harvested input. No peculiarities have been found.

Another measure for model quality is compliance with a UML modeling style that is used for code generation. It appeared that the harvested models were not directly usable for code

---

40 Amesys is an application of Fortis ASR, another Fortis insurance company in The Netherlands
generation: the harvested models did not comply completely with the used modeling style. Several small adaptations were necessary, which could all be automated. For example, associations must have an explicit multiplicity on both ends and documentation may not contain comment delimiters (such as `*/` which denotes end of comment in Java).

### 12.2.6 Code Quality Assessment

A measure for code quality is the number of errors during compilation. Code generated from both Haebop models contained 12 compile errors, regarding too many parameters for a method and formatting errors for attribute initialization values. These errors could easily be prevented by refactoring the models used for code generation.

Another measure for code quality in the case study is compliance to the Fortis Software Development Organization standards and guidelines. This has not yet been assessed, because it is formally part of a project parallel to this case study. It was not completed during writing of this thesis.
13 Harvesting Results

This chapter covers several harvesting results that have been generated in the case study.

13.1 System Size

The generated harvesters natively support to generate several statistics based on an AST. From these statistics, the size of a system can be indicated. Table 12-1 shows some figures for Haebop.

Table 13-1 Haebop system size

<table>
<thead>
<tr>
<th>Element Count</th>
<th>Harvester</th>
</tr>
</thead>
<tbody>
<tr>
<td>Construct</td>
<td>Tables</td>
</tr>
<tr>
<td>Tables</td>
<td>163</td>
</tr>
<tr>
<td>Fields</td>
<td>4,052</td>
</tr>
<tr>
<td>Constraints</td>
<td>4,160</td>
</tr>
<tr>
<td>Triggers</td>
<td>468</td>
</tr>
<tr>
<td>Packages</td>
<td></td>
</tr>
<tr>
<td>Procedures</td>
<td></td>
</tr>
<tr>
<td>Statements</td>
<td>6,977</td>
</tr>
<tr>
<td>Expressions</td>
<td>4,088</td>
</tr>
<tr>
<td>Database statements</td>
<td>813</td>
</tr>
</tbody>
</table>

The number of constraints is considerable, as is the number of triggers, especially when migrating this application to an environment that does not natively support triggers and constraints (representing these constructs in UML is already non-trivial).

Note that according to this measurement, there are more than 60 tables in Haebop, as would have been expected based on the anticipated architecture. In fact, there are 163 tables that have been harvested. This shows that expectations may not always be reflected in reality: the source code contained the only true answer to the question ‘how many tables does the system contain?’

13.2 System Complexity

Complexity analysis has been performed on all Haebop package procedures. For this a GenericAST analyzer was developed that calculates McCabe’s Cyclomatic Complexity Number (CCN) for a given procedure, which measures the number of potential branches (or choices) in a procedure [Sei 00]. A high value indicates a complex procedure, which is potentially error-prone and badly maintainable. Table 13-2 shows the complexity numbers that were found for Haebop. Note that the table shows complexities for each package procedure including inner-procedures; that is why the total number of package procedures is larger than mentioned in table 12-1.
Table 13-2 Haebop package procedure complexities

<table>
<thead>
<tr>
<th>McCabe’s CCN</th>
<th>Number of statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 – 99</td>
<td>100 – 199</td>
</tr>
<tr>
<td>100 – 199</td>
<td>200 – 299</td>
</tr>
<tr>
<td>200 – 299</td>
<td>300 – 399</td>
</tr>
<tr>
<td>300 – 399</td>
<td>400 – 499</td>
</tr>
<tr>
<td>...</td>
<td>700 – 799</td>
</tr>
<tr>
<td>1 – 10</td>
<td>455</td>
</tr>
<tr>
<td>11 – 20</td>
<td>67</td>
</tr>
<tr>
<td>21 – 30</td>
<td>10</td>
</tr>
<tr>
<td>31 – 40</td>
<td>1</td>
</tr>
<tr>
<td>41 – 50</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>71 – 80</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>171 – 180</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>534</td>
</tr>
</tbody>
</table>

As mentioned before, a high value indicates a complex procedure which is hard to maintain and also hard to migrate. Applying a categorization as shown in table 13-3 (taken from [Sei 00]), we see that only a small part of Haebop package procedures is qualified as high risk (only 6 %). This part requires extra attention when migrating Haebop and should be considered for refactoring.

Table 13-3 Haebop complexity overview

<table>
<thead>
<tr>
<th>McCabe’s CCN</th>
<th>Risk Evaluation</th>
<th>Haebop Procedure Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 – 10</td>
<td>Simple procedure without much risk</td>
<td>465</td>
</tr>
<tr>
<td>11 – 20</td>
<td>More complex, moderate risk</td>
<td>76</td>
</tr>
<tr>
<td>21 – 50</td>
<td>Complex, high risk procedure</td>
<td>30</td>
</tr>
<tr>
<td>&gt;50</td>
<td>Untestable procedure, very high risk</td>
<td>1</td>
</tr>
</tbody>
</table>

13.3 Structural Overview

The generated UML models for Haebop give insight in the structure of the application. Diagrams of the model can show tables, relations between tables and triggers on tables.

Figure 13-1 shows a diagram of the data structure of a certain Haebop module. It has four tables with two associations. All data fields have been hidden in order to give a high-level overview of the data structure.
Figure 13-1 Haebop data structure example (in UML)

Figure 13-2 shows a similar diagram, only with extra details for the table 'premie_percentage_09'. It shows the data fields (e.g. 'pr9_beroepsklasse'), the constraints (e.g. 'ck_pr9_update_teller') and triggers (e.g. 'TR_PREMIE_PERCENTAGE_09_AIUD') that are associated with the table. The model contains all information; the diagram is a view on model in which the level of details can be chosen at will.

Figure 13-2 Haebop data structure example with more detail (in UML)

Figure 13-3 shows a diagram of the package model with PL/SQL packages with stored procedures. This kind of diagrams gives insight in the structure of the behavior of Haebop. A PL/SQL package with stored procedures is mapped to a UML class with methods.

Figure 13-3 Haebop behavior structure example (in UML)

To verify Haebop with its anticipated architecture harvested dependencies have been aggregated to a UML package level. In this process a dependency between two UML
packages is created, if a dependency exists between children of both packages. The Haebop module overview with aggregated data dependencies is shown in figure 13-4. Each dependency has a name with ‘dd’ for derived dependency and a number, which indicates the weight of the dependency. For example, if a derived dependency is caused by three dependencies it is given the name ‘dd_3’. The documentation of the dependency contains information about which dependencies caused it (showing for each dependency the name, the source and target model element).

![Figure 13-4 Haebop data structure module overview (in UML)](image)

It is clear that the anticipated architecture is violated in Haebop. More detailed analysis has shown that this is caused by only few tables which contain data that is used by multiple products. Refactoring might help to remove this unexpected dependency, for example by moving such tables to the common module.

13.4 Behavioral Overview

The generated UML models for Haebop give some insight in the behavior of the application. Because UML 1.4 is not suited for representing behavioral information directly, only derived information is shown in diagrams. For example the database operations performed by stored procedures, the direct call dependencies between stored procedures, and the functional dependencies of packages and modules based on direct call dependencies.

Figure 13-5 shows two Haebop PL/SQL packages with stored procedures and the database operations the procedures perform. In the figure, only the database ‘update’ and ‘read’ operations are shown (modeled as a UML class) for simplicity. Every dependency indicates a specific operation on the database; its documentation contains the exact statement as it appears in the original source code. Figure 13-5 shows for example that procedure ‘pp_converteer’ in PL/SQL package ‘conv$verzuim’ may perform an update on the database.
Figure 13-5 Haebop database operations example (in UML)

Figure 13-6 shows three PL/SQL packages with direct call dependencies (a dependency for each method call from caller to callee). We can see that procedure ‘P_ANW’ in PL/SQL package ‘PK_EURO_SAMENVATTING’ may call procedure ‘P_SET_CONTROLE’ in PL/SQL package ‘PK_DUPL_CTRL’. In PL/SQL package ‘PK_AANVRAAG’ internal package dependencies are also shown (e.g. from ‘P_INDICATIE_APP_UIT’ to ‘P_HERBEREKENEN_AAN’).

Figure 13-6 Haebop direct call dependencies example (in UML)

Direct call dependencies may not give best insight in behavior, because it is hard to answer questions like ‘when calling this operation, what will happen before the call is returned?’ From direct call dependency information UML collaboration diagrams can be generated, that show for a given method call, what other methods can be called and in what order. It provides a more detailed view what other classes and methods are needed for executing a method, including indirect call dependencies (e.g. method A calls method B, and method B calls method C; thus method A indirectly depends on method C). Note that there is no information about how many times a method will be invoked, because it might be in a conditional or a looping invocation. Currently the order in a sequence of method invocations is arbitrary, because the direct call dependencies have not been harvested with a specific order. The order in the call stack (thus A calls B calls C) is correct, on the other hand. It has been decided that reharvesting with explicit order is simple but takes much time, therefore the effort of reharvesting is not worth the gain in the collaboration diagrams for the case study.

Figure 13-7 shows a simple example of a generated collaboration diagram. A box denotes a class, a line denotes the usage of a method (there is one line for each unique method), and an arrow with text (a number and a method name) denotes a method invocation. The number of the arrow shows the order: the method invocation with the lowest number is done first. If a method invocation is caused by another method invocation, it is shown by a ‘.’ and a number. It is similar to a call stack: ‘0.2’ means ‘the third method invoked in the first method invoked’.
The element ‘Dummy’ initiates the call to the class that owns the method (indicated by an arrow starting with ‘0’ and the method name). The called method may again call other methods, for example in another class (indicated by the arrow starting with ‘0.2’ because it follows after call ‘0’). Next it calls an internal method (indicated by the arrow starting with ‘0.4’). Calls to global PL/SQL functions (such as mathematical functions) have been hidden from the diagram for simplicity.

Figure 13-7 Haebop call dependencies as collaborations – simple (in UML)

Part of a more complex collaboration diagram is shown in figure 13-8, which shows all method calls for calculating a pension fee, more than 1000 in total. Calls to global PL/SQL methods are hidden for simplicity, although the diagram is still very complex and provides the insight that this calculation is complex.

A static diagram of this complexity might not give an optimal insight in the behavior because there is still too much information on it. An option for solving this is for example to provide a higher-level view by extracting parts of the diagram that can be used as an atomic operation (internally it may call other methods which are not shown in the diagram). Animating the diagram might also help, simply by incrementally adding all methods to the diagram to show the method invocation sequence.
To verify Haebop with its anticipated architecture existing dependencies have been aggregated to a UML package level, similar to the structural analysis of Haebop; this is shown in figure 13-9.

Figure 13-9 Haebop behavior module overview (in UML)

It is clear that there are more dependencies between modules than could be expected based on the anticipated architecture. It also appears to be more chaotic than the data structure dependencies.
In the case study the feasibility of using the generic harvesting framework for harvesting a large application has been assessed. The application was a 178 KLOC PL/SQL production application with high business value for De Amersfoortse Verzekeringen, called Haebop.

The generic harvesting framework was used to develop four harvesters (one for each main construct) and four M2M-transformations to a GenericAST model. To counter scalability issues with the current M2M-transformation engine implementation, the input has been split up in several smaller parts to be harvested in parallel, which was a successful approach. The resulting GenericAST models have been merged and finally transformed to a UML model, giving insight in the structure of Haebop. A single iteration for harvesting Haebop took about five working days on two desktop computers.

The harvested models give insight in the structure of Haebop, for example in existing data and/or behavioral dependencies. With aggregated dependencies, the models allow views on the system on various levels of abstraction: application view with modules, module view with tables and packages, table view with fields, or package view with stored procedures. These views can provide a valuable base for documentation of the current application implementation. UML collaboration diagrams give insight in the total sequence of method calls for a given method call, derived from direct call dependencies.

Code has been successfully generated from harvested models using standard MDA functionality provided by ArcStyler, after several small, automated modifications. Quality of the generated code has not yet been assessed with respect to the Fortis Software Development Organization. This is an activity that is formally part of another project which has branched from this case study.

The Haebop harvesters have proven to be adaptable to other PL/SQL systems with limited effort. For a showcase the Haebop harvesters were adapted to harvest Amesys, another PL/SQL application used at Fortis, within hours. The initial harvesting results provided the Amesys project team with new insight in the structure of their application.

14.1 Workbench Evaluation

The generic harvesting framework has proved to be suitable for harvesting large applications, provided that the source code structure can be described in a grammar. The most time intensive operation is a M2M-transformation (of which many are needed in a harvesting project) which can be attributed to the current implementation of the M2M-transformation engine AIM.

The way grammars are written proved to be very useful during the case study: within a day four working grammars for 178 KLOC PL/SQL have been developed and tested. The grammars used many utility features, such as case insensitive tokens, implicit token declarations and production element naming. These features resulted in relatively compact grammars and informative repositories with meaningful names, which made M2M-transformation development easier.

Performance of the workbench is very high: harvesters are generated within minutes (including tool startup time). Harvester generation time is negligible compared to the runtime of generated harvesters. Performance of generated harvesters is high: parsing 73 KLOC PL/SQL and populating a MOF repository takes approximately 3 minutes. Performance of M2M-transformations is currently low: generating a high-detail GenericAST model from 5 KLOC PL/SQL takes in the order of hours. M2M-transformations are the current bottleneck for the harvesting process, of which many are needed, unfortunately. The performance problem can be attributed to the current implementation of the M2M-transformation engine,
which is a development release and therefore not optimized for performance. An improved way of doing M2M-transformations is an essential prerequisite for large scale harvesting in acceptable time.

The GenericAST proved to be a valuable step in the harvesting process, although it was not able to represent every concept that would have been useful in the case study. Reusability of analyses and M2M-transformations from GenericAST to UML has proved to be feasible and this saved much time in the case study.

14.2 Recommendation

During the case study it appeared that probably the most important thing for a harvesting project is having a (very) specific question or problem to solve. The more specific the question, the more effective can be harvested and the more useful the resulting models are. Without a specific question, harvesting will not provide the answer and is in that case a waste of resources.

When migrating a database-oriented application to another environment (for example, J2EE) it is important to have a clear plan of how to map database specific constructs (such as constraints and triggers) to the new environment, which may support these constructs natively. In the example of PL/SQL to J2EE, most constraints can be specified in the model, but some may not. Then it must be decided how these constraints are implemented in J2EE and how it is made sure that the constraints always hold.

In an MDA-based migration project it makes sense to reason about application semantics and to provide non-trivial mappings between the source and target environment. For example in the case study, it appeared that many triggers are used for auditing functionality. Since auditing should be an application or even company-wide decision, it should be considered as something that holds for every persistent storage element. Thus, using the MDA philosophy, a transformation should be developed that automatically generates auditing functionality for each persistent data storage element. This implies for the migration project that the auditing triggers can safely be removed from the model, because they are now located in a transformation.
15 Future Work

This case study has several issues that are still left open and can (or should) be solved in a later project.

In the case study harvested models have not been used to generate (or manually compose) system documentation. The harvested models could be displayed in diagrams, which can put into static documentation (e.g. a paper document). Although the models are the main documentation, a static version could be convenient because it does not require a tool to be accessed.

To improve usability of generated UML collaboration diagrams for insight in the behavior of Haebop, Haebop should be harvested such that call dependencies have an explicit order. Complex collaboration diagrams should be split up in different parts to simplify usage. The diagrams could be enriched with information about database operations, such as reading from or updating a certain database table, giving a more detailed view of the behavior. The ability to animate diagrams could help to give insight in complex call sequences by selectively and incrementally showing information. Other UML diagrams could provide new insights, for example sequence diagrams to show the behavior of procedures.

No part of Haebop has actually been implemented in a J2EE environment, due to timing constraints in the case study. This is an important showcase to show that the approach for migrating a database-oriented application to a new environment (e.g. J2EE) works using MDA-based reengineering. For this a suitable mapping from typical database constructs (e.g. triggers) must be made to the new environment, such that all functionality can be migrated. Next to a showcase, assessment of the quality of generated code with respect to the Fortis Software Development Organization is an important activity at Fortis. It will indicate the usability of MDA-based harvesting in a migration project at Fortis, as well as the potential of MDA-based forward engineering in general. This project has already been initiated during the case study.

Behavioral information represented in a GenericAST model has not yet been used for MDA code generation purposes. This is an interesting possibility that should be investigated, because it might increase the migration process automation. It should be investigated what part of implementation logic can be used for code generation and how much code can be generated from that. In an ideal world, the generated code works perfectly without manual alterations; in practice it might occur that the code needs manual improvement (either syntactically or semantically). This too should be investigated.

An interesting development is the strategic merge of OMG and BPMI.org regarding their business process management activities. Both are non-profit standardization organizations supported by a broad industry. It enables support for a standardized business process modeling language in MDA, which can be used to represent harvested business processes as implemented by existing systems. It would be interesting to assess the feasibility of harvesting business processes from existing systems to such models.

16 RELATED WORK

This chapter gives a brief (not necessarily complete) overview of other research and development activities in the area of reengineering. It has been subdivided into an industry part and an academics part, because both are active in this area.

16.1 In Industry

Many companies are working on reengineering degraded software, some of which have grouped into the ADM taskforce of the OMG. The taskforce aims at providing open standards for the subject matter. We discuss it below and compare it to our solution because both deal with similar concepts. A remarkable initiative from industry using artificial intelligence is discussed in section 16.1.2.

16.1.1 ADM – Architecture Driven Modernization

In 2003 the OMG started the Architecture Driven Modernization Task Force\textsuperscript{42}, which aims at extending MDA practices and standards to existing systems. MDA uses a top-down approach for developing new systems: from models to systems. ADM tries to work bottom-up by extracting architectural models from existing systems, which can be used in a top-down MDA development process. Briefly stated, ADM is similar to what we call harvesting.

This section describes the similarities between our concept of harvesting and the ADM proposals. It was out of scope for this project to comply with the ADM specifications because the first of seven specifications was due in June 2005, when this project was already halfway. Information has been taken from\textsuperscript{43} and specifically from [OMG 05].

The goals of ADM are:

- The ultimate goal: revitalization of existing applications
- Make existing applications more agile
- Leverage existing OMG modeling standards and the MDA initiative
- Consolidate best practices leading to successful modernization

These goals must be achieved while complying with existing OMG standards, especially MOF.

ADM consists of seven different proposals (RFP’s) targeted at seven different activity areas [OMG 04]:

1. **Knowledge Discovery (KDM)**
   - Defines a meta-model for representing systems in a generic way, but not below the procedure level.
   - Is to be adopted in June 2005.

2. **Abstract Syntax Tree (ASTM)**
   - Defines a meta-model for representing systems below the procedure level. It forms a basis for KDM such that it can represent a system completely.
   - Is to be adopted in 2006.

\textsuperscript{42} http://adm.omg.org
\textsuperscript{43} http://adm.omg.org
3. **Analysis (AP)**
   Facilitates the examination of structural meta-data to derive behavioral information. It focuses on semantics rather than syntax of the system.
   Will be initiated in 2005.

4. **Metrics (MP)**
   Facilitates the examination of structural meta-data to derive metrics.
   Will be initiated in 2006.

5. **Visualization (VP)**
   Focuses on visualizing the contents of ADM models.
   Will be initiated in 2007.

6. **Refactoring (RP)**
   Focuses on refactoring systems represented in ADM models. It aims to improve system quality without redesigning it.
   Will be initiated in 2008.

7. **Target Mapping and Transformation (TMT)**
   Focuses on the mapping of ADM models to specific targets, which include MDA.
   Will be initiated in 2009.

It is clear that the ADM task force has a time frame much longer than this project. We cannot comply with the ADM standards currently because there are none. However, the generic harvesting framework touches many subjects mentioned in the RFP’s. Table 16-1 shows the seven RFP’s and what part of our project concerns it.

**Table 16-1 ADM compared to GenericAST**

<table>
<thead>
<tr>
<th>ADM RFP</th>
<th>Generic Harvesting Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. KDM</td>
<td>GenericAST – structural part</td>
</tr>
<tr>
<td>2. ASTM</td>
<td>GenericAST – behavioral part</td>
</tr>
<tr>
<td>3. AP</td>
<td>GenericAST analysis framework facilitates</td>
</tr>
<tr>
<td>4. MP</td>
<td>GenericAST analysis framework facilitates</td>
</tr>
<tr>
<td>5. VP</td>
<td>GenericAST GUI</td>
</tr>
<tr>
<td>6. RP</td>
<td>(future work)</td>
</tr>
<tr>
<td>7. TMT</td>
<td>GenericAST target model transformation collection</td>
</tr>
</tbody>
</table>

We see that the GenericAST and surrounding tools and frameworks touch nearly all ADM RFP’s. We do not claim to provide complete solutions however. The GenericAST meta-model is an initial, pragmatic implementation of both KDM and ASTM meta-models. The analysis framework supports development of analyzers which may implement AP and MP conform functionality. The GenericAST GUI is an initial visualization application for visualizing GenericAST models. We mention refactoring in the future work section. The TMT RFP is similar to what we call the target model transformation collection.

### 16.1.2 Automated Legacy Transformation

A remarkable initiative to software system migration uses artificial intelligence to achieve a high-level of automation [ND 01]. The authors claim to be able to achieve 99% automation for migration of legacy systems and databases to an object oriented environment, using four steps for their migration:

1. Convert the legacy source code into a knowledge based abstract syntax tree
2. Generate an initial OO model
3. Refactor the OO model
4. Generate web-enabled target code in Java or C++

The approach taken is certainly interesting. Using AI for migration and automation instead of human effort could be an effective way of making legacy system migration economically more attractive, both in time and money. In their conclusions, however, they weaken their automation claim somewhat by mentioning that human effort is still valuable and required in certain areas.

Application of AI technologies is out of scope for this project; still we mentioned it as a possibility for the classification of GenericAST model elements. We also strongly believe that human assessment is necessary to improve the systems architecture and design, which regards semantics rather than syntax. Many semantics are present in comments, or in the minds of developers, maintainers and business users. This unstructured or even implicit knowledge may be hard to feed to an AI application, but can be used by a human engineer while harvesting and refactoring.

16.2 In Academics
The scientific community is very active regarding reengineering. We give a brief overview of several other research projects without comparing them to our solution, which will be done in the contributions chapter. This section has been subdivided into different topics for clarity.

16.2.1 Fundamentals
[BKV 97] show the need for generic programming language technology for reengineering. They present a classification of reengineering tools: language independent, language dependent and language parameterized (or generic). Our workbench approach would fall into the language parameterized category. They claim generic language tools are crucial to do fundamental reengineering.

[AK 04] present a framework for language neutral representation of source code. They mention the term Generic AST as part of the program representation framework. XML is the main language. This has the advantage of being a light weight solution, but comes with meta-model discovery problems. The meta-model must be hardwired into the programs that use it.

[CM 04] present a view on language support for MDA. They argue for a Generic Annotated AST to solve certain model driven development problems. They promote an explicit meta-representation for programs in an AST-like structure and the possibility for users to add their own annotations to this AST. The need for a new language (with an explicit meta-level and support for user annotations) is shown.

[BDD 05] present a paper on bridging different engineering areas, specifically that of model and ontology engineering. They discuss the topic with respect to the meta-meta-models that reside on the M3 meta-data level (cf. MOF meta-data architecture) and present mappings on that level. [Mer 04] presents a similar idea on systems engineering and software engineering.

[ZG 04] discuss the technological needs for supporting legacy system evolution in a model driven software engineering context. They show a prototype of a model driven program transformation application that supports the migration process.

[KNT 02] present how to represent procedural source code in UML models. They use UML profiles for representing aspects of procedural languages, comparable to our GenericAST approach. Given the right transformation engines, they can generate target code on a large variety of platforms. They have an example using PL/SQL as their source language.

[NM 04] show an attempt of generating a platform independent model from a platform specific model in an MDA context. It supports the reverse engineering process by providing an additional abstraction on the platform specific models, which usually result by initial reverse engineering with an MDA target context.
Our literature study (done as a preparation for this project) discusses several research papers regarding methods and techniques on extraction, abstraction and presentation for the reengineering process [Reu 05].

### 16.2.2 Architecture Recovery

[Kru 95] presents a model for representing architectures for software systems. He proposes four different views combined with overlapping scenarios. It allows the representation of both functional and non-functional requirements for a variety of stakeholders.

[HRY 95] show a framework for recovering a high-level architecture based on system source code. They use reverse engineering techniques for extracting the required information from the source code.

[BHB 99] present a case study for the extracted architecture of the Linux operating system. They started with a conceptual architecture and refined it with extracted information from the source code. They used standard Linux tools for the extraction process.

[DM 01] show an approach for application evolution by using an extracted architecture. They use an incremental approach for extracting the architecture.

[Riv 00] presents a case study on reverse architecting. Reverse architecting aims to extract software architecture models from system source code.

[HH 00] present a specialized case study for extracting system architectures. They aim to define a reference architecture for web servers based on extracted web server architectures.

### 16.2.3 Software System Migration

[DHK 04] present a view-driven process for software architecture reconstruction. They present a framework for architecture reconstruction that is problem-driven and show experience gained in different case studies.

[ACR 05] present a framework for automatic legacy system migration in MDA. They use rewriting logic as their transformation engine. The results are UML models of the legacy system.

[WSV 05] present an approach and a tool to cope with the complexity of software system migration. They decompose the system to different migration units using static and dynamic analyses. They prioritize each migration unit to decide which to migrate first in a progressive migration approach.

[AT 04] focus on the business reasons for legacy system migration. They state that technical reasons alone do not suffice for migration. They present a strategy for extracting legacy system migration requirements from business process evolution requirements.

[BCM 01] present a process for gradually reengineering a legacy system, without freezing or duplicating it. They use a component-wise migration approach therefore minimizing maintenance request delays.

[EH 05] discuss some interesting problems in the domain of legacy system migration towards a service-oriented architecture. They discuss problems of identifying, describing and modeling services in legacy source code.

[Goo 02] describes in substantial detail the process around legacy transformation. He describes the transformation process, how to develop a strategy for legacy transformation and discuss business aspects for migration.

[MNS 03] present an industrial case study for integrating legacy systems into the new systems and technologies in the MDA context. They use UML models as target models for the reverse engineering step.
[BLW 99b] present two methodologies for migrating legacy information systems. The first is the Chicken Little strategy, which is an 11-step incremental process of migration. The legacy system is gradually rebuilt in the new environment using modern development techniques. The Butterfly method assumes that while the legacy system must remain operable during migration, there is no need for cooperation between parts of the old and the new system.

[SPL 03] have devoted a whole book on a case study on migration of legacy systems. They present the methodology ‘risk managed modernization’, which is an incremental migration strategy targeted at minimizing the risks involved in the process.

16.2.4 Model Refactoring

[GSM 03] present a minimal extension to the UML meta-model such that source-consistent refactorings are possible. The extension allows MDA-tools to provide model-based refactorings while keeping the underlying source code synchronized and consistent.

[MDB 03] show the current research status and future trends in the refactoring area. They identify the challenges regarding refactorings on different levels of abstraction in an MDA context, while keeping all artifacts (models, source code) consistent.

16.2.5 Integration & Maintenance

[CC 04] present an evolution strategy for Java programs in the Eclipse Modeling Framework (EMF). The solution they present is more light-weight than a full MDA environment. It is limited to the Java language (both as source and target) imposed by EMF. EMF is a meta-meta-modeling standard very similar to MOF, used by the Eclipse community.

[Heu 04] discusses a methodology for integrating new business application with legacy systems using wrapping techniques. The proposed method is in line with the MDA/ADM philosophy. The aim is to integrate new developments with current (legacy) systems to prevent implementation of already existing functionality. For that, the legacy systems are reverse engineered into a platform independent model, which can be matched and linked to the new development models.
17 Contribution

Reengineering is a popular theme in academics and industry: numerous initiatives have been
done on creating reverse engineering tools, abstraction algorithms and presentation options.
Slowly MDA-based reengineering gets attention, most notably in the ADM taskforce of the
OMG.

Until now, not much practical effort has been put into using MDA in the reengineering
process. This is remarkable, because MDA is an attempt to raise the level of abstraction while
developing software systems. With MDA models are the main software assets, instead of
huge amounts of code with some loosely coupled documentation. It is a serious initiative to
prevent the degradation of software systems, the systems we are trying to reengineer.

We have shown in this project that it is possible to tie together different tools to build a
working MDA-based harvesting environment. This solution opens the way for MDA-based
reengineering. We can effectively harvest existing software assets into a commercial MDA-
tool and use it for MDA-based forward engineering.

We have shown we can map EBNF grammars to usable MOF meta-models. Based on a
grammar a MOF-repository is generated, which can be filled and accessed in any JMI or
MOF capable environment. This mapping was the prime prerequisite of the whole harvesting
framework.

We explored the concept of a generic intermediate model, which allows reusing
transformations and analyses in multiple harvesting projects. It even enables harvesting
systems of different sources into one representation, if required. Being an optional extra step
in the harvesting process, it makes investments in more complex transformations and analyses
economically more attractive.

In the case study the workbench has been thoroughly tested on a large production application
at De Amersfoortse Verzekeringen. We have shown that using the workbench, harvesters
could be developed quickly for the given system and that they could easily be adapted for a
similar system. The harvesters produced interesting results, though having some scalability
problems in the current M2M-transformation engine implementation.

For the analyzed system, UML models have been generated that give insight in the structure
and behavior of the system. Structural information has been shown in UML class diagrams,
behavioral information has been shown in UML class, state chart and collaboration diagrams.
The harvesting solution has no conceptual restriction on generated target models, although
currently only class, collaboration and state chart diagrams have been created.
18 Conclusion

This chapter summarizes the conclusions stated in the parts about the harvesting framework (in chapter 8 at page 74) and the case-study (in chapter 14 at page 107).

The generic harvesting framework provides a flexible toolset and a process to effectively harvest existing source code. The source code structure is described in a grammar and from the grammar a specialized harvester and repository are generated. Without additional coding the repository can be populated with relevant source constructs, which is then available in any JMI or MOF compliant environment.

The generic intermediate model allows us to efficiently develop harvester technology and experience. A collection of transformations and analyses can be built up, to support reuse in different projects. For specific harvesters a transformation can be developed to generate a generic model. The collection of transformations and analyses can be applied to the generic model to get the desired results. The generic intermediate model comes with a collection of tools and frameworks that support its usage. It provides means for interacting with the models using a graphical user interface, for quickly building model analyzers, and manually annotating models for enriching the models with human assessment. Annotations can be saved separate from the model and be reapplied when the model gets regenerated.

The case study has shown that the harvesting workbench works for a production system of 178 KLOC: UML models have successfully been generated that give insight in structure and behavior. These models can be used for both documentation and forward engineering, because code has successfully been generated based on the harvested models, although no part of Haebop has been fully migrated. Before that can be done, a suitable mapping must be defined for specific database constructs that are not natively supported in the target environment.

The GenericAST has proven its use to reuse transformations and analyses, although it could not represent every construct that appeared in the case study. The GenericAST enabled parallel harvesting which was the only feasible way of harvesting large amounts of input. Evolution of the meta-model is still required, using the experience of the case study as a starting point.
19 DISCUSSION

I found the project very interesting to do: combining academics with a business environment is a very challenging approach. On the one hand the results and approach must be scientifically sound; on the other hand the results must work. It was not merely a theoretical exercise but every choice had practical implications, which made the project very challenging. A clear example of this is that the academic thesis reviews resulted in different views and suggestions than the reviews done by the companies, whereas both were very valuable.

The first part of the project I performed at Interactive Objects. It has lead to what I believe is a sound approach of solving the harvesting problem in an MDA context. The solution is by no means finished, exhaustive or complete, but it provides basic means to harvest textual source code to MDA, provided it can be described by a suitable grammar. It must be seen as an attempt to MDA-enable existing systems with greater flexibility than before.

The workbench architecture made the project flexible. It left many choices open until they were required to make. For example, I could develop the GenericAST without changing the other workbench parts. It allows replacing the parser generator by a more powerful one when required.

The choice for the parser generator was not my initial intention. I wanted to be able to use island grammars for source structure specification because it is a very powerful and promising concept. Unfortunately, no suitable parser generator was available for using island grammars. It led to evaluating other parser generators and choosing the ‘second best’ option. It is by no means a bad option. It could be very well adapted to provide convenient grammar development and full code generation for our harvesters.

The case study at De Amersfoortse Verzekeringen, an expert in the area of insurances, showed that harvesting a real, large application is different compared to harvesting only a small part of it. During harvester development it appeared that both grammars and M2M-transformations grew large and modularization support would have been welcome, unfortunately, this feature had to be manually mimicked. The real pain of harvesting a large application proved to be the M2M-transformation, caused by the current M2M-transformation engine implementation. Every step in the harvesting process was in the order of seconds, maybe minutes, but M2M-transformations were strictly in the order of hours, some ran more than 24 hours before completing.

The results of harvesting a large application are very attractive to the eye: seeing hundreds of classes in modules, with thousands of dependencies all being extracted from a large pile of source code is a wonderful sight. On the other hand, it took about 3 iterations before the models were found to be sufficient, requiring some patience and a strong feeling of ‘this time it goes right’. Such large results are also somehow scary: how to handle so much information, what should be done with it? It was not trivial how to proceed: where to start? But after some more thorough model reviews it became slowly clear how the information could be made useful, either for documentation or forward engineering.

According to the initial plan a part of Haebop would have been reengineered to runnable code, but during the project this plan was altered. Forward engineering would only be useful if it could be done according to the process of the Fortis Software Development Organization, an assessment that required detailed knowledge on the process. The credibility of the assessment would also be much higher if done by an independent third party. Therefore, the assessment of model quality for code generation and code quality for the development organization has been put in a separate project, branching from the case study.
This chapter summarizes the future work that has been discussed in the part on the harvesting framework (chapter 9 at page 75) and the case study (chapter 15 at page 109).

The first part of the generic harvesting framework can be made more powerful and flexible by integrating a suitable GLR-parser generator in the workbench. This allows the workbench to make use of island grammars, which have several advantages in harvesting context over traditional grammars. It allows us to incrementally develop partial grammars, tailored for a specific project.

The parser could be adapted such that we can selectively filter which parts of the AST are populated into the generated MOF-repository. Currently this can be done on grammar-level, but source information which appears during parsing could also be used. It allows us to use the classification information from the generic intermediate model to apply filtering. It results in a model with just the right level of details for each AST subtree.

The generic intermediate model can be tested more thoroughly on languages different than PL/SQL. This will lead to further development and refinement of the model and increase its value. Effort should also be put in minimizing the performance issues when using larger models.

The whole harvesting concept could be implemented as a lazy harvester. This implies a strong cooperation between the various tools and therefore vetoes the workbench approach. It allows for more efficient (i.e. demand-based) harvesting, instead of the currently implemented push-based harvesting.

The annotation framework of the generic intermediate model could be extended to a complete refactoring framework. It could be integrated with ArcStyler to keep the harvesting process repeatable, while preserving manual alterations to keep the generic model and the UML views consistent. This allows the more detailed information in the generic model to be used when generating code with ArcStyler, while respecting the applied refactorings.

An important activity for the case study is forward engineering part of Haebop to demonstrate the feasibility of the migration approach. For this, a suitable mapping must be made between typical database constructs and the target environment. The generated code should also comply as much as possible to the guidelines as set by the Fortis Software Development Organization.

Based on the experience gained from the case study, the GenericAST meta-model should be evolved.


Bibliography

127
<table>
<thead>
<tr>
<th>Reference</th>
<th>Authors and Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reu 05</td>
<td>Thijs Reus, 2005, “Methods for automated extraction of models from a legacy software system – assessment and evaluation”, Research Assignment for TU Delft</td>
</tr>
<tr>
<td>Sei 00</td>
<td>The Software Engineering Institute, 2000, “Cyclomatic Complexity” <a href="http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html">http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html</a></td>
</tr>
</tbody>
</table>

A Glossary

ADM Architecture Driven Modernization – OMG’s initiative to the reverse engineering challenge in MDA context

AIM Atomistic Information Mapping – ArcStyler’s M2M-transformation engine

ArcStyler Commercial, state of the art MDA-tool by Interactive Objects GmbH

AST Abstract Syntax Tree – Tree-like representation of source code


Carat Cartridge Architecture – architecture for developing ArcStyler cartridges using MDA

CIM Computation Independent Model – Business level model in the MDA context, independent of platform and computation model

EBNF Extended Backus Naur Form – Formalism for describing textual structures in a grammar

Forward engineering Traditional process of moving from high-level abstractions and logical, implementation independent designs to the physical implementations of a system [CC 90].

GenericAST Generic intermediate model for representing structure and behavior independent of specific source code structure

GLR Generalized Left-to-Right bottom-up parsing – a certain class of grammars, stronger than LR(k), LALR(k) and LL(k)

GUI Graphical User Interface – an interface to the user in some graphical way

Haebop Het Amersfoortse Employee Benefits Offerte Programma – The system that has been used in the case-study of this project

Harvesting Equivalent to reengineering, but with more marketing impact

Initial model The first model in a harvesting process that has lost dependencies with the source structure. Can be a GenericAST or a UML model, for example.

J2EE Java2 Enterprise Edition – component architecture standard for developing business applications on the Java platform

JMI Java Metadata Interface – the Java interface for MOF

LALR(k) Look-Ahead Left-to-Right bottom-up parsing – a certain class of grammars, stronger than LL(k), with a look-ahead of k

Legacy system can be defined as any information system that significantly resists modification and evolution [BLW 99a].

LL(k) Look-Ahead Left-to-Right top-down parsing – a certain class of grammars with a look-ahead of k
<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LR(k)</td>
<td>Left-to-Right bottom-up parsing – a certain class of grammars, stronger than LALR(k) and LL(k), with a look-ahead of k</td>
</tr>
<tr>
<td>M2M</td>
<td>Model-to-Model – special kind of transformation, using one or more source models to generate a target model</td>
</tr>
<tr>
<td>M2T</td>
<td>Model-to-Text – special kind of transformation, using one or more source models to generate a target textual representation</td>
</tr>
<tr>
<td>MDA Enabling</td>
<td>Creates precise MDA/UML models from existing code and infrastructure. It creates reusable models according to a well-defined MDA Modeling Style thus enabling the advantages of high-end MDA immediately from existing infrastructures [Arc 05b]. Similar to MDA Harvesting.</td>
</tr>
<tr>
<td>MDA Harvesting</td>
<td>Process of extracting information from existing software assets to MDA-enable existing systems. The opposite of MDA [Arc 05a].</td>
</tr>
<tr>
<td>MDA</td>
<td>Model Driven Architecture – OMG’s initiative to raise the level of abstraction in software development by considering models as the key software asset as opposed to code</td>
</tr>
<tr>
<td>MDH</td>
<td>Model Driven Harvesting – see MDA Harvesting</td>
</tr>
<tr>
<td>MDR</td>
<td>Model Driven Recycling – see MDA Harvesting</td>
</tr>
<tr>
<td>Meta-model</td>
<td>A model of a model</td>
</tr>
<tr>
<td>MOF</td>
<td>Meta-Object Facility – OMG’s standard for meta-modeling</td>
</tr>
<tr>
<td>Modeling style</td>
<td>Defines what subset of models are valid in a given context. It defines the precision, completeness and consistent abstraction levels of models [Arc 05b].</td>
</tr>
<tr>
<td>.Net</td>
<td>component architecture standard for developing business applications on the Microsoft platform</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group – an open membership, non-profit consortium that produces and maintains computer industry specifications for interoperable enterprise applications</td>
</tr>
<tr>
<td>PIM</td>
<td>Platform Independent Model – architecture model of a software system in MDA context, independent of a specific platform</td>
</tr>
<tr>
<td>PL/SQL</td>
<td>Procedural Language/Structured Query Language – procedural extension of the SQL language by Oracle</td>
</tr>
<tr>
<td>Platform</td>
<td>In MDA context, a certain implementation technology such as J2EE or .Net</td>
</tr>
<tr>
<td>Population</td>
<td>The process of filling a repository with data; also as a verb: to populate.</td>
</tr>
<tr>
<td>PSM</td>
<td>Platform Specific Model – design model of a software system in MDA context, specific for a certain platform</td>
</tr>
<tr>
<td>Redocumentation</td>
<td>Creation or revision of a semantically equivalent representation of a system within the same relative abstraction level. The resulting forms of representation are usually considered alternative views [CC 90]. Special case of reverse engineering for informational purposes.</td>
</tr>
<tr>
<td>Reengineering</td>
<td>Also known as both renovation and reclamion, is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form [CC 90]. It is the</td>
</tr>
</tbody>
</table>
concatenated process of reverse engineering, optional restructuring, and forward engineering.

<table>
<thead>
<tr>
<th>Restructuring</th>
<th>Transformation from one representation form to another at the same relative abstraction level, while preserving the subject system’s external behaviour (functionality and semantics) [CC 90]. Refactoring is a way of restructuring.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reverse engineering</td>
<td>Process of analyzing a subject system to identify the system’s components and their interrelationships and create representations of the system in another form or at a higher level of abstraction [CC 90]. Equivalent to harvesting in general (thus without the MDA target context).</td>
</tr>
<tr>
<td>RFP</td>
<td>Request for Proposal</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language – standard language for querying relational databases</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language – OMG’s standard modeling language</td>
</tr>
<tr>
<td>XMI</td>
<td>XML Metadata Interchange – OMG’s standard for representing models of MOF meta-models in XML</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language – a markup language for documents containing structured information</td>
</tr>
</tbody>
</table>
B Parser Generator Comparison

Several parser generators have been compared to select a simple, but powerful and portable tool for the harvester generator. Finally the Grammatica parser generator has been chosen and modified to maximally support the harvesting process. Table B-1 shows the comparison of the considered parser generators. We refer to section 6.3.1 at page 28 for more information on the usage of the parser generator.

Table B-1 Parser generator comparison

<table>
<thead>
<tr>
<th>Parser Generator</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASF+SDF meta environment</td>
<td><a href="http://www.cwi.nl/projects/MetaEnv/">http://www.cwi.nl/projects/MetaEnv/</a></td>
</tr>
<tr>
<td></td>
<td>+ GLR parser generator</td>
</tr>
<tr>
<td></td>
<td>+ Scannerless parser</td>
</tr>
<tr>
<td></td>
<td>+ Active tool (development is ongoing)</td>
</tr>
<tr>
<td></td>
<td>+ Successfully built island parsers</td>
</tr>
<tr>
<td></td>
<td>- Requires Linux</td>
</tr>
<tr>
<td></td>
<td>- Only partially written in Java (rest in C)</td>
</tr>
<tr>
<td></td>
<td>- Does not generate Java (but parse table for other tool)</td>
</tr>
<tr>
<td></td>
<td>- Proprietary grammar specification (SDF)</td>
</tr>
<tr>
<td></td>
<td>+ Mature tool</td>
</tr>
<tr>
<td></td>
<td>+ Standard grammar specification (EBNF)</td>
</tr>
<tr>
<td></td>
<td>- GLR parser generator (but non-scalable implementation)</td>
</tr>
<tr>
<td></td>
<td>- Not written in Java (but in C)</td>
</tr>
<tr>
<td></td>
<td>- Does not generate Java (but C)</td>
</tr>
<tr>
<td></td>
<td>+ Generates Java</td>
</tr>
<tr>
<td></td>
<td>+ Standard grammar specification (EBNF)</td>
</tr>
<tr>
<td></td>
<td>- No GLR (but LALR)</td>
</tr>
<tr>
<td></td>
<td>- Not written in Java (but in C)</td>
</tr>
<tr>
<td></td>
<td>- Old tool (last update in 2002)</td>
</tr>
<tr>
<td>Parser Generator</td>
<td>Information</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>ElkHound</td>
<td><a href="http://www.cs.berkeley.edu/~smcpeak/elkhound/">http://www.cs.berkeley.edu/~smcpeak/elkhound/</a> (GLR parser generator) + Active tool (development is ongoing)</td>
</tr>
<tr>
<td>Jaccie</td>
<td><a href="http://www2-data.informatik.unibw-muenchen.de/Research/Tools/JACCIE/index.html">http://www2-data.informatik.unibw-muenchen.de/Research/Tools/JACCIE/index.html</a> + Generates Java + Written in Java + GUI</td>
</tr>
</tbody>
</table>

- Not written in Java (but in C++)
- Does not generate Java (but C++)
- Only LL(k) with k → ∞ (no GLR)
- Hard to write island grammar
- Not written in Java (but in C++)
- Does not generate Java (but C++)
- Non standard grammar specification (C++ alike)
- No GLR (only LL(1), LR(0), LR(1), SLR(1), LALR(1))
<table>
<thead>
<tr>
<th>Parser Generator</th>
<th>Information</th>
</tr>
</thead>
</table>
| PRECC            | http://www.afm.sbu.ac.uk/precc/  
+ Standard grammar specification (EBNF)  
- LL(k) with k → ∞ (no GLR)  
- Not written in Java (but in ANSI-C)  
- Does not generate Java (but ANSI-C) |
| PyGgy            | http://www.lava.net/~newsham/pyggy/  
+ GLR parser generator  
- Not written in Java (but Python)  
- Does not generate Java (but Python)  
- Immature tool (version 0.4 currently) |
| SLK              | http://home.earthlink.net/~slkpg/  
+ Generates Java  
- Not written in Java (but available for Win & Linux)  
- Only LL(k) with limited k (no GLR)  
- Non standard grammar specification (but EBNF alike) |
This appendix describes the matching algorithm that is used by the repository population algorithm in the harvesters. We refer to section 6.3.3 at page 35 for more information on the usage of the matching algorithm. The matching algorithm tries to do the following:

**Given**
- LEN\(\(k\)\) : a list of \(k\) typed encountered nodes
- EL\(\(n\)\) : a production alternative sequence of \(n\) typed elements
  - Each element has a multiplicity of 1, 0..1, 0..* or 1..*

**Find out**
- If a match is possible for the given parameters
- For each \(i\) determine the exact number of nodes that go into EL\(i\)

Below the matching algorithm is shown in pseudo-code. The following ‘methods’ are used by the algorithm:

- **maxLength**
  Finds out the length of the longest sequence of nodes on a given list of nodes, of a given type, backwards from a given position. If the first tested node is not of the given type, the length is 0.

- **validSequenceWithNodeCount\_i**
  Determines if a given choice for nodeCount \(i\) is valid. It calls MatchFrom with the proposed parameters. If null is returned, the choice is not valid. Otherwise, the result contains the complete nodeCounts array which we can return immediately.

---

### Match()

**Input**
- EL: the list of types that are expected for the alternative, with multiplicity
- LEN: the list of encountered nodes

**Output**
- nodeCounts: an array that has for each EL\(i\) the number of corresponding nodes in LEN or null if it was no match

**Do**
- # Start recursion
- nodeCounts = array of integer
- return MatchFrom(
  - EL, LEN, nodeCounts,
  - count(EL), count(LEN))

**Od**
MatchFrom()

**Input**

- **EL**: the list of expected types with multiplicities
- **LEN**: the list of encountered nodes
- **nodeCounts**: array with node counts for each EL(i)
- **elementNr**: the index of EL we should check
- **nodeNr**: the index of LEN we should check from

**Output**

- **nodeCounts**: an array that has for each EL(i) the number of corresponding nodes in LEN or null if it was no match

**Do**

- # Stop recursion if needed
  - if elementNr < 0 then
    - if nodeNr < 0 then return nodeCounts
    - else return null
  - if nodeNr < 0 then
    - if this and all other elements can be zero then return nodeCounts
    - else return null

- # Find out the number of nodes that match the current element
  - if elementMultiplicity equals exactlyOne then
    - if element-type not equal to node-type then return null
    - else nodeCounts[elementNr] = 1
  - else if elementMultiplicity equals oneOrZero then
    - size = maxLength(LEN, elementType, nodeNr)
    - if size >= 1 and validSequenceWithNodeCount_1 then return nodeCounts
    - else nodeCounts[elementNr] = 0;
  - else if elementMultiplicity equals atLeastOneOrZero then
    - size = maxLength(LEN, elementType, nodeNr)
    - for each s in [size…element-minimal-occurrence] do
      - if validSequenceWithNodeCount_s then return nodeCounts
    - if not validSequenceWasFound then return null

- # Round up by selecting the next stack element and node number
  - nodeNr -= nodeCounts[elementNr]
  - elementNr -= 1
  - return MatchFrom(EL, LEN, nodeCounts, elementNr, nodeNr)

**Od**
This appendix shows a high-level overview of the GenericAST meta-model. A brief description is provided for each diagram to explain the main concepts; attributes of model elements are hidden for simplicity. We refer to chapter 7 at page 49 for more information on the GenericAST and associated frameworks & tools.

**Structural Part**

This section describes the structural part of the GenericAST meta-model. Figure D-1 shows the package structure for the structural part. In the following sections the model elements are described for each package.

**Core Package**

The Core package contains the main elements for the GenericAST. It defines the root element ‘Element’, which may have tagged-values and a source reference. A tagged-value is model element with a name and value, which is a light-weight meta-model extension mechanism for storing unanticipated information in GenericAST models. A source reference contains a reference to the original source code, as a link or a textual fragment.

The element ‘NamedElement’ adds a name to ‘Element’ and a collection of aliases. It is the base class for most model elements in the GenericAST meta-model.

The element ‘Visibility’ defines visibilities for GenericAST model elements. It is equivalent to UML visibilities, such as public, package or private.

Figure D-2 shows the contents of the Core package in the structural part of the GenericAST meta-model.

**Container Package**

The Container package defines model elements that contain other model elements as their sole purpose. The element ‘Container’ can be used for describing a Java package structure.

The element ‘HiddenContainer’ can be used for storing internal GenericAST model elements, such as types or visibilities.

Figure D-3 shows the contents of the Container package in the structural part of the GenericAST meta-model.
Entity Package

The Entity package provides model elements for representing concepts such as Java classes or database tables. An entity can have structural and behavioral features. It can have inner-entities and may inherit from multiple other entities. It may implement several entities comparable to Java interfaces, which are entities with only behavioral feature specifications. An entity may have a number of triggers associated with it. Triggers specify behavior on certain events, such as the creation of an entity instance.

Figure D-4 shows the contents of the Entity package in the structural part of the GenericAST meta-model.

Feature Package

The Feature package defines two features: structural feature and behavioral feature. A structural feature can be used for representing attributes or table columns. They provide data storage for an entity. A behavioral feature can be used for representing methods or operations. They provide logic for an entity. A behavioral feature may have multiple parameters to control the behavior. Both features may have associated triggers which specify behavior on certain events. A foreign key constraint is a constraint that refers to another feature. A behavioral feature has an association to Block which is the entry point for the behavioral part of the GenericAST. It specifies the behavior of the feature.

Figure D-5 shows the contents of the Feature package in the structural part of the GenericAST meta-model.
**Constraint Package**

The Constraint package provides a simple constraint for specifying constraints using a string. In the Feature package there is a foreign key constraint to specify foreign keys.

Figure D-6 shows the contents of the Constraint package in the structural part of the GenericAST meta-model.

**Trigger Package**

The Trigger package provides a base class for triggers. Triggers for specific model elements are located in the respective packages. A trigger specifies behavior and has therefore an association to Block, which is the entry point for the behavioral part of the GenericAST.

Figure D-7 shows the contents of the Trigger package in the structural part of the GenericAST meta-model.

**Type Package**

The Type package contains model elements for specifying types. We created an initial type-hierarchy to provide more semantic models. Some types specify other types, such as ‘ParameterizableType’ or ‘CollectionType’. The latter supports Java5 generics, for example.

Figure D-8 shows the contents of the Type package in the structural part of the GenericAST meta-model.
Behavioral Part

This section describes the behavioral part of the GenericAST meta-model. Figure D-9 shows the package structure for the behavioral part added to the structural part. In the following sections the model elements are described for each behavioral package.

Behavior-Core Package

The behavior Core package provides the entry point for representing behavioral information. It defines Block which contains statements as its sole purpose. It also defines Scope and ScopeItem which can be attached to a statement. It defines the known elements in the scope of the statement. A scope can have a super-scope which contains known items from a level higher.

Figure D-10 shows the contents of the Core package in the behavioral part of the GenericAST meta-model.
Behavior-Statement Package

The behavior Statement package contains specializations for different statements. For several general programming language constructs a specific statement has been defined. For example, there is a ConditionalStatement which represents a Java ‘if’-statement but also a Java ‘switch’-statement. A JumpStatement refers to another statement, to which it jumps. A ConditionalStatement contains other ConditionalStatements for representing ‘else if’ constructs.

Implementation specific statements, which generally are IO-statements, have been abstracted by several IO-statement classes. We can for example represent database IO-statements in a separate model element without implementation specific details. It makes the GenericAST less generic but keeps it of manageable size. Future meta-model evolutions may elaborate the IO-structure.

Figure D-11 shows the contents of the Statement package in the behavioral part of the GenericAST meta-model.

Behavior-Expression Package

The behavior Expression package provides representations for expressions. An expression can be evaluated and results in a value, although currently the GenericAST does not natively define expression semantics. A ConditionalExpression results in a Boolean value, whereas a CalculationExpression results in a numeric value. Both have other expressions as operands.
and a list of operators to connect the operands. There are several PrimitiveExpressions for representing numbers, strings and Booleans. The SimpleExpression allows us to represent any expression by a string; a LookupExpression represents a scope lookup or a type lookup.

We have not implemented evaluation functionality because it is out of scope for this meta-model. If required we may develop a model analyzer or processor for evaluating expressions.

Figure D-12 shows the contents of the Expression package in the behavioral part of the GenericAST meta-model.

![Figure D-12 GenericAST behavioral Expression package (in UML)](image)

**Behavior-Operator Package**

The behavior Operator package defines operators which connect operands in conditional and calculation expressions. Each operator is defined by a model element and an enumeration value. There are conditional operators, such as AND, OR and NOT. There are calculation operators, such as ADD, SUBTRACT and MULTIPLY. CUSTOM provides a light-weight extension mechanism for representing operators that are not predefined by the meta-model.

Figure D-13 shows the contents of the Operator package in the behavioral part of the GenericAST meta-model.

![Figure D-13 GenericAST behavioral Operator package (in UML)](image)

**Behavior-Property Package**

The behavior Property package provides representation of properties that can be attached to model elements. The intended purpose is model analysis. Model elements can be enriched with properties, which are known to be valid. Using model analysis properties can be derived for a model element by looking at properties of its child model elements.

This package has not yet been tested; it has been provided as a start for future work.

Figure D-14 shows the contents of the Property package in the behavioral part of the GenericAST meta-model.
Appendices 147

Figure D-14 GenericAST behavioral Property package (in UML)
This appendix lists the used information sources for the system used in the case study, Haebop. Table E-1 shows the interviews and official meetings that have been organized to gather information about Haebop and provide feedback to the stakeholders. Table E-2 shows the documents about Haebop that have been used for information gathering.

### Table E-1 Interviews and official meetings with Haebop stakeholders

<table>
<thead>
<tr>
<th>Date</th>
<th>Topic</th>
<th>Present</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 March 05</td>
<td>Environment of Haebop</td>
<td>Raymond Kolfschoten (De Amersfoortse)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Thijs Reus</td>
</tr>
<tr>
<td>5 April 05</td>
<td>Introduction to Haebop</td>
<td>Bert Torn (VDA)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Remco Zomer (VDA)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lex van de Geest (De Amersfoortse)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Thijs Reus</td>
</tr>
<tr>
<td>2 May 05</td>
<td>Inner details of Haebop</td>
<td>Bert Torn (VDA)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Remco Zomer (VDA)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lex van de Geest (De Amersfoortse)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Thijs Reus</td>
</tr>
<tr>
<td>20 June 05</td>
<td>Project status update</td>
<td>Bert Torn (VDA)</td>
</tr>
<tr>
<td></td>
<td>Future directions</td>
<td>Remco Zomer (VDA)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lex van de Geest (De Amersfoortse)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Thijs Reus</td>
</tr>
<tr>
<td>5 September 05</td>
<td>Presentation + live demo Case study kick-off</td>
<td>Bert Torn (VDA)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Remco Zomer (VDA)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lex van de Geest (De Amersfoortse)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>John Adegeest (De Amersfoortse)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Adriaan Quakkelaar (De Amersfoortse)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rob Vogelzang (Interactive Objects)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Thijs Reus</td>
</tr>
</tbody>
</table>

### Table E-2 Documents about Haebop

<table>
<thead>
<tr>
<th>Document</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FoAPP980909</td>
<td>The main functional design of Haebop. First increment with modules ‘Optimaat’, ‘Ongevallen’ and ‘Reis’. 60 pages.</td>
</tr>
<tr>
<td>Document</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>OnPembaER-990618</td>
<td>Functional design of additional module ‘Pemba Eigen Risico’. 38 pages.</td>
</tr>
<tr>
<td><strong>Technical documentation</strong></td>
<td></td>
</tr>
<tr>
<td>TDHAEBOP2991021</td>
<td>The main technical design of Haebop. First and second increment. 639 pages.</td>
</tr>
<tr>
<td>TDHAEBOP3en4991021</td>
<td>Technical design of additional modules. 259 pages.</td>
</tr>
<tr>
<td><strong>Miscellaneous documentation</strong></td>
<td></td>
</tr>
<tr>
<td>Database and PLSQL Style Guide</td>
<td>Style guide defining code style and naming conventions used by VDA. 14 pages.</td>
</tr>
</tbody>
</table>