Bridging Technical Spaces: Model Translation from TA to XMI and Back Again

by

Kristina Diew Hildebrand

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2006

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

There are many different techniques and notations for extracting architecturally interesting information from the source code of existing software systems. This process is known as reverse engineering. One current problem with reverse engineering techniques is that models of software systems cannot easily be transferred from one notation and storage format to another. We refer to this as the problem of bridging *technical spaces*.

In this work, we approach the issue of bridging between the SWAG technical space and the UML technical space. The SWAG technical space, named after the Software Architecture Group at the University of Waterloo, consists of fact extractors, fact manipulators, schemas, and a fact storage language - the Tuple-Attribute language (TA). The UML technical space consists of the UML metamodel, the XML Metadata Interchange (XMI) format for encoding UML models, and various UML modeling tools. We have designed and implemented a plugin for MagicDraw UML, which will import, export, and merge between XMI-encoded UML models and TA-encoded Function-Level Schema models.

We document evidence of what is referred to as a *bridge domain* - a technical space which exists between two encodable spaces. The metamodels of the two notation languages that we have focused on are very rich and flexible, but neither technical space is capable of fully expressing an accurate architectural model of any given software system; however, each technical space is capable of maintaining certain semantic information relevant to that technical space through multiple merge operations.

Acknowledgements

I would like to thank my supervisor, Andrew Malton, for his guidance throughout the process of completing this work, and for helping to keep me on track. Thank you also to my readers, Steve MacDonald and Krzysztof Czarnecki, for their feedback and suggestions for my work.

In addition, I want to express my gratitude to my colleagues in the Software Architecture Group, for their support and for making my time here more enjoyable. I'd particularly like to thank Jingwei Wu for providing me with the QL source code and for his help with learning how to use it.

Special thanks to Robert Munsch, for his input on the problem of expressing the same concepts within different languages and cultures.

I would also like to thank my parents for encouraging me throughout my studies, and for helping me get to the point where I could enter the MMath Program at the University of Waterloo.

Thanks also to the Natural Sciences and Engineering Research Council of Canada, for the financial support to pursue a Master's degree, in the form of a Canada Graduate Scholarship.

Last but certainly not least, I would like to thank Jason Taylor for his love and support over the past three years, and his promise to continue to do the same for the rest of our lives.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Sharing information between two cultures is frequently a difficult problem – this is true in human cultures, as well as the "cultures" of different software model storage formats. Robert Munsch's story *A Promise Is a Promise* [37], published in both English and Inuktitut, exemplifies the difficulties associated with adaptation required for different cultures. Munsch describes the cultural issues that arose when writing this story: "I tried to make it into a story that would work in both Inuit and Southern worlds" [36].

The story is about an Inuit girl, Allashua, who defies her mother and goes down to the beach and onto the sea ice. She is caught by the *qallupilluit*, a mythical Inuit creature that lives among the ridges and cracks in the ice on Hudson Bay. She is released but only after promising to return with her brothers and sisters. Michael Kusugak wrote the core of the story, which describes a child's encounter with the qallupilluit. The story was later modified by Munsch to better suit the various target Canadian cultures. The original version has several children narrowly escaping the sea creature after being told by their

mothers not to go near the ice. It was a simple cautionary tale about the dangers of the sea ice [31]. Such a moral would have less impact on children in many other parts of Canada since they do not encounter sea ice in their daily lives. Munsch modified the story, with the promise made by Allashua to the qallupilluit, subsequently making a broken promise the central theme. "The big problem with my version was that Alashua was impolite to the Quallupilluit. Kids of Michael's generation were always polite to Elders, even if the Elder was going to kill them" [36].

As we can see in Munsch's story and its need to be adapted, some concepts are not easily expressed within a given culture or language. In addition, not all concepts are relevant to every culture. In software engineering, we refer to this as the problem of bridging between *technical spaces*. This term refers to the notation, schema, domain, and tools with which a programmer or software architect is working. Translating or merging information between two technical spaces is a difficult but necessary task and thus motivates our research.

1.1 State of the Art

Abstraction is the act of creating a representation of the most important features of a set of complex information and "is the primary way we as humans deal with complexity" [3]. Abstraction is required in the field of software engineering because it helps analysts design the high-level architecture of a new system, aids new developers in understanding an existing system when they join a development team, and allows maintainers of legacy systems to understand the structure and behaviour of a system that they may need to modify. In *Object Oriented Design with Applications*, Grady Booch refers to Miller's ex-

periments which show that an average person is capable of understanding between five and nine pieces of information at a time [4]. This applies to any type of information, including the components of a software system. We can aid comprehension by organising complex information into visual categories and hierarchies, limiting the amount of information that must be understood at any moment.

There are many different techniques for abstracting information about complex software systems. The result of applying these abstraction techniques is a model of the software being investigated. Along with this variety of techniques comes a wide variety of notation languages, model storage formats, model interchange languages, and visualisation techniques. Some of the languages used for storing models about software include Entity-Relationship models [9], XML Metadata Interchange (XMI) [41], the Tuple-Attribute Language (TA) [21], and general-purpose graph description languages, such as GXL [23] and DOT [29]. Each of these storage languages generally has its own metamodel and visualisation techniques. For example, XMI is used to record UML models, and GXL is intended as a graph description language.

Visualisations of software are useful because they provide a high-level description of a complex system and allow a developer to see the important parts of the software at a glance. This can improve a developer's understanding of the software. These visualisations can be created by extracting information from the source code of existing pieces of software. This is called *reverse engineering*, and is a very active field of research within software engineering. There are currently several well-developed and well-tested reverse engineering tools and techniques, including a wide variety of commercial reverse engineering tools for UML [5] [25] [39], and tools and techniques developed by researchers at various academic

institutions. For example, researchers at the University of Victoria have developed a suite of tools, called Rigi, which includes reverse engineering tools, a graph model, a scripting language for manipulating the graphs, and a rigiedit, a graph editor [35]. The Software Architecture Group (SWAG) at the University of Waterloo, has developed a similar suite of tools, hereafter referred to as SWAGkit [45]. The usual output from using SWAGkit is one or more files in the TA format, conforming to the *Function Level Schema* (FLS). TA is a language for recording information about certain types of graphs, and the Function Level Schema is a schema created by SWAG to give meaning to graphs about the static structure of software systems. Using this schema, we can record details about relationships between various entities in a piece of software. These TA files can be viewed in LSedit, a software landscape editor and visualisation tool and also can be manipulated using a relational algebra calculator known as *grok*.

The Unified Modeling Language (UML) is a standardised notation language, primarily intended for designing object-oriented software systems. It is also useful for architectural description [19]. The Object Management Group (OMG) is a consortium that maintains computer industry standards for software development. The OMG has adopted XMI as a standard for recording UML models. One of the primary benefits of UML is that it is the industry standard notation for designing object-oriented software systems. When the various participants in the software development process use a common language, they are able to communicate better. As a standard, UML can facilitate communication between different stakeholders in the software design, development and maintenance processes. We are interested in producing XMI-encoded UML models from information extracted using SWAGkit and the reverse, producing input for the SWAG tools from XMI-encoded UML

models. This process will require us to provide the ability to import and export both types of models, as well as the ability to merge information between existing models.

Many techniques have been developed to bring together information from multiple stakeholders and merge different types of information. In any development group with more than two members, different developers may be working on multiple versions of source code concurrently. When they commit changes to a central source code repository, a correct merge must take place, since possible conflicts may arise between the different versions of the code. Several techniques have been developed by various researchers to deal with these conflicts. Zündorf *et al.* [55] have developed an object structure based technique for merging the source code of object-oriented systems that overcomes many of the problems associated with text-based merging. Malton *et al.* [34] developed a technique for factoring source code into different streams of text, such as code and comments, then merging them back together after making changes to the individual streams. Westfechtel [53] has developed a technique for merging information from various types of software documents, such as source code and design documents. This system uses an interface to each document type, allowing new document types to be easily added to the system.

There has also been a lot of research into developing techniques for *Round-Trip Engineering* in recent years. Round-trip engineering refers to the process of turning a design into source code then extracting the design, possibly after the source source code has been modified [38]. FUJABA [28], developed by researchers at the University of Paderborn, is an example of one such system. FUJABA, short for "From UML to Java And Back Again," consists of tools for developing a model in UML, exporting the model to Java source code, and then recreating the UML model from modified source code [28]. Some of the decisions

involved in the design of FUJABA are similar to those that were involved in our work.

1.2 Motivation

Different modeling tools are useful for a wide variety of different purposes. A developer may wish to sketch out a general design before committing it to a model or a particular notation language. Simple graph-drawing or diagramming tools, such as OmniGraffle [18], Microsoft Visio [11], and Dia [32] are good for these purposes. A developer may later wish to extract information about the design from existing source code, manipulate it, obtain more detailed information about interactions between the various components, and explore a visualisation of the software. These purposes would be well-served by tools similar to those available in SWAGkit [45], such as fact extractors (BFX [50], LDX [51], CPPX [22]), relational calculators (grok [46], QL [47]), and software landscape editors (LSedit [48]). A developer may also wish to view and edit the design of their software in a tool with a standardised notation language, pattern-matching capabilities, the ability to export to source code, and the ability to interface with a variety of other development tools; tools such as MagicDraw UML [39] and Rational Software Architect [25] may serve these purposes. Each of these sets of tasks require the information about a software system to be available within different technical spaces.

In the specific case we approach with this work, we are working with the UML technical space and the SWAG technical space. Both of these spaces will be described in detail in Chapter 2. If a user starts with a UML model, it would be useful to be able to use the tools within SWAGkit to perform queries on the facts in the model and to manipulate the

model using one of the relational algebra calculators. On the other hand, if the model is being extracted from source code, an architect could use the SWAG tools to extract the facts about it then produce a set of UML diagrams documenting the system in a standard modeling language.

A software system can be described using a model that conforms to one of many different metamodels. The important information about the software – that is, the model – can be encoded in a language that relies on a particular metamodel to give the model meaning. This model can then be visualised in a tool designed to understand the particular metamodel and encoding language. Figure 1.1 shows the metamodels, encoding, and visualisation tools for three technical spaces. These features, represented as boxes in this diagram, show the various components of a technical space. In each box, we have shown specific tools and languages for the UML space, the SWAG space, and the Rigi space.

The UML was originally intended for designing object-oriented systems but can be used to model the architecture of a system that was not designed with object-oriented principles in mind. SWAGkit currently uses a non-standard notation for describing software architecture, and for this reason we are interested in making it possible to move this model information between different technical spaces. Specifically, we aim to be able to move model information between the SWAG and the UML technical spaces. Ideally, it should be possible to work in multiple technical spaces at once and have any changes made in one technical space be reflected in the other corresponding spaces.

We believe that a system can be represented in several technical spaces at once, and it is possible to translate between these technical spaces. No one technical space fully reflects the actual software, and each technical space provides different views of the software system.

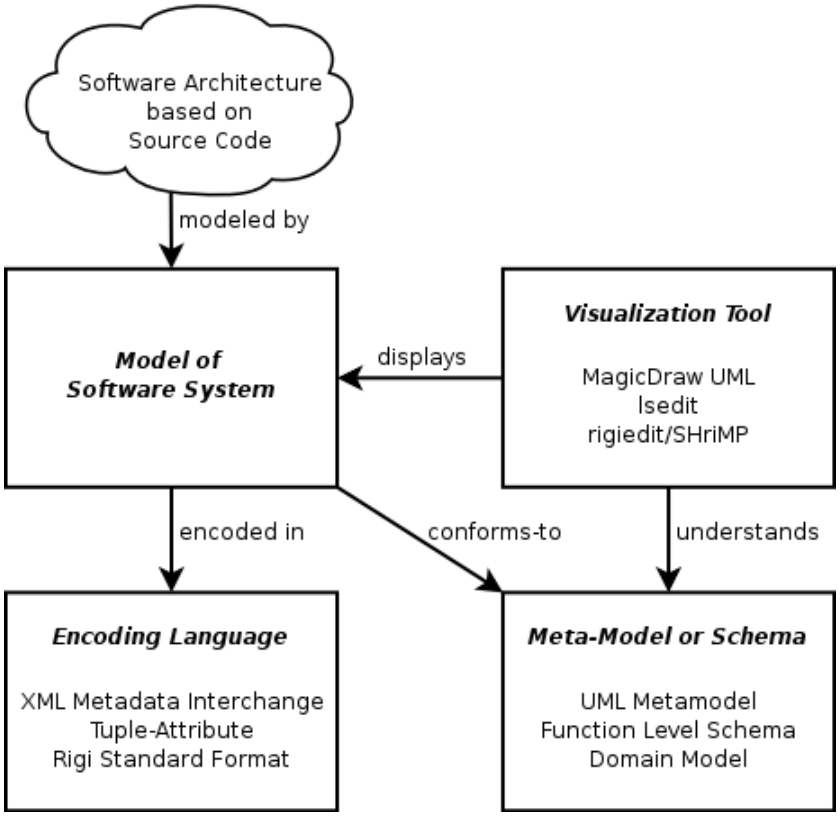
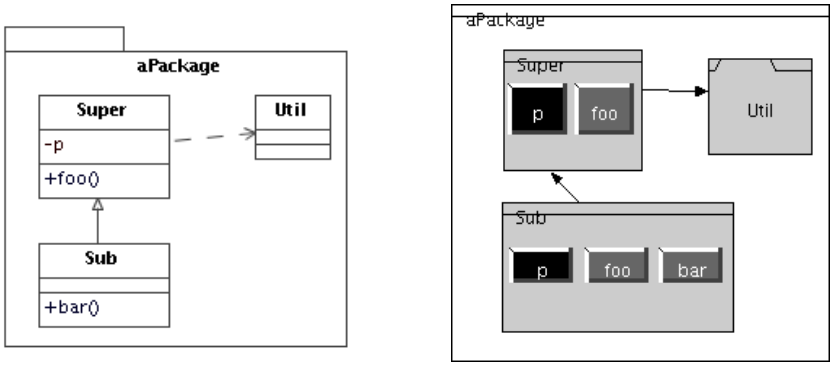


Figure 1.1: Relationships between elements of a Technical Space



(a) UML Class Diagram (b) LSedit View

Figure 1.2: A simple translation example from UML to LSedit

We will describe the process for moving models between UML's standard interchange format, XMI, and TA – the format required by SWAG's architectural tools.

1.3 An Example

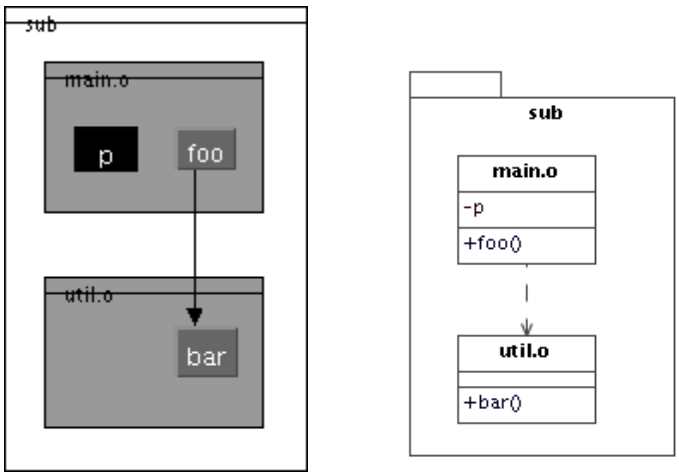
In this section, we will discuss two simple examples. Figure 1.2 shows an example of a UML diagram and its LSedit equivalent. This diagram was created in MagicDraw UML, then was exported to TA, using our tool. The TA is shown below. This segment of TA uses the function-level schema, which will be described in detail in Chapter 3.

```
FACT TUPLE :
$INSTANCE /aPackage/Super/foo cFunction
$INSTANCE /aPackage/Sub/foo cFunction
$INSTANCE /aPackage/Sub/bar cFunction
$INSTANCE /aPackage/Super cObjectFile
$INSTANCE /aPackage/Util cObjectFile
$INSTANCE /aPackage/Sub cObjectFile
```

```
$INSTANCE /aPackage/Super/p cObject
$INSTANCE /aPackage/Sub/p cObject
$INSTANCE /aPackage cSubSystem
contain /aPackage /aPackage/Super
contain /aPackage /aPackage/Util
contain /aPackage /aPackage/Sub
contain /aPackage/Super /aPackage/Super/p
contain /aPackage/Super /aPackage/Super/foo
contain /aPackage/Sub /aPackage/Sub/p
contain /aPackage/Sub /aPackage/Sub/foo
contain /aPackage/Sub /aPackage/Sub/bar
cLinks /aPackage/Super /aPackage/Util
cLinks /aPackage/Sub /aPackage/Super
```

The Function Level Schema for TA does not have the capability of recording different types of relationships, other than containment and cLinks. Other schemas can be developed for TA, but we have chosen to use this one, as it is the most commonly used within SWAGkit. As demonstrated in Figure 1.2, when a UML-based generalization is exported to the Function Level Schema, the internal elements of the superclasses are copied to the subclass. This is done in order to preserve the semantics of this relationship. Translations such as this one must be created, because the metamodels of these two technical spaces have different sets of features. In this example, `p` is a member variable of the class `Super`, `foo` is a function in this same class, and `bar` is a function defined only in the class `Sub`. The translation copies the variables and functions of the superclass into the subclass, then defines a cLinks relationship between the two classes – or cObjectFiles. The packages become SubSystems.

Figure 1.3 shows a slightly different example. In this case, the example model starts in LSedit, and we have imported it using our tool into a MagicDraw UML model. The



(a) LSedit Software Visualisation

(b) Imported UML Diagram

Figure 1.3: A simple translation example from LSedit to UML

main difference between these two views of the system is the position of the dependency between foo and bar. In the LSedit diagram, the links relationship exists between individual functions. In the MagicDraw UML version of this model, the dependency exists between the classes, rather than the individual operations.

1.4 Contributions

Our primary goal was to create a method for producing UML diagrams from TA-based factbases. To this end, we have provided a tool for translating models from the TA language used by SWAG to the XMI storage format for UML models. We have focused on the package and class diagrams of UML since they are the most commonly used and the most useful for documenting software architecture [19]. Further, we have contributed to the

understanding of the issues involved in bridging between technical spaces. We now believe that neither technical space fully reflects reality, resulting in the concept of a *bridge domain*. This bridge domain exists between the technical spaces being bridged. It can be thought of as a mental-only model, or as the reality reflected in the actual implementation of the software system. Each modeling notation has its own features that contribute to the bridge domain, but none fully capture it.

We wish to promote the use of well-developed and well-researched extraction and manipulation tools such as those found in SWAGkit. These tools include a variety of fact extractors, a relational calculator, and a software landscape navigator and editor. The techniques developed by SWAG have been validated with numerous case studies and have proven to be quite useful in studying the architecture of a wide variety of software systems [6] [54]. We believe that these tools would enjoy more widespread use in industry if they could produce visualisations in UML, a modeling language widely recognised as the industry standard.

We eventually hope to be able to work in any technical space and have the changes made by the developer or architect in one space be reflected in a meaningful way in all other spaces. While moving toward this goal, we have created a tool for moving model information back and forth between TA and XMI. We have also documented the process for building these tools, which will allow others to follow our procedure for other modeling notations.

1.5 Structure of Thesis

We begin this thesis with some background on the Unified Modeling Language (UML), XML Metadata Interchange (XMI), the Function Level Schema (FLS) and the Tuple Attribute Language (TA), in Chapter 2. These technologies make up the two technical spaces that are relevant to our research. We will then describe our overall approach to bridging these two domains, in Chapter 3. Chapter 4 will discuss the concept of a *Bridge Domain* – that is, a technical space that exists between the two encodable spaces. Finally, Chapter 5 summarises our contributions, and describes some areas for future work related to our research.

Chapter 2

Background

We will frequently use the term *technical space*, which refers to the notation, schema, domain, and tool with which a programmer or architect is working. A technical space defines the framework in which a program understanding task is being accomplished [49]. In our work, the two main technical spaces we have studied are: the XMI format for storing UML models, and the Tuple-Attribute language (TA). TA is used for storing the Function-Level Schema (FLS) information extracted using the tools developed by the Software Architecture Group (SWAG) at the University of Waterloo.

Any data format should provide and conform to a schema or meta-model in order to give structure to the data. The data is given meaning through the documentation of the schema or meta-model. Schemas can be internal or external, and implicit or explicit [27]. In TA, schemes are explicit and external; that is, they are explicitly defined outside of the data being recorded. TA's schemes are usually contained within the same file as the data but in a separate section from the actual data. While working in the TA technical

space, we will be restricting ourselves to the FLS, the most commonly used schema in this space. This schema and the TA language in general will be described in more detail in Sections 2.2.1 and 2.2.2. The metamodel for UML is also explicit and external, but is defined separately from the data, in the specification documents produced by the OMG. The UML meta-model is very large; we will only concentrate on a subset of it, which will be described in more detail in Section 2.1.2.

This chapter will provide detailed background information for the reader. We begin with a discussion of the UML and the UML-based tool we have chosen to work with. This is followed by details on the SWAG technical space, including the tools involved, the TA language, and the FLS. We conclude this chapter with some discussion of related research on merging sets of related data and some techniques for promoting software engineering tool interoperability.

2.1 The UML Technical Space

The UML "is a family of graphical notations, backed by single meta-model, that help in describing and designing [...] software systems built using the object-oriented (OO) style" [16]. The UML has also successfully been used to describe software architectures in the past. It works particularly well for communicating the static structure of software. In *Describing Software Architecture with UML*, Hofmeister *et al.* argue that a conceptual view, a module view, and a code view can be readily described using standard UML diagrams, such as Class & Object Diagrams, Package Diagrams, and Component Diagrams [20]. In *Using the UML for Architectural Description* [19], Hilliard describes how the UML can be used

within the context of the IEEE *Recommended Practice for Architectural Description*. In this paper, Hilliard points out that it has become standard practice to use multiple views to describe software architecture. This need for multiple, consistent views is satisfied by the different types of diagrams defined by the UML.

Throughout the history of software modeling techniques, there have been many different decomposition and abstraction approaches developed and used. These historical approaches correspond to several of the views of software architecture proposed by Kruchten in *The 4+1 View Model of Software Architecture* [30]. The *logical view* describes the object model of the software and is concerned with the data associated with the software. This view of software systems can be documented using the data-based Entity-Relationship diagrams proposed in 1976, by Peter P. Chen [9]. The *process view* describes the concurrency and synchronisation aspects of the software system, which could be described, in part, using the Jackson System Development specifications proposed by Cameron in 1985 [8]. The *physical view* describes the way in which the software is mapped onto the hardware. This view has does not have a historical notation designed specifically for it, though simple network diagrams perform this task well. The *development view* describes the static organisation of the software, as the developers see it. This view can be encoded using the approach developed by SWAG, performing static architectural analysis then producing a boxes-and-arrows diagram of the static relationships between compilation units.

The UML provides standard notations for each of these types of views, as well as many others. The UML symbols and notations used for the representation of these views have evolved out of their historical ancestors. For example, a deployment diagram shows the physical view of a system. A package diagram provides the development view. The

process view can be obtained from a component diagram annotated with thread and process stereotypes. The logical view can be described using a class diagram, since classes are the fundamental data structure in object-oriented systems. The UML also introduces a notation for use cases, satisfying the *+1* scenario view described by Kruchten.

2.1.1 XML Metadata Interchange (XMI)

XMI was designed to allow software engineers to easily exchange model metadata between software tools. It was primarily developed by the OMG as an interchange language for UML models, but can be used for any model information that can be described using *Meta-Object Facility (MOF)*. MOF is another OMG standard, intended to support Model Driven Engineering. Full XMI support within UML modeling tools is not currently very widespread – this may be due to the significant changes that have been made between the 1.X standards and the 2.X series of standards, the latest of which was released in September of 2005 [41]. The more popular UML tools partially support XMI, at least for the more common and straightforward features of the UML, which are all that is required for our work.

2.1.2 UML Metamodel

We will describe the subset of the UML metamodel that is relevant to our work. The diagrams in this section use the UML class diagram notation and come from the UML 2.0 Infrastructure document [17].

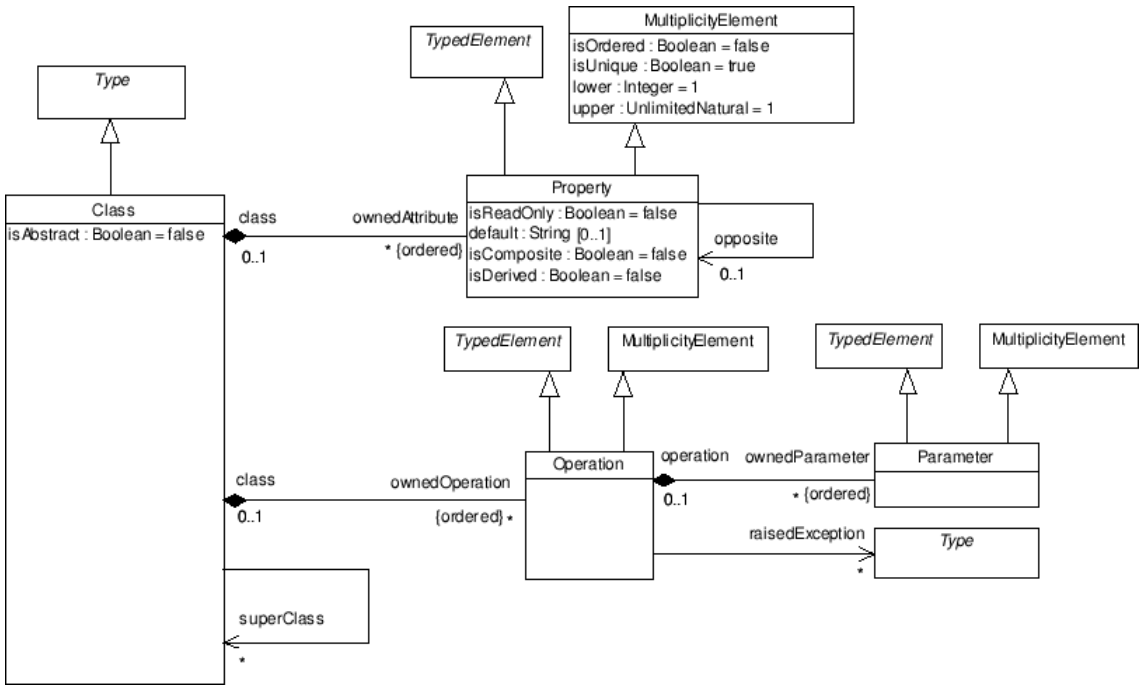


Figure 2.1: UML Class Metamodel [17]

Class

Figure 2.1 shows the elements that can be related to a Class in UML. Classes are Types that are made up of Properties and Operations and occasionally other Classes. An Operation is analogous to a function, which is performed using the data stored in the properties of the Class. The Properties are basically variables that may have different values for different instances of the class. Classes can have relationships with other Classes, as outlined below.

Package

Figure 2.2 shows the metamodel for UML Packages. Packages can contain Types and other

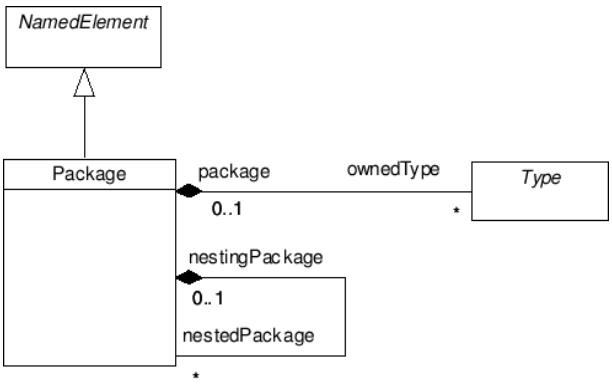


Figure 2.2: UML Package Metamodel [17]

Packages. As NamedElements, they can also be related to other model elements.

Relationships

Figure 2.3 shows the metamodel for UML Relationships. Relationships are associated with one or more elements, such as Classes and Packages. Generalization, Association, and Dependency are all types of Relationships and are the Relationships that we will focus on throughout this work. Other types of Relationships include Composition and Aggregation. Generalization is a relationship which means that a subclass is an instance of a superclass. This relationship is designated by a line with an empty arrowhead pointing to the superclass. An Association is designated by a solid line, possibly with arrows, labels, and multiplicities at either end. An Association has no strict implementation in object-oriented Programming languages. Dependencies are designated by a dashed arrow, pointing from one model entity to another, towards the entity that is depended upon.

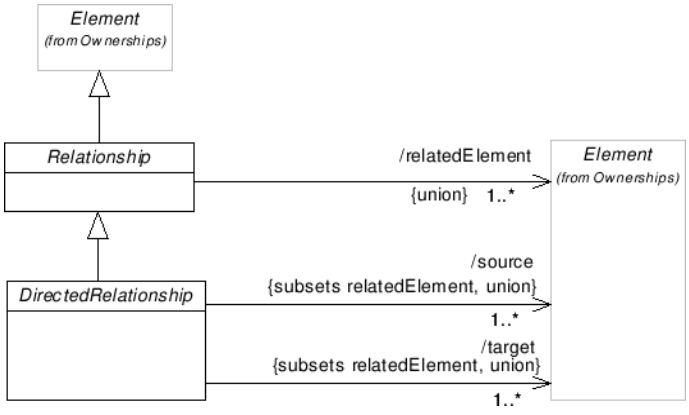


Figure 2.3: UML Relationships Metamodel [17]

2.1.3 UML Diagrams

UML defines several types of diagrams, many of which do not require the programmer to think in an object-oriented manner. UML diagrams are made up of entities and relationships between the entities. The entities include: classes, packages, objects, and components, among others, whereas the relationships include: dependencies, messages, state transitions, associations, and generalization. Each different type of UML diagram is composed of different subsets of these diagram elements. For example, a Class Diagram can contain classes, packages, and a variety of relationships. All of the UML diagrams may contain comments attached to any model element.

Static Diagrams

The Class, Package and Component diagrams defined by the UML specification are the most useful for software architecture description [20]. These diagrams most readily represent static structure, which is the structure extracted using the SWAG tools. We will focus

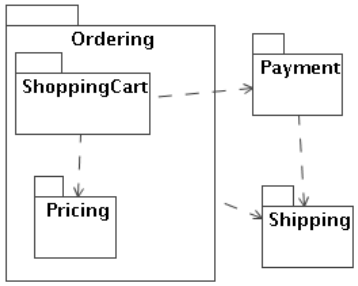


Figure 2.4: UML Package Diagram

on the Class and Package diagrams, as these are sufficient for representing the information available in the FLS-based factbases extracted by SWAGkit.

Package diagrams can help identify the higher-level dependencies in an application. These diagrams basically show the packages in a software system, as well as the relationships between those packages. Normally, these relationships will be determined based on the relationships between the contents of the packages then lifted to the package level. Figure 2.4 is an example of a very simple UML Package Diagram. It shows two packages (Ordering and Payment) that depend on the Shipping package, ShoppingCart depends on Payment and Pricing, and Pricing and ShoppingCart are sub-packages of Ordering.

Figure 2.5 is an example of a UML Class Diagram. This is far from being a complete specification for any system, but is used here to demonstrate some of the features of Class Diagrams. On the left side of the diagram is a generalization hierarchy. Payments can be of two types: Cash or Credit. Each Order is associated with any number of Payments and is composed of one or more OrderLines. At the OrderLine end of this Composition relationship there is a role name, lineitems. This notation is used on associations to give more details about the relationship. On the far right of this diagram is the class Customer,

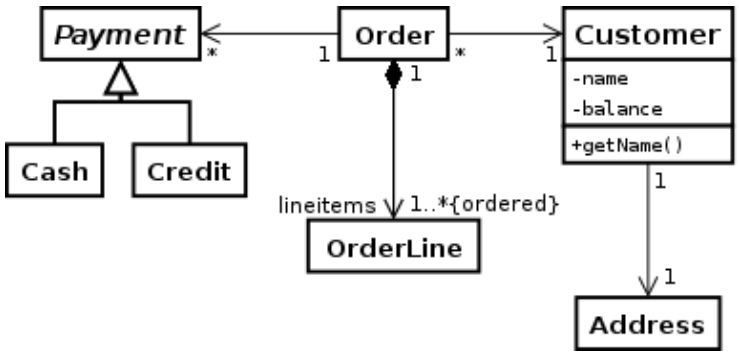


Figure 2.5: UML Class Diagram

which has a one-to-one association with an Address. Customers have the properties name and balance, and the operation getName().

Dynamic Diagrams

The UML also defines a number of diagrams that are not about static structure, but rather about behaviour of the software system. Recent activities [13] [7] [52] [43] within the software engineering research community have been aimed at extracting and understanding dynamic software architecture information, and therefore it would be useful to consider the ability to put this information into a standard visualisation format such as UML. Specific dynamic interaction traces can be documented using UML's sequence diagrams. Figure 2.6 is an example of a sequence diagram. An instance of the class Order sends getQuantity and getProduct messages to each OrderLine. The message getPricingDetails is sent to a Product, then the Order object makes a method call to itself to calculate the price of the order. A similar notation could be used for describing non-object-oriented sequences of events, using compilation units in place of classes, and function calls and returns between

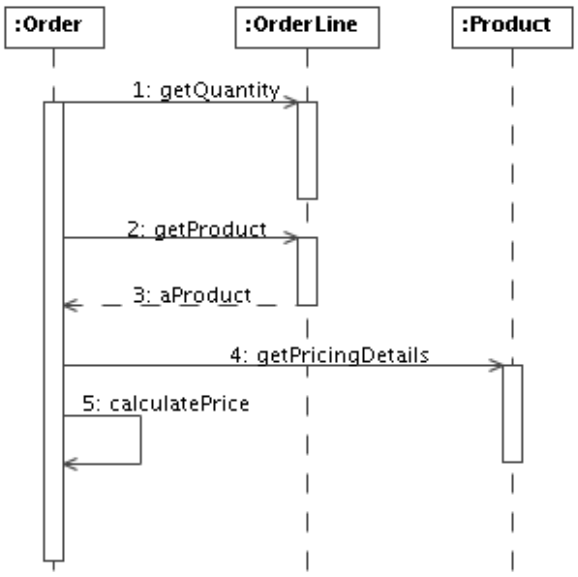


Figure 2.6: UML Sequence Diagram

those compilation units. The same sorts of techniques could be applied to Collaboration and Activity Diagrams. Statecharts can also be applied to non-object-oriented information, but that level of detail is not generally considered to be interesting with respect to the software's architecture.

2.1.4 UML Tool Choice

After studying and using several UML modeling tools, including IBM's Rational Software Architect [25], various Eclipse-based plugins and tools [42] [14] [15], Poseidon for UML [1], ArgoUML [10] and Visual Paradigm [44], we have chosen MagicDraw UML (MD UML) [39] as our target platform. This is for a number of reasons. First, it uses XMI as its standard model storage format. As previously noted, XMI is the OMG standard for UML

interchange. MD UML also features the *OpenAPI*, which makes creating and editing UML models a fairly straightforward programming task. This eliminates a need for separate XML tools or our own software to create an XMI-compatible model, thus avoiding unnecessary steps in translation that could remove flexibility and introduce errors. The use of the OpenAPI will be discussed in some detail in Appendix B. MD UML closely follows the latest published standards for UML and XMI, and has one of the best layout algorithms for UML class diagrams [12]. The other tools that were part of our survey have limited or non-existing support in at least one of the above areas.

The main drawback to using an existing tool, such as MD UML, is that the developers of the tool have their own set of interpretations of how the UML notation is intended to be used. For example, an association is an abstract concept that is not tied to a particular implementation in object-oriented programming languages. Each developer may implement an association in a different way. MD UML records navigable associations within the model information as properties within the associated classes, but non-navigable associations have no assumed implementation in MD UML. The MD UML interpretation for particular parts of the UML notation, such as associations, may not match ours, and both may not match the reader's. We have chosen to accept MD UML's implementation of these types of concepts, rather than attempting to impose our own interpretation. In addition, MD UML users will understand and expect the interpretation that this tool imposes.

2.2 The SWAG Technical Space

The SWAG technical space consists of several software architecture tools, the Tuple-Attribute (TA) Language, and several schemas for TA. In our work, we concentrate on one particular TA schema for software systems. The tools involved in the SWAG technical space have been used to extract and study the architectures of a wide variety of software systems, including the Linux Kernel, PostgreSQL, and OpenSSH [54], among others. This section describes the elements of the SWAG technical space.

2.2.1 The Tuple Attribute Language (TA)

The TA language, developed by Ric Holt [21], is a data format that records information about nodes and edges in a graph. These nodes and edges can also be thought of as entities and relations. Information recorded in a file in the TA format is called a TA program. TA programs are separated into a *scheme* section and a *fact* section – often referred to as the *factbase*. The scheme section describes the meaning of the facts. Both the scheme and fact sections consist of Tuple and Attribute sub-sections.

The Tuple Sub-Language

Each line in the tuple sub-language of TA can be interpreted as information about an edge in the graph. This could include information such as function calls, variable references, or containment relationships. For example, the following TA fragment shows that P contains Q and R, and R calls Q.

```
contain P Q
```

```
contain P R
call R Q
```

The Attribute Sub-Language

The Attribute sub-language defines information about each node. This can include labels, size, description, and position of a given node. For example, the following TA shows that P has the attribute *label*, with the value "foo".

```
P { label = "foo" }
```

Schemes

A TA program should also define a *scheme*, which describes the shape of the graph. The data has no meaning without the scheme. A scheme identifies what types of nodes can be related, what edges are allowed between them, as well as the list of types allowed for nodes in the factbase. The scheme also defines the allowed attributes for each node and optional default values for these attributes. The scheme encodes a set of constraints that we expect the graphs encoded in the fact level of the language to satisfy. We say that a graph conforms to a scheme when the fact level graph uses the entities defined in the scheme in the way the scheme allows. As an example, the piece of scheme information below shows that a `contain` edge can exist between a subsystem and a file, meaning that a `subsystem` can contain any `file`.

```
contain subsystem file
```

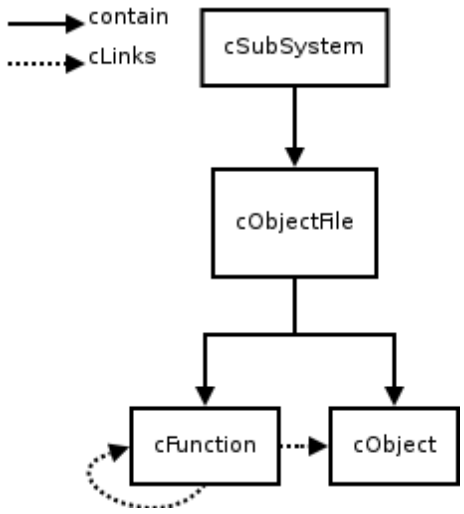


Figure 2.7: FLS for TA

2.2.2 Function Level Schema (FLS)

In this work, we will be concentrating on the *Function Level Schema (FLS)*, which is used to define certain information about software systems. Other schemas can be developed to describe other types of information, but we have chosen to use this one as it readily describes the static structre of a system under study. The output from the QLDX pipeline, described in Section 2.2.3, is a set of facts about the piece of software. Several QL scripts are included with the QLDX pipeline and are normally used to process the extracted facts, producing a TA factbase that conforms to either the FLS or that has been lifted to the file level. In this section, we will describe each of the entities and relations allowed by the FLS. It is summarised in Figure 2.7. This schema was designed for describing information extracted from compiled C source code, which explains the naming conventions involved.

cSubSystem

Each cSubSystem normally corresponds to file system directories containing source code files. These subsystems can be interpreted as components or modules of the software system.

cObjectFile, cExecutable, cArchiveFile

As the names suggest, Object Files, Executables, and Archive Files created from C source code are represented by cObjectFile, cExecutable, and cArchiveFile. In our work, we have concentrated on the cObjectFiles, as these are the most relevant in the static structural diagrams.

cObject

cObjects are the variables that are declared within a given `.c` file. These will not be included in the factbase if the facts have been lifted to the file level.

cFunction

cFunctions directly correspond to actual functions. These are contained within cObject-Files and can be linked to other functions, or can refer to objects. As with cObjects, cFunctions will not be included in the factbase if the facts have been lifted to the file level.

cLinks

cLinks can occur between the various entities in the FLS. This relation is used to document relationships like function calls and variable references. In the FLS, cLinks will normally

occur between cObjects and cFunctions. If the facts have been lifted to the file level, cLinks will occur between files, rather than entities within those files.

contain

This relation is used to define a containment hierarchy, identifying which entities are contained by others. For example, subsystems will normally contain the various types of files. The files, in turn, will usually contain functions and objects.

2.2.3 SWAG Tools

The SWAG architectural tools that are relevant to this work include BFX (a fact extractor), QL (a relational calculator), and LSedit (a software landscape editor). This set of tools is often referred to as the Build/Comprehend pipeline [24]. BFX is used to extract architecturally interesting information from compiled object files, with the option to include additional information by using a custom linker (LDX) [54]. The extracted facts include information such as function calls, variable names, and containment hierarchies. This data is stored in a TA file. The relational calculator QL understands the TA file format, and allows the user to manipulate the database of extracted facts. For example, using QL, a containment hierarchy can be added, and relationships can be lifted to the file level, rather than the function and variable level. A set of pre-written QL scripts included with the SWAGkit can help with these tasks. The landscape editor, LSedit, reads a TA file and displays it in a coloured "boxes and arrows"-style set of navigable diagrams.

Since QL is written in Java, it was easy to leverage the source code for it and integrate it into our MagicDraw UML plugin. We use QL to read and interpret the TA programs

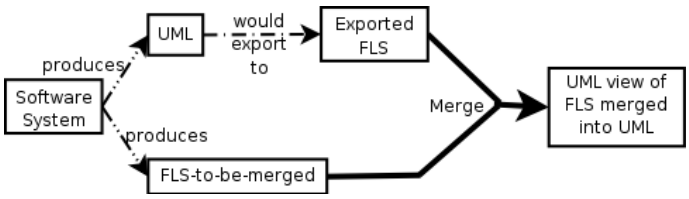


Figure 2.8: Overview of Merging FLS into UML

and to perform transformation operations on the facts, enabling us to compare sets of facts to information in the UML models. Pre-made QL scripts, included in the SWAG kit, have also been used to prepare the factbases for use in our tool.

2.3 Merging Data

In this thesis, we attempt to solve the problem of bridging between two technical spaces. To accomplish this bridge, we must have the ability to import, export, and merge information from one space to another. Import and export operations are relatively simple because they do not require the ability to map existing entities across different technical spaces. Merging, on the other hand, requires these types of mappings.

Figure 2.8 shows an overview of the process involved in merging two sets of facts. The case we have illustrated shows the merging of an existing set of facts using the FLS into an existing UML model, encoded in XMI. We show three sets of facts: the existing UML, the FLS-based set of facts to be merged, and what we call the *exported FLS*. Both the UML and the "to be merged" FLS factbase are models of the same system, which may have been modified in either of these technical spaces. The end result of this merge is a model that contains the relevant information from both technical spaces, and can be viewed in

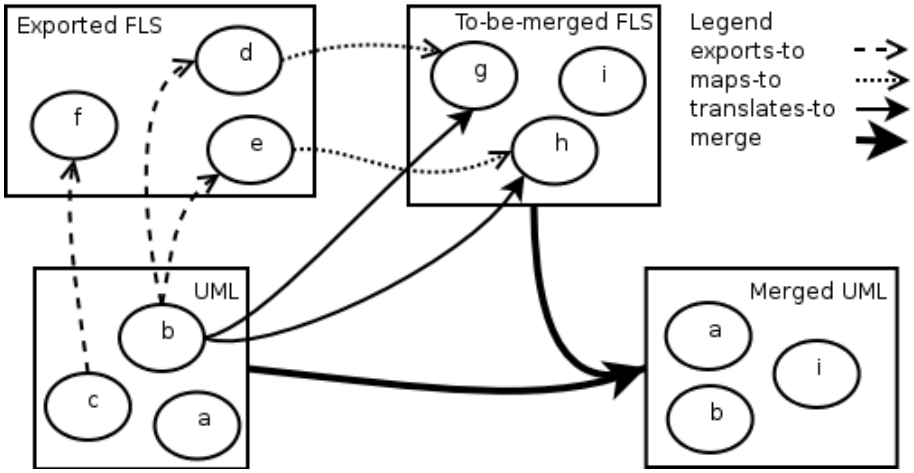


Figure 2.9: Merging Existing FLS into Existing UML Model

the UML space. The exported FLS refers to the FLS information that would result from exporting the UML model using our export technique. This export does not actually take place during a merge operation, but it is used here to show how the mappings are achieved.

Figure 2.9 shows how mappings are determined between the entities in each of these models. The dashed arrows from `b` to both `d` and `e` show the mapping between some UML entities and their equivalent exported TA entities. In this case, this might be a situation where `b` is a method in a superclass that exports to two functions in the FLS – one for the superclass, and one for the subclass. The dotted arrows show which entities in the exported FLS match up with entities in the to-be-merged FLS-based factbase. By doing this matching, the dashed and dotted arrows can be combined to become the solid arrows, which identify those entities in the to-be-merged TA that already exist in the UML model.

After obtaining this list of entities which already exist in the UML model, it can be used

to determine which entities in the to-be-merged TA must be added to the UML model. This information can also be used to delete those items in the UML model that exist in the exported FLS, but do not have a corresponding entity in the to-be-merged FLS-based factbase. If the UML entities have no possible equivalent in the FLS, they are not touched. In our example, `i` would need to be added to the UML model, `c` would be deleted, and `a` and `b` would not be touched.

2.4 Related Work

We describe previous research work in the areas of bridging technical spaces, merging information from multiple sources, and promoting the interoperability of software architecture tools.

Staikopoulos and Bordbar [49] created a metamodel refinement approach for bridging technical spaces. To apply their technique, the technical spaces involved must be well-established and well-formalised. They discuss the process of determining the mappings between metamodels. Their focus is on solving the problem of what to do when some metamodel elements from the source cannot be directly mapped to metamodel elements in the target. In our work, each technical space may have such un-mappable metamodel elements. We do not focus on mappings in one particular direction, but mappings in both directions, as well as the problem of deciding which information should be ignored or acknowledged in the case of a merge.

Zundorf *et al.* [55] have developed a technique for merging graph-like object structures. They argue that text-based methods for merging of object structures have signifi-

cant limitations and propose a graph-based approach to deal with these perceived problems. Specifically, they state that if the order of the information in text documents changes, most systems will not properly recognise matching objects, unless a unique ID is assigned to each object. In contrast, we need not be concerned about global unique identifiers, but we can identify and map to entities based on their location within the containment structure of the model.

Westfetchel [53] has developed a multilingual structure-oriented technique to merge different types of software documents written in arbitrary languages. His technique uses the idea of a Document Interface for each document type, to retrieve structural information from the documents. This merge technique takes the underlying syntactic and semantic structures into consideration. The interfaces provide a bridge into the documents, thus providing access to, and a translation for, the information relevant to the merges. The use of interfaces allows additional document types (and languages) to be added to the system without significant changes to the underlying environment.

Some researchers are using the process of model synchronisation to record information about the evolution of software systems. For example, Ivkovic and Kontogiannis [26] have developed a technique to track changes to objects based on a unique ID for each object. Recording this change information provides them with traceability information about the changes that occur during software evolution. While doing this work, they developed some key terminology and techniques for performing model synchronisation. They argue that there are two types of mappings between sets of related data. A mapping can be explicit, which means that specific entities in the source are explicitly mapped to specific entities in the target. The other type of mapping is implicit. This style of mapping is done between

metamodels and is the type of mapping we will be performing in this work. Relations between specific entities are implied, based on the mappings between the metamodels. These two researchers argue that propagating changes between software models is a "first step towards maintaining consistency between architectural, design, and implementation models" [26].

Round-trip engineering is another related aspect of software engineering. For example, FUJABA is a tool that translates information from UML to Java and back to UML. The tool must handle the issues of interpreting the semantics of UML and the mappings required from UML to source code. We face similar issues in our work. For example, there are several different ways to implement associations in Java. The FUJABA project has chosen to implement associations as private attributes and access methods in the associated classes [28]. We have made a similar decision for associations in our mapping choices.

Aside from model synchronisation, some other techniques for tool interoperability have been developed. Notably, *GXL* and the *Dagstuhl Middle Metamodel (DMM)* have been designed as interchange languages for models of software. GXL is an XML-based exchange format designed by Holt *et al.* [23] as a standard for describing and interchanging graphs about software. The DMM was initiated by a group of researchers at the Dagstuhl Seminar on Reverse Engineering Tool Interoperability, held in January 2001 [33]. The DMM was designed to be a schema for interchanging information about software between different tools. The authors of this metamodel suggest that it can be used to describe the objects and their relationships, and that the choice of encoding syntax is up to the user. They recommend the use of GXL or TA for encoding purposes.

Bézivin *et al.* [2] have a completely different approach for tool interoperability. They

argue that model engineering should not be based on single, monolithic languages like UML, but rather on small Domain Specific Languages defined by focused metamodels. In their approach, interoperability is attained by using model transformations. Our approach is similar, in that we argue that different notations, tools and diagrams are useful for different purposes – bridging between the different modeling notations is the interesting part of this problem.

Chapter 3

Model Translation

In this chapter, we describe the process involved in bridging between the UML technical space and the SWAG technical space. This bridging requires that we provide the ability to import, export, and merge between models encoded in each space. We begin this chapter by discussing alternative approaches to solve this problem. We follow this with some examples of bridging in each direction. We then describe the mapping choices we have made between the entity types in each of these spaces, and discuss the complications we encountered while building this tool. We conclude this chapter with an overview of the algorithms we have used for the merge procedures.

3.1 Design Decisions

There are several ways the bridging process could be performed. The simplest solution, but the least useful, would be to develop a TA scheme for UML, and create QL scripts to

translate from the FLS to the UML schema. With this approach, a translation would then be required between a TA factbase conforming to the UML schema and an actual XMI encoding of the model, meaning that the provision of a UML schema for TA would not get us any closer to solving the problem.

An alternative would be to create an XML schema for the FLS, create an XML encoding of the FLS-based TA factbase, then create a transformation from the FLS-based XML to XMI. This would require us to understand all of the details of the XMI specification and to create an XMI encoding that would be compatible with a specific UML tool. This would also require a tool to perform the translation from FLS-based TA to XML, and a transformation from the XML to XMI. Again, this introduces unnecessary steps.

Finally, there is the option of using existing pieces of software, each of which understand the details of their respective technical spaces, and creating a simple tool to bridge between these two pieces of software. This is the approach we have taken. QL, written in Java by Jingwei Wu at the University of Waterloo [54], is a tool for interpreting and manipulating TA factbases. A major benefit is that we have access to the source code for QL, and we have made significant use of this. MagicDraw UML is a UML modeling tool developed in Java by MagicDraw that allows users to add their own plugins to extend it. Plugins can use a Java API called OpenAPI [40] to create and edit UML models programmatically. We used these two facilities to create a MagicDraw UML plugin to bridge the UML and SWAG technical spaces.

3.2 Translation Scenarios

There are four possible scenarios for translating between the UML and SWAG technical spaces. Model information can be imported from the TA format into a MagicDraw Model, and vice-versa. Two models can also be merged, in either direction. For instance, a FLS model can be merged into an existing UML model, or a UML model can be merged into an existing FLS-based TA factbase. We have implemented all four of these scenarios in one plugin for MD UML. This section covers examples of each of these scenarios.

3.2.1 Import a FLS Factbase to a UML Model

Figure 3.1 shows an LSedit visualisation of a TA factbase containing the extracted architecture of Gaim 1.5.0. Many of the details are not shown in this diagram, but those that are shown are important to note. For example, we have shown the inner details of `sha.o`. In this part of this figure, we can see the details of cLinks between functions. We can also see which functions are being called from outside of `sha.o`. LSedit allows us to view as many or as few details about each entity in the model as desired.

Figure 3.2 shows the architecture of Gaim 1.5.0, as imported into a UML model, from the FLS-based TA factbase shown in Figure 3.1. The contents of the subsystem `src` have been mostly suppressed due to space limitations. Some details are shown between the package `protocols` and the class `sha.o`. In the UML view of this model, we can see that `sha.o` depends on itself, but we cannot see the internal details of which functions rely on which other functions. This information isn't relevant within the UML technical space, so it is discarded. As will be seen, it is still maintained when a UML model is merged into

an existing TA factbase.

3.2.2 Merge a UML Model into Existing FLS Factbase

Starting from the model shown in Figure 3.2, we have made some modifications to the UML model. In Figure 3.3, we have added a class named SuperSha as a superclass to sha.o. We have also moved the method shaInit() from sha.o to its new parent. The resulting modified section of the model is shown in Figure 3.3. After using our tool to merge this model back into the existing FLS Factbase shown in Figure 3.1, the resulting LSedit visualisation is shown in Figure 3.4. As can be seen in this figure, the method in the superclass has been copied to the subclass, and all other relationships remain intact.

3.2.3 Export a UML Model to FLS

Figure 3.5 shows a portion of the model of the Java source code for QL. Because QL is written in Java and thus contains object-oriented concepts from the beginning, it is a more appropriate example for the translation in this direction. The portion of code modeled in these diagrams is essentially those classes in the QL API which we have directly used to write our tool. Each of these classes is described in more detail in Appendix A. The UML model in Figure 3.5 was reverse engineered from Java byte code, using the reverse engineering tools built into MagicDraw UML.

We then exported this model to a FLS-based TA factbase, and the LSedit visualisation of this factbase is shown in Figure 3.6. This figure shows the generalisation (between EdgeSet, NodeSet and TupleSet) and realization (between Tuple and TupleImpl) relationships

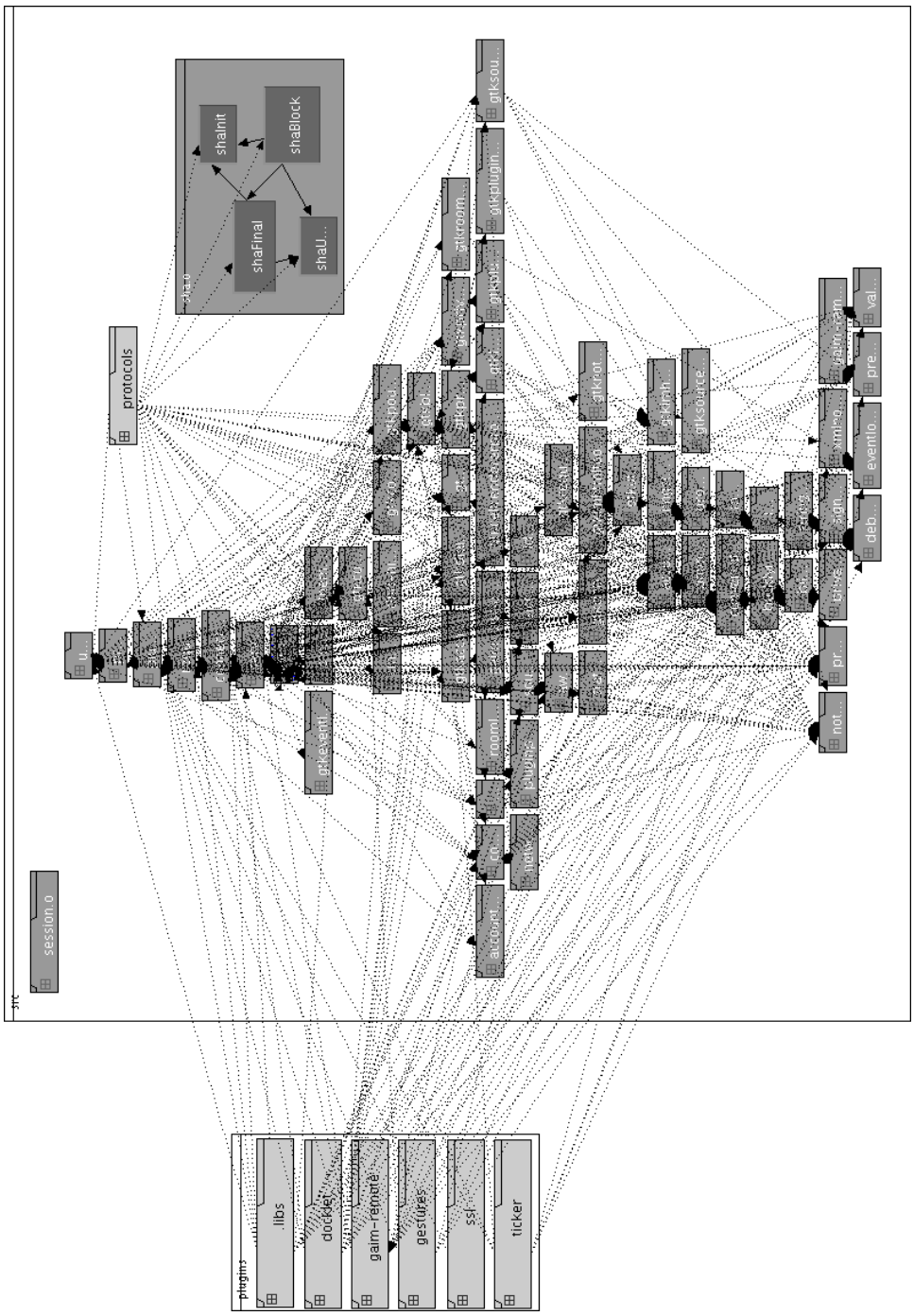


Figure 3.1: LSedit view of Gaim 1.5.0, from a TA factbase

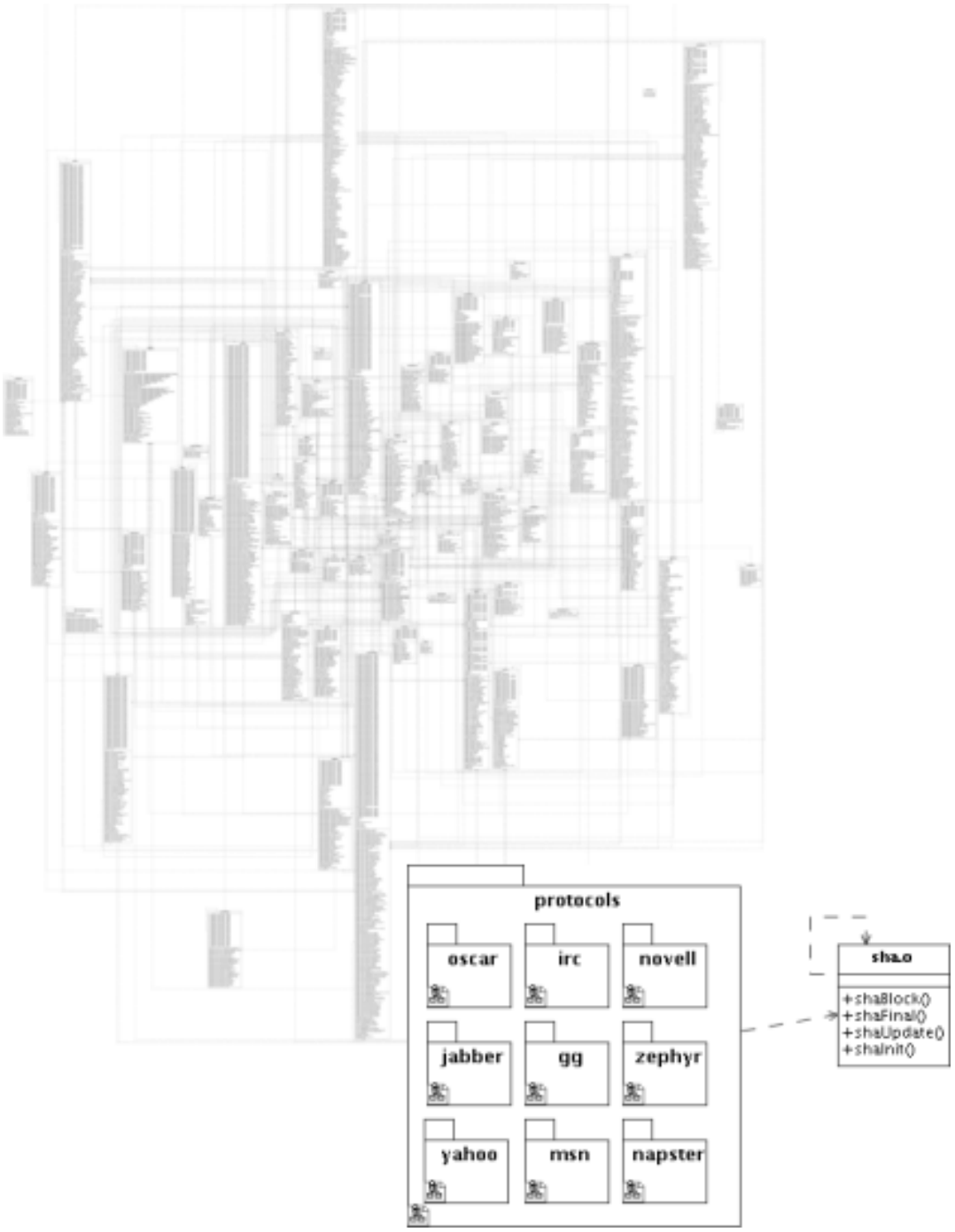


Figure 3.2: MagicDraw UML view of Gaim 1.5.0 imported from factbase in 3.1

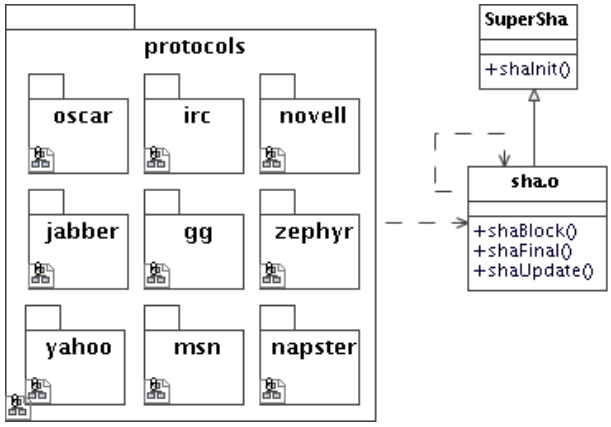


Figure 3.3: Edited UML model of part of Gaim 1.5.0

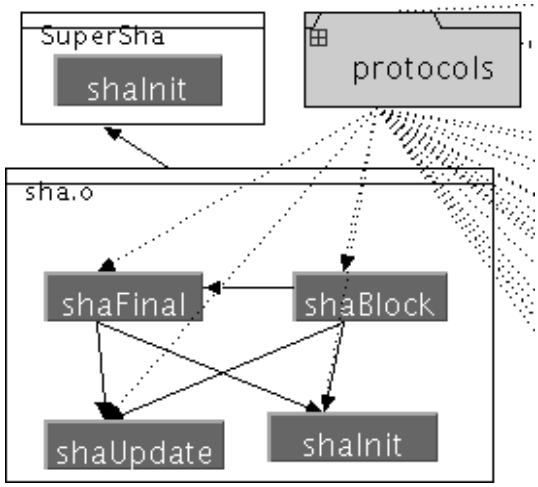


Figure 3.4: Portion of merged Gaim 1.5.0 model in LSedit

as cLinks between the classes. Because of the restrictions of the SWAG technical space, we are required to map the different types of UML relationships to a semantic equivalent in the SWAG space. We have shown the contents of TupleSet and EdgeSet to show the operations that EdgeSet has inherited from TupleSet. In this case, it is important to note that the dependencies shown in Figure 3.6 exist between classes, and not between individual functions or operations.

3.2.4 Merge a FLS Factbase into Existing UML model

As was demonstrated in the previous section, exporting a UML model containing a generalization relationship to the FLS will create a copy of a superclass' operations in each of its subclasses. When browsing the LSedit visualisation, a user may decide that these copies of the operations are redundant and should be removed. We have shown this in Figure 3.7. In this case, we have removed the constructor TupleSet, and operations trySort, newSet, appendDB, appendTA, print, printTA, sort, sortDom, and sortRng from the subclass EdgeSet. Figure 3.8 shows the merged results in TupleSet and EdgeSet. For the sake of clarity, we have omitted the rest of the elements as seen in Figure 3.5. In Figure 3.8, we can see that the deleted functions have been removed from the subclass, and the rest of the inherited operations have been added to the subclass.

3.3 Mapping of Entities

The general translation between UML and the FLS for TA follows the basic mappings as defined in Table 3.1. This mapping groups the functions and variables contained within a

Figure 3.5: Partial UML model of QL

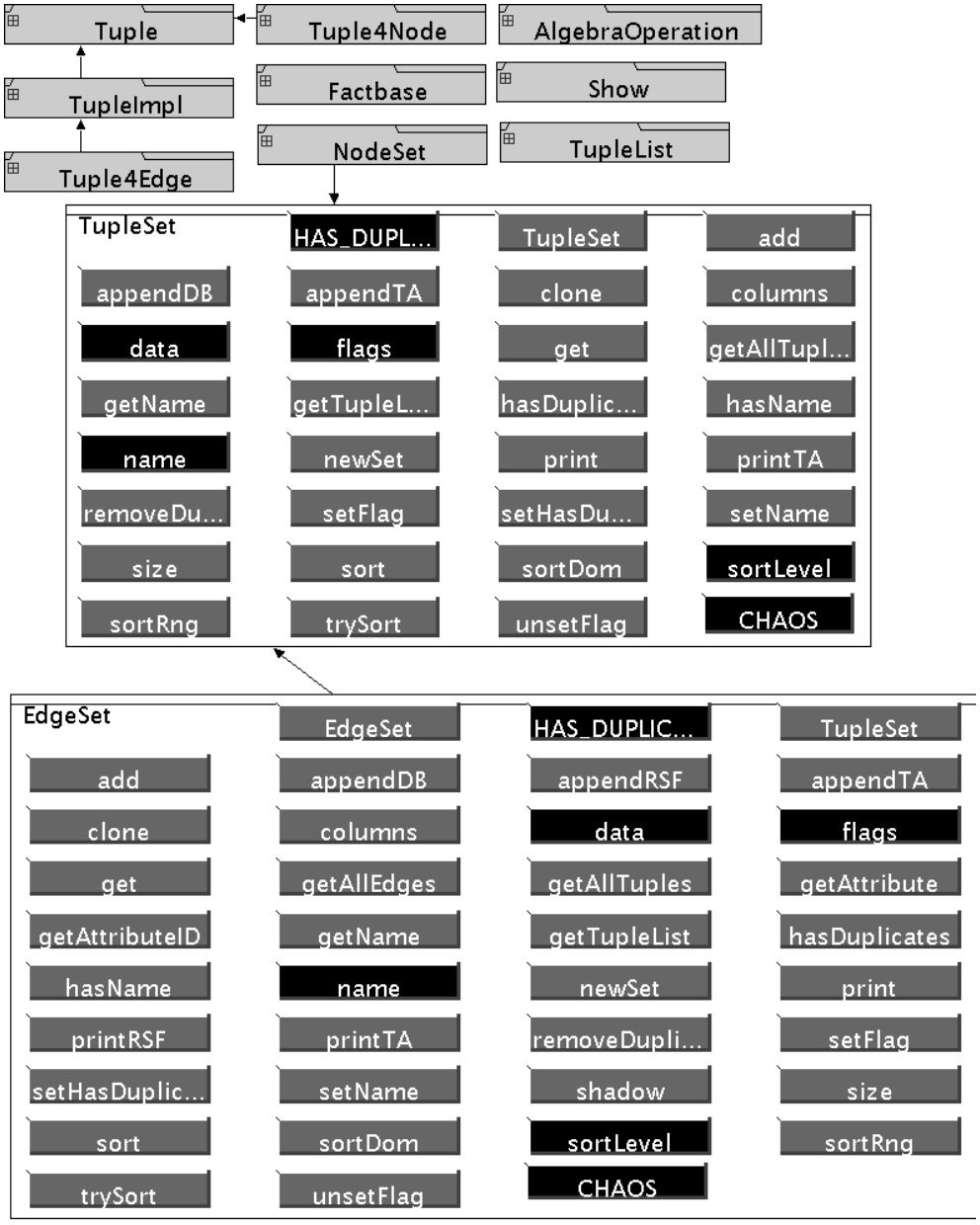


Figure 3.6: LSedit view of partial model of QL

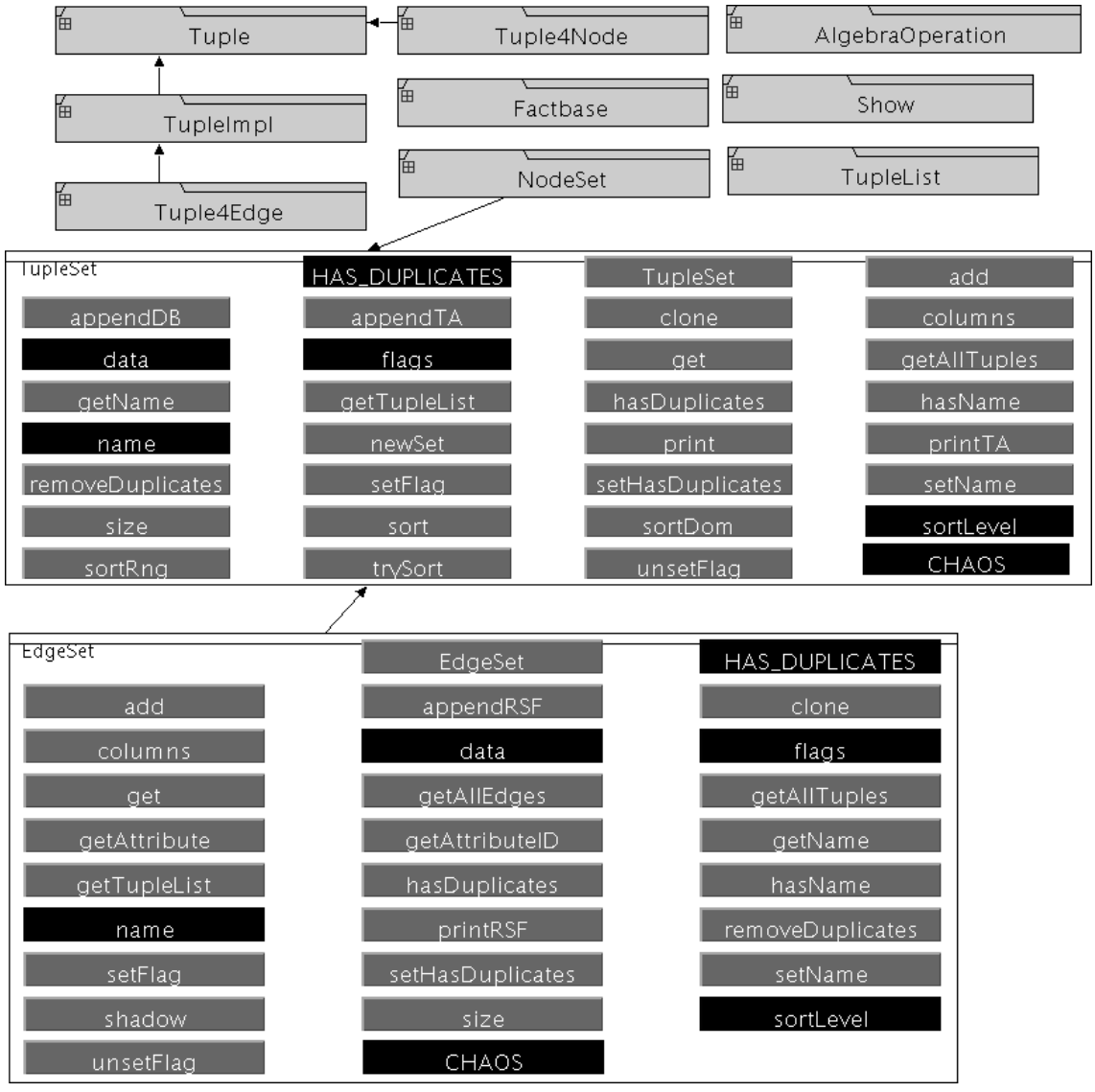


Figure 3.7: Modified partial LSedit view of QL

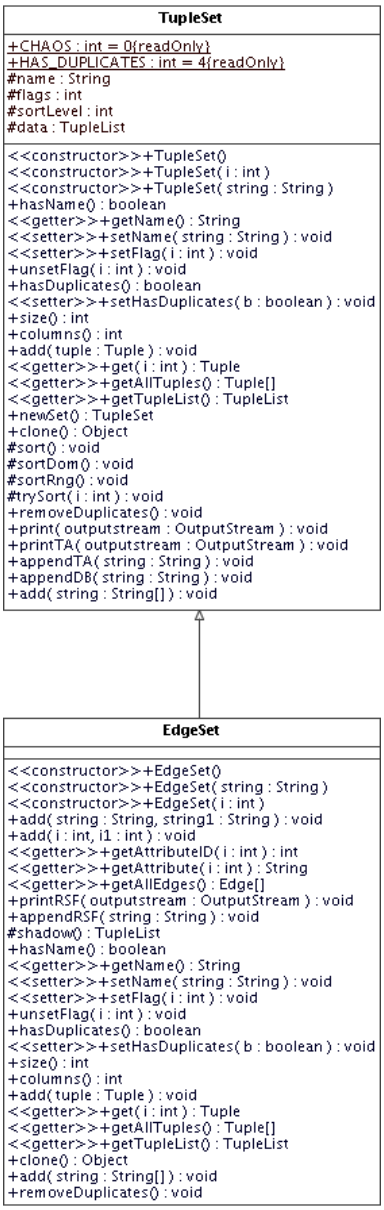


Figure 3.8: Portion of merged QL model in UML

FLS Entity	UML Entity
cSubSystem	Package
cObjectFile	Class
cObject	Property
cFunction	Operation
cLinks	Dependency

Table 3.1: Basic Mappings between FLS Entities and UML Entities

given source file into a class. This is validated by the founding principle of Object Oriented Development, which is to group related properties and operations into classes.

Section 2.2.2 contains more specific details regarding each of the entity types in the function-level schema, and Section 2.1.2 describes the UML entities. We will now describe the relationships between these two sets of entities. Each directory in the source code is identified as a Subsystem during fact extraction. This is semantically equivalent to a Package, since packages tend to also be based around directory structure. ObjectFiles refer to each .o file generated during compilation. These are compilation units, just as Classes are compilation units. ObjectFiles contain "Objects" and Functions. In this case, some confusion can arise due to naming choices. An Object in the FLS technical space is simply a variable of some type, contained within an ObjectFile. This is similar to Objects in object-oriented programming, since an Object is an instance of some particular Class. A FLS "Object" can be thought of as a member variable of a Class. Functions are units of executable code, similar to Operations in Classes. The FLS refers to a reference from one function to another or from a function to a variable as a Link. We consider this to be semantically equivalent to a UML Dependency, except that dependencies are usually between classes rather than operations, as will be discussed in Section 3.4.2. Figure 3.9

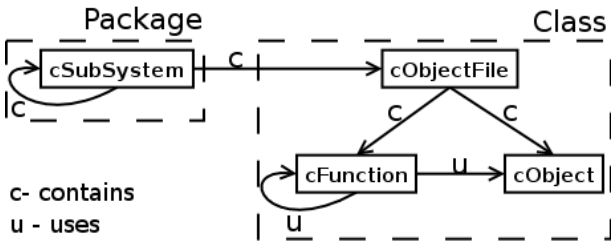


Figure 3.9: Schema Mappings

summarises this mapping. The "uses" edges in this figure represent links or dependencies.

One limitation of this basic mapping is that it does not take into account some of the central principles of object-oriented programming, such as Generalization. This is because procedural languages, such as C, have no equivalent concept. Since the SWAGkit was originally designed with languages such as C in mind, we have been forced to develop our own technique for dealing with Generalization when it is introduced into the UML version of a model. This will be discussed in detail in Section 3.4.3.

Associations present another interesting decision. MD UML has its own assumed implementation for some types of associations. It distinguishes between navigable and non-navigable associations. Navigable associations are implemented as variables in the classes at either end of the association. Non-navigable associations have no default implementation in MD UML. For simplicity, our tool follows the MD UML implementation in this case. For navigable associations, we add the appropriate variables to the classes at the association endpoints. This decision is supported by other researchers who have made similar decisions regarding the implementation for associations, such as those involved with the FUJABA project [28]. If the associations are not navigable, we ignore them, since as-

sociations are an abstract concept with no standard implementation. The same situation applies for composition and aggregation, which are specific types of associations.

Another limitation is that object files are not entirely semantically equivalent to classes. In an executing C program there will only be one instance of any given object file in memory; there is no reason for there to be more. By definition, Classes are instantiated multiple times, in order to promote re-use. We can think of the classes in a UML model of a software system written in a non-Object Oriented language like C as being singletons; alternately, they can be thought of as classes with only static members. We have chosen not to add any stereotypes to make this distinction in our generated UML models, instead leaving the architect free to think about the "classes" in the way he or she chooses.

3.4 Complications in Translation

When performing these import, export and merge operations, we run into several complications, requiring our tool to consist of more than a simple translation mechanism. The names of entities in a TA factbase have no semantic impact on the model, whereas in UML, the naming scheme is entirely based on the fully qualified path to a specific entity in the containment structure. Also, in UML static structure models, relationships tend to exist between classes, whereas the FLS in TA is fully capable of documenting and displaying relationships between entities at a lower level, between functions and variables. The final complication that will be discussed in this section is the problem of dealing with generalization semantics, when moving information from the UML to the FLS in TA.

3.4.1 Naming Schemes

When a FLS-based TA factbase is produced by the QLDX pipeline, the entities contained within each cObjectFile – such as cFunctions and cObjects – are named by a coding system, with names such as `napster.o[.text+0x13e0]`. When LSedit, the software landscape editor, reads in a TA factbase, it uses labels from the `FACT ATTRIBUTE` section of the factbase as the human-readable name of the function or variable, such as `gaim_init_plugin`. If no labels exist, LSedit will use the last part of the name as the display name. For example, if the name of the entity is `/napster/.libs/napster.o[.text+0x13e0]`, LSedit will use `napster.o[.text+0x13e0]` as the display name. On the other hand, UML uses the actual operation or property name as part of the fully qualified identifier. The same name is also displayed on diagrams. For example, a function named `gaim_init_plugin` within the file `napster.o` would be identified as `napster.o::gaim_init_plugin`.

This difference between these two naming schemes is most notable when comparing an initial TA factbase with one that has been imported using our plugin and then exported again. In an exported factbase, the `FACT ATTRIBUTE` section of the factbase is basically removed and merged into the `FACT TUPLE` section. A factbase will be generated by the BFX/QLDX pipeline containing information such as follows:

```
FACT TUPLE :
$INSTANCE .../napster.o cObjectFile
$INSTANCE .../napster.o[.text+0x13e0] cFunction
FACT ATTRIBUTE :
.../napster.o[.text+0x13e0] { label = "gaim_init_plugin" }
```

After this information has been imported via our MD UML plugin, then exported back to TA, it will have been modified to the following:

```
FACT TUPLE :
$INSTANCE .../napster.o cObjectFile
$INSTANCE .../napster.o/gaim_init_plugin cFunction
```

This modification has no negative impact on the usefulness of the factbase, since every cFunction or cObject within a cObjectFile must have a unique name. This change also significantly reduces the size of the TA file. If there are no labels in the `FACT ATTRIBUTE` section of the factbase, the representations in LSedit will be unaffected, as LSedit resorts to using the last item in the path for labels in the visual representation of the facts.

This complication is simple to handle in an import or export situation, but the merge process becomes slightly more complicated. When merging, our plugin must check to see whether the entity in question exists in any form, in the file that the model is being merged with. To merge UML entities with TA factbases, we must match up individual entities. These must match based on both the label name for a function and the containment structure, in order to locate the entity in the model.

3.4.2 Level of Calls Relationships

Another minor complication arises from the fact that UML models normally do not document dependencies between individual operations, and instead document dependencies at a higher level – that is, between classes. For this reason, we have chosen to *lift* the links in the TA factbase before adding them to the UML models. This means that we raise the links to be between object files, rather than between functions. This is accomplished by a couple of straightforward composition operations, using relational algebra operations from QL, as outlined below. The `o` is a composition operation between two EdgeSets, and `inv`

returns the inverse of the EdgeSet.

```
lifted = (containment o dependencies) o inv containment
```

In this example, the `containment` EdgeSet might contain something like:

```
contain ../napster.o ../napster.o/gaim_init_plugin cFunction
contain ../accounts.o ../account.o/gaim_accounts_init cFunction
```

and `dependencies` might look something like:

```
cLinks ../account.o/gaim_accounts_init ../napster.o/gaim_init_plugin
```

The resulting TupleSet, `lifted`, would then contain the following, for the above example:

```
../accounts.o ../napster.o
```

This TupleSet records the links between the files that contain the functions that participated in the original links relationship. Duplicates are removed. This lifted information can then be used to create dependencies in a new UML model, or could be compared against existing dependencies or links in the case of a merge. The interesting result of this complication will be discussed in more detail in Section 4.2. It is worth noting that lower-level calls relationships between functions in the TA factbase are not replaced when merging back in the opposite direction.

3.4.3 Generalization

The FLS for TA was not designed to handle object-oriented concepts such as generalization relationships. As a result, conversion from a FLS-based factbase to UML, and back to the FLS – without making changes in the UML model – will not contain any generalization hierarchies. However, the goal is to allow the user to modify the model in some way, in

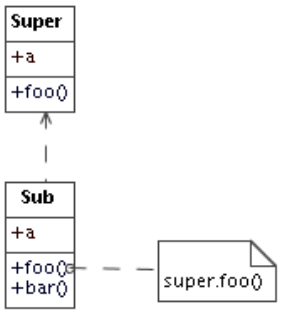


Figure 3.10: TA "Inheritance" Semantics

any tool or notation, and have the changes be reflected in the other technical space. Since the metamodel of the SWAG technical space does not support any semantic equivalent to the object-oriented concept of generalization, we have been forced to create our own FLS-compatible translation for this concept.

Generalization means that the member operations of the superclass are inherited by the subclass. As a result, in the FLS, we have converted generalization relationships to links and copied the methods of the superclass into the subclass. This preserves the semantics of the generalization concept. Figure 3.10 is a UML diagram showing the meaning of this translation. The inherited operations of the superclass are copied to the subclass. If the inherited operation had not previously been redefined in the subclass, we can think of the subclass version as being a simple call to the superclass version. This is the reason for the links arrow in the FLS. This not only shows a function call, but also shows that these two compilation units are related in some way. This translation will be discussed in greater detail in Chapter 4.

When performing a merge, our tool determines whether an entity already exists in the

model, and if it does, it does not get replaced. The generalization arrow is one such entity when merging from TA to UML. This maintains the generalization semantics in a merge. Unfortunately, there is no way to preserve this information when performing an import to an empty model, since the additional semantic information is not there.

It is worth noting that we have not taken actual implementation details into consideration. If the models were intended to be used to generate code for a non-object oriented language, such as C, a prefix may be necessary for the functions created in the subclass, in order to prevent ambiguity during linking.

3.5 Merge Algorithms

In this section, we will provide an overview of the algorithms we have used to perform the merge operations in our tool.

3.5.1 Merging an FLS-based Model into UML

The algorithm to merge a FLS-based model into an existing UML model is a fairly simple two-step procedure. The steps are outlined below.

1. Using the Visitor pattern, visit relevant Elements in the UML model:
   (a) If no FLS equivalent is found, delete the UML Element.
2. Loop through the entities in the FLS-based factbase:
   (a) If no corresponding UML Element exists, add one.

When compared to the merge requirements described in Section 2.3, the only difference in the resulting models is that when a UML model is exported to TA, then merged back in again, some information will be added to the UML model using our technique. More specifically, the methods of the superclass will be copied into the subclass if a generalization relationship exists. This does not affect the meaning of the model nor does it violate any object-oriented principles. The appearance of a superclass method in a subclass normally shows that the method has been overridden in some way. The actual contents of the subclass version could be a simple call to the superclass version, as previously described in Section 3.4.3.

3.5.2 Merging UML into a FLS-based Model

In the reverse direction, to merge a UML model into an existing FLS-based TA program, we follow the algorithm outlined below. The merge in this direction is slightly more complicated, due to the differences in naming schemes, as previously described in Section 3.4.1.

1. Visit UML Elements:
   (a) Add Packages, Classes, Operations, and Properties to output factbase as $INSTANCES.
   (b) Create contains relationship between parents and children, as they are added.
   (c) For Generalisations:
      i. Add to the factbase, as described in Section 3.4.3.
      ii. Remove duplicate cFunctions, cObjects and cLinks.
2. Loop through the cLinks in the old factbase.

(a) Using the old containment structure, determine the UML name for each entity involved in a given cLinks relationship.

(b) If a corresponding (lifted) dependency exists in the UML, add the cLinks using the new entity names.

3.6 Chapter Summary

In this chapter, we have described the obstacles we encountered during the translation process, as well as the solutions we have implemented for these particular problems. The translation activities have been described for the two specific tools we have studied. Similar problems and similar solutions should be applicable in implementing a translation procedure and tool for bridging between other technical spaces. This information should be useful to other researchers or developers who would like to bridge between another technical space and the ones we have bridged in this work.

Chapter 4

The Bridge Domain

In this chapter, we discuss our evidence for the existence of a bridge domain. This evidence supports the claim that the real architecture of a software system cannot be fully expressed in any one specific notation or modeling language. We argue that a software system can be best understood through multiple views of the software, with different technical spaces providing each view. This set of views essentially defines the bridge domain.

Even with two very expressive and flexible schemas, there are certain concepts that can only be described in one or the other, but this information can be maintained through merge operations between the two technical spaces. The fact that these types of concepts exist points to the existence of a bridge domain. This bridge domain cannot be expressly encoded, but is held as a mental model or as a collection of views seen through several technical spaces. One could argue that the bridge domain is automatically encoded in the source code, but there is a problem with this perspective. A program may be written in a procedural language like C, yet have object-oriented concepts embedded in it. To include

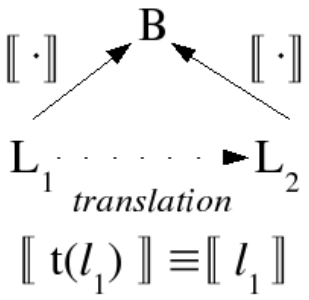


Figure 4.1: "Bridge Domain" arises through translation

these concepts in a program understanding task, the software must be abstracted into a model and viewed within some technical space that includes a mechanism for demonstrating these concepts.

Figure 4.1 shows how the bridge domain relates to the notation languages used to model the software. L refers to a notation language. B refers to a Bridge Domain. A translation from L_1 to L_2 means that some transformation can be performed on the models L_1 and L_2 to represent them in equivalent encodings in the bridge domain. The actual encoding of this bridge domain is not possible, except through both languages, as each notation language contains information that is not expressible in the other. This information can be maintained during a merge operation, but not during import or export operations.

We begin this chapter with an example of how the UML and SWAG technical spaces can be used together to investigate various properties of a real software system. This example shows that information viewable in each technical space can be important for a program understanding task, thus demonstrating the conceptual importance of the bridge domain.

4.1 A Real-World Example

Throughout the course of the development of our tool, we have been using the Gaim Instant Messaging client, version 1.5.0, as a testbed for our tool. Gaim is a multi-protocol instant messaging client, which means that it allows users to register accounts for various instant messaging protocols, such as ICQ, MSN, AIM, IRC, Jabber, Napster, and Yahoo, among others. Architecturally speaking, this is a small program (151 KLOC). It was chosen for its small size, which made it possible to develop and test our tool relatively quickly. It uses the GTK rendering engine, and is a standard application in the Gnome desktop environment for Linux. There are also versions available for Windows, BSD, and Mac OS X.

Certain aspects of the implementation of Gaim have made it an interesting testbed for our work. For instance, there are many pairs of source files with similar names, such as account.o and gtkaccount.o, log.o and gtklog.o, blist.o and gtkblist.o, privacy.o and gtkprivacy.o, and conversation.o and gtkconv.o. We speculated that these files may have code duplication and could potentially be thought of as super and subclasses. It is clear from examining the LSedit visualisation that the GTK versions of these files depend on the non-GTK version and no dependencies exist in the other direction. This is shown in Figure 4.2.

A closeup of the privacy.o and gtkprivacy.o portion of the UML diagram is visualised in Figure 4.3. In this visualisation, we can easily see that there appears to be several functions that have similar names, such as `gaim_gtk_privacy_init()` and `gaim_privacy_init()`. It seems likely that the dependency we see in the UML diagram is related to these similarly named functions. Therefore, we return to the visualisation in LSedit to attempt to validate

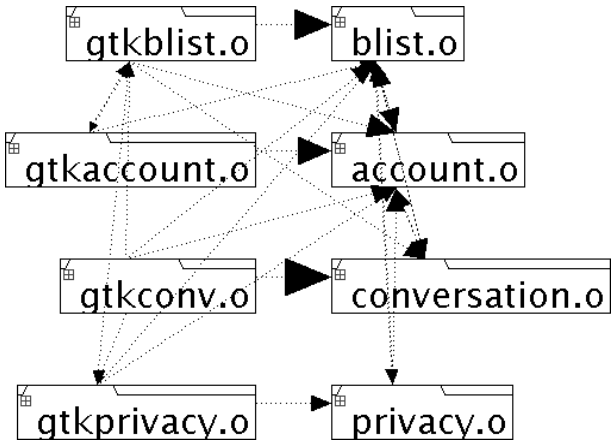


Figure 4.2: Interesting Relationships

this theory.

Figure 4.4 shows the relevant portion of a backward query performed on the privacy.o file, from within gtkprivacy.o, in LSedit. This query shows which functions in gtkprivacy.o depend on functions contained in privacy.o. In this case, it shows that `clear_cb()`, `remove_cb()` and `confirm_permit_block_cb()` are the only functions in gtkprivacy.o that depend on functions in privacy.o. This disproves our theory about the nature of the dependencies, but it shows how these two tools can be used together to investigate certain properties of a software system.

LSedit is useful for performing queries on the entities, and has the ability to show the strength of a dependency based on the size of the arrowhead. The size of the arrowhead is directly proportional to the number of cLinks relationships from one entity to the other. UML makes it easy to see which functions exist in each source file and provides the ability to add meaning to the structure of a software system by adding things like generalization

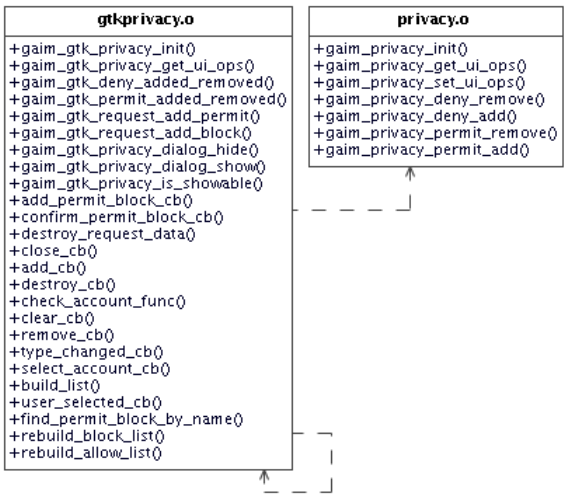


Figure 4.3: Privacy Source in UML

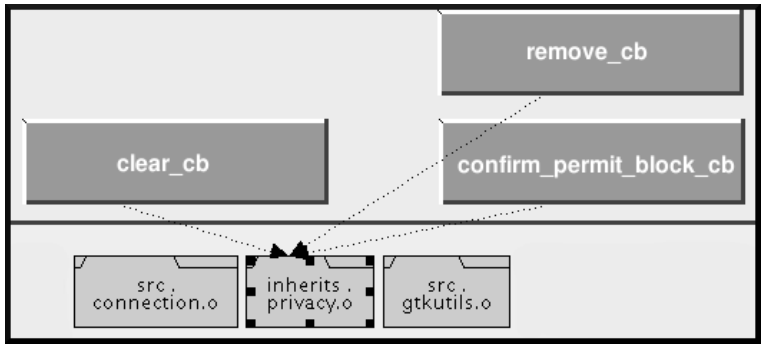


Figure 4.4: Backwards Query on privacy.o

hierarchies. UML also provides the ability to easily see multiple levels of the containment structure at once, which provides printable diagrams, whereas LSedit is a more interactive visualisation and navigation tool. The use of these two technical spaces, together, brings out information that would be difficult to discover by viewing the model of the software in one technical space by itself.

4.2 Dependencies

Part of demonstrating the existence of a bridge domain that cannot be expressly encoded is showing that there are concepts in each of our technical spaces which cannot be accurately expressed in the other. In this section, we discuss dependencies, which can be expressed in more detail in the SWAG technical space than in a UML model. When merging dependency information between TA and XMI, the TA "error-corrects" some of the information that cannot reasonably be held in the UML model. In the LSedit visualisation, it useful to know which functions call which other functions. The LSedit visualisation is dynamic – it changes depending on which parts the user is looking at – and can visualise this detailed information very well. The UML Class Diagram has no reasonable way to display information about calls between functions. Thus, we require the ability to maintain the detailed information about call relationships through a merge with a UML model.

We will demonstrate this using an example from Gaim 1.5.0. Figure 4.5 shows the contents of the .libs SubSystem for IRC in LSedit. The call relationships shown that end somewhere off-diagram are references to functions or variables in other subsystems. Figure 4.6 shows the same package in UML. One advantage to LSedit is that it is possible to tell

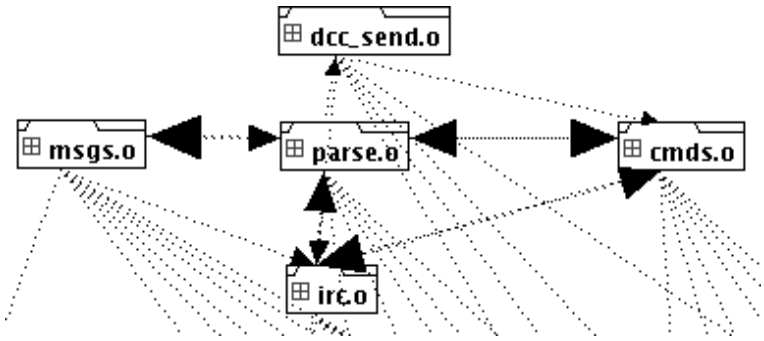


Figure 4.5: The IRC .libs SubSystem in LSedit

the strength of a dependency based on the size of the arrowhead. A larger arrow denotes more links between the functions contained in these compilation units; UML has no such distinction. Since a UML Class Diagram with multiple dependencies contains the same information as one with single dependencies between classes, it makes sense to omit this multiple dependency information from the XMI encoding of the model.

Figure 4.7 shows the internal structure of the parse.o file within the subsystem shown in Figure 4.5. In this case, it is possible to visualise the dependencies contained within an object file. Since LSedit is able to visualise dependencies between individual functions and objects, it makes sense to attempt to retain this information during a merge operation.

Figure 4.8 shows the difference between the dependencies in UML and LSedit. When merging links information into an existing UML model, the process is simple. It's a matter of lifting each links relationship to the file level and – if the involved entities exist in UML and the dependency does not – creating the dependency. To lift the links to the file level, one can use a combination of a few simple relational algebra calculations, demonstrated in the code below.

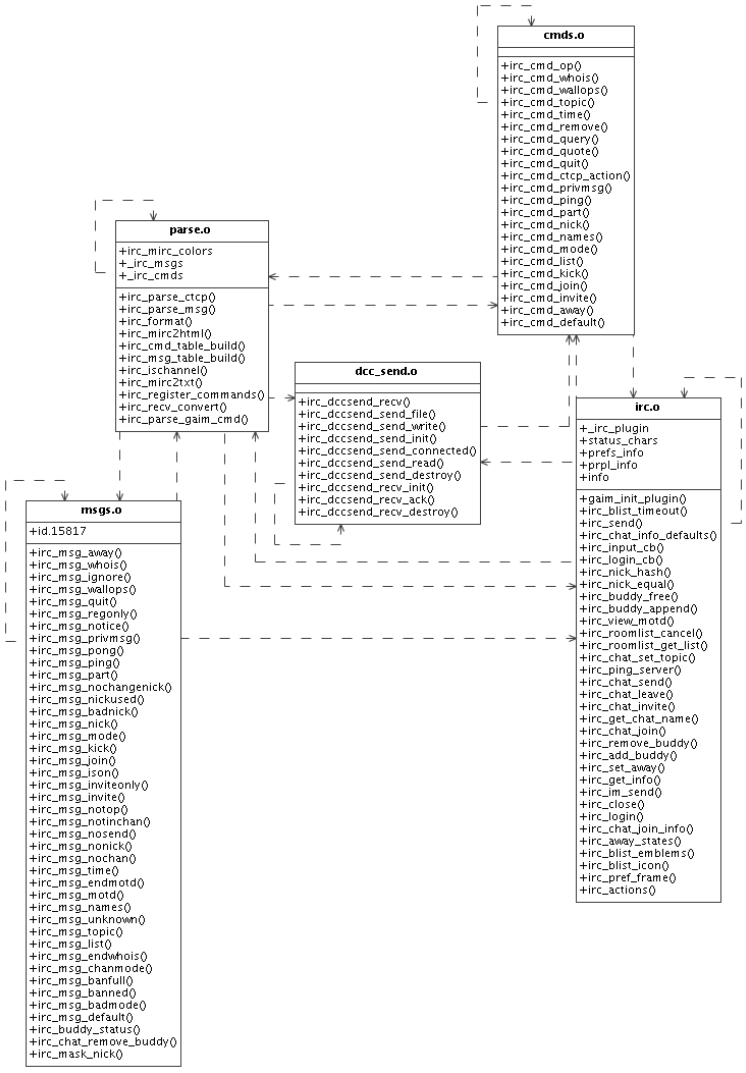


Figure 4.6: The IRC libs Package in UML

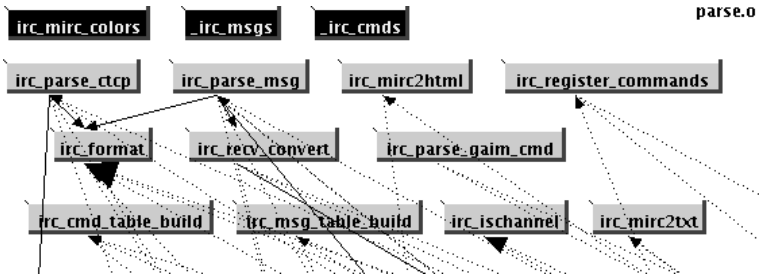


Figure 4.7: The IRC parse.o file in LSedit

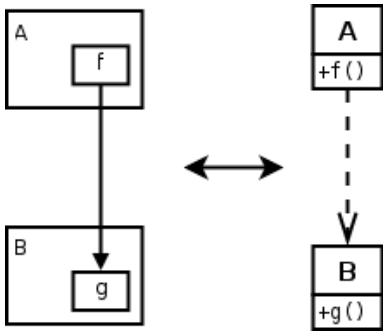


Figure 4.8: Dependencies in LSedit and UML

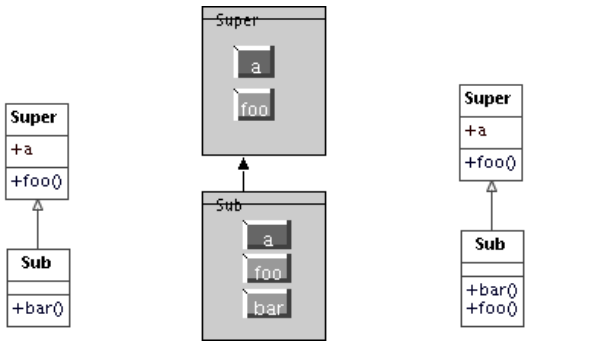
```
TupleSet lifted = AlgebraOperation.composition(
  AlgebraOperation.composition(hierarchy, calls),
  AlgebraOperation.inverse(hierarchy));
```

In the opposite direction, merging relationship information becomes more complicated. As described in Section 3.4.1, the naming schemes for entities in the UML and the FLS are different. Before comparing the links relationships, the names must be made consistent, using the containment structure of the existing TA factbase. Once this is complete, we go through the links in the existing factbase and individually lift it to be between the files, check to see if the lifted dependency exists in the UML, and if so, create the pre-lifted link in the new merged factbase using the new consistent entity names. If a dependency exists in the UML with no corresponding lower-level link in the factbase, we add a link between files to the merged factbase.

When exporting an existing UML model to FLS-based TA (rather than merging), the dependencies are exported as links between files rather than between functions, since there is no information in an existing factbase to provide more details. Importing a factbase into a new UML model is similar in that the existing links in the factbase are lifted to relationships between files and simply added as dependencies. These procedures guarantee the maintenance of as much detailed information about dependencies as possible during imports, exports and merges.

4.3 Generalization

Similar to the details concerning dependencies, information about generalization can be maintained during merge operations. The UML records certain information that the FLS



(a) UML (b) Exported TA (c) Merged UML

Figure 4.9: Translation of Generalization

for TA cannot. For example, in UML a generalization relationship is denoted by a particular arrow type. As described in Section 3.4.3, when a generalization relationship is exported to TA it becomes a regular links relationship between the .o files, and the functions in the superclass are copied to the subclass. When this information is merged into an existing UML model, we wish to maintain the generalization relationship.

Figure 4.9 shows an example of generalization in the two different notations, as well as what the UML model would look like after the exported TA factbase has been merged back in. As described in Section 2.3, when merging the new TA factbase back into UML, some information will be added to the UML model – Figure 4.9(c) shows the added information. The generalization arrow will not be replaced, as the existing UML model maintains this piece of semantic information. The methods that had to be added to the TA factbase to translate the generalization semantics, will be added to the existing UML model. It would be possible to avoid adding this information to the model, but we have chosen not to do so. A user may have intentionally left the subclass version of the function in the FLS-based

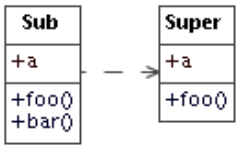


Figure 4.10: Imported TA Factbase

factbase, so we do not override the user's decision. On the other hand, if a subclass version of a function has specifically been *removed* from the FLS-based factbase, it will also be removed from the subclass in the UML model during a merge. Data members will not be re-added to subclasses because subclasses will never have a reason to redefine variables.

Figure 4.10 shows the differences between importing and merging a FLS-based factbase to a UML model when there has been generalization information added to the factbase. The primary difference between this figure and Figure 4.9(c) is that during a merge the arrow type is preserved, whereas with a simple import it is not. In addition, member variables will be added to the "subclass", contrary to the semantics of generalization.

Again, as with the detailed information about dependencies between functions, generalization relationships are one type of information that can be maintained during a merge but not during imports and exports. This is because each technical space has the ability to record semantic information that is irrelevant to another technical space. The fact that this information can be maintained during a merge implies the existence of a bridge domain that captures all of the architectural details of the software system; however, this bridge domain cannot be expressly encoded, and in order to get a complete and accurate picture of a piece of software, a view through multiple technical spaces must be considered.

Chapter 5

Conclusions

Our original motivation for this work was to improve the usability and usefulness of the SWAG architectural tools. Specifically, our goal, which we have accomplished, was to provide the ability to produce UML models from facts extracted using the fact extraction tools provided by SWAGkit.

There are many reasons why interoperable software tools are more useful than those that stand on their own. Different tools are suited to serving different purposes. It should be possible to share model information between different software design and architecture tools. Models can only be compared to each other when they are in the same format. A clear architectural picture of a software system is best obtained through a collection of views reflecting different features of the system. Each of these views may be managed by different tools, but they all work within the same bridge domain.

We believe it should be possible to work in any technical space and have the changes made by the developer or architect be reflected in a meaningful way in all other spaces.

We have made this a reality for the UML and SWAG technical spaces, as described in this work. It has been made clear to us through the completion of this work that no one technical space can fully reflect the actual architecture of a software system.

5.1 Contributions

We have provided a tool for bridging between the UML and SWAG technical spaces. Users can import a FLS-based TA factbase into a new UML model using MagicDraw UML. An XMI or MagicDraw-based UML model can also be exported to a new FLS-based TA factbase. These models can also be merged in either direction. Certain semantic information, as described in Chapter 4, can be maintained during the merge process, even though that information is not encoded in the space from which the information is being merged. This maintenance of semantic information that cannot be stored in both models is what points to the existence of a bridge domain.

Along with providing this tool for bridging these two technical spaces, we have provided the documentation on how to bring other tools into this work. In Appendices A and B, we have provided documentation on how to use the two APIs that are used in our MagicDraw UML plugin. The problems we encountered during this work and their solutions have also been documented, in the hope of allowing others to bypass a lot of difficulties when bridging between another technical space and one of the two for which our tool was developed.

This tool can be used for several applications. The most obvious is to get a clearer understanding of the actual structure of a piece of software. If a user starts with a UML model, it would be useful to be able to use the tools within SWAGkit to perform queries

on the facts in the model and to manipulate the model using one of the relational algebra calculators. On the other hand, if the model is being extracted from source code, an architect could use the SWAG tools to extract the facts about it then produce a set of UML diagrams documenting the system in a standard modeling language. Another potential application would be to use these tools to perform a very rough object discovery on an existing non-object-oriented system. Testing this particular application is left for future work.

5.2 Future Work

One obvious extension of this work would be to add the translation of layout information to our tool. Currently, we import model information programmatically, but use the tools built into MagicDraw UML to produce the UML class diagrams. These diagrams can be created programmatically through the OpenAPI instead. LSedit has the built-in ability to use layout information contained in the Attribute section of the TA factbase. The OpenAPI has the ability to determine where items are in the diagrams, and where to put diagram elements based on input from the TA factbase. As a result, translating the layout from the UML to LSedit would be a matter of generating the UML diagrams programmatically, determining a mapping from the MagicDraw coordinates to LSedit coordinates, and outputting the position information in the attribute section of the TA factbase. The opposite direction would be similar.

There are several encoding languages for models of software architecture that we would like to see brought into this collection of tools. GXL is an XML-based exchange format

designed by Holt *et al.* [23] as a standard for describing and interchanging graphs about software. RSF, part of the Rigi toolset developed at the University of Victoria [35], is an encoding language similar to TA. DOT is a language used for recording basic information about graphs. Bringing any of these languages into this tool would be useful. In addition, many diagramming tools are capable of producing UML diagrams of software, such as Dia or OmniGraffle. Both of these tools have the ability to export diagrams to text-based encoding languages, which means it would be possible to read in the diagram data and output it to TA or XMI.

Another possible extension would require some work on the fact extractors and schemas available in SWAGkit. It should be possible to extract some other types of information from the compiled (or pre-compilation) source code, such as enumerations and state transitions on those enumerations. Using this information, we could produce UML state transition diagrams. Using some of the work that has recently been done in SWAG to extract runtime information from executing programs, we could also produce UML sequence diagrams. These extensions would require new TA schemes for these new types of information, and new translation options for the MagicDraw UML plugin we have implemented.

Bibliography

[1] Gentleware AG. Poseidon for UML. http://gentleware.com.

[2] Jean Bézivin, Hugo Brunelière, Frédéric Jouault, and Ivan Kurtev. Model engineering support for tool interoperability. In *WISME 2005 - 4th Workshop in Software Model Engineering*, 2005.

[3] Grady Booch. Handbook of software architecture. http://www.booch.com/architecture.

[4] Grady Booch. *Object Oriented Design with Applications*. Benjamin/Cummings Pub. Co., Redwood City, California, USA, 1991.

[5] Borland Software Corporation. Borland Together. http://www.borland.com/us/products/together/.

[6] Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster. Linux as a case study: Its extracted software architecture. In *ICSE '99: International Conference on Software Engineering*, Los Angeles, California, USA, May 1999.

[7] L.C. Briand, Y. Labiche, and Y. Miao. Towards the reverse engineering of UML sequence diagrams. In *WCRE 2003: Proceedings of the 10th Working Conference on Reverse Engineering*, 2003.

[8] John R. Cameron. An overview of JSD. *IEEE Transactions on Software Engineering*, SE-12(2), Feb 1985.

[9] Peter Pin-Shan Chen. The entity-relationship model – toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[10] CollabNet. ArgoUML. http://argouml.tigris.org.

[11] Microsoft Corporation. Visio 2003. http://office.microsoft.com/en-us/FX010857981033.aspx.

[12] Holger Eichelberger. Aesthetics of class diagrams. In *Proceedings of the* 1^{st} *International Workshop on Visualizing Software for Understanding and Analysis*. IEEE, 2002.

[13] Mohammad El-Ramly, Eleni Stroulia, and Paul Sorenson. Mining system-user interaction traces for use case models. In *IWPC 2002: Proceedings of the 10th International Workshop on Program Comprehension*, 2002.

[14] The Eclipse Foundation. Eclipse Modeling Project. http://www.eclipse.org/modeling.

[15] The Eclipse Foundation. UML2. http://www.eclipse.org/uml2.

[16] Martin Fowler. *UML Distilled : A Brief Guide to the Standard Object Modeling Language*. Addison Wesley, Reading, Mass., 3rd edition, 2003.

[17] Object Management Group. *Unified Modeling Language: Infrastructure version 2.0*, March 2006.

[18] The Omni Group. Omnigraffle. http://www.omnigroup.com/applications/omnigraffle/.

[19] Rich Hilliard. Using the UML for architectural description. In *UML '99 - The Unified Modeling Language: Beyond the Standard, Second International Conference*, volume 1723, pages 32–48, Fort Collins, CO, USA, October 1999. Lecture Notes in Computer Science.

[20] C. Hofmeister, R. L. Nord, and D. Soni. Describing software architecture with UML. In *Proceedings of the First Working IFIP Conference of Software Architecture*, 1999.

[21] Ric Holt. TA: The tuple attribute language, February 1997.

[22] Ric Holt, Andrew Malton, and Tom Dean. CPPX: Open source c++ fact extractor. http://www.swag.uwaterloo.ca/∼cppx/.

[23] Ric Holt, Andy Schürr, Susan Elliott Sim, and Andreas Winter. Graph eXchange Language, July 2002. http://www.gupro.de/GXL/.

[24] Richard C. Holt, Michael W. Godfrey, and Andrew J. Malton. The build / comprehend pipelines (position paper). In *Second ASERC Workshop on Software Architecture*, February 2003.

[25] IBM. IBM Rational Software. http://www-306.ibm.com/software/rational/.

[26] Igor Ivkovic and Kostas Kontogiannis. Tracing evolution changes of software artifacts through model synchronization. In *ICSM '04: Proceedings of the* 20^{th} *IEEE International Conference on Software Maintenance*, pages 252–261, Washington, DC, USA, 2004. IEEE Computer Society.

[27] Dean Jin, James R. Cordy, and Thomas R. Dean. Where's the schema? A taxonomy of patterns for software exchange. In *IWPC '02: Proceedings of the* 10^{th} *International Workshop on Program Comprehension*, page 65, Washington, DC, USA, 2002. IEEE Computer Society.

[28] Thomas Klein, Ulrich A. Nickel, Jörg Niere, and Albert Zündorf. From UML to Java And Back Again. Technical report, University of Paderborn, Paderborn, Germany, September 1999.

[29] Eleftherios Koutsofios and Stephen North. Drawing graphs with dot, February 2002. http://www.graphviz.org/Documentation/dotguide.pdf.

[30] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[31] Michael Kusugak. Qallupilluit, 1986. From Personal Correspondence with Robert Munsch.

[32] Alexander Larsson. Dia a drawing program. http://www.gnome.org/projects/dia/.

[33] Timothy C. Lethbridge, Sander Tichelaar, and Erhard Ploedereder. The dagstuhl middle metamodel: A schema for reverse engineering. In *ateM 2003: Proceedings of the International Workshop on Meta-Models and Schemas for Reverse Engineering*, May 2004.

[34] Andrew Malton, Kevin A. Scheneider, James R. Cordy, Thomas R. Dean, Cousineau Darren, and Jason Reynolds. Processing software source text in automated design recovery and transformation. In *IWPC '01: Proceedings of the* 9^{th} *International Workshop on Program Comprehension*, pages 127–134, Toronto, ON, Canada, 2001. IEEE.

[35] Hausi A. Müller, Scott R. Tilley, and Kenny Wong. Understanding software systems using reverse engineering technology: Perspectives from the Rigi project. In *CASCON '93: Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research*, pages 217–226. IBM Press, 1993.

[36] Robert Munsch. Personal Communication, February 2006.

[37] Robert Munsch and Michael Kusugak. *A Promise Is a Promise*. Annick Press (Classic Munsch), 1992.

[38] Ulrich A. Nickel, Jörg Niere, Jörg P. Wadsack, and Albert Zündorf. Roundtrip engineering with FUJABA. In *WSR: Proceedings of the 2^{nd} Workshop on Software-Reengineering*, Bad Honnef, Germany, August 2000. Fachberichte Informatik, Universität Koblenz-Landau.

[39] No Magic, Inc. MagicDraw UML. http://www.nomagic.com.

[40] No Magic, Inc. *MagicDraw OpenAPI UserGuide*, February 2006.

[41] Object Management Group. *Meta Object Facility (MOF) 2.0 XMI Mapping Specification, v2.1*, September 2005.

[42] Omondo. EclipseUML. http://www.omondo.com.

[43] Atousa Pahlevan. Enhancing static architecture design recovery by lightweight dynamic analysis. Master's thesis, School of Computer Science, University of Waterloo, 2006.

[44] Visual Paradigm. Visual Paradigm for UML. http://www.visual-paradigm.com/product/vpuml.

[45] University of Waterloo Software Architecture Group. About SWAG kit. http://www.swag.uwaterloo.ca/swagkit/index.html.

[46] University of Waterloo Software Architecture Group. Grok. http://www.swag.uwaterloo.ca/tools.html#grok.

[47] University of Waterloo Software Architecture Group. Grokdoc. http://www.swag.uwaterloo.ca/∼nsynytskyy/grokdoc/index.html.

[48] University of Waterloo Software Architecture Group. Lsedit. http://www.swag.uwaterloo.ca/lsedit/index.html.

[49] Athanasios Staikopoulos and Behzad Bordbar. A metamodel refinement approach for bridging technological spaces, a case study. In *WISME 2005: Bridging Technical Spaces and Model-Driven Evolution*, 2005.

[50] Nikita Synytskyy. Setting up and using the bfx pipeline. http://www.swag.uwaterloo.ca/qldx/README.bfx.html.

[51] Nikita Synytskyy. Setting up and using the ldx-bfx pipeline. http://www.swag.uwaterloo.ca/qldx/README.ldx.bfx.html.

[52] Robert J. Walker, Gail C. Murphy, Bjorn Freeman-Benson, Darin Wright, Darin Swanson, and Jeremy Isaak. Visualizing dynamic software system information through high-level models. In *OOPSLA '98: ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1998.

[53] Bernhard Westfechtel. Structure-oriented merging of revisions of software documents. In *SCM-3: Proceedings of the* 3^{rd} *International Workshop on Software Configuration Management*, pages 68–79, New York, NY, USA, 1991. ACM Press.

[54] Jingwei Wu. *Open Source Software Evolution and Its Dynamics*. PhD thesis, School of Computer Science, University of Waterloo, 2006.

[55] Albert Zündorf, Jörg P. Wadsack, and Ingo Rockel. Merging graph-like object structures. In *(SCM-10) Proceedings of the* 10^{th} *International Workshop on Software Configuration Management*, Toronto, Ontario, Canada, May 2001.

Appendix A

Using the QL API

In this appendix, we describe the Java classes from the implementation of QL [47] that we used in implementation of our tool. This information is provided for readers who may wish to extend our work. In order to use these classes, `ql.jar` and `java_readline.jar` must be on the Java classpath. These files are both available upon request to Jingwei Wu, the author of QL (http://swag.uwaterloo.ca/∼j25wu/).

A.1 ca.uwaterloo.cs.ql.io.TAFileReader

This class is used to read in the information from a TA file. The `read(String filename)` method returns a `Factbase`, ready to be used as described below.

A.2 ca.uwaterloo.cs.ql.fb.Factbase

A `Factbase` contains a collection of `Tuple`s. These can be accessed in several ways. An `EdgeSet` containing all `Edge`s of a particular relation can be obtained with the method `getEdgeSet(String name)`. For instance, to obtain all `call` relations, one would use `getEdgeSet("call")`. An `EdgeSet` containing all nodes in the graph can be retrieved with `getEdgeSet("$INSTANCE")`. On the other hand, a `NodeSet` containing all instances of a particular type of node can be retrieved with a call to `getNodeSet(String name)`.

A.3 Tuple Classes

A.3.1 ca.uwaterloo.cs.ql.fb.TupleSet

`ca.uwaterloo.cs.ql.fb.EdgeSet` and `ca.uwaterloo.cs.ql.fb.NodeSet` are both subclasses of `TupleSet`. As the name suggests, a `TupleSet` is a set of `Tuple`s. In addition to being a container, the `TupleSet` class provides the functionality to output element data in TA form, using `printTA(java.io.OutputStream out)`. This class also provides the functionality to return a `TupleList` (`getTupleList()`), return an array of `Tuple`s (`getAllTuples()`), and remove duplicate items from the set.

A.3.2 ca.uwaterloo.cs.ql.fb.TupleList

The `TupleList` class simply provides the regular operations expected from a `List`. It will return a `Tuple` given an index, return an `Iterator` for the elements of the list, and so on.

A.3.3 ca.uwaterloo.cs.ql.fb.Tuple

The `Tuple` class is probably the most important class for directly interacting with the data. This class provides the ability to get the domain and range of the tuple, using `getDom()` and `getRng()`. It also provides matching `set` methods. These methods get and set integer values, which are internal node IDs. To get the `String` names of the entities, one must use the static method `ca.uwaterloo.cs.ql.fb.IDManager.get(int id)`. `IDManager` also provides the opposite method, for retrieving an ID: `getID(String name)`.

A.4 ca.uwaterloo.cs.ql.fb.Show

`Show` can be used to obtain information about a node's attributes. `Show.getAtt(int nodeID, EdgeSet att)` will return a node's attribute value. The `EdgeSet` required is the `EdgeSet` from the factbase containing all of the attributes of a particular type. The attributes are referred to using "@" in front of the attribute name. For example, if one wanted to retrieve all of the labels on a graph, one would use `fb.getEdgeSet(``@label'')`. To obtain a particular label, the static method `getAtt()` must be used, as described above.

A.5 ca.uwaterloo.cs.ql.fb.AlgebraOperation

`AlgebraOperation` provides static methods for the operations that can be performed between `TupleSet`s, `EdgeSet`s and `NodeSet`s that are provided by QL. These operations include composition, inverse, intersection, transitive closure, difference, and so on. The JavaDoc for this class should be consulted for more details on the individual operations.

Appendix B

Using the MagicDraw UML 10.5 OpenAPI

In this Appendix, we describe the classes that are relevant to developing a plugin similar to ours, in MD UML using the OpenAPI. For the sake of clarity, we have omitted the package names, unless they are relevant to the discussion. The JavaDoc included in an installation of MagicDraw makes it easy to determine the package of a particular class. More details can be found in the Open API User's Guide [40].

B.1 Creating MD UML Plugins with a Menu

To create a plugin in MD UML, one must begin with a class that inherits from Plugin. This class requires `init()`, `close()` and `isSupported()` methods. If menu options are desired, the init method should create an object that is a specialisation of `AMConfigurator`, and

add it to the `ActionsConfiguratorsManager`. The specialisation of the `AMConfigurator` is where menu options are added.

The actual plugin consists of the Java .jar file containing the compiled code, and a plugin.xml file telling MD UML where to find the class that inherits from `Plugin`. The plugin.xml file should also contain information on where to find any .jar libraries that are needed to run the code. More information on the formatting of the plugin.xml file can be found in [40].

B.2 Dealing with UML Models and Elements

UML models are made up of `Element`s. There are several classes included in the OpenAPI to assist with handling `Model`s and their `Element`s.

`ModelElementsManager` is a utility class for adding, moving, and removing model elements.

`ModelHelper` is a utility class for finding parents of `Element`s, and getting and setting client and supplier `Element`s on `Relationship`s.

Other utility classes include `RepresentationTextCreator`, which is useful for retrieving various information about an `Element`. For example, `getFullUMLName(Element)` will return the path to the root element, with path items separated by '::'. For example, a method named 'method' in the class 'Class' in the package 'package' will have a UML Name of "package::Class::method". As the name suggests, `getPathToRoot(Element)` will return the full path to the root element, which can be useful for certain operations.

In the MagicDraw implementation, visiting model elements is based on the Visitor

Design Pattern. Every `Element` can accept a `Visitor`. To implement a specific `Visitor`, you simply override the visit methods for each relevant element type in your own extension of the `Visitor` class. Using this method, you can restrict your application to only visit the types of model elements that are relevant to your interests. For example, in our export operation, we visit Packages, Classes, Dependencies, Generalisations, Associations, Properties, and Operations. We can then perform specific operations based on which type of model element is being visited.

Creating model elements can be done using the `ElementsFactory` class. Instances of the model `Element`s are created, then the attributes can be modified using methods specific to the type of the `Element`. `VisibilityKindEnum` provides `PACKAGE`, `PRIVATE`, `PROTECTED`, and `PUBLIC` visibility types for setting the visibility of a property or operation.

B.3 UML Model Elements

The package `com.nomagic.uml2.ext.magicdraw.classes.mdkernel` contains classes for most of the various types of elements in UML. `Element` is the superclass of each of these types. The methods available in each of these classes are extensive. For this reason, we will not document them here, other than to state that these classes exist. More details on each individual class can be found in [40]. These classes directly follow the UML2 specifications.

1. Class
2. DirectedRelationship
3. Generalization

4. Operation

5. Package

6. Property

The class for Depenencies is found in:

`com.nomagic.uml2.ext.magicdraw.classes.mddependencies.Dependency`.