An Integrated Development Environment for Business Object Notation

Ralph Skinner

A dissertation submitted in part fulfilment of the degree of

MSc. in Advanced Software Engineering

Supervisor: Dr. Joseph Kiniry

UCD School of Computer Science and Informatics

College of Engineering Mathematical and Physical Sciences

University College Dublin

April 16, 2010

Table of Contents

Abstract

This dissertation describes the design and implementation of an integrated development environment for the Business Object Notation specification language (BON). Particular emphasis is given to the visual aspect of BON, whereby a BON specification can be represented in full fidelity as a graphical diagram. The BON IDE forms an integral part of a software engineering methodology that provides the tools and practices for system design, from high-level specification to implementation.

The project takes a model-driven design approach, where the central component is a meta-model of the BON language from which the implementation is largely derived, using Eclipse-based technologies. The metamodel aims to capture not only the elements of the BON language and the relationships between them, but also the behavioural and syntactical characteristics of the language in the form of constraints on the model.

Acknowledgments

I would like to thank my thesis supervisor Dr. Joseph Kiniry, who has been consistently supportive and positive with regard to this project. I always came away from our meetings re-energized, re-focused, and more knowledgeable. I would also like to thank Fintan Fairmichael for taking the time out of his own schedule to meet with me and provide much help, advice, and code. Finally, I wish to acknowledge the patience and consideration of my wife Karen and our children, Jenna, Rían and Sam.

List of Figures

List of Tables

Chapter 1: **Introduction**

1.1 Motivation

Model driven development (MDD) is widely seen as the next level of software development [5], and is starting to see increased use in the development of complex systems [9]. Ideally, MDD moves the focus from a program expressed as computer language code, to a more abstract representation that is closer to the problem domain [10]. In this way, models can help make a complex system easier to understand, which should pay dividends in the quality of software design, and ultimately, in the quality of implementation. However, MDD is not without its challenges, one of the most pressing being the need to extend the relevance and lifetime of models beyond the initial design phase, into the implementation and maintenance phases.

The Business Object Notation (BON) is a set of concepts for modelling object-oriented software [11], which is aimed at addressing this and other issues in MDD (section 2.1). One of the guiding principles in BON is simplicity, both in general principles and also in terms of the notation used to express BON. The BON notation is a key feature of the language that can render a BON specification both textually and graphically. The graphical representation is more compact, and conveys a complex design more quickly and concisely than the textual representation. In addition, the graphical notation is designed to show different levels of detail in different contexts, a feature that demands automated tool support.

1.2 Objectives

To this end, the objectives are to deliver a visual development environment for BON, which can:

- Render a textual BON specification in its graphical format.
- Validate the correctness of the diagram according to the syntax of the BON language.
- Display visual feedback related to validation errors.
- Show ancillary views, such an outline view and a project navigation view.

1.3 Outline

The remainder of this document is divided into four chapters and appendices as follows:

- **Chapter 2** introduces BON concepts, the BON notation, and related work.

- **Chapter 3** explores the technologies that were selected to implement the BON IDE.

- **Chapter 4** describes the details of the implementation.

- **Chapter 5** presents conclusions and suggestions for future work.

- **Appendices**

Chapter 2: **Background**

This chapter presents BON and its key concepts. The intention is to give a high-level overview of the BON methodology in order to pave the way for subsequent chapters, and to illustrate the purpose served by a BON IDE. Two pieces of related work are also described: BONc, a parser and typechecker for BON that forms part of the BON IDE, and the BON Development Tool (BDT).

2.1 BON

BON is general purpose methodology for the production of object-oriented software. It provides recommended processes and guidelines, and a notation that can be expressed textually and graphically. In some respects, BON is similar to other modelling and specification languages such as UML, however, it contains several key features designed to improve the quality and reliability of the produced software. Three of these features that are fundamental to the BON approach are seamlessness, reversibility, and software contracting.

Seamlessness refers to the concept of integrating the analysis, design and implementation of software into a smooth, unbroken process. The challenge is to provide a high-level design notation that is simple to understand and easy to use, but rich enough to be suitable for translation into the programming language of choice.

Fundamental to the idea of seamlessness is reversibility: the ability to have full-fidelity translation from design specification to implementation code and back again. Reversibility prevents the divergence between the design specification of a software system and the inevitable modifications that are introduced in during its implementation and maintenance.

Software contracts allow designers to define the semantics of objects in the system by asserting before and after conditions for class operations (pre- and post-conditions) and the validity of the overall state of a class (invariants). Software contracting is key to the production of reliable, high quality software [8].

Central to BON is the notation that is used to specify a software system by describing the software objects in the system, the relationships between them, and the semantics of those objects in terms of behavioural constraints. In object oriented terms, BON allows the formal specification of classes, class methods, inheritance and client-supplier relationships between classes, and software contracts. The next section introduces the BON notation with examples in textual and graphical format.

2.1.1 BON Notation

The fundamental elements in a BON specification are classes and clusters. Classes are conventional object oriented classes; a cluster is a container that groups together related classes and other clusters.

Figures 2.2 and 2.1 show an example of the BON textual and graphical notation, respectively. The code in figure 2.2 describes a class `CAGE`, which is subdivided into four primary subsections: indexing, inheritance, features and class invariants. The indexing clause consists of one or more index entries and corresponding keywords. The inheritance clause specifies super classes, if any. For each feature, the number arguments and the return type are specified. A feature may have one or more preconditions, delineated by the `require` keyword, which are predicates that must be true when the feature is invoked by a client. Similarly, post-conditions, delineated by `ensure`, are predicates that must be true when the feature has been executed and control is returned to the client [11]. The class invariant(s) express consistency constraints for the entire class that must be true before a feature is executed, and after execution is completed.

The diagram in figure 2.1 shows the graphical equivalent of the `CAGE` class. Within the main figure, each of the four primary sub-sections is represented by its own compartment. This aids readability and also allows for compartment to be collapsed, hiding the detail within and simplifying the overall diagram. Preconditions and postconditions are expressed in their more concise symbolic format.
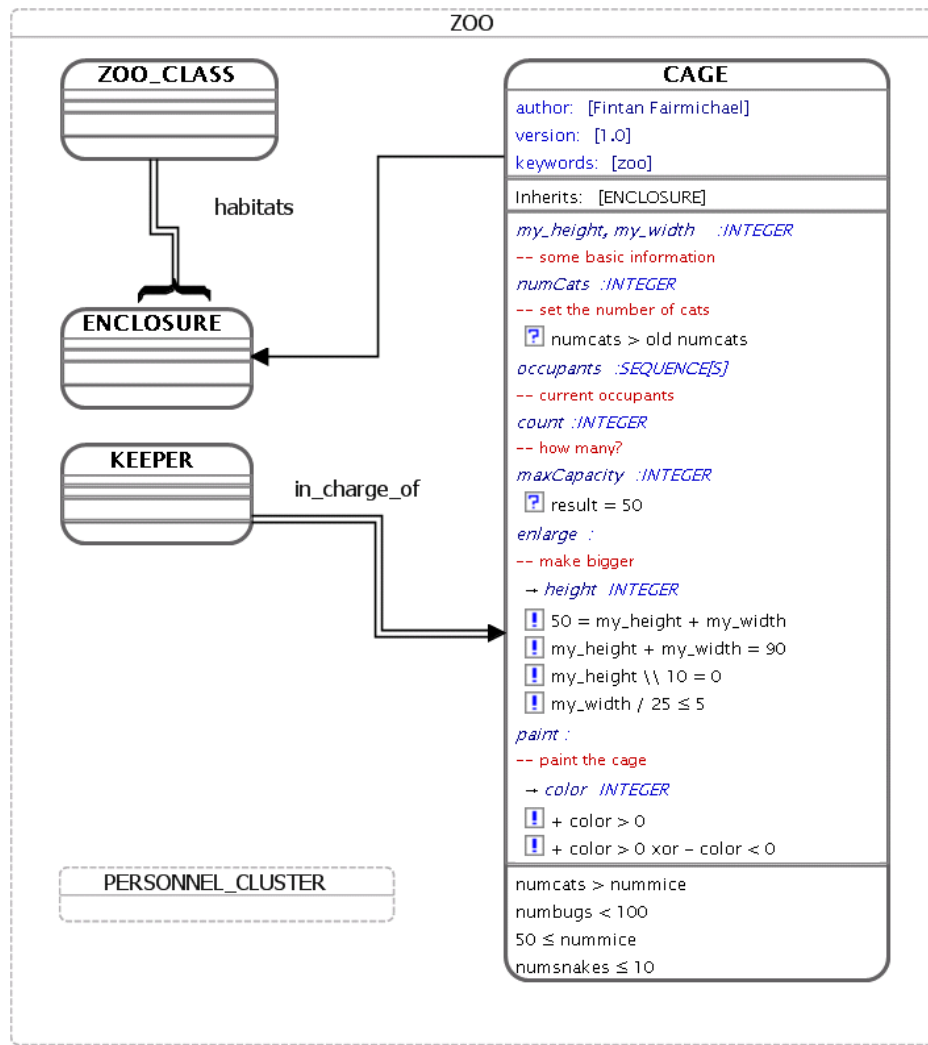

Figure 2.1: Graphical BON example

Figure 2.1 illustrates some additional aspects of BON over and above that shown in figure 2.2. The `ZOO_CLASS`, `ENCLOSURE`, and `KEEPER` classes are displayed with all their compartments collapsed, simplifying the display at the expense of a reduction in the amount of detail shown.

The connections between `CAGE` and the other classes indicate the relationship between the classes, specifically, `CAGE` is inherited from `ENCLOSURE` (single line connection), `ZOO_CLASS` aggregates one or more `ENCLOSURE`s (a double line with a curly brace), and `KEEPER` is associated with an instance of the `CAGE` class (double line with an arrow). Finally, figure 2.1 also indicates that all four classes are grouped together into the `ZOO` cluster, and that the `ZOO` cluster itself contains a nested cluster, `PERSONNEL_CLUSTER`, which is collapsed to hide its members.

The full specification for BON notation is given in [11].

```
class CAGE
   indexing
      author: "'Fintan Fairmichael"';
      version: 1.1
      keywords: zoo
   inherit
        ENCLOSURE
   feature
      my_height, my_width: INTEGER  -- some basic information
      numCats: INTEGER -- set the number of cats
         ensure
            numCats > old numCats;
         end
occupants: SEQUENCE[S] -- current occupants
maxCapacity: INTEGER -- how many?
   ensure
      Result = 50;
   end
enlarge -- make bigger
   -> height: INTEGER
      require
         50 = my_height + my_width;
         my_height + my_width = 90;
         my_height \\ 10 = 0;
         my_width / 25 <= 5;
      end
paint -- paint the cage
   -> color: INTEGER
      require
         +color > 0;
         +color > 0 xor -color < 0;
      end
invariant
   numCats > numMice;
   numBugs < 100;
   50 <= numMice;
   numSnakes <= 10;
   end
```

Figure 2.2: Textual BON example

2.2 Related work

2.2.1 BONc

BONc is a parser and typechecker for BON that is under development by Fintan Fairmichael [6]. BONc can read one or more files and/or input from standard input in the BON textual format, parse the input and typecheck it. The parser also has a pretty-printer that formats and indents the code and displays color syntax and error highlighting. BONc provides the textual view of a BON specification that corresponds to the graphical view in the BON IDE.

BONc exposes an API that allows clients access to the abstract syntax tree (AST) that is the in-memory representation of a BON specification. The BON IDE uses the AST when constructing a diagram from BON code as outlined in section 4.4.

2.2.2 The BON CASE Tool (BCT)

The BCT is a Java-based application for BON development that was created by Jason Lancaric under the supervision of Jonathan Ostroff and Richard Paige [7]. Although the BCT shares some functionality with the BON IDE, it is not Eclipse-based, and it was developed pre-GMF and so does not leverage that technology. The BCT is not currently in active development.

Chapter 3: **Design Approach and Key Technologies**

This chapter describes the technologies that were used to implement the BON IDE, and gives the rationale for the choices made. From the outset it was decided to target the development as a plug-in for the Eclipse platform. Eclipse is a proven, widely-used technology on which to build tools. It is highly extensible and configurable, with broad support for the kind of features required by an IDE, such as project navigation and outline views. One of the most useful features of the Eclipse framework is its ability to develop, debug and execute plug-ins in the run-time workbench, in other words, an Eclipse plug-in can be developed from within Eclipse itself.

3.1 Model Driven Design

In addition to providing a graphical or notational interface for the BON language, it is required to be able to check the validity of the BON diagram being generated or edited. The BON IDE has to capture not just the display characteristics of the static diagram, but the syntax and semantics of the underlying BON specification as well. For this reason, the starting point of the design is a metamodel that describes a BON specification. The metamodel can capture and evaluate the syntactic constraints to which a valid model must adhere, and can also be leveraged to provide the underlying model for the graphical editor.

To achieve the goals of building an Eclipse-integrated, metamodel-based, graphical environment with model validation, requires a set of integrated software technologies, namely EMF, GEF, GMF, and M2T. The following sections give a brief introduction to each of these technologies.

3.2 Eclipse modelling Framework (EMF)

EMF is a modelling framework and code generation facility for building tools and other applications based on a structured data model or domain model [1]. The EMF-generated code has support for model creation and manipulation, including serialization and deserialization of the model. It also contains a mechanism for change notification that automatically informs listener classes when the model is modified. EMF supports validation of model constraints by providing a framework that validation adapters can plug into (section 3.5). Most critically for this project, EMF provides the foundation for interoperability with other EMF-based tools and applications such as GEF and EMF.

In our case the structured data model at the core of EMF is the BON metamodel upon which the graphical BON IDE is based. In EMF, this model is called an *Ecore* model. Eclipse provides a plug-in, the graphical Ecore editor, that is used to create and edit Ecore

models. Figure 3.1 shows the Ecore editor including the display area and tool palette. Using the Ecore Editor, a detailed model is specified that describes classes, class attributes and their type, and class relationships such as inheritance and aggregation.
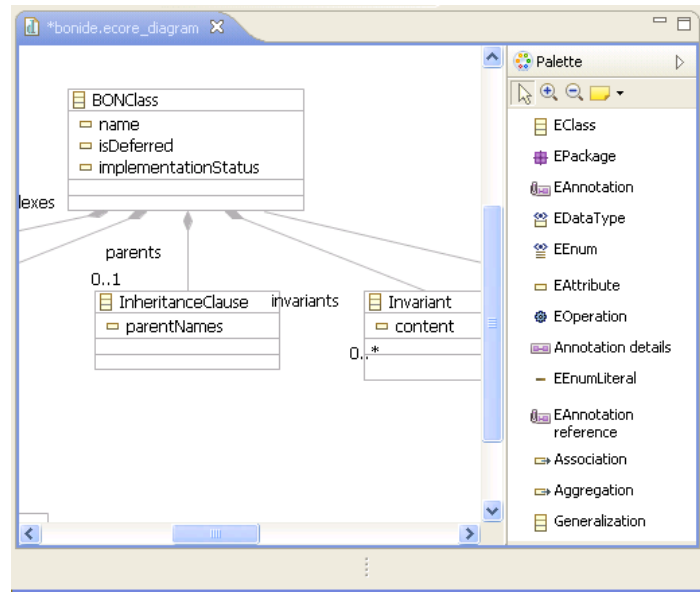

Figure 3.1: Graphical Ecore editor

A second model, called a *genmodel*, is used to control the code generation process and allows fine control over class packaging, package names, class names, and code formatting..

From the Ecore model and the genmodel, the EMF generator creates a set of Java classes for the model, a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor. The generated code is described in more detail in section 4.1.1.

The generated Java code can be edited if required to add methods and instance variables to the classes. Marking the manually edited code as `@generated NOT` allows the code to be regenerated from the model and merged with the user code, without overwriting the users changes.

3.3   The Graphical Editing Framework (GEF)

GEF is a layout and rendering toolkit for the development of graphical interfaces that support the display and editing of an underlying domain model [2]. Whilst GEF is essentially application-neutral, it lends itself to the development of CAD-like editors that provide tools that end users can use to select, create, edit and connect graphically distinct elements in a visual way.

The GEF framework contains classes that support the rendering of figures and connections, the provision of a palette for user tools for element selection and creation, and a command framework with undo/redo facilities.

Further details of GEF are explored from the perspective of GMF, with which it is heavily integrated.

3.4 Graphical modelling Framework (GMF)

GMF builds a so-called *generative bridge* [3] between EMF and GEF. Taking input from a domain model expressed as an Ecore model, and integrating the information from a graphical definition, tooling definition and mapping definition, GMF can generate the Java code that forms the basis of a graphical editor. The generated code is based largely on classes from the GEF framework. Figure 3.2 shows an overview of the process of building a GMF-based plug-in, indicating which models are developed in parallel and which are interdependent. The following sections take a closer look at the models used by GMF.
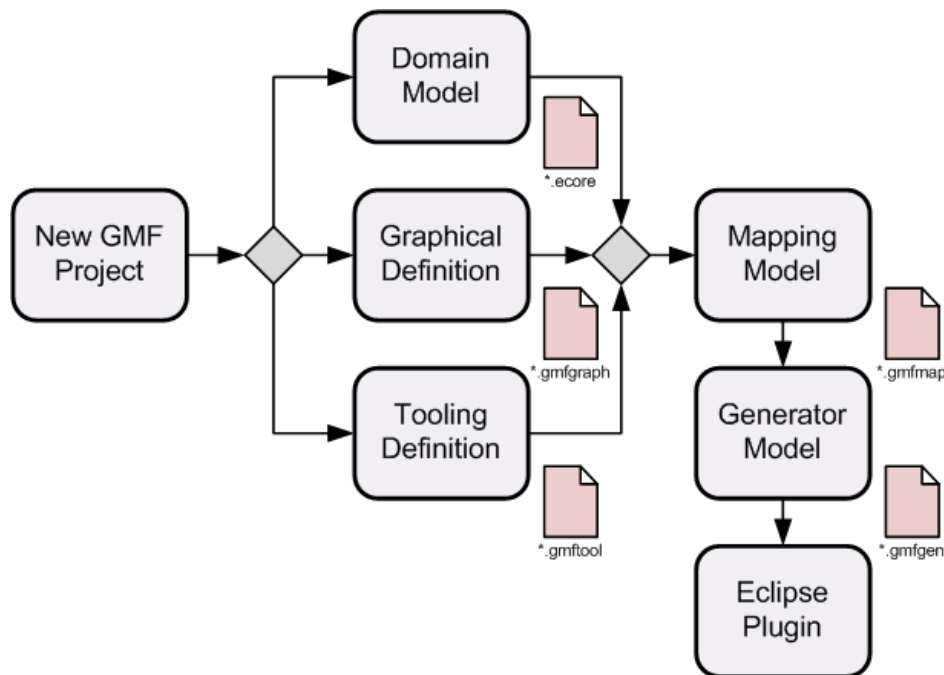

Figure 3.2: GMF process overview

3.4.1 Graphical Definition Model

The graphical definition model is used to specify the appearance of the static diagram. All metamodel elements that have a graphical representation have their shape, size, color and other visual features defined here. In model-view-controller fashion, the graphical definition model deals only with graphical elements and has no mapping to elements in the domain model.

The primary elements in the graphical definition model are nodes, connections, labels, and compartments.

- **Nodes** are visual diagram elements such as clusters, classes, or features. Nodes may have a hierarchical structure, for example, a class node can contain one or more feature nodes, which can contain argument nodes.

- **Connections** are segmented lines that connect nodes. Aspects of a connection such as its routing behaviour and anchoring behaviour can be specified.

- **Labels** are text, such as class and cluster names, that appear in the diagram.

- **Compartments** are container elements that add layout and behaviour to nodes, for example, a compartment can be collapsed, hiding the items contained within it.

3.4.2 Tooling Definition Model

The tooling definition model specifies tools that are available to the user to edit the graphical model. The tools defined are available in a palette or toolbar that becomes part of the final Eclipse plug-in. The toolbar provides conventional graphical editing tools such as selection and zoom tools, and creation tools that are used to add new elements to the diagram.

3.4.3 Mapping Model

Thus far we have seen independent models that address different aspects of the target diagram editor; the meta (Ecore) model, the visual (graphical definition) model and the tools (tooling definition) model.

The mapping model integrates these models and binds together the semantic (meta) objects with their concrete visual counterparts. Each mapping in the mapping model connects the metamodel element from the Ecore model with its corresponding visual representation from the graphical definition model, and to a tool from the tooling definition model. The mapping model also defines parent-child relationships between elements, which are manifested in the graphical diagram as containment.

3.4.4 Generator Model

The genmodel is the final model in the GMF workflow. It's purpose is to act as a link between the mapping model that encapsulates the design intent, and the actual generated Java code. Its functionality is analogous to the EMF genmodel (section 3.2), that is, it specifies code generation details like class names and package names and some implementation-specific parameters to allow fine-tuning of generated code. For example, properties of the genmodel can be set to enable validation in the generated code (section 4.5).

The high-level architecture of the generated plug-in is shown in figure 3.3. The code is based on the model-view-controller design pattern where the model is decoupled from its visual representation. The generated code is described in more detail in section 4.3.1.

3.5 Model-to-Text and Check

M2T is a toolkit that focuses on the generation of textual artifacts, such as programming language code, from models [4]. M2T provides languages that can be applied in different contexts in model driven development. One of those languages is **Check**, a domain specific language that is specialized on model validation.

Check is based on a type system and an expression language. The type system combines built-in types and the types of metamodel implementations, such as Ecore, that are registered with
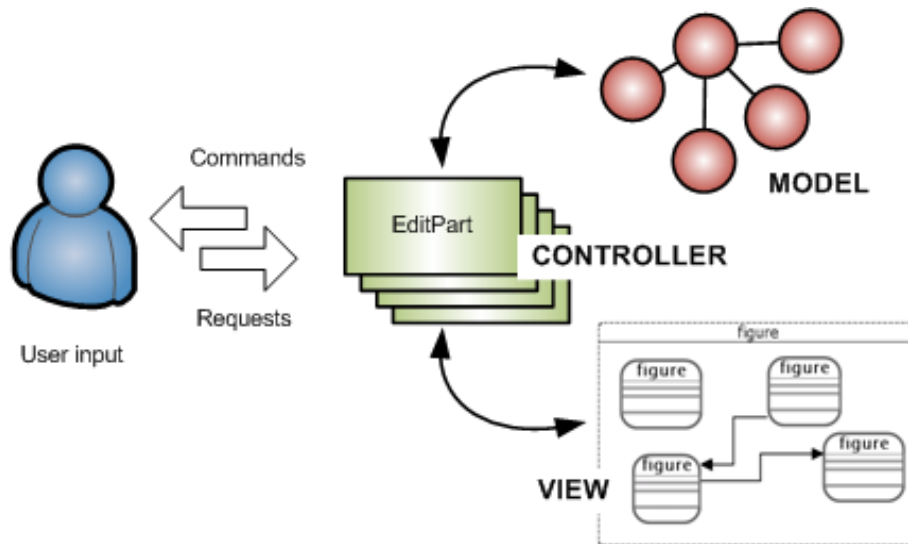
Figure 3.3: GMF architecture

Check. In this way, Check provides an abstraction layer over the metamodel. The statically typed expression language, syntactically a mixture of Java and OCL, defines a concrete syntax for executable expressions, using the type system.

Check is used to define semantic constraints over and above the constraints implied by the structure of the metamodel itself and as such becomes an extension of the metamodel.

Chapter 4: **Implementation**

This chapter presents the implementation of the BON IDE using the technologies that were introduced in Chapter 3.

4.1   Metamodel

The BON metamodel is implemented as an Ecore model. The metamodel described by Paige and Ostroff [9] was used as the initial basis for the Ecore metamodel. The Paige and Ostroff model is designed with formal checking of BON programs as its eventual purpose; our metamodel is required to provide static checking only, therefore the elements of the Paige and Ostroff model concerned with dynamic checking can be omitted. However, the BON IDE metamodel is required to capture information related to graphical artifacts, such as the expanded or collapsed state of a class or cluster, so those model elements must be added. The full BON Ecore metamodel is shown in its graphical and its hierarchical text format in Appendix A. The following is a description of the metamodel.

The top level element in the metamodel is the `Model`, which can be regarded as a container that contains all other model elements. As will be seen, in the graphical representation, the `Model` will become the canvas of the diagram upon which the diagram is rendered (section 4.2). As depicted in the Paige and Ostroff model, a BON specification consists primarily of a collection of *Abstractions* and a collection of *Relationships*.

An abstraction is a class or a cluster in the BON specification. A cluster may contain classes and other nested clusters. In the Ecore model, this potentially endless recursive hierarchy is modeled by inheriting both `BONClass` and `Cluster` from `StaticAbstraction`, and aggregating an unbounded number of `StaticAbstractions` by `Cluster`. This relationship is shown in figure 4.1.
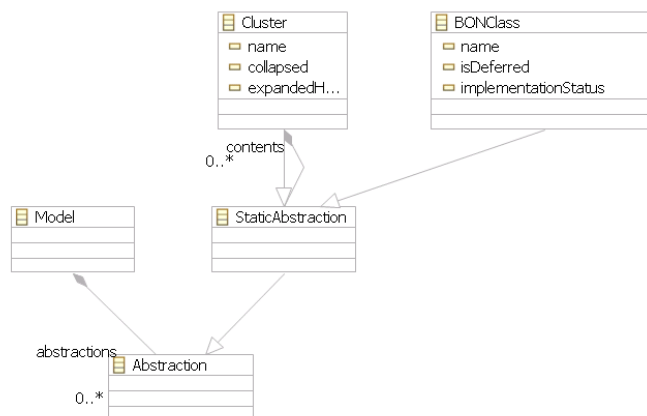

Figure 4.1: Ecore model of BON classes and clusters

There are two types of object oriented relationships between objects that are defined in a

BON specification: inheritance relations and client relations. Client relations are further defined as association, shared association, or aggregation relations [11].

Relations will be represented in the static diagram as links or connections between objects. The graphical display of association and shared association relations is identical, therefore we can simplify the model and represent both simply as association relations. Consequently, there three distinct visual styles: inheritance, association and aggregation. Figure 4.2 shows the Ecore model of BON relations.
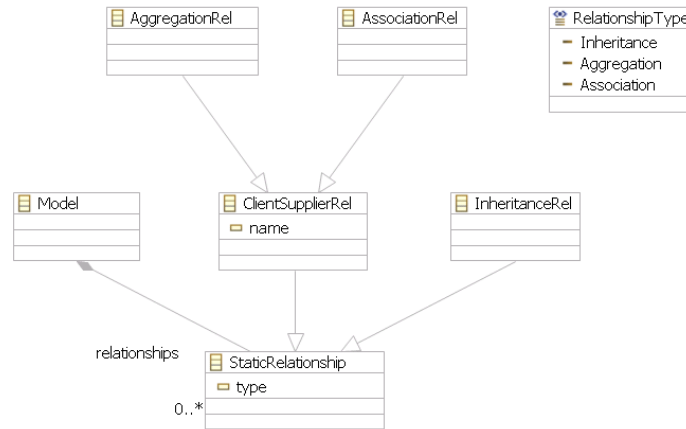

Figure 4.2: Ecore model of BON relations

4.1.1   EMF Code Generation

From the Ecore model, the EMF generator will generate Java interfaces and concrete implementations for the classes in the model. All generated classes implement the `EObject` interface, the base interface of the EMF reflective API. This allows EMF to provide a generic notification and persistence framework that will work with any model.

The Ecore model defines one EPackage, `bonIDE`. For the `bonIDE` EPackage, five Java packages are generated. Three packages contain classes that define the model and related utility classes: `bonIDE`, `bonIDE.impl`, and `bonIDE.util`.

For each `EClass` in the `EPackage`, an interface is generated in the base package, and a Java class that implements it is generated in the `impl` package. If the `EClass` inherits from another `EClass`, then the generated interface and implementation extend the interface and implementation generated for the base class type. For each attribute in the model, accessor methods are generated in the interface and implementation class. Attributes are resolved to the appropriate EMF types, for example `EString` or `EInteger` for single value types, or `EList` for multi-valued attributes. Method signatures are generated in the interface and implementation classes for each operation in the model.

The `bonIDE.provider` package contains adapter classes that convey model change notifications to clients, and the `bonIDE.presentation` package contains classes that use the adapters in `bonIDE.provider` to provide a basic view on the model.

A summary of the EMF generated packages and their contents are listed in table 4.1.

Package	Description	Example
bonIDE	Interfaces for model objects and the factory to create the Java classes.	Cluster, BONClass, Feature.
bonIDE.impl	Concrete implementation of the interfaces defined in the model.	ClusterImpl, BONClassImpl, FeatureImpl.
bonIDE.util	A factory class for creating adapters.	BonIDEAdapterFactory
bonIDE.provider	ItemProvider adapter classes that adapt the model for viewers.	ClusterItemProvider, BONClassItemProvider
bonIDE.presentation	Viewer classes.	BonIDEEditor, BonIDEActionBarContributor.

Table 4.1: EMF generated packages

4.2 GMF

4.2.1 Graphical Model

Elements from the metamodel that have a visual counterpart are defined in the graphical definition model. The top level element is a canvas, which contains two child elements: a figure gallery, and a collection of views, as seen in Appendix B.

A view is a node, connection, compartment or label that may be displayed in the diagram. Context appropriate behavioural properties are set for each view, such as resize constraints for nodes, or collapsible/non-collapsible behaviour for compartments. A view may correspond to a figure from the figure gallery, or in some instances, a child component of a figure. For example, the `BONClassView` corresponds to a `BONClassFigure`, but the `FeatureCompartment` view corresponds to `FeatureRectangleFigure`, which is a child of `BONClassFigure`. This reflects the fact that the compartment for features is contained within a rectangular region within the `BONClassFigure`.

The figure gallery defines in detail the appearance of each figure, including the shapes, lines, and labels that make up the figure, along with their color, line style and line weight. Layout information for figures that have child elements are also defined. Figure 4.3 shows the detail of the `FeatureFigure`, which contains as its direct children a `Rectangle` figure to contain the feature signature, a `Label` to display a comment, and three more `Rectangles` to hold lists of feature arguments, preconditions and postconditions. The `FeatureSignatureRectangleFigure` in turn contains child `Labels` to contain the component strings that comprise the feature signature.

4.2.2 Tooling

Nodes and connections that can be added to the diagram by the user will require a tool in the diagram palette and therefore an entry in the tooling definition model. Table 4.4 shows the list of tools defined.

4.2.3 Mapping

In the mapping model, the metamodel elements are linked with their graphic representations and tools. The mapping model also defines graphic containment relationships, which generally
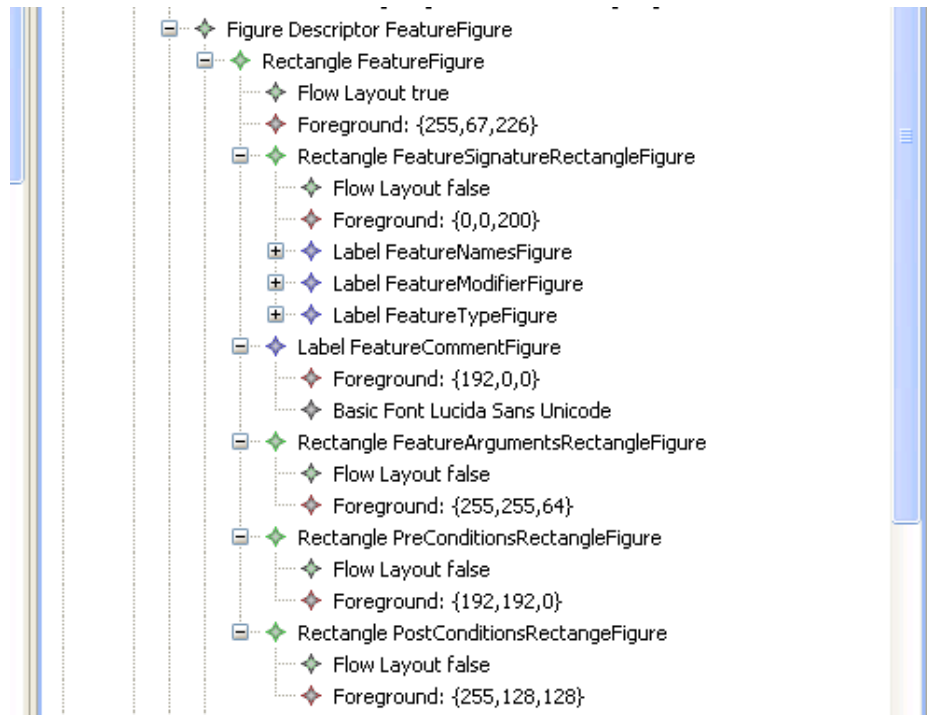
Figure 4.3: Detail of the graphical definition model

Cluster
BONClass
Feature
Index
Inheritance
Argument
PreCondition
PostCondition
Invariant
Inheritance
Aggregation
Association

Figure 4.4: Elements defined in the GMF tooling definition

correspond to parent/child relationships in the metamodel.

Figure 4.5 shows a portion of the mapping model and serves to illustrate the properties of several key elements, as follows:

The left pane shows the overall structure of the mapping model. At the top level, there are two `Node References` and three `Link Mappings` that represent clusters, classes and the three types of relationships that can be added to the diagram (inheritance, association, and aggregation). The `BONClass` node is expanded to show its child elements. The immediate child nodes of the `BONClass` node are a `Label` that serves to display the class name, four `Child References`, and four `Compartments`.

Each `Child Reference` maps to an element or list of elements that is aggregated both visually and semantically by `BONClass`, namely indexes, parent classes, class features and class invariants. The `Compartments` are mapped to regions within the overall class boundary, and help to layout the items pointed to by `Child References`. In addition, `Compartments` can

be collapsible, hiding the contained elements.

To right pane shows the property pages for a `Node Mapping`, a `Child Reference`, a `Feature Label` and a `Compartment Mapping`. The property pages reveal the practical detail of the mapping for each type.

For the `Node Mapping`, the `Element` property gives the mapping to the metamodel element, and the `Diagram Node` and `Tool` properties give the link to corresponding items in the Graphical Definition model and Tooling Definition model, respectively.

The `Child Reference` properties define the mapping to the metamodel element and the Graphical Definition model element (`Feature`/`Feature`), the `Compartment` that contains the `Features`, the model element in the parent that stores the `Feature` children, and the type of the children (`BONClass.Features:Feature`).

The properties of the `Feature Label` (note in this context 'Feature Label' is GMF nomenclature and is not to be confused with a BON feature), define the underlying metamodel textual element that the label displays, in this case a concatenation of the String collection `Feature.names`, and give the mapping to the Graphical Definition label.

Finally, the `Compartment` properties bind together the visual element that will define the boundaries of the compartment given in the Graphical Definition, and the `Child Reference` items that the compartment will contain, in this case the `Features` child reference.
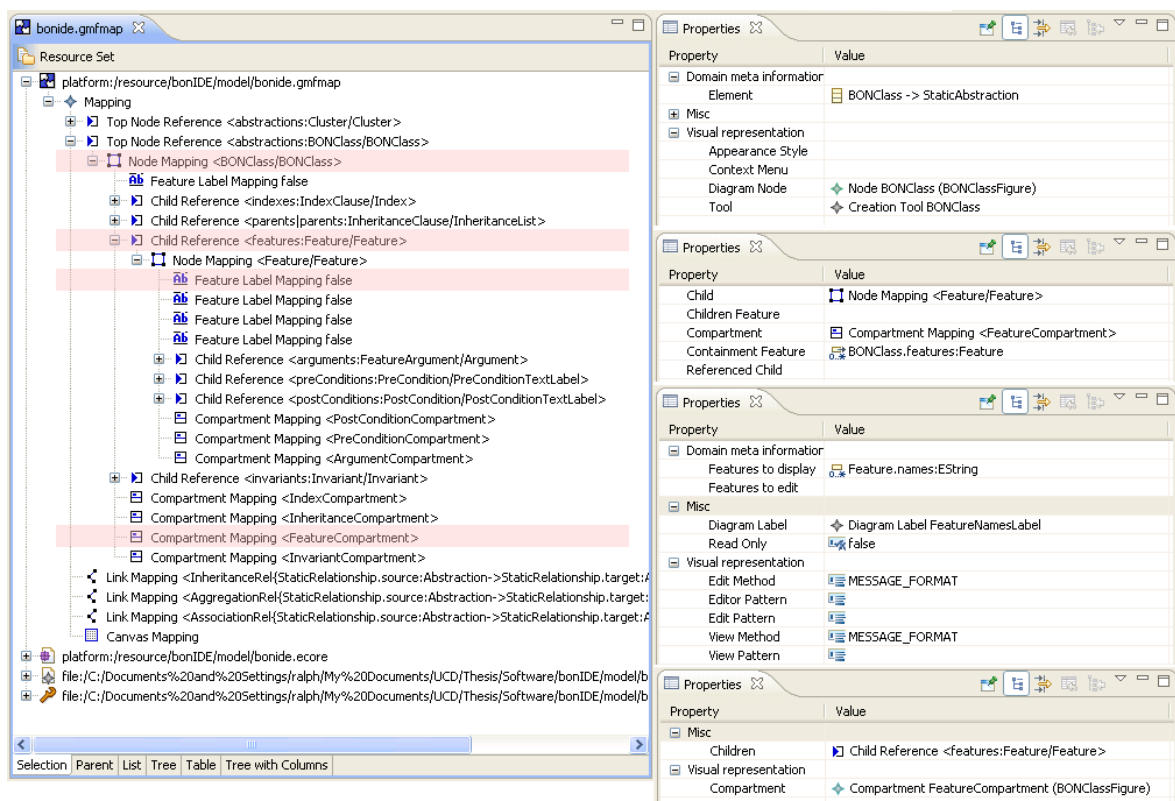

Figure 4.5: Detail of the GMF mapping model

Package	Contents
bonIDE.diagram.edit.commands	Undoable commands that are used to modify model elements.
bonIDE.diagram.edit.parts	Classes that represent the controller in the MVC architecture.
bonIDE.diagram.edit.policies	Classes that manage what can and cannot be performed on an Edit-Part.
bonIDE.diagram.parsers	Parser(s) for processing input from diagram labels.
bonIDE.diagram.part	Diagram-level features such as wizards and file handling.
bonIDE.diagram.preferences	Preference pages for printing and formatting.
bonIDE.diagram.providers	Factory providers that create classes like EditParts and Icons.
bonIDE.diagram.sheet	Diagram property page that shows properties for the selected element.

Table 4.2: GMF generated packages

4.3   Code Generation

4.3.1   GMF Code Generation

The GMF code generator creates a set of Java classes that build upon the EMF-generated classes (section 4.1.1) and provide a graphical interface based on the model. An overview of the generated packages and their contents is shown in table 4.2.

4.4   Code Customisation

The code generated by GMF provides a working graphical editor, but manual customisation of the code is required to implement some of the more BON-specific features. This section contains examples of generated code that was edited and code that was hand-written in its entirety.

4.4.1   Diagram Construction

To render a diagram from a BON file, the BON IDE invokes the BONc parser introduced in section 2.2.1. BONc builds an abstract syntax tree (AST) from BON code. Once built, the AST is queried to obtain collections of clusters and classes, and the details of each class.

To create the visual representations for elements in the AST, the corresponding model elements are constructed and the attributes of each model element are populated with data from the AST. Listing 4.1 shows the code to create a cluster model element and set its name attribute.

```
    private bonIDE.Cluster createBONClusterElement(Cluster ASTClusterNode) {
        ClusterImpl newCluster = (ClusterImpl) BonIDEFactoryImpl.eINSTANCE.createCluster();
        newCluster.setName(ASTClusterNode.getName());
        return (newCluster);
    }
```

<div align="center">Listing 4.1: Creating a model element</div>

Since the mappings between model and graphical elements have been defined, it is possible
to utilise the GMF command framework to create the appropriate graphical representation
for the model elements. Listing 4.2 shows the instantiation and execution of a command to
create a cluster, and its contained classes, on the diagram canvas.

```
private void createBONClusterAndContentsNotationView(bonIDE.Cluster clusterModel) {

    CreateViewRequest.ViewDescriptor viewDescriptor = new CreateViewRequest.ViewDescriptor(
        new EObjectAdapter(clusterModel),
        org.eclipse.gmf.runtime.notation.Node.class,
        BonideElementTypes.getElementType(ClusterEditPart.VISUAL_ID)).getSemanticHint(),
        true,
        getDiagramPreferencesHint());

        viewDescriptor.setPersisted(true);

        CreateViewRequest createViewRequest = new CreateViewRequest(viewDescriptor);
        Command createViewCommand = modelEP.getCommand(createViewRequest);

        getDiagramEditDomain().getDiagramCommandStack().execute(createViewCommand);
}
```

<div align="center">Listing 4.2: Creating a diagram element</div>

4.4.2 Optional Elements

GMF labels that show text in the diagram are, by default, fixed entities, and will take up
space in the diagram whether or not they have any content. This means that labels showing
Feature comments, for example, will appear as blank labels for features that do not have a
comment. These elements will upset the layout of the diagram. What is required is to remove
unwanted, blank, elements.

EditPart (section 3.3) classes are responsible for creating and maintaining diagram fig-
ures, and EditParts are notified of changes to the model, such as the deletion of an el-
ement's content. Therefore to manipulate the visibility if optional Feature elements, the
FeatureEditPart class is modified. Listing 4.3 shows the hideComponent method that is
added to FeatureEditPart. A corresponding showComponent method is also implemented
in the FeatureEditPart class.

```
/**
 * @generated NOT
 */
public void hideComponent(GraphicalEditPart component) {
    if (component instanceof FeatureCommentEditPart) {
        fFigureFeatureCommentFigure.setVisible(false);
    } else if (component instanceof FeaturePreConditionCompartmentEditPart) {
        fFigurePreConditionsRectangleFigure.setVisible(false);
    } else if (component instanceof FeaturePostConditionCompartmentEditPart) {
        fFigurePostConditionsRectangeFigure.setVisible(false);
```

```
    } else if (component instanceof FeatureArgumentCompartmentEditPart) {
        fFigureFeatureArgumentsRectangleFigure.setVisible(false);
    }
}
```

Listing 4.3: Hiding child elements of a feature

A new method, `adjustVisibility`, is added to each of the child `EditParts` of a feature, namely `FeatureCommentEditPart`, `FeatureArgumentEditPart`, `FeaturePreConditionEditPart` and `FeaturePostConditionEditPart`. Listing 4.4 shows the code of the `adjustVisibility` method.

A call to the `adjustVisility` method is added to the end if the generated `refresh` method for each child EditPart. When a child `EditPart` has its content updated, the `adjustVisibility` method will hide the component if it has no content.

```
protected void adjustVisibility() {
    if (this.getChildren().size() == 0) {
        FeatureEditPart parentFeature = (FeatureEditPart) this.getParent();
        parentFeature.getPrimaryShape().hideComponent(this);
    } else {
        FeatureEditPart parentFeature = (FeatureEditPart) this.getParent();
        parentFeature.getPrimaryShape().showComponent(this);
    }
}
```

Listing 4.4: Showing/hiding an optional element

4.5   Model Constraints

As described in section 3.5, the Check language is used to add constraints to the model. This section presents implementation examples of typical constraints.
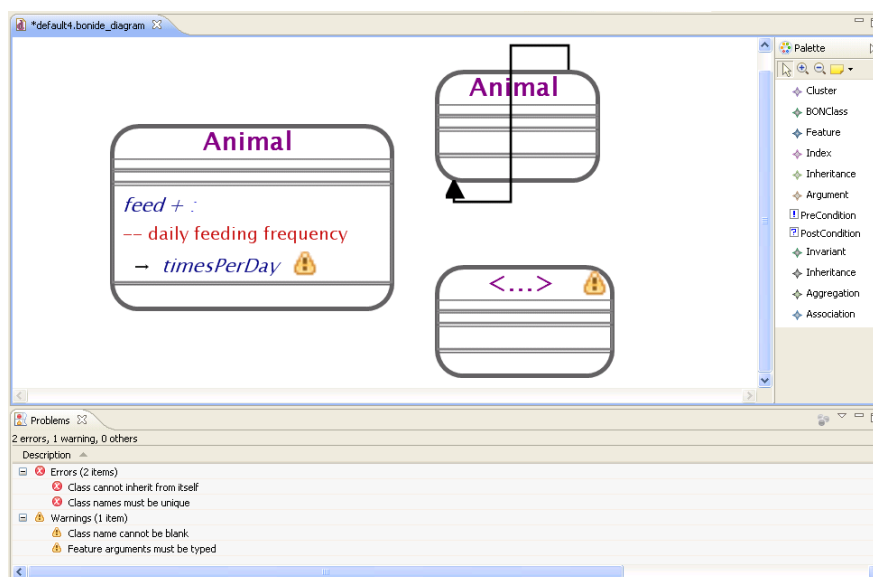

Figure 4.6: Validation errors in the UI

The syntax of a Check constraint is as follows:

```
context CLASS ERROR|WARNING "message string":
   predicate;
```

context is the metamodel class to which the constraint applies, for example, **Cluster** or **Model**. The severity of the constraint is specified as **ERROR** or **WARNING**. The **message string** is the text that appears on the Eclipse Problems pane if the constraint is violated. Finally, the **predicate** is the condition, specified by a Check expression, that is evaluated.

Listing 4.5 is the Check code for the constraint that class names in the BON specification cannot be blank.

```
context BONClass WARNING "Class name cannot be blank":
    name.length < 10;
```

Listing 4.5: Non-blank class name constraint

Listing 4.6 illustrates the **forAll** statement used on the collection of feature arguments to verify that all arguments are typed.

```
context BONClass WARNING "Feature arguments must be typed":
    features.arguments.forAll(arg|arg.type != "");
```

Listing 4.6: Feature arguments type constraint

Listing 4.7 validates that inheritance relations do not have the same **source** and **target**, that is, a class cannot inherit from itself.

```
context Model ERROR "Class cannot inherit from itself":
    !relationships.typeSelect(InheritanceRel)
        .exists(relation| (relation.source == relation.target));
```

Listing 4.7: Inheritance constraint

Listing 4.8 uses the **typeSelect** statement to restrict the context of the constraint to a subclass of the **Model.abstractions** collection. The **Model.abstractions** collection contains both **BONClasses** and **Clusters**, but the constraint is applied to a subset of **Model.abstractions** that contains only **BONClasses**.

```
context Model ERROR "Class names must be unique":
    abstractions.typeSelect(BONClass)
        .forAll(class1|!abstractions.typeSelect(BONClass)
            .exists(class2| (class1 != class2) && (class1.name == class2.name)));
```

Listing 4.8: Unique class names constraint

Chapter 5: **Conclusions**

───────────────

This project has a cyclical aspect, in that model driven design techniques were employed to build a tool that promotes the use of BON, which is itself a tool for MDD. The work demonstrates the practicality of MDD, and the need for tools like the BON IDE.

The MDD approach has enormous benefits. EMF and GMF and can significantly reduce the development effort required to design and build a graphical editor by allowing the designer to concentrate efforts on the problem domain instead of on implementation details. In particular, the well integrated nature of the main frameworks - Eclipse, EMF, GMF and M2T - eases the development workload by maximising the benefit of automated code generation while minimising the impact of changes to the design.

An issue highlighted by this project is that automatic code generation frameworks, like the modelling frameworks that invoke them, are necessarily generic, since they strive to be relevant across a range of problem domains. Although the design may be elegant, this generic nature can result in generated software that is overly complex, and difficult to modify and maintain. Whilst more time consuming, a hand-written solution may produce software with a smaller and simpler codebase. The BON principle of reversibility should help to combat this issue, by completing the loop between design and implementation and feeding back implementation details into the design.

5.1 Future Work

To achieve complete synchronicity between the textual and graphical versions of BON notation, the functionality to merge diagram changes back into textual BON is required. The BON IDE currently provides the tools to create BON notation elements such as clusters, classes, and features, and to edit their properties using standard Eclipse property sheets. But, due to time constraints, the mechanism to convey the changes to the textual format was not implemented. Such a mechanism could leverage the existing GMF model changes notification framework to capture visual edits, apply the changes to the existing abstract syntax tree, and propagate the results to the textual BON plug-in.
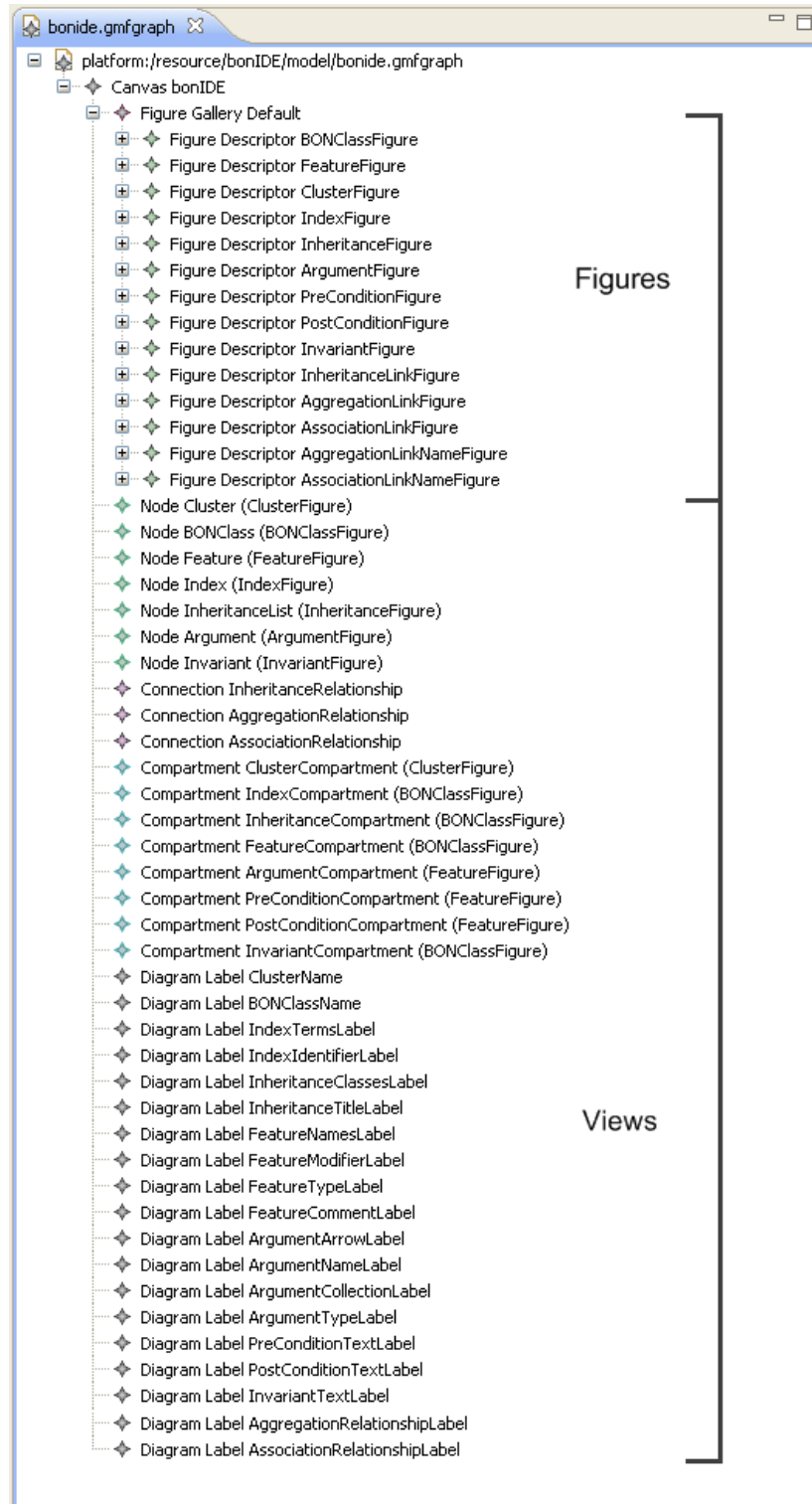
Bibliography

[1] Eclipse modeling framework project. `http://www.eclipse.org/modeling/emf/`.

[2] Graphical editing framework. `http://www.eclipse.org/gef/`.

[3] Graphical modeling framework. `http://www.eclipse.org/modeling/gmf/`.

[4] Model to text. `http://www.eclipse.org/modeling/m2t/`.

[5] Michael Azoff. The benefits of model driven development, 2008.

[6] Fintan Fairmichael. The BONc Tool homepage. `http://kind.ucd.ie/products/opensource/BONc/`.

[7] Jason Lancaric. The homepage of the bon case tool. `http://www.cse.yorku.ca/~eiffel/bon_case_tool/`.

[8] Bertrand Meyer. Applying design by contract. 1992.

[9] Richard F. Paige and Jonathan S. Ostroff. Precise and formal metamodeling with the business object notation and pvs, 2000.

[10] Bran Selic. The pragmatics of model-driven development, 2003.

[11] Kim Walden and Jean Marc Nerson. *Seamless Object-oriented Software Architecture: Analysis and Design of Reliable Systems*. Prentice-Hall, United Kingdom, 1994.

Appendix A: **BON Ecore Model**
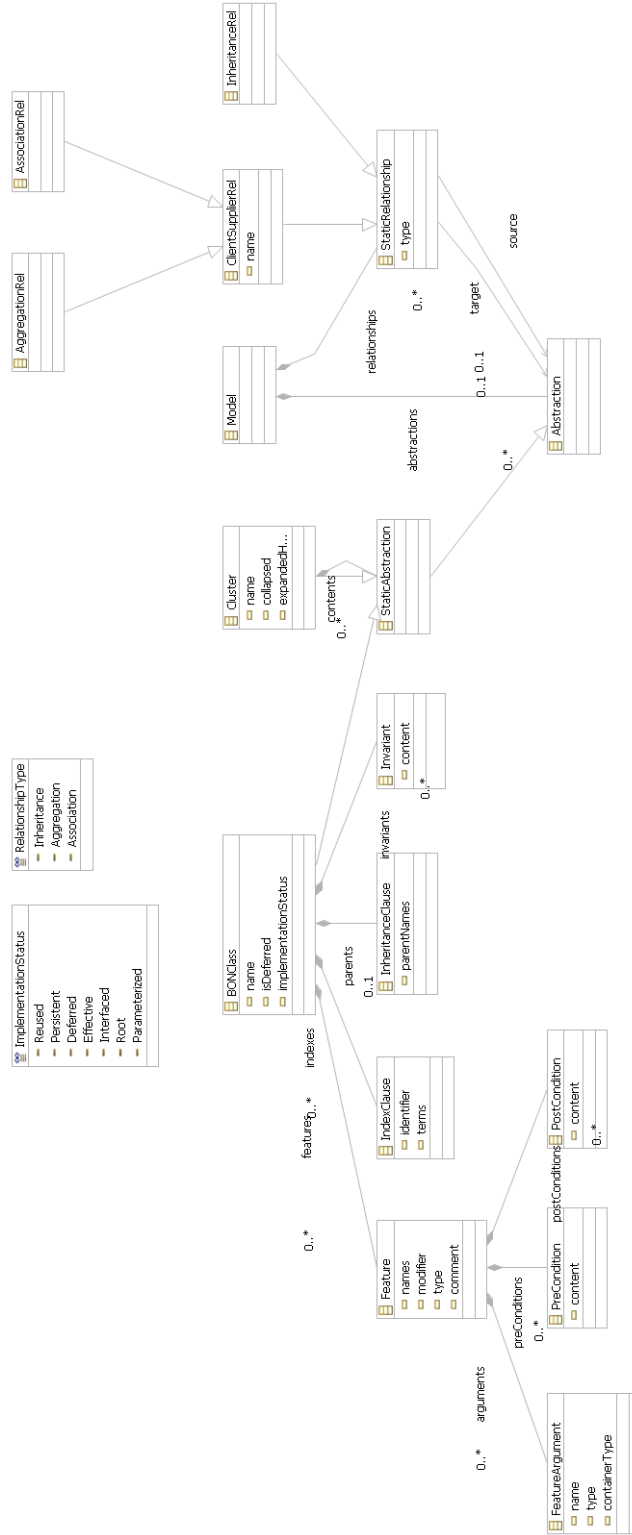
Appendix B: **GMF Graphical Definition Model**

Appendix C: **GMF Mapping Model**