Modelling Architectural Primitives in UML

Author: Nick Kirtley

Supervisors: Paris Avergiou
Ahmad Waqas Kamal

Date: 1st December 2007

Organisation: Department of Mathematics and Computing Science
University of Groningen
Table of Contents

1 Introduction .................................................................................. 7
   1.1 Software Architecture ............................................................... 7
      1.1.1 The Four Views ................................................................ 8
   1.2 Model Driven Architecture .......................................................... 8
   1.3 Architectural Patterns ................................................................. 9
   1.4 Architectural Primitives ............................................................... 10
   1.5 Modelling Frameworks ............................................................... 11
   1.6 Research Goals ........................................................................ 11
      1.6.1 Problem Statement .............................................................. 12
      1.6.2 Research Questions ............................................................. 12
   1.7 Proposed Solution ..................................................................... 13
   1.8 Overview .................................................................................. 13

2 Choice of Framework ...................................................................... 14
   2.1 General Requirements ............................................................... 14
   2.2 Results ..................................................................................... 14
      2.2.1 ACME ............................................................................. 15
      2.2.2 Eclipse MDT ..................................................................... 15
      2.2.3 Visio ............................................................................... 15
      2.2.4 Argo UML ........................................................................ 15
      2.2.5 Archium .......................................................................... 16
   2.3 Framework Selection .................................................................. 16

3 The Eclipse Platform ....................................................................... 17
   3.1 Plug-in Development ................................................................. 18
   3.2 Plug-in Types ........................................................................... 18
   3.3 Eclipse UML2 Plug-in ................................................................. 18
   3.4 Eclipse OCL Plug-in ................................................................. 19
   3.5 Eclipse UML2Tools Plug-in ....................................................... 19
   3.6 Primus Plug-in ......................................................................... 20
      3.6.1 Events ............................................................................ 20
      3.6.2 Validation ....................................................................... 21
      3.6.3 Creating Primitives ........................................................... 22
      3.6.4 Eclipse Wizard ................................................................. 22

4 Unified Modelling Language .......................................................... 23
   4.1 Four Layer Meta-Model ............................................................. 23
   4.2 Metamodel ............................................................................... 24
   4.3 UML Elements used by primitives ............................................ 25
      4.3.1 Packages ......................................................................... 25
      4.3.2 Components .................................................................... 26
      4.3.3 Ports ............................................................................... 26
      4.3.4 Classes ........................................................................... 26
      4.3.5 Interfaces ....................................................................... 26
4.3.6 Connectors ................................................................. 27
4.4 Extending UML ............................................................ 27
4.5 UML Modelling ............................................................. 28
  4.5.1 Required and Provided Interface ................................. 28
  4.5.2 Connector ............................................................. 29
  4.5.3 Association .......................................................... 29
5 Object Constraint Language ................................................. 31
  5.1 General OCL Commands Constructs ............................. 31
    5.1.1 All Instances .................................................... 31
    5.1.2 Iterate .......................................................... 31
    5.1.3 List Types ...................................................... 32
    5.1.4 Tuple ........................................................... 32
    5.1.5 Types .......................................................... 32
    5.1.6 Select .......................................................... 32
    5.1.7 Any ............................................................... 33
6 Primitives ............................................................................ 34
  6.1 Callback ................................................................. 34
    6.1.1 General Description .............................................. 34
    6.1.2 Eclipse UML2 Modelling M2 Level .......................... 34
    6.1.3 Eclipse UML2 Modelling M1 Level .......................... 35
    6.1.4 Callback Validation ............................................. 35
    6.1.5 Near Callback .................................................... 38
  6.2 Indirection ............................................................... 39
    6.2.1 General Description .............................................. 39
    6.2.2 Eclipse UML2 Modelling M2 Level .......................... 39
    6.2.3 Eclipse UML2 Modelling M1 Level .......................... 40
    6.2.4 Indirection Validation .......................................... 41
    6.2.5 Near Indirection ................................................ 43
  6.3 Grouping ................................................................. 43
    6.3.1 General Description .............................................. 43
    6.3.2 Eclipse UML2 Modelling M2 Level .......................... 44
    6.3.3 Eclipse UML2 Modelling M1 Level .......................... 44
    6.3.4 Grouping Validation ........................................... 44
    6.3.5 Near Grouping ................................................... 45
  6.4 Layers ......................................................................... 45
    6.4.1 General Description .............................................. 45
    6.4.2 Eclipse UML2 Modelling M2 Level .......................... 46
    6.4.3 Eclipse UML2 Modelling M1 Level .......................... 46
    6.4.4 Layers Validation ................................................ 46
    6.4.5 Near Layers ....................................................... 48
  6.5 Aggregation Cascade .................................................... 48
    6.5.1 General Description .............................................. 48
    6.5.2 Eclipse UML2 Modelling M2 Level .......................... 49
10 Conclusion and Future Work ................................................................. 88
  10.1 Conclusions .................................................................................. 88
  10.2 Future Work .................................................................................. 90
    10.2.1 Increased amount of Options for Adding Primitives to Model ....... 90
    10.2.2 Improved Visualisation ............................................................. 90
    10.2.3 More Extensive Testing of Queries ........................................... 91
    10.2.4 Code Generation ..................................................................... 91
    10.2.5 Defining New Primitives ........................................................ 91
    10.2.6 Mining Patterns in Software Design using Primitives ............... 92

Appendix A  Project Plan
Appendix B  High Level Design
Appendix C  Detailed Design
Appendix D  Updated Design
Appendix E  Developer and User Guide
Abstract

Architectural primitives are small recurring logical building blocks of architectural patterns. They are recurring in the sense that they can be found in many architectural patterns. This paper proposes to implement the concept of architectural primitives in a UML framework. In order to implement architectural primitives in a UML framework it is necessary to define the exact UML elements and stereotypes needed to make a primitive. Architectural primitives will have to be validated using automated model checking to make sure that they have been applied properly. Validation of primitives consists of checking that they have been applied properly to the UML model and that they adhere to all of the constraints defined in [1].

**Keywords:** Architectural Patterns, Architectural Primitives, Modelling, UML.
1 Introduction

This paragraph will introduce the research problem. An essential part of understanding the problem lies in understanding the domain in which it resides and the direction that the domain is taking. The domain of this project is Software Architecture and in particular Model Driven Architecture.

The following sub-paragraphs will describe what Software Architecture and MDA is so that the context of the research goals will be understood when described later on. Architectural primitives will also be introduced as they are an essential part of the research goals. The end of the paragraph consists of an overview of the rest of the document.

1.1 Software Architecture

Software architecture is a relatively new technical field. It consists of designing software at a higher level of abstraction than detailed designs that most are familiar with. Software architects can be compared to with architects of buildings or civil engineering in that they create the designs of the system or building and the programmers or builders build the system or building. In fact, Christopher Alexander is seen as the ‘father’ of pattern languages and is an architect [11].

There are also differences between software architecture and ‘normal’ architecture. Most notably, software architecture is dynamic and ‘normal’ architecture is relatively static. In software architecture the state of the system can change considerably during runtime. This requires methods to be able to map this change. The actual design of the software architecture can also change considerably. For example, if a system needs some sort of upgrade, then the system design can change to model the changes produced by the upgrade.

Software architecture is as much about designing the structures of the high level system as it is about communicating the many issues of an architectural design with the stakeholders. The types of issues communicated are dependent on the type of stakeholder, for example, the end-user stakeholder group would be interested in ease of use of the user interface. In fact communicating with the stakeholders, who may not have a technical background, is important because it can highlight problems in the requirements of the system at an early stage and thus save time and money.

Software architecture as an abstraction is achieved by encapsulating many of the details that are needed in the final design into elements within an architectural design. One drawback or difficulty to such a solution is that it is possible that important concepts relevant to the architectural design are left out and encapsulated within an element. This can only be avoided by prudent choices from the architect which comes with experience. This shows that problems can arise when a system is ‘over’ abstracted.

Even when applying best practices of software architecture it is still impossible to create a perfect system. Each system solves a certain problem and the solution also consists of trade-offs that the
architect must make. Examples of trade-offs are performance versus extendibility or design simplicity versus source code optimization. A good system is one where the right trade-offs have been made for the right reasons. Moreover, it is important that these trade-offs are documented so that architects and developers know why certain decision have been taken.

1.1.1 The Four Views

Software architecture is often separated into different views [3]. Each view shows a different aspect of the architectural design. This is done to help people deal with the complexity of large systems. The four views consist of the code view, module view, execution view and the conceptual view. A short description of each view will be given:

The code view deals with the source code of the system. In the past the source code of a program was fairly small but as the complexity of programs has increased so has size of the source code. The code is often split into many different types of files and many different versions and therefore requires its own view.

The module view splits systems into modules to help deal with complexity due to size and the ability to split work amongst groups.

The execution view deals with the problems that arise when large and distributed systems share resources and handle coordination and synchronization between components.

The conceptual view is the most relevant to this project because it deals with the components and the interconnections of those components within a system. This is the domain in which the project will take place. It may also be referred to as the components and connectors view. The conceptual view can play a primary role over the other three views.

1.2 Model Driven Architecture

Model Driven Architecture is a way of describing a system by using models at its centre [12]. This is a major shift in describing systems as opposed to using procedural or object-oriented programming. MDA has been proposed by the OMG and uses UML and MOF (Meta-Object Facility) as the language that describes the models. It is beyond the scope of this document to go into too much detail about MDA but it is important to realise that models could be used instead of programming languages for describing the general structure of a program/system.

A concept important to MDA and to this project is the Modelling Maturity Levels. The MML describes the maturity level of a model ranging from it having no structure or meaning at all to it being able to provide an exact specification:

MML0: No specification
MML1: Textual specification, a specification written in a natural language
MML2: Text with models, a specification written in a natural language along with basic models
MML3: Models with text, models describe the specification with additional text
MML4: Precise models, models describe the specification and are very precise
MML5: Models only, precise specification with models that allow for automatic code generation

MML4 is what OMG calls Model Driven Architecture. MML4 is important because it allows a modeller to model a specification or system at a sufficient level of precision without the need for natural language and interpretation by an engineer. Precision is better than interpretation because interpretation can differ from person to person and thus lead to differing solutions.

At MML4 the model is precise enough to have a link to actual code [2]. Examples of UML to code mapping include UML operations, with parameters and return type, to code methods and UML attributes become a code field or variable. The ultimate goal is to go to MML5 which will allow for a system to be generated directly from the model(s) without any coding. UML models are also constrained using Object Constraint Language (OCL) to achieve more precision. More information about OCL can be found in the OCL and primitives paragraph.

This project is not directly concerned with MDA but it is partly about modelling precise models that could one day be used in MDA and also indicates the general direction that industry and academia is taking.

1.3 Architectural Patterns

When developing a system it is highly unlikely that it is the first system of its kind or the first system with a certain structure to have been built to date. Engineering is usually a case of incrementally improving existing systems or structures. It is also safe to say that the problems encountered when building new systems are not the first time that they have appeared. Patterns make use of this fact and document problem and solution pairs.

Every pattern consists of a context, problem and solution. The context describes the general situation that the problem resides in. The problem describes the difficulty that repeatedly arises in a given context and a solution describes a proven solution that architects have developed in the past. Patterns can also be divided into categories based on general functionality. For example, some patterns are used for concurrency and networking issues and some patterns are used for distributed computing issues.

Patterns exist at a design and architectural level. Design patterns are used to describe problem/solutions to smaller problems usually within the subsystems of software architecture [13]. Architectural patterns differ in that they are used to describe problem/solutions for system-wide issues [14].
Architectural patterns are templates for concrete software architectures that specify a solution to a specific problem. The solution concerns system-wide issues at a high-level of abstraction. The details of the solution are not of interest and can be solved at a later stage. Architectural patterns are of particular interest to this project because the goal of this project is essentially to model these patterns using primitives (See next sub-paragraph).

1.4 Architectural Primitives

Architectural primitives are recurring primitive abstractions found in patterns [1]. Due to the fact that they are found in many patterns means that they can be used as building blocks to construct patterns and architecture designs. They are called architectural primitives because they are the smallest logical building blocks at an architectural design level [1].

The UML will be extended in order to define architectural primitives. It is possible to extend the UML in a ‘soft’ and ‘hard’ way. The hard way consists of creating new elements and the soft way consists of extending the existing elements using stereotypes, tag definitions and constraints in the form of OCL queries. The soft way is the approach that will be taken in this project because it has the advantage that existing modelling programs work with the standard UML meta-model and because developers and architects are familiar with standard UML.

[1] Identifies a set of primitives that have been identified by examining architectural patterns. This list is not a definitive list as there are no doubt other primitives needed to model patterns that haven’t yet been discovered. The primitives and their purpose can be found in Table 1. This is only an introduction to primitives and a more detailed explanation can be found in the primitives paragraph and in the paper by Zdun and Avgeriou [1].

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Callback</td>
<td>A mechanism that allows for a component to store a reference from another component so that it can be invoked later on</td>
</tr>
<tr>
<td>Indirection</td>
<td>A mechanism that allows for redirection of a message/communication to a target component with the purpose of hiding the location of the target component</td>
</tr>
<tr>
<td>Grouping</td>
<td>A group of components with a logical commonality</td>
</tr>
<tr>
<td>Layers</td>
<td>Groups of components ordered with constraints on communication between layers</td>
</tr>
<tr>
<td>Aggregation Cascade</td>
<td>An aggregation cascade consists of a whole-part relationship between components</td>
</tr>
<tr>
<td>Composition Cascade</td>
<td>Builds on aggregation cascade with further constraints that only allow a component to be part of one composite</td>
</tr>
<tr>
<td>Shield</td>
<td>Shields a group of components from the outside world. Only to be accessed through a special access point component</td>
</tr>
<tr>
<td>Typing</td>
<td>A mechanism that allows for components to be typed and super typed.</td>
</tr>
<tr>
<td>Virtual Connector</td>
<td>A logical connection between two components. They are not connected with a connector but have some connection between intermediaries</td>
</tr>
</tbody>
</table>

Table 1: Architectural primitives general description
1.5 Modelling Frameworks

Modelling frameworks are used to model systems. There are many types of modelling frameworks based on the domain of the required model. The purpose of a modelling framework is to be able to model a system, often graphically, that will describe how a system can be designed and built. Once a system has been modelled it is also possible to test a model. A model can be tested for particular behaviour under certain circumstances and allows modellers to test a model for aspects or concerns that they deem important. It is important to understand that a model has semantic meaning. Each modelling element in a model has a concept or behaviour associated with it. It is also important to realise that with semantic meaning comes rules. The rules are used to make sure that the modelling elements behave properly according to their function and are connected in a logical fashion.

Modelling frameworks that are of particular interest to this project are modelling frameworks for software architecture and design. There are many examples currently available: Microsoft Visio, IBM’s Rational Rose, IBM’s Rational Software Architect, Eclipse Omondo and Eclipse MDT to name a few. The modelling capabilities of the various frameworks listed vary in scope. For example, IBM’s Rational Rose is a large and complicated system that can model the intricacies of a software system at an architectural level and Microsoft Visio only offers limited support for software systems but has the ability to present models that are aesthetically pleasing.

A modelling framework will be chosen to implement the concept of architectural primitives. A comparison of frameworks will be made to determine which will be best to work with.

1.6 Research Goals

Architectural patterns can improve the quality of a software system by providing known solutions to problems encountered within a similar context. However, the solution provided can have many variability issues and requires the architect to model the solution in his or her own way. One solution can therefore be modelled in many different ways. This variation in modelling can lead to problems because the variations are not tried and tested. In a way this is the same problem that is encountered when not using architectural patterns at all.

Architectural primitives can solve this problem by offering tried and tested building blocks to model architectural patterns that allow for variation in pattern variability instead of modelling pattern variability in an ad-hoc way.

The goal of this thesis is to implement architectural primitives in a development environment. The development environment has yet to be defined and should be chosen based on how well it can support the development of architectural primitives and the quality of the meta-model implementation.
1.6.1 Problem Statement

Theoretically the principle of architectural primitives has potential in improving the way that architectural patterns are modelled. However, architectural primitives have not been implemented in an actual modelling environment and it is possible that certain unforeseen problems exist in their development.

The practicality of using architectural primitives should also be validated once they have been developed in a modelling environment. This can be achieved by modelling an architectural system based on architectural patterns using architectural primitives. A peer-to-peer system called Leela has already been defined in [1].

Implement the principle of architectural primitives in a modelling framework and validate the implementation through a non-trivial case study.

1.6.2 Research Questions

An existing modelling framework is needed to implement architectural primitives. This framework should support a meta-model for its model elements (Preferably one that is known in the architectural community), a constraint system (Preferably one that is known in the architectural community) and it should be extensible to allow for development work.

1. What framework best suits the needs of modelling architectural primitives?

Architectural primitives are defined by many different types of elements interacting with each other. Each meta-model has a different way of describing how, for example, two components are connected to each other. Primitives have been defined by Zdun and Paris in [1] but how should they be defined in a modelling framework?

2. How will the primitives, which have been defined using UML elements and stereotypes, be modelled in the framework that has been chosen?

Preferably a constraint or query language is used to find the primitives in a model and to make sure they have been constrained properly. Are there any implementation issues associated with using a constraint/query language instead of a general purpose language like Java?

3. How to perform model checking, which consists of assessing the validity of the primitives and defining which UML elements make up the primitives, as efficiently as possible?
The current list of primitives is not exhaustive and more primitives are likely to be found in the future. This means that primitives will also have to be added to the implementation. The question is how easily this can be achieved.

4. *How to make the framework as extensible as possible?*

### 1.7 Proposed Solution

The proposed system is called Primus, which is Latin for primitive, and will consist of a soft extension to the UML meta-model to define architectural primitives. It will have the 9 primitives from Table 1 defined so that they can model architectural patterns. OCL will be used to locate the primitives and check that the proper constraints have been met so that the user can quickly find the primitives used and to check that they have been applied properly.

### 1.8 Overview

The rest of the document is used to describe the steps taken to investigate the problem statement and to answer the research questions.

Paragraph 2 describes the steps taken to choose a framework suitable for implementing the principle of architectural primitives.

Paragraph 3 describes Eclipse and the plug-ins of Eclipse used in this project.

Paragraph 4 describes UML 2.0 in general and the steps needed to model primitives in Eclipse UML2 by using the editors and in Java code.

Paragraph 5 describes OCL in general and the programming constructs needed to properly query and constrain the architectural primitives.

Paragraph 6 details the architectural primitives purpose, precise UML modelling and OCL queries.

Paragraph 7 describes the results of the project by using the primitives to model an actual software architecture design.

Paragraph 8 discusses an improved method of validation that can be applied to primitives.

Paragraph 9 discusses related work from other research groups.

Paragraph 10 concludes, answers research questions and discusses future work that can further the development of architectural primitives and its implementation in UML modelling.
2 Choice of Framework

The main objective of the project is to implement the principle of architectural primitives in a modelling framework that has a comprehensive meta-model and preferably one with the UML meta-model to allow for proper UML modelling. This required an investigation to find such a framework that would also meet other requirements needed to fulfil the project goals. All (General) requirements and needs of a framework to model architectural primitives are described in this paragraph. A shortlist of five frameworks has been made based on the fact that they all deal with modelling (UML) concepts.

2.1 General Requirements

The following lists the criteria needed for a framework that will implement the principle of architectural primitives. An explanation is also provided per point of criteria.

Documentation regards the documentation of the design and development of the tool and not how the tool is used.

Stability is based on how long the tool exists and in what phase tool development is in.

Community is based on the size of the developers and users community.

Model complexity regards the complexity of the current meta-model. The meta-model will have to be extended to achieve full UML support and architectural primitive support.

Adaptability is based on the ease of changing the existing plug-in. This based on which underlying IDE it uses.

Reusability is based on how much of the existing plug-in we can use. The more that can be reused the less will have to be programmed from scratch.

Maintainability is the flexibility of the selected tool to accommodate our future requirements.

2.2 Results

The results of the analysis are based on a review from information found on the Internet for each framework.

<table>
<thead>
<tr>
<th>Criteria</th>
<th>ACME</th>
<th>Eclipse/MDT</th>
<th>Visio</th>
<th>ArgoUML</th>
<th>Archium</th>
</tr>
</thead>
<tbody>
<tr>
<td>Documentation</td>
<td>-</td>
<td>++</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Stability</td>
<td>+</td>
<td>++</td>
<td>++</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Community</td>
<td>-</td>
<td>++</td>
<td>-</td>
<td>+</td>
<td>--</td>
</tr>
<tr>
<td>Model complexity</td>
<td>0</td>
<td>+</td>
<td>0</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Adaptability</td>
<td>++</td>
<td>++</td>
<td>-</td>
<td>++</td>
<td>+</td>
</tr>
<tr>
<td>Reusability</td>
<td>++</td>
<td>+</td>
<td>--</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Maintainability</td>
<td>+</td>
<td>++</td>
<td>0</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>
Table 2: Results of tool analysis

Scale:
0: Not Studied
-: Negative
--: Very Negative
+: Positive
++: Very Positive

2.2.1 ACME
ACME clearly lacks in documentation and there is virtually no community. Due to the lack of documentation it is impossible to determine the structure of the underlying meta-model. ACME scores high in adaptability (eclipse) and reusability. The reusability is the main advantage of ACME.

Bottom line: ACME is a close match with the SADT project but the serious lack of documentation means that we would be very dependent on their input.

Solution:
Proper documentation and personal assistance will be needed from ACME studio developers.

2.2.2 Eclipse MDT
The MDT project is very large and so is the user base. This means that the documentation and community support is excellent. MDT is obviously model based so the match is very close to SADT. In fact the meta-model used is actually based on the UML 2.x metamodel. Another added advantage of EMF is that it provides a wide range of plug-ins; some of these plug-ins are developed to draw/visualise UML diagrams which can be further extended to fulfil our software requirements.

Bottom line: Eclipse MDT has excellent support and the reusability is high due to the model based features.

2.2.3 Visio
Visio is primarily a drawing tool and does not offer enough support to develop a SADT plug-in.

Bottom line: not suitable

2.2.4 Argo UML
Argo has fairly high support for documentation but it is not yet very stable. The meta-model is not as well described as EMF. There have been many quotes of slow fix rate of bugs.

Bottom line: A basic environment that is not as good as EMF.
2.2.5 Archium

Archium is an ADL based on design decisions. The documentation and support is poor but has the advantage that it was developed locally in our department and thus questions can be answered in person.

2.3 Framework Selection

The analysis shows that there are two suitable tools; Eclipse MDT and ACME studio. An essential part of this project is to use the UML 2 metamodel. ACME studio provides no information as to what metamodel is used and so it is very difficult to say if it closely matches the UML 2 metamodel. ACME studio does not provide any documentation or source code to adapt the existing program so extending it could prove to be very difficult. So even though ACME is a suitable tool it does not support two very important criteria: UML 2 metamodel and support (Documentation and community activity) to update the tool.

The Eclipse framework is an open source tool and thus has the advantage that all source code is openly available and can be openly distributed upon change. Eclipse MDT has a fair amount of documentation that covers creating UML models, creating profiles to extend the metamodel, basic OCL implementation and basic visual interface usage. The documentation is not perfect but it helps to understand the basics. Support is also available in the form of forums and this is by far the best way to learn about what Eclipse MDT is capable of and how to extend it in any way. This support is far better than support offered from ACME. The MDT package has a number of plug-ins and the relevant ones include – UML2, OCL and UML2Tools. The UML2 plug-in offers UML 2 metamodel support, the OCL plug-in offers OCL model checking support and the UML2Tools plug-in offers visualisation support. This clear separation of functionality has the advantage that it is clear what part of the package needs to be used for the relevant functionality of the project. The structure of the packages in ACME is not documented so this is another advantage of Eclipse MDT over ACME.

Another advantage of the Eclipse framework is the fact that it has the UML2 plug-in that offers the UML 2.1 metamodel. The fact that the UML2 plug-in is based on the UML 2.1 metamodel is a huge advantage because it means that any modellers using the Primus system will know exactly what the metamodel consists of because of the very detailed specification supplied by OMG [12].

Based on the results the decision was made to use Eclipse MDT.
3 The Eclipse Platform

The Eclipse platform was designed to offer developers a means to build an Integrated Development Environment [6][9]. Giving developers an opportunity to build their own IDE is a very different philosophy than other IDE providers that offer their solution as to what an IDE should be. This has led to an IDE that can work with a number of popular programming languages, offers various tools for debugging, service oriented architectures, modelling and more.

The fact that so many features are available to Eclipse is due to the central plug-in architecture of Eclipse. By itself Eclipse is just a framework for allowing plug-ins to be added to it. This basically consists of a runtime layer that is responsible for activation and discovery of plug-ins. The runtime layer is based on the Open Services Gateway initiative which is widely used in the hardware and software industry. This ensures compatibility with many hardware systems and runtime execution environments. The core version of Eclipse also comes with a small number of standard plug-ins that allow for Java development and plug-in development.

Figure 1 shows the way in which plug-ins extend Eclipse and that plug-ins can extend plug-ins. The plug-in architecture offers enormous benefit to developers because existing functionality developed by the (Developer) community can be reused. Figure 1 also shows JDT and PDE. JDT is crucial to Java development and PDE allows for development of new plug-ins.
3.1 Plug-in Development

All important information about a plug-in is kept in the manifestation and the plugin.xml file. The manifestation file contains properties such as version number, run-time class path and exported packages. The plugin.xml file contains information relevant to extensions and extension-points.

Extensions and extension-points are key to developing plug-ins and extending existing plug-ins. When a new plug-in is developed it is almost always going to use existing plug-ins. How the existing plug-ins can be extended is determined by the extension-points. The extension-points define what is required of an extension in order to be a valid extension. Information that is required can include window names, icon paths and classes that implement interfaces. The names and locations of the classes is obviously important because it tells Eclipse what code to execute when an event occurs that requires the new plug-in.

In theory a plug-in can be developed by only using text editors and updating the manifest and plugin.xml files that point to the code that requires execution. This would not be a wise thing to do but it shows that this is what Eclipse requires of a plug-in. Actual plug-in development in Eclipse is achieved using the Plug-in Development Environment which in turn creates and updates the necessary files.

3.2 Plug-in Types

There are three types of plug-in types in Eclipse: The standard plug-in, the fragment plug-in and the feature plug-in. The standard plug-in is a full-fledged plug-in with its own manifest and plugin.xml file. The fragment plug-in is part of an existing plug-in or target plug-in as it is called. It is essentially a merge between the existing plug-in and the new plug-in. It is normally used for localising a plug-in for a specific language, to provide platform specific functionality or to provide extra functionality without the need for a full-fledged plug-in. The feature plug-in is used to package plug-ins into a product so that it can be downloaded as a new program or product. For example, an embedded programming feature version could be used to group tools related to embedded programming.

3.3 Eclipse UML2 Plug-in

The Eclipse UML2 plug-in is a UML 2.1.x implementation (Specification is to be finalised soon) of the meta-model for the Eclipse framework [10]. With it developers can use the plug-in to develop modelling tools. The plug-in can be used with the knowledge that the meta-model used is the one specified by the UML 2.x specification. Constraints specified in the UML 2.x specification are also implemented in UML2. The constraints have been implemented using Java code and not using OCL as one might expect.

The UML2 plug-in also has a basic editor to define UML2 models. It shows a tree notation of any given model and allows for basic interaction using context-menus. The tree notation is not very suitable
for large scale development of models but it can be used to provide functionality in case it is not
offered by the UML2Tools plug-in (Which offers easier interaction with models).

Figure 2: UML2 tree view

Figure 2 shows how UML2 depicts a UML model. In this model various components are shown to be
owned by the main package and some of the components own ports and connectors.

3.4 Eclipse OCL Plug-in

The OCL plug-in provides integration of OCL with UML models from the UML2 plug-in. The
following functionality is provided by the OCL plug-in (As described by the plug-in description) [10]:

- Defines APIs for parsing and evaluating OCL constraints and queries on EMF models.
- Defines an Ecore implementation of the OCL abstract syntax model, including support for
  serialization of parsed OCL expressions.
- Provides a Visitor API for analyzing/transforming the AST model of OCL expressions.
- Provides an extensibility API for clients to customize the parsing and evaluation environments
  used by the parser.

The first item of functionality is of particular importance to the Primus project because it allows for
querying of UML models (UML models are an implementation of EMF models). The queries will be
used to detect and validate primitives in the UML model.

3.5 Eclipse UML2Tools Plug-in

The UML2Tools project allows for visualisation of the UML models in the form of UML diagrams.
The following diagrams are supported:

- Activity diagram
- Class diagram
- Component diagram
- Composite Structures diagram
Architectural Primitives Project          Edition 1.0

- Deployment diagram
- Profile diagram
- State machine diagram
- Use case diagram

Primus requires that models are visualised in a component diagram form. This is supported by UML2Tools. However, the component diagram is not yet complete and it is not clear when it will be finalised. Most of the problems regarding the component diagram are related to the connectors and specifically assembly connectors. This problem has been solved with a quick fix as part of this project. However, this is not a solution that is suitable for integration with the repository.

### 3.6 Primus Plug-in

The Primus plug-in is a fragment plug-in that will extend the component diagram plug-in of the UML2Tools plug-in (The UML2Tools plug-in actually consists of many sub-plug-ins). The Primus plug-in should allow modellers to model primitives in UML models and to validate them as described in the introduction paragraph. The list of primitives is not yet complete and primitives are likely to be added in the future. This means that the plug-in should allow for easy extension of more primitives.

The plug-in has been developed by having functionality separated so that it can be easily understood by other developers and so that primitives can be added by copying relevant classes and adapting them.

The remainder of the paragraph will describe how Primus works in terms of (Java) code structure.

#### 3.6.1 Events

All Primus actions start with an event that is triggered by an underlying plug-in. The underlying plug-in in this case is the UML2Tools plug-in because interaction with Primus occurs in the component diagram. To ‘make something happen’ it is necessary to catch the relevant event. Relevant in this case is a context menu within the component diagram. There are many different types of events in the Eclipse framework due to its plug-in structure and so each event has a code that is passed along to all listeners in the plug-ins. To add a listener class to the Eclipse framework the plug-in.xml file has to be updated to include the relevant classes. This can be done via a configuration page or by updating the xml file directly. The configuration page is a part of the PDE discussed earlier.
Figure 3: Plug-in Extensions

Figure 3 shows the extension menu of the Primus fragment.xml file. One of the extensions is ‘opened’ and shows the details of that extension. The extension is question is the contributionItemProvider. This extension allows the context menu to be appended and tells Eclipse which class should be invoked if the event is triggered. This is a good example of how Eclipse is extended by plug-ins and how plug-ins extend plug-ins.

3.6.2 Validation

If the user selects a validation menu then validation is carried out. The validation code is split into two segments/classes: The actual OCL code is defined and run in one class and the interpretation of the OCL results is carried out in another. This separates functionality and makes the code easier to understand and extend for future adaptations.

The OCL code can be found in the OCLCheckAll java class. To execute a validation the constructor should be called and this initialises the necessary OCL helper classes. Based on the choice of primitive it will validate the relevant primitive by retrieving the primitive specific OCL code and executing it.

Each type of primitive is different and this means that the return value of the location of a primitive is also different. This requires that an interpretation of the results is needed so that the user can be presented with a simple message describing the location of a primitive instead of a complicated list of components and packages (To fully understand the concept of return values of OCL code see Primitive paragraph). The interpretation is carried out in the RegisterPrimitive java class.

When a validation of primitives is carried out there are two queries that are executed. The first query is a strict validation of the relevant primitive and the second query is a less strict version that is needed to find ‘near’ primitives. Near primitives are primitives that have not validated properly and thus tell the modeller to look at that primitive and to fix whatever problem that primitive has. The results of the near validation are compared to with the results of the proper validation and if the ‘near’ validation has found something that the proper validation has not found then this information will be presented. This also occurs in the RegisterPrimitive java class.
3.6.3 Creating Primitives

In order for Primus to create a primitive it has to create a number of UML elements, apply stereotypes to some of them and to add them to the UML model. A considerable part of the project has consisted of defining exactly which elements will make up a primitive. With this information it is just a question of creating the necessary elements, applying the stereotypes and adding to the model. In the primitive paragraph a list of the exact elements and stereotypes that need to be applied is given.

Primus offers the ability of creating primitives automatically without adding the elements individually. This means that Primus adds the necessary elements as described in the primitive paragraph.

3.6.4 Eclipse Wizard

Adding primitives to a model requires that the user make a number of decisions and the easiest way to present the user with the relevant information is by using a wizard. Implementing wizards in Eclipse is a relatively easy task. It requires that a special wizard class is created and that pages are added with each page having selection options and input elements like buttons and text fields. Selections that are made should be saved in an object or file that stores the information so that when the wizard is complete the action can retrieve necessary information from the storage class.
4 Unified Modelling Language

Modelling is an essential part of developing software systems and its importance is growing. UML has become the de-facto modelling language and is therefore a good choice when modelling any kind of system and thus architectural primitives as well. The broad domain of UML has helped in its popularity because modellers in many different fields can use UML. Unfortunately, it also means that it lacks detail for any given domain as well. This problem has been solved by being able to extend UML with profiles. The extension process as well as the exact way in which UML was used to model primitives (General UML constructs and not specific per architectural primitive) will be described in this paragraph.

4.1 Four Layer Meta-Model

The modelling architecture of UML consists of four layers. Using layers to define models is a proven architecture for complex modelling languages [4] [5].

The top layer, known as the meta-metamodel or M3 level (MOF), is where the definition of UML elements begins. Elements in this level are instances of themselves. Its purpose is to define a language for the metamodel. This layer is the highest level of abstraction of the UML architecture.

The third layer, known as the metamodel or M2 level (MOF), is used to specify the language of the model and is an instance of the meta-metamodel. This is the layer in which primitives will be defined by using soft extensions.

The second layer, known as the model or M1 level (MOF), is used to specify the model and is an instance of the metamodel. This is the layer in which architects design an actual system and the layer in which primitives will be used.

The bottom layer, known as the runtime or M0 level, contains the objects, called user objects or domain objects that are instances of the model level.
Figure 4: Four level modelling architecture with bank account example [5]

Figure 4 shows how a layered architecture works. In this case ‘checking12345’ is an instance of checkingAccount.

4.2 Metamodel

The metamodel used by the UML2 plug-in is that of the UML 2.1 specification (The latest version of UML2 is based on the, yet to be finalised version of the, UML 2.2 specification). The metamodel is rather large and thus cannot be depicted in one diagram so the diagrams shown in this paragraph are the parts of the metamodel that are most relevant to this project.
Figure 5 shows that the component element is a specialisation of the classes element with the added ability to require and provide one or more interfaces.

![Diagram of Component Element with Added Ability](image)

Figure 6: The metaclasses that define the component wiring constructs [7]

![Diagram of Metaclasses](image)

Figure 7: Connector [7]

A component is designed to be an independent module within a system but it still has dependencies. These dependencies can be modelled by using interfaces or connectors. Figure 6 shows the metaclasses of the connector. Figure 7 shows the relationship between the connector metaclass and other relevant classes like ConnectorEnd.

### 4.3 UML Elements used by primitives

#### 4.3.1 Packages

A package is used to group elements together. The model always starts with a package so that other entities can be added. A package can also contain other packages [7][8].

![Diagram of Package Symbol](image)

Figure 8: Package symbol
4.3.2 Components

A Component is a modular unit that is replaceable within its own environment. They are designed to act as independently as possible so that other components can be added to the model or modelled to replace an existing component without complicated dependency issues. Components are not totally independent and have required and provided interfaces to model dependency between themselves. When a component is dependent on another component it is modelled using a ‘required interface dependency’ linked to a ‘provided interface dependency’. The required and provided connections are usually exposed using ports and this will always be the case in Primus.

![Figure 9: Component symbol](image)

4.3.3 Ports

A port is a property of a classifier, which is always a component in the case of Primus, and specifies the interaction between the internal parts of the classifier (Component) and its environment. Ports are owned by a classifier.

![Figure 10: Component with a port (Smaller Square)](image)

4.3.4 Classes

A class is used to specify a collection of objects with similar or the same structure and behaviour. Classes are the most widely used classifier and easily the most recognizable for programmers.

![Figure 11: Class symbol](image)

4.3.5 Interfaces

An interface acts as a contract between classifiers. It states what can be expected from classifiers and most importantly what should be provided and what should be required. An interface cannot be instantiated and must be implemented by a classifier.
4.3.6 Connectors

There are two types of connectors: delegation and assembly.

Delegation connector: A delegation connector connects the port of a component to the internal realization. Specifically it does not connect the port but rather uses the port to connect to the specific behaviour and expectancy stated by the contract (As defined by the required or provided interface).

Assembly connector: An assembly connector connects components together. This requires that the required and provided interfaces are defined (The ‘contract’ between the two components must be defined).

The Primus project will only use assembly connectors.

4.4 Extending UML

A stereotype defines how existing UML elements can be extended. By using them it is possible to add information or features but not to hide information or disable features of existing elements. Stereotypes are used to extend UML in order to define primitives. Stereotypes have to be defined in the M2 level before they can be applied to elements in the M1 level.

The following steps describe the process of defining stereotypes:

1) Import the base element that needs to be extended
2) Create a stereotype
3) Create tag definitions within the stereotype if needed
4) Create an extension between the base element and the stereotype

Extending UML also consists of adding constraints. This will be achieved using OCL and this process will be described in the OCL paragraph.

4.5 **UML Modelling**

This sub-paragraph will describe how general UML modelling concepts are applied within Eclipse UML2 in order to create primitives. It will not describe how each primitive type is modelled. This is needed because UML implementations (Those from frameworks) can differ. For example, the way that connectors are implemented can differ (There are many ways of defining a connector between two components).

4.5.1 **Required and Provided Interface**

Required and provided interface dependencies are needed to describe the dependencies between components. They are also needed if components are to be connected using connectors. In the Primus project all required and provided interface dependencies will be modelled in the following way:

Two components each have an owned port and have a connector between them. This requires the components to have a required/provided dependency between them and in particular between the two ports. Two ‘helper’ classes are created to refer to the actual dependencies – one will have an interface realization and one will have a usage referred to it. The interface realization refers to the provided dependency and the usage refers to the required dependency. The component that needs the ‘requires dependency’ (The component that initiates the connection and owns/will own the connector) will have its port typed to the class that has a usage and the component that needs the ‘provided dependency’ will have its port typed to the class that has an interface realization.

An exact step-by-step explanation:

1) Create two components
2) Create an interface
3) Create an owned port for each component
4) Create two classes
5) Type each port to a class
6) Add an Interface Realization to one of the classes and set the supplier property to the interface
7) Add a usage to the model and set the client to the other class and set the supplier property to the interface
4.5.2 Connector

There is no explanation or ‘how to’ in the UML superstructure to create a connector between two components (More specifically – between the ports of components). This means that modellers can have different ways of creating connectors between components.

In Eclipse UML2 a connector must be defined in the following way:

Two components need to be connected with a connector. A connector has to be owned by a component and thus the connecting component should own the connector. The connecting component is the component that is on the required side of the connection. Two properties are needed to define the ends of the connection. This is indicated by typing the properties with the components that need to be connected. This construct is required because the connector ends must be owned by the owner of the connector. Thus, it is not possible to simply choose the ports or components as the roles of the connector ends.

An exact step-by-step explanation:

1) Create two components
2) Create an owned port for each component
3) Create two properties on the component that will own the connector. This should be the component that will make the connection and should thus be the required side.
4) Create a connector on the connecting component
5) Set the type of one of the properties to the connecting component
6) Set the type of the other property to the other component
7) Create two connector ends for the connector from step 4)
8) Set the role of one connector end to the property from step 5)
9) Set the role of the other connector end to the property from step 6)
10) Set the partWithPort property from connector ends to the relevant ports

4.5.3 Association

An association is an important concept in UML and it is also used by some primitives. Associations are also modelled in various ways just as connectors are. In the Primus project all associations will be modelled in the following way:

Two components need an association between them. Each component has a property added to them with the aggregation property set to either none, shared or composite based on the type of association. An association should be defined in the model and the properties defined within the components should have the association property set to the association created.

An exact step-by-step explanation:
1) Create two components
2) Create a property for each component
3) Create an association
4) Set the association property of the first property to the association
5) Set the association property of the second property to the association
5 **Object Constraint Language**

OCL is a notational language for analysis and design of software systems. It is a part of the UML standard and thus usually used with UML models. With OCL it is possible to define exact constraints and queries without the ambiguities of natural language and without the complexities of mathematical definitions.

OCL is commonly used to define and enforce constraints or to perform expressions on models. Applying constraints on elements within a model consists of defining a constraint that a certain kind of element must adhere to. OCL is then able to check whether such a constraint is met or violated. Performing expressions consists of changing the state of a model by performing a query that specifies a pre and post state. This means that if a pre state is met then the post state will occur after the query has been executed.

OCL queries will be used in the Primus plug-in to validate the existence of primitives and near primitives. A near Primitive is a Primitive that has not met all of the constraints associated with a Primitive yet does have sufficient modelling structure to justify it being a close match.

The queries differ from normal constraint enforcement and executing expressions because interest lies in the location of the primitives and not solely checking for constraints or changing the model (expressions). Another reason for not using constraints directly is that normal UML modelling must also be possible alongside primitive modelling and thus the constraint cannot form a barrier by not allowing certain UML modelling constructs.

5.1 **General OCL Commands Constructs**

This paragraph will be used to describe general OCL commands and constructs used to find primitives.

5.1.1 **All Instances**

The allInstances() command returns all instances of the defining parameter.

Example: Component.allInstances()

The example returns all components in the model.

5.1.2 **Iterate**

The iterate command iterates through a set, collection or bag without returning a value. This is useful as compared to select or any because it is possible to define a return value that meets a specific demand. This is certainly the case with primitives because the return value of the primitives is unique because the structure of a primitive is unique.

Example: Component.allInstances().iterate(i | i.code)
The example iterates through the list of components with the ‘i’ variable holding the current component.

### 5.1.3 List Types

The collection OCL type is the abstract super type of all OCL predefined collection types:

- The set type is a non-ordered collection of objects that do not have any duplicates in it
- The bag type is a non-ordered collection of objects that allows for duplicates
- The sequence type is an ordered collection of objects that allows for duplicates

Example: Set(Component)
The example shows how a set can be defined as a return type.

### 5.1.4 Tuple

By using tuple it is possible to return tuples of related elements. This is exactly what is needed in the case of primitives because a primitive consists of multiple related elements. For example, a Callback consists of two components, by using a tuple with two components it is possible to determine which component is the caller and which component is the callback. The amount of related elements in a tuple will vary based on the primitive.

Example: Tuple(c1:Component, s:Set(Component))
The example defines a tuple with a component and a set of components per tuple.

### 5.1.5 Types

In OCL it is possible to assign the return value with a type. For example, if the return value is a component then it is possible to tell OCL that the object returned is a component. This is achieved using the oclAsType command.

It is also possible to test if a return value is of a particular type. For example, if a return value could be empty or a component and this could have further implications then it can be checked and dealt with in an if statement using the oclIsKindOf command.

Example: i.oclIsKindOf(Package)
The example returns true if ‘i’ is a package.

### 5.1.6 Select

The select command selects objects from a list based on whether the objects in the list pass the criteria specified. This obviously requires a list (collection, bag or set) to be present before the select command.

Example: Component.allInstances().select(i | i.name='something')
The example returns components with the name ‘something’.

5.1.7 Any

The any command is the same a select except, that it does not return a list of objects that pass the criteria; rather it returns one object from the selection of objects that pass the criteria.

Example: Component.allInstances().select(i | i.name='something')

The example returns a maximum of one component even if multiple components have the name (Although names should be unique in a UML model).
6 Primitives

This paragraph will describe the primitives that have been defined in [1]. Each primitive will have a definition of the profile needed for that primitive and the modelling elements and stereotypes needed to apply a primitive to a model. The validation of each primitive will also be described by presenting the validation code and an explanation of how the code works.

6.1 Callback

6.1.1 General Description

A callback consists of a component calling another component. The latter component stores the reference of the former component so that it can be called back when necessary. The connector should be stereotyped by Callback and is also the minimum requirement for finding the primitive using the ‘near’ query (The near query finds primitives but does not validate them). The ports of the two components should be stereotyped: The caller port should be stereotyped by EventPort and the target port should be stereotyped by CallbackPort. Callback requires two interfaces each with a provided/required dependency from the two components because of the two way communication between the components. The interfaces are stereotyped by either IEvent or ICallback (See Figure 11 to see which is stereotyped by which).

![Figure 15: Callback](image)

6.1.2 Eclipse UML2 Modelling M2 Level

Callback requires that interface, port and connector be extended. A list of stereotypes and extensions as needed in Eclipse UML2:

1. Element Import Interface
2. Element Import Port
3. Element Import Connector
4. Stereotype IEvent
5. Extension Interface_IEvent
6. Stereotype ICallback
7. Extension Interface_ICallback
8. Stereotype EventPort
9. Extension Port_EventPort
10. Stereotype CallbackPort
11. Extension Port_CallbackPort
12. Stereotype Callback
13. Extension Connector_Callback

6.1.3 Eclipse UML2 Modelling M1 Level
The Callback primitive is modelled as follows in Eclipse UML2 (An indentation means that the entity is owned by the last decreased indented entity):

1. Caller component
   1.1. Port stereotyped by EventPort and type set to Class from 3.
   1.2. Property typed by Caller component
   1.3. Property typed by Callback component
   1.4. Connector stereotyped by Callback
      1.4.1. Connector end with role set to property from 1.2. and partWithPort set to Port from the Caller component
      1.4.2. Connector end with role set to property from 1.3. and partWithPort set to Port from the Callback component
2. Callback component
   2.1. Port stereotyped by CallbackPort and type set to Class from 4.
3. CallerPort Class
   3.1. Interface Realization with supplier set to ObserveEvent interface
4. CallbackPort Class
   4.1. Interface Realization with supplier set to Update interface
5. Interface ObserveEvent stereotyped by IEvent
6. Interface Update stereotyped by ICallback
7. Usage with client set to CallerPort Class from 3. and supplier set to interface Update
8. Usage with client set to CallbackPort Class from 4. and supplier set to interface ObserveEvent

6.1.4 Callback Validation
The algorithm must search all components for the special relationship that exists between two components that constitute a Callback. The main component per loop will be assumed to be the caller component and the search for applied stereotypes and constraints will begin from there. The ‘search parameters’ in this case includes the proper application of the ports, connector and interfaces with the relevant stereotype (As depicted in Figure 1). If such a relationship is found between two components then they are added to the return results in a tuple. The return result consists of a set of tuples. Each
tuple consists of a component (The Caller component) and a set of components (The callback components).

Figure 16 shows three possibilities of a Callback in a model. The Callback primitive can consist of a single caller and a single Callback, a single caller and multiple Callbacks or multiple callers and a single Callback. The white components are the components of interest and the black component are other random components in the model. The algorithm should work for all situations.

It is helpful to start with pseudo-code due to the complexity of the search algorithm:

1. (Line 1) Iterate through all components
2. (Line 1) Set return parameters to a component and a set of components within a tuple
3. (Line 3) The following tests will be performed on all connectors from current component.
4. (Line 4) Define the location of the caller port of the current connector
5. (Line 5) Define the location of the callback port of the current connector
6. (Line 7) Test whether Callback stereotype is applied to connector (Within current component)
7. (Line 8) Test whether connector has two ends
8. (Line 9) Test whether port of caller side of connector has Eventport stereotype
9. (Line 10) Test whether port of caller side has provided interface with stereotype IEvent
10. (Line 11) Test whether port of caller side has required interface with stereotype ICallback
11. (Line 12) Test whether port of Callback side has CallbackPort stereotype
12. (Line 13) Test whether port of Callback side has provided interface with applied stereotype ICallback
13. (Line 14) Test whether port of Callback side has required interface with applied stereotype IEvent
14. (Line 15) Test whether provided interface on caller side points to same interface as required interface on Callback side (While making sure that the ports caller and Callback side are used)
15. (Line 16) Test whether required interface on caller side points to same interface as provided interface on Callback side (While making sure that the ports from caller and Callback side are used)

16. (Line 23-29) If the results of the tests listed are true then it is true that the current component is the caller side of a Callback and that there is at least one Callback component. We can thus add the current component and the Callbacks (There may be multiple Callback components with one caller) to the Callback set defined in 2

OCL code:

```
Component.allInstances().iterate(pairs : Set(Tuple(c1 : Component, s : Bag(Component))) = Set() |
    let comp : Component = c1.oclAsType(Component),
    stemp : Bag(Component) = comp.ownedConnector->select(i |
        let callerPort : Port = i.oclAsType(Connector).end.partWithPort->any(owner=i).oclAsType(Port),
        callbackPort : Port = j.oclAsType(Connector).end.partWithPort->any(owner<>i).oclAsType(Port) in
            if
                j.oclAsType(Connector).getAppliedStereotypes()->any(name='Callback')->notEmpty() and
                j.oclAsType(Connector).end->size()==2 and
                callerPort.getAppliedStereotypes()->any(name='EventPort')->notEmpty() and
                callerPort.getProvideds()->getAppliedStereotypes()->any(name='IEvent')->notEmpty() and
                callerPort.getRequireds()->getAppliedStereotypes()->any(name='ICallback')->notEmpty() and
                callbackPort.getAppliedStereotypes()->any(name='ICallback')->notEmpty() and
                callbackPort.getProvideds()->getAppliedStereotypes()->any(name='IEvent')->notEmpty() and
                callbackPort.getRequireds() = callbackPort.getProvideds() and
                callerPort.getRequireds() = callbackPort.getProvideds()
            then
                true
            else
                false
            endif
    )
)
```

<table>
<thead>
<tr>
<th>#</th>
<th>Status</th>
<th>Description</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Passed</td>
<td>An event port is typed by <code>IEvent</code> as a required interface</td>
<td>Constraint check in line 14</td>
</tr>
<tr>
<td>2</td>
<td>Passed</td>
<td>An event port is typed by <code>ICallback</code> as a required interface</td>
<td>Constraint check in line 11</td>
</tr>
<tr>
<td>3</td>
<td>Passed</td>
<td>A callback port is typed by <code>ICallback</code> as a provided interface</td>
<td>Constraint check in line 13</td>
</tr>
<tr>
<td>4</td>
<td>Passed</td>
<td>An event port is typed by <code>IEvent</code> as a provided interface</td>
<td>Constraint check in line 10</td>
</tr>
</tbody>
</table>
A Callback connector has only two ends. Constraint check in line 8

A Callback connector connects an EventPort of a component to a matching CallbackPort of another component. An EventPort matches a CallbackPort if the provided IEvent interface of the former matches the required IEvent interface of the latter, and the required ICallback interface of the former matches the provided ICallback interface of the latter:

Constraint check in line 15 and 16

<table>
<thead>
<tr>
<th>5</th>
<th>Passed</th>
<th></th>
<th>6</th>
<th>Passed</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Callback connector has only two ends.</td>
<td>A Callback connector connects an EventPort of a component to a matching CallbackPort of another component. An EventPort matches a CallbackPort if the provided IEvent interface of the former matches the required IEvent interface of the latter, and the required ICallback interface of the former matches the provided ICallback interface of the latter:</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Callback constraints [1]

6.1.5 Near Callback

The algorithm can search in the same way as the validation algorithm but it should not enforce every constraint check. Here we only check for the callback stereotype as this is the essence of the callback primitive. The code is very similar to the validation algorithm so an explanation will not be given.

```
Component.allInstances()->iterate((pairs : Set(Tuple(c1 : Component, s : Bag(Component))) = Set{} |
let comp : Component = i.oclAsType(Component),
stem : Bag(Component) = comp.ownedConnector->select(j | 
if j.oclAsType(Connector).getAppliedStereotypes()->any(name='Callback')->notEmpty() and 
then true else false endif 
stem->isEmpty() then 
pairs 
else 
pairs->including(Tuple(c1=comp, s = stem )) 
endif
) )
```
6.2 Indirection

6.2.1 General Description

Indirection is a method of redirecting a message to a target component. Indirection consists of three component types – client, indirector/proxy and target components. The client calls the indirector/proxy component. The client can consist of multiple components. The indirector/proxy component redirects the message. The target is the component that the client actually wanted to message/communicate with. The target can consist of multiple components and therefore the indirector/proxy component would require some internal logic to reroute the message to the proper target. In [1] Avgeriou, Zdun suggest that the indirector/proxy can also consist of multiple components but this is not the case in this implementation and multiple indirector/proxy components imply that there are multiple indirection primitives present.

The client and indirector/proxy can connect using a standard connector with an interface that is stereotyped by IIndirector. The connection between the indirector/proxy component and the target component requires that the connector be stereotyped by Indirection. The port of the indirector/proxy should be stereotyped by IndirectionPort and the port of the target should be stereotyped by IndirectionTargetPort.

![Figure 17: Indirection](image)

6.2.2 Eclipse UML2 Modelling M2 Level

Indirection requires that interface, port and connector be extended. A list of stereotypes and extensions as needed in Eclipse UML2:

1. Element Import Interface
2. Element Import Port
3. Element Import Connector
4. Stereotype Indirector
5. Extension Interface_Indirector
6. Stereotype ITarget
7. Extension Interface_ITarget
8. Stereotype IndirectionPort
9. Extension Port_IndirectionPort
10. Stereotype IndirectionTargetPort
11. Extension Port_IndirectionTargetPort
12. Stereotype Indirection
13. Extension Connector_Indirection

6.2.3 **Eclipse UML2 Modelling M1 Level**

The indirection primitive is modelled as follows in Eclipse UML2:

1. Component Indirector
   1.1. Port stereotyped by IndirectionPort and type set to Class from
   1.2. Property typed by Indirector
   1.3. Property typed by IndirectionTarget
   1.4. Connector stereotyped by Indirection
       1.4.1. Connector end with role set to property from 1.2. and partWithPort set to Port from
             the Indirector component
       1.4.2. Connector end with role set to property from 1.3. and partWithPort set to Port from
             IndirectionTarget component
2. Component IndirectionTarget
   2.1. Port stereotyped by IndirectionTargetPort and type set to Class from
3. Component IndirectionClient
   3.1. Port typed by Class from
4. Indirector Class
   4.1. Interface Realization with supplier set to IndirectedOp
5. IndirectionTarget Class
   5.1. Interface Realization with supplier set to TargetOp
6. IndirectionClient Class
7. Interface IndirectedOp stereotyped by Indirector
8. Interface TargetOp stereotyped by ITarget
9. Usage with client set to Indirector Class from 4. and supplier set to interface TargetOp
10. Usage with client set to IndirectionClient Class from 6. and supplier set to interface IndirectedOp
6.2.4 Indirection Validation

The algorithm must search all components and check to see if indirection exists. The starting point of indirection is the indirector/proxy type component. So when the algorithm iterates through the components it will assume that the current component is an indirector/proxy. From there it will look for clients and targets. If at least one client and target is found then the indirection primitive is added to the return result. The return result consists of a tuple consisting of a component, a set of client components and a bag of target components.

Figure 18 shows possibilities of an indirection primitive. The variation points consist of a single or multiple clients and a single or multiple targets. The black components show that the algorithm must be able to find the primitive amongst non-primitive components.

Pseudo code:

1. (Line 1) Iterate through all components
2. (Line 1) Set return parameters to a component, a set and a bag within a tuple
3. (Line 7) Test whether the required interface of client (relevant port) is stereotyped by IIndirector
4. (Line 8) Test whether the provided interface of indirector (relevant port) is stereotyped by IIndirector
5. (Line 9) Test whether required interface of client and provided interface of indirector are matched
6. (Line 15) Test whether connector(s) between indirector and target(s) is stereotyped by Indirection
7. (Line 16) Test whether connector(s) between indirector and target(s) has two ends
8. (Line 17) Test whether the relevant port of the target is stereotyped by IndirectionTargetPort
9. (Line 18) Test whether the provided interface of the target is stereotyped by ITarget
10. (Line 19) Test whether the indirector port is stereotyped by IndirectionPort
11. (Line 20) Test whether the required interface of the indirector is stereotyped by ITarget
12. (Line 21) Test whether required interface of indirector and provided interface of target are matched
13. (Lines 23-29) If at least one client and target are found then the indirection primitive found is added to the return result.

**OCL code:**

```ocl
cOMPONENT.allInstances() ->iterate({pairs : Set(Tuple(c1 : COMPONENT, client : Set(COMPONENT), target : BAG(COMPONENT))) = Set() | 
  let comp : COMPONENT = ioclAsType(COMPONENT), 
  clienttemp : Set(COMPONENT) = COMPONENT.allInstances() ->select(a | 
    a.ooclAsType(COMPONENT).ownedConnector ->any(b | 
      let clientPort : PORT = b.oclAsType(PORT).end.partWithPort ->any(owner=a).oclAsType(PORT), 
      otherPort : PORT = b.oclAsType(PORT).end.partWithPort ->any(owner<>a).oclAsType(PORT) in 
    clientPort.getRequireds() ->any(name='Indirector').getAppliedStereotypes() ->any(notEmpty()) and 
    otherPort.getProvideds() ->any(name='Indirector').getAppliedStereotypes() ->any(notEmpty()) and 
    clientPort.getRequireds() = otherPort.getProvideds() and 
    otherPort.owner.oclAsType(COMPONENT) = comp 
  ) ->notEmpty() and 
  a.ooclAsType(COMPONENT) <> i), 
  stemp : BAG(COMPONENT) = comp.ownedConnector ->select(c | 
    let indirectorPort : PORT = c.oclAsType(CONNECTOR).end.partWithPort ->any(owner=i).oclAsType(PORT), 
    targetPort : PORT = c.oclAsType(CONNECTOR).end.partWithPort ->any(owner<>i).oclAsType(PORT) in 
    c.oclAsType(CONNECTOR).getAppliedStereotypes() ->any(name='Indirection').getAppliedStereotypes() ->any(notEmpty()) and 
    targetPort.getProvideds() ->any(name='ITarget').getProvideds() ->any(name='ITarget').getProvideds() = targetPort.provided 
  ) ->notEmpty() and 
  indirectorPort.getProvideds() ->any(name='ITargetPort').getProvideds() ->any(name='ITargetPort').getProvideds() = targetPort.provided 
  indirectorPort.required = targetPort.provided 
  ) ->notEmpty() and 
  indirectorPort.owner.oclAsType(COMPONENT) = comp 
  ) ->notEmpty() and 
  pairs ->including(Tuple(c1=comp, client = clienttemp, target = stemp )) 
  endif 
})
```

<table>
<thead>
<tr>
<th>#</th>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Passed</td>
<td>An IndirectionTargetPort provides an ITarget interface</td>
</tr>
<tr>
<td>2</td>
<td>Passed</td>
<td>The IndirectionPort requires an ITarget interface</td>
</tr>
<tr>
<td>3</td>
<td>Passed</td>
<td>The IndirectionPort provides an Indirector interface</td>
</tr>
<tr>
<td>4</td>
<td>Passed</td>
<td>An Indirection connector has only two ends</td>
</tr>
<tr>
<td>5</td>
<td>Passed</td>
<td>An Indirection connector connects an IndirectionPort of a proxy component to</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a matching IndirectionTargetPort of the target component. An Indirection</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Port matches an IndirectionTargetPort if the provided ITarget interface of</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the latter matches the required ITarget interface of the former.</td>
</tr>
</tbody>
</table>

**Table 4: Indirection constraints [1]**
6.2.5 Near Indirection

In the following algorithm the constraints are not checked. The purpose is to check for near indirection primitives. The algorithm is similar to the validation algorithm. The difference is that the constraints are not checked.

1. Component.allInstances() -> iterate((pairs : Set(Tuple(c1 : Component, client : Set(Component), target : Bag(Component)))), target : Bag(Component))) = Set()
2. let comp : Component = ioclAsType(Component),
3. clienttemp : Set(Component) = Component.allInstances() -> select(a |
4. a.oclAsType(Component).ownedConnector -> any(b |
5. let clientPort : Port = b.ownedConnector.end.partWithPort -> any(owner=a).oclAsType(Port),
6. otherPort : Port = b.ownedConnector.end.partWithPort -> any(owner<>a).oclAsType(Port) in
7. ) -> notEmpty() and a.oclAsType(Component) <> i),
8. stemp : Bag(Component) = comp.ownedConnector -> select(c |
9. c.oclAsType(Connector).getAppliedStereotypes() -> any(name='Indirection') -> notEmpty() and
10. ).oclAsType(Connector).end.role.type -> select(c | c <> comp).oclAsType(Component) in
11. if
12. stemp -> isEmpty() or clienttemp -> isEmpty()
13. then
14. pairs
15. else
16. pairs -> including(Tuple(c1=comp, client = clienttemp, target = stemp ))
17. endif
18.

6.3 Grouping

6.3.1 General Description

The grouping primitive is used to group components together logically. Usually this would mean that the grouped components all have a common task or, in the case of the shield primitive (see shield sub-paragraph), they all have the same common attribute – all protected by shield. A group is defined by applying the group stereotype to a package.

![Figure 19: Grouping](image-url)
6.3.2 Eclipse UML2 Modelling M2 Level

Grouping requires that package be extended. A list of stereotypes and extensions as needed in Eclipse UML2:

1. Element Import Package
2. Stereotype Group
3. Extension Package_Group

6.3.3 Eclipse UML2 Modelling M1 Level

The grouping primitive is modelled as follows in Eclipse UML2:

1. Package stereotyped by Group
   1.1. Element Import set to Component from 2.
   1.2. Element Import set to Component from 3.
   1.3. Element Import set to Component from 4.
2. Component
3. Component
4. Component
5. Package stereotyped by Group
   5.1. Element Import set to Component from 3.
   5.2. Element Import set to Component from 4.

6.3.4 Grouping Validation

The algorithm should return all packages with a group stereotype applied. The package may not own any members and the imported members should all be components to adhere to the constraints.

Figure 20: Grouping scenarios

Figure 20 shows that many packages can exist within a model. The algorithm must find the packages that are groups and the groups must adhere to the group constraints. It also shows that components are
linked to the groups. These are the group members. A component can be a member of multiple components.

Pseudo code:

1. (Line 1) Start with all packages and select the ones that pass the following tests
2. (Line 3) Test whether package owns any packages
3. (Line 4) Test whether all imported members are components
4. (Line 5) Test whether package has group stereotyped applied

OCL code:

```
1   Package.allInstances().select(i | 
2      ioclIsKindOf(Package) and 
3      ioclAsType(Package).ownedMember.size()=0 and 
4      ioclAsType(Package).importedMember.forAll(c | c.oclIsTypeOf(Component)) and 
5      ioclAsType(Package).getAppliedStereotypes().any(name='Group').notEmpty() 
6   )
```

<table>
<thead>
<tr>
<th>#</th>
<th>Status</th>
<th>Description</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Passed</td>
<td>A Group does not own any members</td>
<td>Constraint check in line 3</td>
</tr>
<tr>
<td>2</td>
<td>Passed</td>
<td>All the imported members of a group are Components</td>
<td>Constraint check in line 4</td>
</tr>
</tbody>
</table>

Table 5: Grouping constraints [1]

6.3.5 Near Grouping

In the following algorithm the constraints are not checked. The purpose is to check for near grouping primitives.

```
1   Package.allInstances().select(i | 
2      ioclIsKindOf(Package) and 
3      ioclAsType(Package).getAppliedStereotypes().any(name='Group').notEmpty() 
4   )
```

6.4 Layers

6.4.1 General Description

Layers builds on Grouping to add logical groups of components together with added constraints. The constraints are related to rules about which layers can be called from which layer. This usually means that layers must call a layer directly above or below the current layer (To be more precise: which components in a certain layer can call which components in another layer).
6.4.2 Eclipse UML2 Modelling M2 Level

Layers require that package be extended. A list of stereotypes and extensions as needed in Eclipse UML2:

1. Stereotype Layer
   1.1. Generalization set to Group from Grouping Primitive
   1.2. Property ‘LayerNumber’ of type Primitive Type – Integer
2. Import of Grouping

6.4.3 Eclipse UML2 Modelling M1 Level

The layers primitive is modelled as follows in Eclipse UML2:

1. Package stereotyped by Layer and Layer Number (Defined in M2 level) set to 1
   1.1. Element Import set to Component from 2.
   1.2. Element Import set to Component from 3.
2. Component
3. Component
4. Package stereotyped by Layer and Layer Number (Defined in M2 level) set to 2
   4.1. Element Import set to Component from 5.
   4.2. Element Import set to Component from 6.
5. Component
6. Component

6.4.4 Layers Validation

The algorithm should return packages with the layers stereotype applied. The algorithm should also check that a member is not part of multiple layers and that components can only connect to components that are in a layer directly below, within the current layer or layers above the current layer.
Figure 22 shows that many packages can exist within a model. The algorithm must find the packages that are layers and adhere to the layers constraints. It also shows that components are linked to the layers. These are the layer members. A component can only be a member of one layer and not multiple layers (As opposed to groups). Another consideration for the algorithm is that a component within a layer should also be able to call components that are not associated with a layer.

**Pseudo code:**

1. (Line 1) Select all packages. The remaining code will check for constraints within the selected packages and assess whether they are layers.
2. (Line 3) Check whether package has the Layer stereotype applied.
3. (Line 4) Check whether package has any owned members. This should not be the case
4. (Line 5,6) Check whether all imported members are components
5. (Line 7) Check whether components that are part of a layer are in fact only part of one layer and not multiple layers
6. (Lines 8-10) Loop through all connectors within the current component. Retrieve the connecting layer (Layer of current component) and the target layer (Layer that connecting component resides in)
7. (Line 12) If the target component has a layer then perform the test on the next two lines
8. (Line 14,15) Get the value of the two layers and subtract the target layer value from the connecting layer value. If that value is less or equal to 1 then true
9. (Lines 16-21) Return true or false values based on results of tests

**Actual code:**

```java
1 Package.allInstances() -> select(z |
2   if
3   zoclAsType(Package).getAppliedStereotypes() -> any(name='Layer') ->notEmpty() and
4   zoclAsType(Package).ownedMember ->size() = 0 and
```
5 \[ \text{zoclAsType(Package).importedMember}\rightarrow\text{forAll}\ i | \]
6 \[ \text{i.oclIsKindOf(Component) and} \]
7 \[ \text{Package.allInstances() \rightarrow select(q | q.getAppliedStereotypes() \rightarrow any(name='Layer') \rightarrow \text{notEmpty()} \rightarrow select(a |} \]
8 \[ \text{aoclAsType(Package).importedMember\rightarrow any(b | b\\rightarrow\text{notEmpty()} \rightarrow \text{size}()=1 \text{ and}} \]
9 \[ \text{ioclAsType(Component).ownedConnector}\rightarrow\text{forAll}(d |} \]
10 \[ \text{let connectingLayer : Package = Package.allInstances() \rightarrow select(q | q.getAppliedStereotypes()\rightarrow any(name='Layer') \rightarrow \text{notEmpty()}) \rightarrow select(l | l.importedMember\rightarrow includes(doclAsType(Connector).end.role.type\rightarrow any(c | c <> i))) in} \]
11 \[ \text{if} \]
12 \[ \text{connectedLayer\rightarrow\text{notEmpty() then} \]
13 \[ \text{if} \]
14 \[ \text{(connectingLayer.getValue(connectingLayer.getAppliedStereotypes()\rightarrow any(name='Layer')\rightarrow LayerNumber').oclAsType(Integer)) \rightarrow any(c | c <> i)) in} \]
15 \[ \text{if} \]
16 \[ \text{true else false endif} \]
17 \[ \text{else true endif} \]
18 \[ \text{) \]
19 \[ \text{) \]
20 \[ \text{true else false endif} \]
21 \[ \)

<table>
<thead>
<tr>
<th>#</th>
<th>Status</th>
<th>Description</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Passed</td>
<td>A Layer member can only be part of one layer and not multiple layers</td>
<td>Constraint check in line 4</td>
</tr>
<tr>
<td>2</td>
<td>Passed</td>
<td>Components in Layer X may only call components in the same Layer and Layer X-1 but not other Layers</td>
<td>Constraint checked in lines 14-15</td>
</tr>
</tbody>
</table>

Table 6: Layers constraints [1]

Grouping constraints also apply to the Layers primitive.

6.4.5 Near Layers

In the following algorithm the constraints are not checked. The purpose is to check for near layers primitives.

1 \[ \text{Package.allInstances()}\rightarrow\text{select(i |} \]
2 \[ \text{i.oclIsKindOf(Package) and} \]
3 \[ \text{ioclAsType(Package).getAppliedStereotypes()\rightarrow any(name='Layer')\rightarrow\text{notEmpty()}} \]
4 \[ ) \]

6.5 Aggregation Cascade

6.5.1 General Description

An Aggregation Cascade models a whole-part relationship whereby constraints are enforced constraining which components can be extended. An aggregation cascade works recursively with an aggregate component calling the next aggregate component and so on. The messaging is thus passed along using a proxy system and this is modelled using the Indirection primitive. The main modelling
aspects of Aggregation Cascade are the stereotyping of the connector between aggregates and the parent being shared by all aggregate components (See Figure 19).

Figure 23: Aggregation Cascade

6.5.2 Eclipse UML2 Modelling M2 Level

Aggregation Cascade requires that Indirection be extended. A list of stereotypes and extensions as needed in Eclipse UML2:

1. Stereotype AggregationCascade
   1.1. Generalization set to Indirection
2. Import of Indirection

6.5.3 Eclipse UML2 Modelling M1 Level

The Aggregation Cascade primitive is modelled as follows in Eclipse UML2:

1. Component MainAgg
2. Component Agg1
   2.1. Generalization set to MainAgg
   2.2. Port stereotyped by IndirectionPort and typed by Class from 4.
   2.3. Property typed by Agg1
   2.4. Property typed by Agg2
   2.5. Property with aggregation set to shared, association set to Association from 4 and typed by Agg2
   2.6. Connector stereotyped by AggregationCascade and Indirection and typed by the association from 6)
      2.6.1. Connector end with role set to property from 2.3. and partWithPort set to Port from the Agg1 component
2.6.2. Connector end with role set to property from 2.4. and partWithPort set to Port from the Agg2 component

3. Component Agg2
   3.1. Generalization set to MainAgg
   3.2. Port stereotyped by IndirectionTargetPort and typed by Class from 5.
   3.3. Property with association set to Association from 5

4. Class Agg1
   4.1. Interface Realization set to Interface IndirectedOp from 9

5. Class Agg2
   5.1. Interface Realization set to Interface TargetOp from 8

6. Association

7. Usage with client set to Class from 4. and supplier set to Interface AggInt from 8

8. Interface TargetOp stereotyped by ITarget

9. Interface IndirectedOp stereotyped by Indirector

6.5.4 Aggregation Cascade Validation

The algorithm must return all aggregation cascades in the model. This can be achieved by looking for the top component of the cascade. The algorithm can then recursively work its way down, checking that each connector is stereotyped by AggregationCascade and that the aggregation is set properly, until the cascade has ended. It must also check that each component generalizes the same component.

Figure 24: Aggregation Cascade scenario

Figure 24 shows how aggregation cascade can reside in a model. The length of the cascade can differ and thus requires a recursive approach to go the length of the cascade.

The Indirection aspects of the validation can be checked using the Indirection algorithm.
Pseudo code:

1. (Line 1) Iterate through all components
2. (Line 1) Set return parameters to two components and a set of components within a tuple
3. (Line 2) Use the closure command to recursively move along the cascade
4. (Line 3) Start a recursive loop using the closure command
5. (Line 4) Check whether the next level of cascade components are all of the same type
6. (Line 6,7,8,9,10) Check whether the current component has a connector with an AggregationCascade stereotype applied. This line of code also checks whether the connector is typed by an association with a shared aggregation and whether the ends of the association are owned by the components.
7. (Line 13) If the checks failed then do nothing
8. (Line 12) Constrain the set of aggregation components further to check that they all have a generalization set to the same component as the top component from the aggregation
9. (Line 15,16,17,18) The algorithm is currently in the iterate loop. The iterate loop has a current component and assumes that the component is the ‘top’ of an aggregation. Therefore it must not have a connector with an AggregationCascade stereotype connected to the current component. This is checked within this line of code.
10. Line (15, 19, 20) Same concept applies here as was the case in the previous line except that the aggregation applies instead of the connector. So the current component cannot have an aggregation applied to it from another component
11. (Line 22-29) If the current component has a shared aggregation and the recursive part of the algorithm is not empty then add the current component as the top of an aggregation

OCL code:

```ocl
1 Component.allInstances()->iterate(i;sel : Set(Tuple(c1 : Component,c2 : Component,s : Set(Component))) = Set{} |
2    let stemp:Set(Component) = i.oclAsType(Component)->closure[j |
3        if
4            j.oclAsType(Component).ownedConnector->select(type<>null).type.memberEnd->
5                select(aggregation=AggregationKind::shared).opposite->forAll(c1,c2 | c1<>c2 implies c1.name=c2.name)
6        then
7            j.oclAsType(Component).ownedConnector->select(c |
8                c.oclAsType(Connector).getAppliedStereotypes()->any(name='AggregationCascade')->notEmpty() and
9                c.oclAsType(Connector).type->size()=1 and
10               c.oclAsType(Connector).type.ownedEnd->isEmpty() and
11               c.oclAsType(Connector)->any(d | d.type->any(memberEnd->exists(aggregation = AggregationKind :: shared))->notEmpty())->notEmpty()
12                ).oclAsType(Connector).end.role.type->select(c | c <> j).oclAsType(Component)
13            null
14        else
15        endif
16    =>select(k | k.oclAsType(Component).general()).oclAsType(Component) = i.general().oclAsType(Component))
17 in
```

51
if
Component.allInstances()->any(a |
a.oclAsType(Component).ownedConnector->any(b |
boclAsType(Connector).getAppliedStereotypes()->any(name='AggregationCascade')->notEmpty() and
b.oclAsTypo(Connector).end.role.type->any(c | c=i and b.owner <> i)->notEmpty())->notEmpty())

ioclAsType(Component).attribute->select(a | a.aggregation = AggregationKind :: none and a.association->notEmpty())->any(b |
boclAsType(Property).type.oclAsType(Component).attribute->any(c | c.aggregation = AggregationKind :: shared and c.type = i))->notEmpty()

i.attribute->any(a | a.aggregation = AggregationKind :: shared)->notEmpty() and
stemp->notEmpty()
then
sel->including(Tuple{c1 = i.general().oclAsType(Component)->any(true),c2 = i.oclAsType(Component),s = stemp})
else
sel
endif

<table>
<thead>
<tr>
<th>#</th>
<th>Status</th>
<th>Description</th>
<th>Solution</th>
</tr>
</thead>
</table>
| 1 | Passed | There is always an association that types the AggregationCascade and that association is an Aggregation. Note that the association being an aggregation implies that it is also binary (only binary associations can be aggregations) | Constraint check in line 6
Note: It may be wise to only check for an association with an aggregation and not specifically shared/composition because composition cascade uses aggregation cascade |
| 2 | Passed | The association is navigable both ways (so the classes own the association ends) | Constraint check in line 5
Note: checking whether the ownedEnds is empty is not the same as checking that the classes own the assoc. ends |
| 3 | Passed | Component A can only aggregate components of the same type B | Constraint check in line 4 |
| 4 | Passed | All components of the hierarchy inherit from the same type | Constraint check in line 12 |

Table 7: Aggregation Cascade constraints [1]

6.5.5 Near Aggregation Cascade

The near aggregation cascade code checks for aggregation without extra constraints. The only aggregation cascade specific construct needed is the connector with an AggregationCascade stereotyped applied to it.

```java
if
Component.allInstances()->iterate(i;sel : Set(Tuple(c2 : Component,s : Set(Component))) = Set() |
let stemp:Set(Component) =
i.oclAsType(Component)->closure(j |
if
j.oclAsType(Component).ownedConnector->any(c | c.oclAsType(Connector).getAppliedStereotypes()->any(name='AggregationCascade')->notEmpty())->notEmpty() |
then
```

52
6.6 Composition Cascade

6.6.1 General Description

Composition Cascade extends Aggregation Cascade by further constraining it. The extra constraint requires that a component only be part of one composite at a time. Composite Associations have a lifetime responsibility for its parts and so if the component is destroyed, its parts (Along the composition cascade) are also destroyed. The Composition cascade uses the Aggregation Cascade primitive by further extending and constraining it. The Indirection primitive is also used to pass the messaging along the cascade.

Note that the Aggregation Cascade constraints are not met as yet and those that are not met will be discussed in the validation sub-paragraph.
6.6.2 Eclipse UML2 Modelling M2 Level

Composition Cascade requires that AggregationCascade be extended. A list of stereotypes and extensions as needed in Eclipse UML2:

1. Stereotype CompositionCascade
   1.1. Generalization set to AggregationCascade
2. Import of AggregationCascade

6.6.3 Eclipse UML2 Modelling M1 Level

The Composition Cascade primitive is modelled as follows in Eclipse UML2:

1. Component Main
   1.1. Port typed by Class from 4.
   1.2. Property typed by Component Composite from 3. with association set to Association from 8.
2. Component Leaf
   2.1. Generalization set to Component Main from 1.
3. Component Composite
   3.1. Generalization set to Component Main from 1.
   3.2. Port typed by Class from 5.
   3.3. Property typed by Component Composite
   3.4. Property typed by Main
   3.5. Property typed by Component Main from 1. with association set to Association from 8.
   3.6. Connector stereotyped by CompositionCascade, AggregationCascade and Indirection and typed by association from 8.
   3.6.1. Connector end with role set to property from 3.3. and partWithPort set to Port from the Composite component
3.6.2. Connector end with role set to property from 3.4. and partWithPort set to Port from Main component

4. Class Main
   4.1. Interface Realization with supplier set to Interface ComplInt from 6.

5. Class Composite
   5.1. Interface Realization with supplier set to Interface indrectedOp from 9.

6. Interface ComplInt

7. Usage with client set to Class Composite from 5. and supplier set to Interface ComplInt from 6.

8. Association

9. Interface IndirectedOp

6.6.4 Composition Cascade Validation

The algorithm for Composition Cascade is similar to Aggregation Cascade. The main loop finds the 'top' (In Figure 21 it would actually be the bottom component) component by making sure that it is not connected to by a connector that is stereotyped by CompositionCascade or connected to by an association with a composite end. From there the algorithm recursively moves along the cascade by checking that the CompositionCascade stereotype has been applied to the connector and that the connector is typed by the composite association.

![Composition Cascade Scenarios](image)

Figure 26: Composition Cascade Scenarios

Figure 26 shows the possible scenarios of Composition Cascade. The length of the cascade is variable like it is with the Aggregation Cascade. The black components represent standard UML modelling and show that it should still be possible to have standard connectors and not mistake it for an extension of the cascade.
Pseudo code:

1. (Line 1) Iterate through all components
2. (Line 1) Set return parameters to a component and a set of components within a tuple
3. (Line 3) Start a recursive loop using the closure command. This will traverse the length of the cascade so long as the constraints per link (connector and association with composite end) are met
4. (Line 5) Check whether a connector exists with a CompositeCascade stereotype applied and that the connector is typed by an association with a composite end
5. (Line 6) Check whether the parent (As defined by generalization) is the same component as the other end of the connector
6. (Line 7) Check whether the connector is connected to the same component as the composite association
7. (Line 9) If the closure should move along the cascade then move along the connector to the next component
8. (Lines 16-22) Iterate through all components and check that no components have connectors with CompositeCascade stereotyped applied or composite associations connected to the current component
9. (Line 26) If the cascade is not empty and the current component has no composition cascade connectors connected to it then add the current Composite Cascade to the list

OCL code:

```oclmultimarkdown
Component.allInstances()->iterate((i;sel : Set(Tuple(c2 : Component, s : Set(Component)))) = Set() |
  let stemp:Set(Component) =
    i.oclAsType(Component)->closure(j |
      j.oclAsType(Component).ownedConnector->any(c | c.oclAsType(Connector)->any(d | d.type->any(memberEnd->exists(aggregation = AggregationKind :: composite))->notEmpty(i)->notEmpty() and c.oclAsType(Connector).getAppliedStereotypes()->any(name='CompositionCascade')->notEmpty())->notEmpty() and
      j.general()->select(oclIsKindOf(Component))->includes(j.oclAsType(Component).ownedConnector->any(c | c.oclAsType(Connector).getAppliedStereotypes()->any(name='CompositionCascade')->notEmpty())->notEmpty())->end.role.type->notEmpty(c | c <> j).oclAsType(Connector)) and
    j.attribute->select(aggregation = AggregationKind::composite).association->select(a | not(a.oclIsUndefined())).memberEnd.owner->includes(j.oclAsType(Component).ownedConnector->any(c | c.oclAsType(Connector).getAppliedStereotypes()->any(name='CompositionCascade')->notEmpty())->notEmpty())->end.role.type->notEmpty(c | c <> j).oclAsType(Connector)) |
  if
    j.oclAsType(Component).ownedConnector->any(c | c.oclAsType(Connector).getAppliedStereotypes()->any(name='CompositionCascade')->notEmpty())->notEmpty() and
    j.oclAsType(Component).ownedConnector->any(c | c.oclAsType(Connector)->any(true))
  else
    null
  endif |
  in
```

56
if
Component.allInstances()->any(a |
aoclAsType(Component).ownedConnector->any(b |
boclAsType(Connector).getAppliedStereotypes()->any(name='CompositionCascade')->notEmpty() and
boclAsType(Connector).end.role.type->any(c | c->i and b.owner <> i)->notEmpty())->notEmpty() and
ioclAsType(Component).attribute->select(a | a.aggregation = AggregationKind :: none and a.association->notEmpty())->any(b |
boclAsType(Component).attribute->oclAsType(Connector).ownedConnector->any(c | c.oclAsType(Connector).getAppliedStereotypes()->any(name='CompositionCascade')->notEmpty() and
boclAsType(Connector).end.role.type->select(c | c <> b.owner)->notEmpty()) and
i.attribute->any(a | a.aggregation = AggregationKind :: composite and c.type = i)->notEmpty() and
stemp->notEmpty() then
sel->including(Tuple{c2 = ioclAsType(Component), s = stemp})
else
sel
endif)

<table>
<thead>
<tr>
<th>#</th>
<th>Status</th>
<th>Description</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Passed</td>
<td>The association that types the CompositionCascade is a CompositeAggregation</td>
<td>Constraint checked in line 5</td>
</tr>
<tr>
<td>2</td>
<td>Failed</td>
<td>Aggregation Cascade constraint: There is always an association that types the AggregationCascade and that association is an Aggregation. Note that the association being an aggregation implies that it is also binary (only binary associations can be aggregations)</td>
<td>Will not work because the association is a composite in this case Possible solution: only check for an association with a shared or composite in Aggregation Cascade</td>
</tr>
<tr>
<td>3</td>
<td>Failed</td>
<td>All components of the hierarchy inherit from the same type</td>
<td>Will not work because Composite Cascade inherits from the component above it</td>
</tr>
</tbody>
</table>

Table 8 : Composition Cascade constraints [1]

6.6.5 Near Composition Cascade

The near Composition Cascade code checks for Composition without extra constraints. The only Aggregation Cascade specific construct needed is the connector with a CompositionCascade stereotyped applied to it.

Component.allInstances()->iterate(csel : Set(Tuple(c2 : Component, s : Set(Component))) = Set() |
let stemp:Set(Component) =
ioclAsType(Component)->closure(j |
if
joclAsType(Component).ownedConnector->any(c | coclAsType(Connector).getAppliedStereotypes()->any(name='CompositionCascade')->notEmpty())->notEmpty() then
joclAsType(Component).ownedConnector->any(c | coclAsType(Connector).getAppliedStereotypes()->any(name='CompositionCascade')->notEmpty())->notEmpty() .oclAsType(Connector).end.role.type->select(c | c <> joclAsType(Component))->notEmpty(true)
else
null
endif
6.7 Shield

6.7.1 General Description

The Shield primitive provides groups of components with a shield mechanism that ensures that components cannot connect to the shielded components without accessing a special intermediary or access point component. This intermediary component could have features to determine if a component may or may not connect to the shielded components. Shielded components are all part of a group and the shield group is specified by the group itself and a reference to it from the intermediary components port. The port is stereotyped by ShieldPort and this allows for a tagged value that references to the shield group. The connecting client should have a connector stereotyped by Shield and the connector should have a matching required/provided interface stereotyped by IShield.

![Figure 27: Shield](image_url)

6.7.2 Eclipse UML2 Modelling M2 Level

Shield requires that interface, port and connector be extended. A list of stereotypes and extensions as needed in Eclipse UML2:

1. Element Import Interface
2. Element Import Port
3. Element Import Connector
4. Stereotype IShield
5. Extension Interface_IShield
6. Stereotype Shield
7. Extension Connector_Shield
8. Stereotype ShieldPort
9. Extension Port_ShieldPort

6.7.3 Eclipse UML2 Modelling M1 Level

The Shield primitive is modelled as follows in Eclipse UML2:

1. Component Client
   1.1. Port typed by Class from 5.
   1.2. Property typed by Component Client
   1.3. Property typed by Component AccessPoint
   1.4. Connector stereotyped by Shield
       1.4.1. Connector end with role set to property from 1.2. and partWithPort set to Port from the Client component
       1.4.2. Connector end with role set to property from 1.3. and partWithPort set to Port from the AccessPoint/Intermediate component
2. Component AccessPoint
   2.1. Port stereotyped by ShieldPort and typed by Class from 6.
3. Component Protected
4. Package stereotyped by Group
   4.1. ElementImport set to AccessPoint
   4.2. ElementImport set to Protected
5. Class Client
6. Class AccessPoint
   6.1. Interface Realization set to Interface ShieldInt from 7.
7. Interface ShieldInt stereotyped by IShield
8. Usage with client set to Class Client and supplier set to Interface ShieldInt from 7.

6.7.4 OCL Solution

The algorithm should find all shields within a model. This has been achieved by looping through all components and assuming that the main component per loop is the access point or intermediary. From that component, checks will be made to look for clients and the shield group which has a reference from the port of the access point/intermediary component. The components that are part of the shield group are also checked to make sure that no components outside of the shield connect to the components within the shield. The return result consists of a tuple consisting of an access point/intermediary component, a package (Shield group) and a set of client components.
Figure 28 shows two shield scenarios. The left scenario consists of multiple clients all connecting through the intermediary/access point component (Using port stereotyped by ShieldPort). This connection is the correct way of accessing a shield. However, the left scenario also shows a component trying to connect a component within the shield group and thus this will not qualify as a validated shield. The right scenario shows that it is also possible to have a shield without protected components.

Pseudo code:

1. (Line 1) Iterate through all components
2. (Line 1) Set return parameters to a component, a package and a set of components within a tuple
3. (Line 3) Set clienttemp as the set that contains all clients. This is tested based on the code from lines 5-18
4. (Line 4) Find a connector that meets the query values from lines 5-18. If a connector is found then
the component on the other end is a client
5. (Line 5) Set thisPort to be the port on the access point/intermediary end
6. (Line 6) Set otherPort to be the port on the client end
7. (Line 7) Check whether the port on the access point/intermediary end has the ShieldPort stereotype applied
8. (Line 8) Check whether the connector has the Shield stereotype applied
9. (Lines 9-13) Check whether the access point/intermediary component provides an interface with
IShield stereotype applied, whether all the features of the interface are public and that a component
from a group has it imported as a member
10. (Line 14) Check whether the client has a required interface that has the IShield stereotype applied
11. (Line 15) Check whether the provided and required interface of the connection is the same
12. (Line 16) Check whether the connector has two ends
13. (Line 17) Check whether the connector is typed (By an association)
14. (Line 20) Set shieldPortStereotype as the stereotype applied to the port (For convenience)
15. (Line 21) Set shieldgrouptemp as a set of components that are part of the shielded group
16. (Line 22) Set shieldedcomponents as a set of components that are shielded (So the access point and intermediaries are not included)

17. (Line 23) Set notinshieldgroup as a set of components that are not allowed to call components that are shielded

18. (Line 27) Check that components within notinshieldgroup do not call components within shieldedcomponents

19. (Line 29) Add the primitive to the tuple if all constraints are met

OCL code:

```ocl
c::Component.allInstances().iterate(c: :Tuple(c : Component, group : Package, client : Set(Component))) = Set[] | 
  let comp : Component = c.oclAsType(Component),
  clienttemp : Set(Component) = c.ownedConnector->any(conn | 
    let thisPort : Port = conn.oclAsType(Connector).end.partWithPort->any(owner=comp).oclAsType(Port),
    otherPort : Port = conn.oclAsType(Connector).end.partWithPort->any(owner<>comp).oclAsType(Port) in
      thisPort.getAppliedStereotypes().any(name='ShieldPort') and
      conn.getAppliedStereotypes().any(name='Shield') and
      thisPort.getProvideds().any(k | k.feature->forAll(f | f.oclAsType(Feature).visibility = VisibilityKind::public) and
        Package.allInstances().importedMember.oclAsType(Component).provided->includes(k) ) and
      otherPort.getRequireds().getAppliedStereotypes().any(name='Shield') and
      conn.end->size()=2 and
      conn.type->size()=1 and
      conn.type.ownedEnd->isEmpty() and
    )->notEmpty() and
  )
  shieldPortStereotype : Stereotype = comp.ownedPort->any(true).getApplicableStereotypes()->
                        any(name='ShieldPort'),
  shieldgrouptemp : Package = Package.allInstances()->any(name=(comp.ownedPort->any(p | p.getAppliedStereotypes()->any(name='ShieldPort'))->notEmpty()/>.getAppliedStereotypes('shieldPortStereotype','shieldGroup').oclAsType(String)),
  shieldedcomponents : Set(Component) = Component.allInstances()->select(s | shieldgrouptemp.importedMember->includes(s) and not(
    Package.allInstances()->any(name=(s.ownedPort->any(p | p.getAppliedStereotypes()->any(name='ShieldPort'))->notEmpty()/>.getAppliedStereotypes('shieldPortStereotype','shieldGroup').oclAsType(String))=shieldgrouptemp)),
  notinshieldgroup : Set(Component) = Component.allInstances()->select(s | shieldgrouptemp.importedMember->
                          excludes(s))->select(s | clienttemp->excludes(s)) |
  in
  if
    clienttemp->notEmpty() and
    notinshieldgroup->select(n | n.oclAsType(Component).ownedConnector.end.role.type.oclAsType(Component)->select(m | shieldedcomponents->
                 includes(m))->notEmpty())->isEmpty()
  then
    sel->including(Tuple(c = comp, group = shieldgrouptemp, client = clienttemp))
```
6.7.5 Near Shield

The near Shield checks for Composition without extra constraints. The only Shield specific construct needed is the port with the ShieldPort stereotyped applied to it.

```
1 Component.allInstances()->iterate((sel : Set(Tuple(c : Component, group: Package, client : Set(Component)))) = Set() |
2   let comp : Component = i.oclAsType(Component),
3   clienttemp : Set(Component) = Component.allInstances()->select(c |
4     c.ownedConnector->any(conn |
5       let thisPort : Port = conn.oclAsType(Connector).end.partWithPort->any(owner=comp).oclAsType(Port),
6       otherPort : Port = conn.oclAsType(Connector).end.partWithPort->any(owner<>comp).oclAsType(Port) in
7       thisPort.getAppliedStereotypes()->any(name='ShieldPort')->notEmpty() |
8     )->notEmpty(),
9   shieldPortStereotype : Stereotype = comp.ownePort->any(true).getApplicableStereotypes()->any(name='ShieldPort')
10   | shieldgrouptemp : Package = Package.allInstances()->any(name=(comp.ownePort->any(p | p.getAppliedStereotypes()->any(name='ShieldPort')
11     )->notEmpty().getValue('shieldPortStereotype','shieldGroup').oclAsType(String)))
12   in
13  clienttemp->notEmpty() |
14  then
15     sel->including(Tuple(c = comp, group = shieldgrouptemp, client = clienttemp))
16   else
17     sel
```
6.8 Typing

6.8.1 General Description

The typing primitive provides a method of modelling domain types that can change at runtime. This is achieved using the TypeConnector and SuperTypeConnector stereotypes. When a TypeConnector is applied it implies that the calling component is typed by the target component. A SuperTypeConnector can only be applied when the calling component is already a type component as depicted in Figure 10.

![Figure 29: Typing](image)

6.8.2 Eclipse UML2 Modelling M2 Level

Typing requires that connector be extended. A list of stereotypes and extensions as needed in Eclipse UML2:

1. Element Import Connector
2. Stereotype TypeConnector
3. Extension Connector_TypeConnector
4. Stereotype SuperTypeConnector
5. Extension Connector_SuperTypeConnector

6.8.3 Eclipse UML2 Modelling M1 Level

The Typing primitive is modelled as follows in Eclipse UML2:

1. Component A
   1.1. Port typed by Class from 4.
   1.2. Property typed by Component A
   1.3. Property typed by Component AType
   1.4. Connector stereotyped by TypeConnector
       1.4.1. Connector end with role set to property from 1.2. and partWithPort set to Port from the A component
       1.4.2. Connector end with role set to property from 1.3. and partWithPort set to Port from the AType component
2. Component AType
   2.1. Port typed by Class from 5.
   2.2. Port typed by Class from 6.
   2.3. Property typed by Component AType
   2.4. Property typed by Component BSuperType
   2.5. Connector stereotyped by SuperTypeConnector
3. Component BSuperType
   3.1. Port typed by Class from 7
4. Class A
5. Class AType1
6. Class AType2
   6.1. Interface Realization with supplier set to TypeInt
7. Class BSuperType
8. Interface TypeInt
9. Interface SuperTypeInt
10. Usage with client set to Class AType from 5. and supplier set to SuperTypeInt from 8.
11. Usage with client set to Class A from 4. and supplier set to TypeInt

6.8.4 OCL Solution

The algorithm should find all Typing and Supertyping primitives within a model. This has been achieved by looping through all components and checking whether TypeConnector or SuperTypeConnector stereotype has been applied. Circularity of the connectors with TypeConnector and SuperTypeConnector has to be checked to make sure that this is not the case.

Figure 30: Typing Scenarios

Figure 30 shows two scenarios of typing. The top scenario shows two TypeConnectors and a SuperTypeConnector. The bottom scenario shows a single TypeConnector. This shows that the length of typing can vary. The algorithm should also differentiate between normal connections.

Pseudo code:

1. (Line 1) Iterate through all components
2. (Line 1) Set return parameters to a component, a bag of components and another bag of components within a tuple
3. (Line 2) Set comp as the current component and assume that it is the start of the Typing/SuperTyping primitive
4. (Line 3) Set c2temp as a bag of components that will be the other end of the connector. The connector in this case is stereotyped by TypeConnector
5. (Line 5) Check whether the connector is stereotyped by TypeConnector
6. (Line 6) Check whether the connector has two ends
7. (Line 9) Select the ‘other’ component of the connectors that have been selected
8. (Line 10) Set c3temp as a bag of components that will be the other end of the connector. The connector in this case is stereotyped by SuperTypeConnector
9. (Line 12) Check whether the connector is stereotyped by SuperTypeConnector
10. (Line 13) Check whether the connector has two ends
11. (Line 16) Select the ‘other’ component of the connectors that have been selected
12. (Line 20,21) Traverse along all connectors that have TypeConnector stereotype applied and make sure that it is not connected in a circularity
13. (Line 25,26) Traverse along all connectors that have SuperTypeConnector stereotype applied and make sure that it is not connected in a circularity
14. (Line 29) If either a TypeConnector or SuperTypeConnector stereotype is applied to an owned connector of comp and there are no circular connections using those stereotypes then add the primitive to the list

```
Component.allInstances()->iterate({ sel : Set(Tuple(c1 : Component, s : Bag(Component), s1 : Bag(Component))) = Set{} |
  let comp : Component = i.oclAsType(Component),
  c2temp : Bag(Component) = comp.ownedConnector->select(j |
    if
      j.oclAsType(Connector).getAppliedStereotypes()->notEmpty().and
      j.oclAsType(Connector).end->size()==2
    then true else false
    endif
  ).oclAsType(Connector).end.role.type->select(c | c <> comp).oclAsType(Component),
  c3temp : Bag(Component) = comp.ownedConnector->select(j |
    if
      j.oclAsType(Connector).getAppliedStereotypes()->notEmpty().and
      j.oclAsType(Connector).end->size()==2
    then true else false
    endif
  ).oclAsType(Connector).end.role.type->select(c | c <> comp).oclAsType(Component) in
  if
    c2temp->notEmpty() and
    comp->closure(l |
      l.oclAsType(Connector).ownedConnector->select(conn | conn.getAppliedStereotypes()->
        any(name='TypeConnector')->notEmpty()).end.role.type->select(m | m <> l).oclAsType(Component))
```
Architectural Primitives Project
Edition 1.0

\[\text{excludes(comp)}\]

\[\text{or}\]

\[\text{c3temp->notEmpty()}\]

\[\text{comp->closure(l | l.oclAsType(Component).ownedConnector->select(conn | conn.getAppliedStereotypes() -> any(name='SuperTypeConnector') -> notEmpty()).end.role.type->select(m | m <> l).oclAsType(Component)}\]

\[\text{excludes(comp)}\]

\[\text{then}\]

\[\text{sel->including(Tuple\{c1=comp, s = c2temp, s1 = c3temp \})}\]

\[\text{else}\]

\[\text{sel}\]

\[\text{endif}\]

<table>
<thead>
<tr>
<th>#</th>
<th>Status</th>
<th>Description</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Passed</td>
<td>A Type Connector has only two ends</td>
<td>Constraint checked in line 16</td>
</tr>
<tr>
<td>2</td>
<td>Passed</td>
<td>A Type Connector might not be applied in circular order</td>
<td>Constraint checked in line 21</td>
</tr>
<tr>
<td>3</td>
<td>Passed</td>
<td>A Super Type Connector has only two ends</td>
<td>Constraint checked in line 13</td>
</tr>
<tr>
<td>4</td>
<td>Passed</td>
<td>A Super Type Connector might not be applied in circular order</td>
<td>Constraint checked in line 16</td>
</tr>
</tbody>
</table>

Table 10: Typing constraints [1]

### 6.8.5 Near Typing

The near Typing code checks for Typing without extra constraints. The only Typing specific construct needed is that the connector has the TypeConnector or SuperTypeConnector stereotyped applied to it.

```cpp
Component.allInstances().->iterate(i; sel : Set(Tuple\{c1 : Component, s : Bag(Component), s1 : Bag(Component)\}) = Set{} |
    let comp : Component = i.oclAsType(Component),
    c2temp : Bag(Component) = comp.ownedConnector->select(j | j.oclAsType(Connector).getAppliedStereotypes() -> any(name='TypeConnector') -> notEmpty())
    then true else false
    endif
    .oclAsType(Connector).end.role.type->select(c | c <> comp).oclAsType(Component),
    c3temp : Bag(Component) = comp.ownedConnector->select(j | j.oclAsType(Connector).getAppliedStereotypes() -> any(name='SuperTypeConnector') -> notEmpty())
    then true else false
    endif
    .oclAsType(Connector).end.role.type->select(c | c <> comp).oclAsType(Component) in
    if
    c2temp->notEmpty() or c3temp->notEmpty()
    then
    sel->including(Tuple\{c1=comp, s = c2temp, s1 = c3temp \})
    else
    sel
    endif
```
6.9 Virtual Connector

6.9.1 General Description
The Virtual Connector primitive provides a method of modelling a connection between two components when they are not actually connector but rather connector through intermediaries and thus form a logical connection. Using the Virtual Connector primitive it is possible to see that the connection is a logical connection rather than an actual connection.

![Virtual Connector Diagram]

**Figure 31: Virtual Connector**

6.9.2 Eclipse UML2 Modelling M2 Level
1. Element Import Connector
2. Element Import Interface
3. Stereotype IVirtual
4. Extension Interface_IVirtual
5. Stereotype VirtualConnector
6. Extension Connector_VirtualConnector

6.9.3 Eclipse UML2 Modelling M1 Level
1. Component A
   1.1. Port typed by Class A from 3.
   1.2. Property typed by Component A
   1.3. Property typed by Component B
   1.4. Connector stereotyped by VirtualConnector
      1.4.1. Connector end with role set to property from 1.2. and partWithPort set to Port from the A component
      1.4.2. Connector end with role set to property from 2.2. and partWithPort set to Port from the B component
2. Component B
   2.1. Port typed by Class B from 4.
3. Class A
4. Class B
   4.1. Interface Realization with supplier set to VirtualConnectorInt from 5.
5. Interface VirtualConnector Int stereotyped by VirtualConnector
6. Usage with client set to Class A and supplier set to Interface VirtualConnectorInt

6.9.4 OCL Solution

The algorithm should find all Virtual Connections within a model. This has been achieved by looping through all components and checking whether Virtual Connector has been applied to an owned connector. The algorithm should also check that the two components that are connected via a Virtual Connector are actually connected through (A series of) normal connectors.

![Diagram of Virtual Connector Scenario]

Figure 32: Virtual Connector Scenario

Figure 28 shows two scenarios of Virtual Connector. The difference between the scenarios is that the lengths of the intermediate connectors are varied. What this shows is that there must be some connection between the virtual connection components and that the length of the connection can vary. It could even be a series of 10 or more intermediate connections.

Pseudo code:

1. (Line 1) Iterate through all components
2. (Line 1) Set return parameters to a component and a bag of components within a tuple
3. (Line 2) Set comp as the current component.
4. (Line 3) Select the connectors that pass the checks from lines 4-14
5. (Line 4) Set callerPort as the port on the caller side
6. (Line 5) Set receiverPort as the port on the receiver side
7. (Line 6) Check whether the connector has the VirtualConnector stereotype applied
8. (Line 7) Check whether the connector has two ends
9. (Line 8) Check whether the required interface of the caller has IVirtual stereotype applied
10. (Line 9) Check whether the provided interface of the receiver has IVirtual stereotype applied
11. (Line 10) Check whether the required interface of the caller matches the provided interface of the receiver
12. (Lines 11-14) Check whether there is actually a path of connectors between the two components
13. (Line 20) Add Virtual Connector if constraints are met

OCL code:

```ocl
Component.allInstances().iterate(sel : Set(Tuple(c1 : Component, s : Bag(Component))) = Set{} |
2  let comp : Component = c1.oclAsType(Component),
3  stemp : Bag(Component) = comp.ownedConnector->select()
4  let callerPort : Port = j.oclAsType(Connector).end.partWithPort->any(owner=i).oclAsType(Port),
5  receiverPort : Port = j.oclAsType(Connector).end.partWithPort->any(owner<>i).oclAsType(Port) in
6  j.oclAsType(Connector).getAppliedStereotypes()->any(name='VirtualConnector')->notEmpty() and
7  j.oclAsType(Connector).end->size()=2 and
8  callerPort.getRequireds().getAppliedStereotypes()->any(name='IVirtual')->notEmpty() and
9  receiverPort.getProvideds().getAppliedStereotypes()->any(name='IVirtual')->notEmpty() and
10 callerPort.getRequireds() = receiverPort.getProvideds() and
11 callerPort.owner->closure(l |
12 Component.allInstances()->select(s | (s<>l and s.ownedConnector->select(t |
13  t.oclAsType(Connector).getAppliedStereotypes()->any(name='VirtualConnector')->isEmpty()).end.role.type->select(m | m<>l)->notEmpty()) or t.oclAsType(Component).ownedConnector->select(t | t<>l)->includes(s))
13 )->includes(receiverPort.owner.oclAsType(Component))
14  j.oclAsType(Connector),end.role.type->select(c | c <> comp).oclAsType(Component) in
15 if
16  stemp->isEmpty()
17 then
18  sel
19 else
20  sel->including(Tuple(c1=comp, s = stemp))
21 endif
22 )
```

<table>
<thead>
<tr>
<th>#</th>
<th>Status</th>
<th>Description</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Passed</td>
<td>A Virtual Connector has only two ends</td>
<td>Constraint checked in line 7</td>
</tr>
<tr>
<td>2</td>
<td>Passed</td>
<td>A Virtual Connector matches the provided IVirtual interface of one component to the matching required interface of another</td>
<td>Constraint checked in line 10</td>
</tr>
<tr>
<td>3</td>
<td>Passed</td>
<td>A Virtual Connector can only be used between two components A and B, if there is a path of components and connectors that link A to B.</td>
<td>Constraint checked in lines 11-14</td>
</tr>
</tbody>
</table>

Table 11: Virtual Connector constraints [1]

6.9.5 Near Virtual Connector

The near Virtual Connector code checks for Virtual Connector without extra constraints. It only checks whether the VirtualConnector stereotype is applied and not whether the components are in a line of communication with each other or whether the connector has two ends.
Component.allInstances().iterate(i; sel : Set(Tuple(c1 : Component, s : Bag(Component))) = Set() |
  let comp : Component = i.oclAsType(Component),
  stemp : Bag(Component) = comp.ownedConnector->select(j |
    let callerPort : Port = j.oclAsType(Connector).end.partWithPort->any(owner=i).oclAsType(Port),
    receiverPort : Port = j.oclAsType(Connector).end.partWithPort->any(owner<>i).oclAsType(Port) in
    j.oclAsType(Connector).getAppliedStereotypes()->any(name='VirtualConnector')->notEmpty() |
    ).oclAsType(Connector).end.role.type->select(c | c <> comp).oclAsType(Component) in
    if
      stemp->isEmpty()
    then
      sel |
    else
      sel->including(Tuple(c1=comp, s = stemp))
    endif
  )}
7 Results

In the paper from Zdun and Paris [1] they show how primitives can be used to model a peer-to-peer system. This system will be modelled in Primus to see how primitives work in practice.

7.1 Leela

Leela [1] is a system of federated peers where all peers are equal. The peers can offer services to each other by letting peers communicate with other peers. Zdun and Paris have modelled the system using patterns and these patterns can be modelled using primitives.

The main patterns used in Leela include: Broker, Layers, Remoting and Client-Server. The primitives used in Leela include: Callback, Indirection, Grouping, Layers, Aggregation Cascade, Shield and Virtual Connector.

A full explanation about how Leela works will not be given but the following list contains the main pattern/primitive related aspects of the system:

- The Remoting pattern requires that the Layers pattern be used. The Layers pattern can be modelled using the Layers primitive.
- The client and server are modelled using the Grouping primitive.
- Each component in each layer of the Broker pattern has a Virtual Connection with the respective client/server side.
- Two kinds of peers exist: Ordinary peer and federation of peers. A peer being part of multiple federations is modelled using the Aggregation Cascade primitive.
- Invocation between client and server is modelled using the InvocationInterceptor components. These components have Callbacks between the invocating components.
- The connection point between the client and server is the RequestHandler component. This component is shielded using the Shield primitive.
- Components in the third layer are shielded.
Figure 33: Leela with primitives

Figure 33 shows Leela modelled using primitives. The layers are divided by the grey dotted lines. The groups ClientBroker and ServerBroker have been modelled without lines to the member components to improve readability of the diagram. An explanation of how the system works and why particular primitives were chosen can be found in [1].

The remainder of this paragraph will discuss the process of modelling Leela and testing the model to check the functioning of the primitives.

7.2 Profiles for Leela

The first step in modelling a system using primitives is to make sure that profiles are available that allow us to apply the proper stereotypes to the UML elements within a primitive. This involves defining the profile and loading the profile into the model that will use the profile.

The Callback primitive is used in Leela and will have to be defined in Primus.
The first step needed is to define a new profile. This functionality is provided by UML2 and can be achieved by selecting a project where the profile should reside in and selecting the new option in the context menu. A wizard will appear and will ask what kind of project is desired. In this case a new UML model is desired and later on in the wizard it is possible to choose a UML profile.

- Create a profile in the current project

  Right-click a project->New->UML model->Profile

![Figure 34: Empty Profile (Primus)](image)

Figure 34 shows the result of creating a new profile.

The second step needed is to reference the UML meta-classes that need to be extended. In the case of Callback this includes Connector, Port and Interface as described in the Callback paragraph. It is possible to reference all UML meta-classes in one go but by only selecting the classes needed it will make for a smaller profile.

- Referencing meta-classes and complete meta-models can be achieved by selecting the profile and selecting the option:

  UML Editor->Profile->Reference Meta-class

![Figure 35: Reference UML Meta-Class (Primus)](image)
Figure 35 shows the menu that allows for referencing meta-classes. The list of meta-classes comes from the UML meta-model and includes all elements from it.

The third step consists of creating the stereotypes needed and extending them from the meta-classes that have been referenced. Callback requires five stereotypes: Callback, EventPort, CallbackPort, IEvent and ICallback as described in the Callback paragraph. To create the Callback port we simply create a stereotype and name it Callback. To extend the connector meta-class we simply create an ‘Extension’ that extends connector and this maps the extension to the Callback stereotype:

- Create stereotype:
  Right click profile->New->Owned Stereotype->Stereotype
- Change stereotype name:
  Properties->name
- Create Extension:
  UML Editor->Stereotype->Create Extension

This should be carried for all of the stereotypes needed. The resulting profile is depicted in Figure 31.

All relevant profiles should be defined and available for import within the Leela model.

### 7.3 Modelling Leela

Now that profiles have been defined it is time to start modelling primitives. The first step needed it to load the profiles and to apply them to the model that will become the Leela component diagram.

- Load Profile
  Right Click package->select Load Resource->Select relevant profile
- Apply Profile
7.3.1 Groups and Layers

The easiest way to start is by creating the components that are part of the groups and layers primitive/pattern. This will also give the model a visual structure that is easy to understand for the modeller by placing the components in a layered fashion. The components needed are: Peer, MarshallerClient, Requestor, Invoker, MarshallerServer, RequestHandler, ProtocolPluginClient and ProtocolPluginServer. The components to the left of centre are part of the client group and the components right of centre are part of the server group. The components of the higher level layers are at the top and the lower level layers at the bottom. The rest of the sub-paragraph will show how a layer can be added. The process for adding a group is the same.

Figure 37: Layer/Broker components (Primus)

Figure 37 shows the components that will initially be added to four layers. The higher the component is in the diagram the higher the layer number.

The Layers primitive is the first primitive that will be added. There are four layers and thus 4 layer primitives are needed. Using the primitive wizard it is possible to create the layer primitives and to assign them their layer number.
Figure 38 shows the layers primitive menu. The new package for layer option lets the user decide whether the layers primitive will be applied to an existing package or a new package. The layer number pull-down menu lets the user select the layer number of the layer that will be created. Components will have to be added to the layers once the layers have been created. This can be done by using the context menu of a component and selecting the relevant layer (or group).

Figure 39 shows the result of creating a layer and adding a component to that layer. As depicted, a line is drawn between the layer and the layer member. The same notation is used to show that components are part of a group.

7.3.2 Shields

There are two shield groups and three access point/intermediary components that need to be added to the model. The shield primitive has four distinctive parts: client(s), access point/intermediary, protected component(s) and the shield group. The Primus primitive menu allows the user to select whether new components or packages should be created for each distinct part or to use existing components or packages.
Figure 40 shows the wizard page for the shield primitive. The combo boxes allow the user to select whether new components or packages should be created. If that is not the case then the selection boxes can be used to select which existing components or packages should be used.
Figure 41 shows a shield that has been applied to the model. The peer component is the client, the Invoker component is the access point/intermediary, the MarshallerServer is the protected component and the ShieldGroup1 package is the shield group. The visualisation is slightly cluttered but all information has been shown in this view to show that various elements have been stereotyped. This can be seen by the use of guillemots ("<< >>"). For example, the port on the Invoker component has a ShieldPort stereotype applied.

### 7.3.3 Callback

Leela requires three callback primitives. Adding callback to the model once again requires the wizard and the options are similar to shield. In the case of the callback between RequestHandler and ProtocolPluginServer it is a matter of choosing these two components.
7.3.4 Virtual Connector

Leela requires five Virtual Connectors.

7.3.5 Aggregation Cascade

The Leela system has an Aggregation Cascade between the Peer component and the Federation component. There is a slight difference between the way that it is modelled in the paper by Zdun and Paris [1] because Zdun and Paris use the Peer component as the component that acts as the main (Parent) component and as the top component in the cascade. However, this would not meet the last constraint from Aggregation Cascade that states that the parent of each component in the cascade

Figure 42: Callback primitive (Primus)

Figure 43: Virtual Connector primitive (Primus)
should be the same. A main component was added and the components in the cascade have a generalization link to it to solve the problem.

Figure 44 shows the Aggregation Cascade. The FederationMain component is the parent of each component and thus acts as the main component.

### 7.3.6 Indirection

Indirection is used in Leela for the Aggregation Cascade because Aggregation Cascade extends Indirection. Indirection is also used between PeerProxy and Peer. Indirection can include a client component that calls the Indirection. In the Indirection between PeerProxy and Peer there is no client and thus in the wizard we must choose to not add a client.

Figure 45 shows an Indirection connection between PeerProxy and Peer. Notice the provided interface that is not matched by a required connection. This is not connected because there is no client.
7.4 Testing Primitives in Leela

The primitives and the validation method have been extensively tested so that they can be used with confidence. Testing the primitives in Leela can show the practicality of using primitives in a model instead of standard UML modelling. This sub-paragraph will test a number of scenarios where a modeller breaks the constraints imposed on the primitives and shows the power of the primitives.

7.4.1 Layers Test

The first test involves the Layers primitive and will try to connect the Peer component to the ProtocolPluginClient component. The Peer component is in the fourth layer and the ProtocolPluginClient is in the first layer and so the Peer cannot connect to the ProtocolPluginClient component without breaking the constraint imposed on it by the Layers primitive.

![Figure 46: Layers Validation (Primus)](image)

Figure 46 shows the result of validating the Layers primitive when a connection between the Peer component and the ProtocolPluginClient is made. It shows that Layer 4 will not validate and only registers as a 'near' layer. This is due to the illegal connection made.

7.4.2 Shield Test

The second test involves the Shield primitive and will try to connect a component directly to a protected component of a shield. This is an illegal connection because the protected component should only be called from access point/intermediary components.

![Figure 47: Illegal Shield Connection (Primus)](image)
Figure 47 shows an illegal connection between a component and MarshallerServer component which is a protected component. Figure 48 shows the result of the validation. The shieldGroup1 group has not validated properly and therefore only register as a near validation.

7.4.3 Virtual Connector Test

A virtual connection signifies a connection between two components that are not actually physically connected but are connected through a series of other connections and components. If the link between the two components is broken then the validation should not be successful.

Figure 48: Shield Validation (Primus)

Figure 49: Illegal Virtual Connector (Primus)
Figure 49 shows three components: ProtocolPluginClient, RequestHandler and ProtocolPluginServer. The ProtocolPluginClient and ProtocolPluginServer components have a virtual connection between them and the actual ‘physical’ connection is via the RequestHandler component. Figure 50 shows the result of the Virtual Connector validation and that the ProtocolPluginClient and ProtocolPluginServer virtual connection is not a valid connection.

Figure 50: Virtual Connection Validation (Primus)
8 Improved Method of Validation

A drawback to the method of validation used in the primitive paragraph is that the primitives are either validated or not validated. If a primitive is not validated then the location of the near primitive will be found by a second query. However, this does not determine what is wrong with the primitive and means that the modeller must look at the primitive details to figure out what is wrong. This can be a difficult task if the modeller is not an expert in the design of a primitive.

This problem can be solved by improving the validation code and changing the interpretation of it. The current validation only returns the location of primitives if they are indeed a properly validated primitive. So in the case of Callback the return value is:

Tuple {component1, Bag (component2)}

This return value has a component and a bag (Collection) of components. So the Callback is between the first component and the Bag of components (Multiple Callback’s are possible originating from one component). Based on this information it is only possible to determine if a Callback exists or not and does not tell us if a Callback has nearly validated.

Callback return value that returns more details:

Tuple{component1,component2,connector1,connector2,port1,port2,callerProvided,callerRequired,callbackProvided,callbackRequired,observeEventMatch,updateMatch}

Using the more detailed return value it is possible to find something that is wrong with a primitive because a number of the return values are associated with a constraint. For example, the return value Connector2 is associated with the constraint that the connector must have two connector ends. In the query the current selected connector is returned if it has two connector ends and null if it has something other than two connector ends. This tells us that if Connector2 is null then the current potential Callback does not adhere to the two connector ends constraint.

The complete algorithm of the new validation method for Callback will be presented:

Pseudo code:

1. (Line 1) Iterate through all components, use collect instead of select because the return value is not a subset of the collection from all instances (Have not used iterate because a double iterate loop has implementation problems)
2. (Line 2) Iterate through all connectors of the current component
3. (Line 3) Retrieve the caller side port and make sure that it is stereotyped by EventPort
4. (Line 4) Retrieve the callback side port and make sure that it is stereotyped by CallbackPort
5. (Line 5) Retrieve the caller side port in case the port is not stereotyped so that the port can be used in following lines of code
6. (Line 6) Retrieve the callback side port in case the port is not stereotyped so that the port can be used in following lines of code
7. (Line 7) Retrieve the connector and make sure that it is stereotyped by Callback
8. (Line 8) Retrieve the connector and make sure that it has two connector ends
9. (Line 9) Retrieve the callback side component
10. (Line 10) Retrieve the caller provided interface and make sure that it is stereotyped
11. (Line 11) Retrieve the caller required interface and make sure that it is stereotyped
12. (Line 12) Retrieve the callback provided interface and make sure that it is stereotyped
13. (Line 13) Retrieve the callback required interface and make sure that it is stereotyped
14. (Line 14) Make sure that the ObserveEvent interface is matched by the required and provided of both ports/components
15. (Line 15) Make sure that the Update interface is matched by the required and provided of both ports/components
16. (Line 16) Return all values

Component.allInstances().collect(i |
  i.ownedConnector.collect(conn |
    let callerPortSter : Port = conn.end.partWithPort.oclAsType(Port).applyStereotypes().name='EventPort').notEmpty(),
  callbackPortSter : Port = conn.end.partWithPort.oclAsType(Port).applyStereotypes().name='CallbackPort').notEmpty(),
  callerPort : Port = conn.end.partWithPort.oclAsType(Port),
  callbackPort : Port = conn.end.partWithPort.oclAsType(Port),
  callbackConn : Connector = conn.applyStereotypes().name='Callback').notEmpty(),
  callbackConn2End : Connector = conn.applyStereotypes().size=2,
  otherComp : Component = if callbackPort.oclAsType(Component).notEmpty() then callbackPort.owner.oclAsType(Component) else null endif,
  callerProv : Interface = callerPort.getProvideds().applyStereotypes().name='IEvent').notEmpty(),
  callerReq : Interface = callerPort.getRequireds().applyStereotypes().name='ICallback').notEmpty(),
  callbackProv : Interface = callbackPort.getProvideds().applyStereotypes().name='ICallback').notEmpty(),
  callbackReq : Interface = callbackPort.getRequireds().applyStereotypes().name='IEvent').notEmpty(),
  observeEventMatch : Integer = if callerPort.getProvideds() = callbackPort.getRequireds() and callerPort.getProvideds().notEmpty() then 1 else 0 endif,
  updateMatch : Integer = if callerPort.getRequireds() = callbackPort.getProvideds() and callerPort.getRequireds().notEmpty() then 1 else 0 endif
in
  Tuple{c1 = i, c2 = otherComp, callbackConn = callbackConn, callbackConn2End = callbackConn2End, callerPort=callerPortSter, callbackPort = callbackPortSter, callerProv = callerProv, callerReq = callerReq, callbackProv = callbackProv, callbackReq = callbackReq, observeEventMatch = observeEventMatch, updateMatch = updateMatch}).asSet()
Figure 51: Validation of callback using improved method (Primus)

Figure 51 shows the result of validating Leela for the Callback primitive using the improved validation method. One of the Callbacks was altered by unapplying the Callback stereotype. As Figure 51 indicates the problem was found and so the modeller can go directly to the relevant connector and fix the problem instead of having to look at the whole primitive.
9 Related Work

The research and development that has taken place in this paper is related to architectural primitives and more specifically the modelling of architectural primitives in UML 2. Modelling of primitives within UML 2 can be split into two areas: defining and applying primitives by using UML profiles and automated checking of stereotypes using OCL. The use of OCL in this project has differed to most other projects because it has been used to query the model and thus check for constraints using queries instead of applying constraints on the model using the OCL constraint method.

This research is based on the work from Zdun and Paris [1]. In their paper they propose the concept of architectural primitives for modelling of architectural patterns and present each primitive and constraints needed per primitive. Their implementation differs in that they do not describe the exact UML elements needed to model a primitive (Although it is very close) and that their validation does not take into account that elements should only be constrained if they have a stereotype applied (And that the relevant stereotype has constraints).

Another difference between the work in [1] and this project is that primitives in this project are searched for (And constrained within the same query) within a model. This is because primitives in [1] are defined in an individual basis and not within a model with other primitives and UML modelling not related to primitives. It is not sufficient to just constrain elements that have a stereotype applied because that does not guarantee that what should be a primitive has had all the relevant stereotypes applied properly.

Using UML to model software architectures is not new. In [15] Giesecke and Marwede propose mapping MidArch, an Architectural Description Language, to UML and thus model software architecture concepts specific to middleware platforms in UML. They also use OCL constraints to improve the precision of the specification. The goal of the [15] project differs somewhat to the goal of this project because this project focuses on modelling architectural primitives which in turn model architectural patterns, as opposed to modelling architectural patterns specifically for middleware platforms. Giesecke and Marwede have a different approach to modelling connectors in that they use collaborations instead of connectors. The reason given is that it is impossible to type connectors and thus UML connectors lack the expressiveness of architectural connectors. Another difference between the two projects is that they use OCL to define constraints and we use OCL to define queries that in turn check constraints.
10 Conclusion and Future Work

10.1 Conclusions

The principle of Architectural Primitives has been implemented in a modelling framework and this allows a modeller to create an architecture design using primitives and to verify that the primitives have been applied properly. The results paragraph has shown that applying a primitive is a relatively easy task and that the modeller can receive instant feedback as to whether any constraints have been violated. This has the advantage that key concepts of an architectural pattern can be checked automatically and thus more confidently than a modeller checking these concepts personally.

The research questions from the introduction will be restated along with an answer and explanation.

*What framework best suits the needs of modelling architectural primitives?*

The Eclipse IDE/framework along with the MDT was chosen as the framework to adapt for use with architectural primitives. It was considered the best choice because Eclipse and the plug-ins are open source and so all code and API descriptions are public which a prerequisite of programming in such an environment. Another advantage of the Eclipse framework as compared to other frameworks is that the documentation and support/community is better than the alternatives. The MDT package provides all the necessary plug-ins needed for this project: UML 2 metamodel, model checking using OCL and UML visualisation.

Having implemented architectural primitives in Eclipse it can be concluded that the Eclipse framework was a good choice but it was not a perfect one. The biggest drawback of using Eclipse is the lack of maturity of the UML2Tools plug-in that is responsible for visualisation and interaction of the component diagram. This is due to the fact that the plug-in is still in the pre version 1 stage of development. At the time of choosing the development environment it was thought that the component diagram would be finished by now (At the time of writing this document) but alas this is not the case. But work is still being carried out on all of the plug-ins so the component diagram should be finished soon. The fact that the plug-ins are constantly being upgraded means that future versions will be up to date and new features will be added. For example, the UML2 plug-in is already including features from the yet to be finalised UML version 2.2. The support offered by the community has been excellent. Problems can be posted on the forum and questions are answered by experts that have had a big role in developing the MDT plug-ins.

*How will the primitives, which have been defined using UML elements and stereotypes, be modelled in the framework that has been chosen?*

Each primitive now has an exact definition of the UML elements needed and all of the primitives validate properly according to the UML2 plug-in constraints which in turn come from the UML 2.1 specification. One of the most surprising aspects of UML modelling is the amount of variability in the
way that modellers and especially architectural modellers model certain constructs. This is especially true for the amount of differing methods of modelling connectors. This is due to the complexity of the semantics involved and the somewhat limited architectural elements offered in UML 2 and the fact that there is still interpretation required in understanding and implementing the UML specification. The exact definition of a primitive is given in the primitives paragraph and consists of two lists per primitive. One list for the M2 level definition and one list for the M1 level definition.

*How to perform model checking, which consists of assessing the validity of the primitives and defining which UML elements make up the primitives, as efficiently as possible?*

Checking the validity of and finding the primitives has proven to work but there is still room for improvement. The current implementation validates primitives or finds primitives that nearly adhere to all of the constraints. However, it would be of great benefit to a modeller if any problems would be highlighted to the modeller instead of just an indication that something is wrong. This problem can be solved by improving the queries using to the method used in paragraph 8.

Finding and validating primitives using OCL was quite a difficult task. It was difficult because of a lack of experience using OCL prior to this project, a lack of query examples from other projects but also because of the language itself compared to an all round language like Java. OCL has a small and simple syntax compared to Java and this leads to large queries that can look complex. The queries themselves are actually smaller than equivalent code would be in Java but that is more to do with the fact that Java files have a lot of overhead in terms of lines of code and OCL queries get straight to the point. A feature that is lacking in OCL, at least in the Eclipse framework, is that it is difficult to get output information if something goes wrong with the query or if the query does not return values that are expected. In Java this can easily be accomplished by using debugging tools, by using system out information or by creating a log file. The only way to solve problems with queries in OCL is to break up the code into smaller segments and to check results of the smaller segments. This is a laborious and tedious task. The Eclipse MDT project has already proposed to add an OCL Tools plug-in and this should address some of the problems with extra tools that will make it easier to develop OCL queries.

OCL does take some time to fully understand and to be able to use efficiently but once this has been achieved it is a good way of querying models. Though Java can do the same as OCL in terms of functionality OCL has the advantage that the queries can be shown to other modellers and quickly understood instead of large chunks of Java code.

*How to make the framework as extensible as possible?*

Primitive types can be added relatively easily by adding to the code of the program. The three main tasks required are to define the query that will find and check the constraints of the primitive, to define the UML elements needed in Java code and to adapt the user interface wizard by adding a page to it. All of the tasks mentioned basically consist of varying the existing code and queries and should therefore not be very difficult.
Adding code for the UI and adding the code to actually create the primitives is an easy task because it is just a case of copying from the existing primitives. Adding the OCL code to find and validate the primitives is a more difficult task but should in most cases (based on whether the functionality of the new primitive is similar to an existing primitive) be very similar to existing code from existing primitives.

10.2 Future Work

The Primus tool is a typical first version experimental tool that does a good job of showing the power of architectural primitives. However, the tool is not very stable, partly due to the underlying UML2Tools plug-in and partly due to the tool itself for situations where the user would make 'odd' choices. An obvious first step in improving the tool would be to improve the stability of the tool. Other improvements to the tool will be explained in the following sub-paragraphs.

10.2.1 Increased amount of Options for Adding Primitives to Model

The current situation allows for modellers to use the wizard to add primitives to a model. Certain options are required when adding a primitive. The main option that is currently available is whether to use existing components/packages or to create new components/packages. This is necessary because some components/packages can be part of multiple primitives and so an existing component/package has to be updated instead of creating a new one.

Future improvements could consist of more options for adding primitives. These options could include using existing ports when a new connector is required or using an existing connector if a connection already exists between two components that will have a primitive or part of a primitive applied. For example, if two components are already connected then a Callback could be applied to the two components while using the existing connector. This is especially necessary if multiple primitives are applied to a single connector (Not the case with the currently implemented primitives).

10.2.2 Improved Visualisation

Visualisation of the component diagram is not perfect because the underlying UML2Tools plug-in is yet to be complete. This is obviously something that needs to be improved. Yet there are more improvements for the visualisation of primitives that are possible. Primitives are currently visualised by the stereotypes that are applied to the various UML elements that are part of a primitive but it would be easier to see a primitive if the whole primitive was coloured or marked somehow. Colour codes could also be applied to signify if a primitive has been applied properly. So if a primitive has an error and the location and nature of the error is known then a red colour could be applied to signify that error. The grouping and layering primitives visualisation could also be improved upon by an option that lets users see, by way of a different colour for components, which components are part of the selected group or layer.
10.2.3 More Extensive Testing of Queries

The primitives have been tested extensively but it is always wise to test usage by someone who was not directly involved in the development itself. Another reason for further testing of the OCL code is that the OCL is rather compact and complex and so a mistake can easily be made. Therefore, someone should test the primitives for any mistakes that could have been made.

10.2.4 Code Generation

An important area of research/development that is currently not supported is code generation of the model and specifically how primitives and the use of primitives will affect code generation. The development aspect will have to take into account what Eclipse has already implemented in terms of code generation. Something that is definitely not available with the current code generation is the implementation of the constraints in code. For example, with the Shield primitive it is not allowed to connect to a shielded component except via the access point/intermediary component. This constraint is checked in the model but should also be implemented in the code so that the constraint is adhered to throughout the life cycle of the component.

10.2.5 Defining New Primitives

The list of primitives currently implemented is not the full list of possible primitives and many more primitives will have to be added to model the complete list of architectural patterns. Adding primitive types to Primus can be achieved by adding to the code of the program: adding code to apply primitives to model, add to the user interface and to add model checking code. Development could be made towards automating this task. It may not be possible to automate the whole task of adding a primitive type but a significant portion should be possible.

Automation of a UI page generator for adding a primitive could consist of loading data from an XML file that specifies how many components and packages are required to make a primitive and so the UI can be generated from that list along with details of the options per component/package.

Generating the code to add a primitive to a model could also be automated via an XML file that states the exact list of UML elements and stereotypes needed to make a primitive. So the automated primitive generator could read the elements from the list and generate them along with the list of options that has been inputted by the user. So the list of generated elements could specify that a component is needed. The user input could state whether the user selected that a component needs to be created or one from the current model will be used. The automated primitive generator will then reuse/create the component.

Automating the process of OCL code generation would probably be too difficult to achieve and would remain a job that a developer would have to do. As mentioned before, the code from existing primitives can be reused and so should not prove to be very difficult.
10.2.6 Mining Patterns in Software Design using Primitives

Primitives are used to model architectural patterns. Each pattern has a list of primitives that could possibly be part of certain patterns (See Appendix A). For example, the Layers primitive can be used (obviously) in the Layers pattern so if a Layers primitive is found in a model then it possible that the Layers pattern has been used.

At the moment it is not possible to automatically determine which patterns have been modelled based on the primitives that have been applied to the model. Given some reason research it should be possible to determine which patterns have been applied with some degree of accuracy. A first implementation can consist of merely stating the possibilities based on the list that has been made in Appendix A. Further implementations could base possibilities on the combination and interaction of primitives with each other to more accurately map the primitives to pattern possibilities.
References


[13] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994

Appendix A

Introduction
This document is the development plan for the SADT project with original start in on the 1st of May and planned end on the 30th of November. This version of the document covers the orientation phase, resulting in one release: release 0.1.

During the project, this plan will evolve. At any point in time it contains:
- The project organisation, communication structure;
- The detailed plan of the current phase;
- The global plan of the remaining phases.

This plan is written and maintained by the Project Leader. This version (0.1) covers the orientation phase.

Customer Requirement specification

Required is a software system that can model software architecture using architectural primitives. Software primitives are fundamental, formalizable modelling elements in representing architectural patterns. Architectural primitives are defined by extending a UML compliant meta-model. The meta-model used should be extensible so that users can add primitives.

The software system should extend existing systems with relevant features to increase industry/academic acceptance and to insure quality of basic functionality. Mechanisms should be built to facilitate (future) development of validators of the architectural model and code generators. Future development of the software system is likely and thus full documentation is required along with easily extensible features of the software system.

Assumptions and risks

Introduction

When writing the project plan, the project manager has to make assumptions with respect to external factors that might influence the project. The action for the project leader is to track if these assumptions become risks. If they become a risk, the following information has to be provided:
- The Probability on a range from 1 (low probability) to 5 (very high probability)
- The Seriousness on a range from 1 (low impact) to 5 (very high impact)

During the weekly progress meeting, open risks are discussed and if needed new risks are issued. These risks will be communicated to the customer by means of the weekly progress meeting. The customer and project define further actions.
Assumptions
List of assumptions:

- Primitives can be adequately described by extending UML metamodels

Risk Identification and assessment

<table>
<thead>
<tr>
<th>RXX1</th>
<th>Severity</th>
<th>Probability</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indicators:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Impact:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reaction(s):</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prevention:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Responsible:</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RXX2</th>
<th>Severity</th>
<th>Probability</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indicators:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Impact:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reaction(s):</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prevention:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Responsible:</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RXX3</th>
<th>Severity</th>
<th>Probability</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indicators:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Impact:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reaction(s):</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prevention:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Responsible:</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
in answers to our questions can become a bottleneck in future.

**Prevention:** Ask critical questions related to tool and environment before the start of actual development

**Responsible:** PM, TL, TM

<table>
<thead>
<tr>
<th>RXX4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Severity</strong></td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td><strong>Description:</strong> Slow development due to high levels of integration of existing plugins.</td>
</tr>
<tr>
<td><strong>Indicators:</strong> Missed deadlines.</td>
</tr>
<tr>
<td><strong>Impact:</strong> Slow development</td>
</tr>
<tr>
<td><strong>Reactions:</strong> Use newsgroups and help from domain experts</td>
</tr>
<tr>
<td><strong>Prevention:</strong> Read development literature (Concentrate on books and tutorials)</td>
</tr>
<tr>
<td><strong>Responsible:</strong> PM, TL, TM</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RXX5</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Severity</strong></td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td><strong>Description:</strong> Realization in later stage of the project that the plugins and framework used is not suitable for primitives’ extension.</td>
</tr>
<tr>
<td><strong>Indicators:</strong> Problems that cannot be solved in the later stage of the project.</td>
</tr>
<tr>
<td><strong>Impact:</strong> Project failure/time overrun</td>
</tr>
<tr>
<td><strong>Reactions:</strong> Use a different framework or program solution instead of using the existing problematic plugin.</td>
</tr>
<tr>
<td><strong>Prevention:</strong> Read development literature (Concentrate on books and tutorials) and make sure that current plugins are suitable for the job.</td>
</tr>
<tr>
<td><strong>Responsible:</strong> PM, TL, TM</td>
</tr>
</tbody>
</table>

P = Probability (1-5)

S = Serverity (1-5)

**Milestones / Delivery Schedule**

**Development Model**
Figure 52: Standard Lifecycle

### Schedule

<table>
<thead>
<tr>
<th>Task Name</th>
<th>Duration</th>
<th>Start</th>
<th>Finish</th>
<th>Predecessors</th>
<th>Resource Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool Development Plan</td>
<td>12 days</td>
<td>Tue 01/05/07</td>
<td>Fri 10/10/07</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem Understanding and Project Goal</td>
<td>24 days</td>
<td>Tue 01/05/07</td>
<td>Fri 10/10/07</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Literature review, pattern language and selection of architectural primitives</td>
<td>12 days</td>
<td>Tue 01/05/07</td>
<td>Wed 10/06/07</td>
<td>Ahmed Waqas(50)</td>
<td></td>
</tr>
<tr>
<td>Related work and current practices of tool development in eclipse platform</td>
<td>3 days</td>
<td>Thu 10/06/07</td>
<td>Mon 21/06/07</td>
<td>Ahmed Waqas(50)</td>
<td></td>
</tr>
<tr>
<td>Short learning to develop tool in eclipse</td>
<td>4 days</td>
<td>Tue 22/06/07</td>
<td>Fri 25/06/07</td>
<td>Rais(75%)</td>
<td></td>
</tr>
<tr>
<td>Proposed approach to modeling patterns by the use of primitives in tool</td>
<td>5 days</td>
<td>Mon 26/06/07</td>
<td>Fri 30/06/07</td>
<td>Ahmed Waqas(50)</td>
<td></td>
</tr>
<tr>
<td>Requirements Analysis</td>
<td>3 days</td>
<td>Mon 04/07/07</td>
<td>Wed 20/06/07</td>
<td>Ahmed Waqas(75)</td>
<td></td>
</tr>
<tr>
<td>System Requirements Specification (SRS)</td>
<td>13 days</td>
<td>Thu 27/06/07</td>
<td>Mon 25/06/07</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Functional Requirements</td>
<td>5 days</td>
<td>Thu 07/06/07</td>
<td>Wed 13/06/07</td>
<td>Ahmed Waqas(50)</td>
<td></td>
</tr>
<tr>
<td>Non-functional Requirements</td>
<td>1 day</td>
<td>Thu 14/06/07</td>
<td>Thu 14/06/07</td>
<td>Ahmed Waqas(75)</td>
<td></td>
</tr>
<tr>
<td>Define environment &amp; Platform selection</td>
<td>2 days</td>
<td>Fri 15/06/07</td>
<td>Mon 18/06/07</td>
<td>Ahmed Waqas(50)</td>
<td></td>
</tr>
<tr>
<td>Use Case</td>
<td>5 days</td>
<td>Tue 15/06/07</td>
<td>Mon 25/06/07</td>
<td>Ahmed Waqas(25)</td>
<td></td>
</tr>
<tr>
<td>Software Design</td>
<td>11 days</td>
<td>Tue 26/06/07</td>
<td>Tue 14/07/07</td>
<td></td>
<td></td>
</tr>
<tr>
<td>High Level Design</td>
<td>3 days</td>
<td>Tue 26/06/07</td>
<td>Thu 28/06/07</td>
<td>Ahmed Waqas(50)</td>
<td></td>
</tr>
<tr>
<td>Software architecture design</td>
<td>6 days</td>
<td>Fri 25/06/07</td>
<td>Fri 08/07/07</td>
<td>Ahmed Waqas(50)</td>
<td></td>
</tr>
<tr>
<td>Quality attributes review</td>
<td>1 day</td>
<td>Mon 05/07/07</td>
<td>Mon 05/07/07</td>
<td>Ahmed Waqas(75)</td>
<td></td>
</tr>
<tr>
<td>Update SRS &amp; Project Plan</td>
<td>1 day</td>
<td>Tue 10/07/07</td>
<td>Tue 10/07/07</td>
<td>Ahmed Waqas(50)</td>
<td></td>
</tr>
<tr>
<td>Support for system evolution</td>
<td>1 day</td>
<td>Tue 26/06/07</td>
<td>Tue 26/06/07</td>
<td>Ahmed Waqas(50)</td>
<td></td>
</tr>
<tr>
<td>Development</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Convert the deliverables of design phase into a working application</td>
<td>69 days</td>
<td>Wed 11/07/07</td>
<td>Fri 06/08/07</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Development of meta model support</td>
<td>5 days</td>
<td>Wed 11/07/07</td>
<td>Thu 17/07/07</td>
<td></td>
<td></td>
</tr>
<tr>
<td>development of graphical user interface interaction (with model)</td>
<td>10 days</td>
<td>Wed 19/07/07</td>
<td>Tue 18/07/07</td>
<td></td>
<td></td>
</tr>
<tr>
<td>development of and primitive module</td>
<td>15 days</td>
<td>Wed 01/08/07</td>
<td>Tue 31/07/07</td>
<td></td>
<td></td>
</tr>
<tr>
<td>development of validation module</td>
<td>10 days</td>
<td>Wed 22/08/07</td>
<td>Tue 04/08/07</td>
<td></td>
<td></td>
</tr>
<tr>
<td>development of code generation module</td>
<td>10 days</td>
<td>Wed 05/09/07</td>
<td>Tue 16/09/07</td>
<td></td>
<td></td>
</tr>
<tr>
<td>usage and extension documentation</td>
<td>10 days</td>
<td>Wed 15/09/07</td>
<td>Tue 02/10/07</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Update SRS &amp; Project Plan</td>
<td>3 days</td>
<td>Wed 03/10/07</td>
<td>Fri 05/10/07</td>
<td>Ahmed Waqas(50)</td>
<td></td>
</tr>
<tr>
<td>Testing</td>
<td>16 days</td>
<td>Mon 08/10/07</td>
<td>Fri 19/10/07</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scenario execution to verify the functionality and remove defects</td>
<td>9 days</td>
<td>Mon 08/10/07</td>
<td>Wed 17/10/07</td>
<td>Ahmed Waqas(50)</td>
<td></td>
</tr>
<tr>
<td>Update SRS &amp; Project Plan</td>
<td>2 days</td>
<td>Thu 16/10/07</td>
<td>Fri 18/10/07</td>
<td>Ahmed Waqas(50)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 53: Schedule

### Delivery Overview

<table>
<thead>
<tr>
<th>Deliverable</th>
<th>Type</th>
<th>Date</th>
</tr>
</thead>
</table>

98
Table 12: Delivery Overview

<table>
<thead>
<tr>
<th>Iteration 1</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Requirements Specification</td>
<td>Document</td>
</tr>
<tr>
<td>Software Design</td>
<td>Document</td>
</tr>
<tr>
<td>Development – Model implementation</td>
<td>Code</td>
</tr>
<tr>
<td>Development – Graphical Modelling Framework</td>
<td>Code</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iteration 2</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRS</td>
<td>Document</td>
</tr>
<tr>
<td>SD</td>
<td>Document</td>
</tr>
<tr>
<td>Development – Model implementation</td>
<td>Code</td>
</tr>
<tr>
<td>Development – Graphical Modelling Framework</td>
<td>Code</td>
</tr>
<tr>
<td>Development – Primitive module</td>
<td>Code</td>
</tr>
<tr>
<td>Development – Validation</td>
<td>Code</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iteration 3</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Requirements Specification</td>
<td>Document</td>
</tr>
<tr>
<td>Software Design</td>
<td>Document</td>
</tr>
<tr>
<td>Development – Primitive module</td>
<td>Code</td>
</tr>
<tr>
<td>Development – Validation</td>
<td>Code</td>
</tr>
<tr>
<td>Documentation</td>
<td>Document</td>
</tr>
<tr>
<td>Test Results</td>
<td>Document</td>
</tr>
<tr>
<td>Thesis</td>
<td>Document</td>
</tr>
</tbody>
</table>

Table 13: Timeline Master Thesis

<table>
<thead>
<tr>
<th>Deliverable</th>
<th>Type</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>First draft problem statement</td>
<td>Document</td>
<td>16/06/07</td>
</tr>
<tr>
<td>PD description of UML + uml metamodel</td>
<td>Document</td>
<td>23/05/07</td>
</tr>
<tr>
<td>PD description of primitives</td>
<td>Document</td>
<td>30/05/07</td>
</tr>
<tr>
<td>PD description of Arch. Patterns</td>
<td>Document</td>
<td>05/05/07</td>
</tr>
</tbody>
</table>

Progress Tracking

Weekly meeting will be held with all project members.

Resources
Non human Development Resources

List of resources:
- Local development platform
- CVS server

Human resources

<table>
<thead>
<tr>
<th>Resource name</th>
<th>Role(s)</th>
<th>Allocation</th>
<th>Time frame</th>
<th>Holidays</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nick Kirtley</td>
<td>PL</td>
<td>90-100%</td>
<td>01/05/07-31/11/07</td>
<td>5 work days</td>
</tr>
<tr>
<td>Ahmad Waqas Kamal</td>
<td>PS</td>
<td>100%</td>
<td>01/05/07-01/06/07</td>
<td>10 work days</td>
</tr>
<tr>
<td>Ahmad Waqas Kamal</td>
<td>PS</td>
<td>20%</td>
<td>01/06/07-31/11/07</td>
<td>?</td>
</tr>
<tr>
<td>Paris Avgeriou</td>
<td>PS</td>
<td>1%</td>
<td>01/05/07-31/11/07</td>
<td>?</td>
</tr>
</tbody>
</table>

Table 14: Human resources

Architectural Primitives to Patterns Mapping

<table>
<thead>
<tr>
<th>Patterns</th>
<th>Callback</th>
<th>Indirection</th>
<th>Grouping</th>
<th>Layering</th>
<th>Aggregation</th>
<th>Cascade</th>
<th>Composition</th>
<th>Cascade</th>
<th>Shield</th>
<th>Typing</th>
<th>Virtual Connector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Observer</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MVC</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reactor</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Event</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interceptor</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Visitor</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Command</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Layers</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indirection Layer</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtual Machine</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interpreter</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adaptor</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Façade</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proxy</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Component Wrapper</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wrapper Facade</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Broker</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Layered System</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object System Layer</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Composite</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cascade</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Organization Hierarchy</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Message Redirector</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type Object</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Knowledge Level</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Requirements

The requirements have been split into areas of interest.

### General

<table>
<thead>
<tr>
<th>Requirement ID</th>
<th>Priority</th>
<th>Description</th>
</tr>
</thead>
</table>
| R-G-1          | High     | Support a variety of architectural primitives that need to be explicitly specified for graphical, syntactical, syntax and semantics support to model patterns in this tool.  

*Graphical support*

Provide menu or stencils support to select architectural primitives in the system.
**Syntax and Syntactical support**
Use existing UML rules to model patterns using architectural primitives e.g. association among components.

**Semantics support**
Use existing relationships among elements of UML meta-model to provide support for modelling of architectural primitives.

<table>
<thead>
<tr>
<th>Requirement ID</th>
<th>Priority</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-G-2</td>
<td>High</td>
<td>The tool should be flexible to accommodate future changes to the design and extension of functionality.</td>
</tr>
<tr>
<td>R-G-3</td>
<td>High</td>
<td>To ensure that a selected architectural primitive is meaningfully representative to all of the patterns associated with the architectural primitive.</td>
</tr>
<tr>
<td>R-G-4</td>
<td>Medium</td>
<td>User should be able to view the patterns associated with a specific architectural primitive.</td>
</tr>
<tr>
<td>R-G-5</td>
<td>High</td>
<td>User should be able to attach architecture primitives to one or more patterns in the system.</td>
</tr>
<tr>
<td>R-G-6</td>
<td>High</td>
<td>User should be able to delete primitives visible in the system.</td>
</tr>
<tr>
<td>R-G-7</td>
<td>High</td>
<td>Customizable interfaces and ports specific to architectural primitives should be specified in the system.</td>
</tr>
<tr>
<td>R-G-7</td>
<td>High</td>
<td>User should be allowed to extend existing architectural primitives to introduce them as new architectural primitives in the system. However, the base architectural primitive should not be affected.</td>
</tr>
</tbody>
</table>

**Model and Primitives**

<table>
<thead>
<tr>
<th>Requirement ID</th>
<th>Priority</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-MP-1</td>
<td>High</td>
<td>The tool should be able to implement at least following architectural primitives: Callback, Indirection, Grouping, Layering, Aggregation cascade, Composition cascade, Shield, Typing and Virtual connector. The implementation of these primitives is scheduled in three parallel iterations:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Iteration 1:</em> Callback</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Iteration 2:</em> Indirection, Grouping, Layering, Aggregation Cascade, Composition Cascade</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Iteration 3:</em> Shield, Typing, Virtual Connector</td>
</tr>
<tr>
<td>R-MP-2</td>
<td>High</td>
<td>UML meta-model will be used to model architectural patterns and architectural primitives.</td>
</tr>
</tbody>
</table>

**Validation**

<table>
<thead>
<tr>
<th>Requirement ID</th>
<th>Priority</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-V-1</td>
<td>High</td>
<td>Standard UML language rules should not be violated in modelling patterns.</td>
</tr>
<tr>
<td>R-V-2</td>
<td>High</td>
<td>Tool should provide support to validate the use of architectural primitives in the system.</td>
</tr>
<tr>
<td>R-V-3</td>
<td>High</td>
<td>UML2 modelling constraints should be available in the tool.</td>
</tr>
<tr>
<td>R-V-4</td>
<td>High</td>
<td>Constraints to architectural primitives should be specified both implicitly and explicitly i.e. standard constraints for architectural primitives should be available in system as well as user should be allowed to define new constraints.</td>
</tr>
</tbody>
</table>

**User Interface**

<table>
<thead>
<tr>
<th>Requirement ID</th>
<th>Priority</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-UI-1</td>
<td>High</td>
<td>All components and connectors should support annotations. The annotation should not only cover textual description of connections but should also define the relationship type among components i.e. multiplicity etc.</td>
</tr>
<tr>
<td>R-UI-2</td>
<td>High</td>
<td>The tool should allow easy and flexible addition of new architectural primitives. For this purpose</td>
</tr>
</tbody>
</table>
wizard support should be provided in the system where user can define new primitives in the system. This will require research as to how feasible this requirement is.

R-UI-3 High A group of check boxes will be provided in the wizard where user can select among a generalized list of constraints for new primitive.

R-UI-4 High Graphical icons should be provided for each standard architectural primitive.

R-UI-5 High User should be able to assign (new) graphical icons to primitives

R-UI-6 High Tool should provide at least following graphical views to the user (in case some existing tool is extended, some of these features will already be available)
- Navigator
- Editor
- Properties
- Palette (for graphical icons)

R-UI-7 Medium User should be able to view all the primitives belonging to a selected pattern.

R-UI-8 High Mouse drag and drop functionality should be available on graphical editor for architectural primitives icons.

R-UI-9 Medium Important functions to be performed on architectural primitives should be available on right mouse click menu.

R-UI-10 High Dragging or moving position of a component in the tool editor should not affect the connections established by the component.

R-UI-11 High A wizard should appear when the user adds a primitive to the model.

**Table 20 : User Interface Requirements**

**Documentation**

| DOC1 | SH | Usage of tool should be described in an easy to understand document:
Documentation should describe how to add primitives
Documentation should describe how to develop a model with the GUI
Documentation should describe the metamodel used
Documentation should describe code generation options |
| DOC2 | SH | Inner working of tool should be described in full:
Documentation should describe existing structure of program
Documentation should describe what each method does
Documentation should describe how to extend the program |
| DOC3 | SH | Considerations for likely extensions to system should be described fully. |
| DOC4 | SH | Explanation of how existing primitives work and how they should be used. |

**Table 21 : Documentation Requirements**

**Key Drivers**

It is important to define key drivers so that focus can be given to these factors during the development of the software system.

Integrability – The project consists of using ideas that have recently been researched and incorporating these ideas into a working software system. Multiple concepts are needed and thus need to be integrated into the software system as a whole. The concepts include: Architectural Primitives, Unified Modelling Language (UML), UML metamodel, Object Constraint Language (OCL), Eclipse Modelling Framework (EMF), Graphical Modelling Framework (GMF) and Architectural Patterns.

Extensibility – The general goal of the project is to develop a software system with basic functionality regarding modelling of architectures using architectural primitives. Future improvements and extensions to the software system are likely and thus the (ease of) extensibility of the software system is an important issue.
Usability – The aim of this software system is to aid architects in the development of software architectures. The software system should therefore be intuitive and easy to use. The usability should also be easily extensible to enable future improvements and to adhere to the extensibility key driver.

Analysis of Existing Modelling Tools

See primitives paragraph
Appendix B

NOTE: This information came from before the actual implementation and is out of date.

Introduction
This document describes the high level design of the SADT plugin.

Module Diagram

Figure 1 shows an abstract view of the modules involved in the SADT plugin. UML2-OCT, UML2 and UML2Tools are existing plugins that are a part of the MDT-UML2 project. In no way can these plugins be altered to ensure that future versions of UML2 can be used without alteration to their code. The SADT plugin consists of three main modules – Validator, Main Module and UI.

The UML2 plugin provides the metamodel and modelling capabilities. This can be used to extend the metamodel and to define models for architectural modelling.

The UML2-OCL plugin defines API’s for parsing and evaluating OCL constraints and queries on EMF models.

The UML2Tools plugin provides the basic tools needed to graphically edit the models. It is essentially a GUI.
The main module provides the usage of the metamodel from the UML2 plugin. This will be the basis for the soft extensions that will describe the primitives. Standard profiles (UML extensions) can be loaded to provide for a basic set of primitives. The functionality to easily add new primitives will also be provided within this module.

The validator module will provide the functionality needed to validate the models. This will include the extra validation rules defined for the extra functionality provided by the primitives.

The Graphics extension/UI will provide the extra graphics icons and extra functionality within an easy to use interface. Primitives can be added during usage and the Graphics extension should also be able to easily add new primitive icons.

**General Interaction Diagram**

Figure 2 shows the line of communication between the modules.

1. User event -> adapt model -> adaptation is made but not final
2. Inform validator to check new model
3. Validation made -> keep change or revert model to previous state
4. Add change to UI or show error

![Figure 55: Module communication](image)

**Components Description**

Figure 3 shows the data models and general functionality of each module described in the preceding paragraphs.
Validation

Data:
Rules/Constraints -> datamodel

Functionality:
Validate

Main

Data:
Metamodel -> can be extended using existing functionality
Model(s) -> Existing model components + inclusion of new primitives (Extend existing datamodel)
Primitives -> datamodel (new)

Functionality:
Add primitive -> Wizard + Creates new elements in primitive datamodel

GUI

Data:
Primitives to Graphics (icons) -> datamodel (new)
Palettes / action buttons

Functionality:
Adding/Connecting components

Figure 56: Components Description

Proposed Layout/Screenshot

Figure 4 shows the proposed layout of the SADT plugin. The highlighted parts show the extra functionality compared to the existing UML2Tools plugin. There will be an extended palette for the new primitives and an extra menu bar or right mouse button menu bar that will allow for primitive specific operations.

Figure 57: Screenshot + proposed add-ons

Questions and Answers
A key driver of the SADT project is integration. The key question is thus how we integrate the plugin with the existing plugins. We can ask questions regarding the various modules:

Question: How do we use the metamodel to add primitives?
Answer: UML2 already offers the ability to add stereotypes. We can therefore use the existing capability without further extending UML2. However, I’m still not sure about extending existing UML metaclasses.
Update: It is possible to extend existing metaclasses.

Question: How do we use the validation features offered by UML2-OCL and how does the OCL feature work; is there a validator or does it have an internal representation of constraints.
Answer: UML2 has a limited selection of constraints from the UML metamodel. They are implemented in Java and are called using a ‘validate’ function. A list of the implemented constraints is still needed. New constraints (constraints based on new features extended from UML metamodel) can be implemented in Java or OCL.

Question: The graphics extension / UI should be used in its existing state and should also be extended. The existing functionality and the extended functionality should work together. The difficulty here is how we can use the existing functionality and extended functionality while only extending UML2Tools and not altering UML2Tools in any way.
Answer: The difficulty lies in developing a plugin that uses existing functionality and adding to that functionality without changing the code form UML2Tools. This is an Eclipse Plugin development issue.
Appendix C

Introduction

Identification and Purpose
This document describes the detailed design for the SEARCH Architectural Design Tool (SADT). It is a follow up of the High Level Design.

Scope

The detailed design describes the design of the plugin and to a limited extent the way that it will interact with the existing plugins that will be extended.

System Design Overview

Modules

There will be three modules – main module, validation module and graphics module, as described in the high level design.

System Requirements

Software – Latest Eclipse version with the following plugins: UML2-OCL, UML2, UML2Tools. The plugins mentioned also have plugin dependencies and the list can be on the eclipse.org website in the downloads section of the above mentioned plugins.

Hardware – A computer that can run the latest Eclipse version.

Application Design Detail

UML Layers
Figure 58: UML Layers

Figure 1 shows the various layers of the UML structure. There is also an M3 layer but this is not relevant to our project. M1 is the modelling layer of our system and M2 is the layer used to add stereotypes and thus primitives to the program. M0 is the layer used to for runtime of the M1 layer, which can be translated into runtime of the modelled system.

Use Cases
Figure 1 shows the use cases of the SADT Plugin. The second tier use cases show the options of the first tier use cases.

Overview

Figure 2 shows a more detailed component diagram. The validate module consists of two components, one for the validation of M1 level constraints and one for the validation of M2 level constraints. This allows for separation of constraint rules.
The main module consists of 3 components. The metamodel component deals with any issues regarding the metamodel. This will consist of any operations on the model and not storing or manipulating the model directly because this is already done in the UML2 plugin. The model component deals with anything regarding the model of the metamodel, once again, the actually model and model manipulation is done in the UML2 plugin. The PrimitivesManipulation component is the component that manipulates the primitives. It does not store the primitives as such because that is done within UML2. The PrimitivesManipulation does contain hash information that links primitives to the stereotypes that have been added. This way there is a map between a primitive and its stereotypes and vice versa.

The GE/UI module consists of one component. The UI component deals with user input and displaying graphics associated with the primitives. The UI component consists of a PrimitivesToIcons component/data structure: this deals with the graphical representations of the primitives.

![Figure 60: Detailed Component Diagram](image)

Class Diagram
Figure 3 shows the class diagram. This is a first draft of the class diagram and is likely to change as more is known about the implementation of the existing plugins (UML2, UML2-OCL and UML2Tools). All connections between classes are associations between each other. The datatypes represent datastructures that will contain relevant information.

Interaction with Existing Framework
Figure 5 shows the interaction and extension of SADT and the eclipse framework. SADT will extend UML2Tools and use UML2 and UML2-OCL and UML2Tools uses UML2 and UML2-OCL.
Appendix D

Updated Design

Requirements

The requirements presented are updated up to the point of the project ending. Requirements that were specified at the start of the project but were not met due to time constraints have been added in a future requirements table.

<table>
<thead>
<tr>
<th>Req ID</th>
<th>Priority</th>
<th>Description</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-G-1</td>
<td>High</td>
<td>Support a variety of architectural primitives that need to be explicitly specified for graphical, syntactical, syntax and semantics support to model patterns in this tool.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>i. Graphical support</td>
<td>Provide menu support to select architectural primitives in the system.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ii. Syntax and Syntactical support</td>
<td>Use existing UML rules to model patterns using architectural primitives e.g. association among components.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>iii. Semantics support</td>
<td>Use existing relationships among elements of UML meta-model to provide support for modelling of architectural primitives.</td>
</tr>
<tr>
<td>R-G-2</td>
<td>High</td>
<td>The tool should be flexible to accommodate future changes to the design and extension of functionality.</td>
<td></td>
</tr>
<tr>
<td>R-G-3</td>
<td>High</td>
<td>User should be able to delete primitives visible in the system.</td>
<td></td>
</tr>
<tr>
<td>R-G-4</td>
<td>High</td>
<td>Customizable interfaces and ports specific to architectural primitives should be specified in the system.</td>
<td></td>
</tr>
<tr>
<td>R-G-5</td>
<td>High</td>
<td>Add and extend primitives to the model (M1 Level).</td>
<td></td>
</tr>
</tbody>
</table>

Table 22: General Requirements

<table>
<thead>
<tr>
<th>Req ID</th>
<th>Priority</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR-G-1</td>
<td>High</td>
<td>To ensure that a selected architectural primitive is meaningfully representative to all of the patterns associated with the architectural primitive.</td>
</tr>
<tr>
<td>FR-G-2</td>
<td>Medium</td>
<td>User should be able to view the patterns associated with a specific architectural primitive.</td>
</tr>
<tr>
<td>FR-G-3</td>
<td>High</td>
<td>User should be able to attach architecture primitives to one or more patterns in the system.</td>
</tr>
<tr>
<td>FR-G-4</td>
<td>High</td>
<td>User should be allowed to extend existing architectural primitives to introduce them as new architectural primitives in the system. However, the base architectural primitive should not be affected.</td>
</tr>
</tbody>
</table>

Table 23: Future General Requirements
Model and Primitives

<table>
<thead>
<tr>
<th>Req ID</th>
<th>Priority</th>
<th>Description</th>
</tr>
</thead>
</table>
| R-MP-1 | High     | The tool should be able to implement at least following architectural primitives: Callback, Indirection, Grouping, Layering, Aggregation cascade, Composition cascade, Shield, Typing and Virtual connector. The implementation of these primitives is scheduled in three parallel iterations:  
  - **Iteration 1**: Callback  
  - **Iteration 2**: Indirection, Grouping, Layering, Aggregation Cascade, Composition Cascade  
  - **Iteration 3**: Shield, Typing, Virtual Connector | Passed |
| R-MP-2 | High     | UML meta-model will be used to model and architectural primitives. | Passed |
| R-MP-2 | High     | User should be able to apply primitive specific profile to the model. | Passed |

Table 24: Model and Primitives Requirements

Validation

<table>
<thead>
<tr>
<th>Req ID</th>
<th>Priority</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-V-1</td>
<td>High</td>
<td>Standard UML language rules should not be violated in modelling patterns.</td>
</tr>
<tr>
<td>R-V-2</td>
<td>High</td>
<td>Tool should provide support to validate the architectural primitives modelled in the design.</td>
</tr>
<tr>
<td>R-V-4</td>
<td>High</td>
<td>Constraints to architectural primitives should be specified both implicitly and explicitly i.e. standard constraints for architectural primitives should be available in system as well as user should be allowed to define new constraints.</td>
</tr>
<tr>
<td>R-V-5</td>
<td>High</td>
<td>Validation should also inform user of any problems and highlight the exact problem with primitives</td>
</tr>
</tbody>
</table>

Table 26: Validation Requirements

User Interface

<table>
<thead>
<tr>
<th>Req ID</th>
<th>Priority</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-UI-1</td>
<td>High</td>
<td>All components and connectors should support annotations. The annotation should not only</td>
</tr>
</tbody>
</table>

Table 27: Future Validation Requirements
cover textual description of connections but should also define the relationship type among components i.e. multiplicity etc.

**R-UI-4**  
High  
Graphical icons should be provided for each standard architectural primitive.  
Failed and not necessary

**R-UI-5**  
High  
User should be able to assign (new) graphical icons to primitives  
Failed and not necessary

**R-UI-6**  
High  
Tool should provide at least following graphical views to the user (in case some existing tool is extended, some of these features will already be available)  
i. Navigator  
ii. Editor  
iii. Properties  
v. Palette (for graphical icons)  
Passed  
Palette exists for standard UML items and not for primitives but that is not necessary

**R-UI-8**  
High  
Mouse drag and drop functionality should be available on graphical editor for architectural primitives icons.  
Passed for UML elements.  
Not possible to move complete primitive but that is not practical.  
Primitive can be added to a specific location by way of assigning it to specific components/packages

**R-UI-9**  
Medium  
Important functions to be performed on architectural primitives should be available on right mouse click menu.  
Passed, all UI events are triggered by the context menu

**R-UI-10**  
High  
Dragging or moving position of a component in the tool editor should not affect the connections established by the component.  
Passed

**R-UI-11**  
High  
A wizard should appear when the user adds a primitive to the model.  
A wizard appears when the user requires it

**R-UI-12**  
High  
User should be able to add a primitive using a step-by-step self explanatory method  
Passed

**R-UI-13**  
High  
User should be able to apply primitives to existing UML elements  
Passed

**R-UI-14**  
High  
User should be able to use single UML elements in multiple primitives.  
Passed

**R-UI-15**  
High  
The user should be able to find primitives applied to the model  
Passed

<table>
<thead>
<tr>
<th>Req ID</th>
<th>Priority</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR-UI-2</td>
<td>High</td>
<td>The tool should allow easy and flexible addition of new architectural primitives. For this purpose wizard support should be provided in the system where user can define new primitives in the system. This will require research as to how feasible this requirement is.</td>
</tr>
<tr>
<td>FR-UI-3</td>
<td>High</td>
<td>A group of check boxes will be provided in the wizard where user can select among a generalized list of constraints for new primitive.</td>
</tr>
<tr>
<td>FR-UI-7</td>
<td>Medium</td>
<td>User should be able to view all the primitives belonging to a selected pattern.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req ID</th>
<th>Priority</th>
<th>Description</th>
</tr>
</thead>
</table>
| DOC1 | SH | Usage of tool should be described in an easy to understand document:  
i. Documentation should describe how to add primitives  
ii. Documentation should describe how to |

Passed, except for code generation options because there are none.
iii. Documentation should describe the metamodel used
iv. Documentation should describe code generation options

| DOC2 | SH | Inner working of tool should be described in full: i. Documentation should describe existing structure of program ii. Documentation should describe what each method does iii. Documentation should describe how to extend the program | Passed |
| DOC3 | SH | Considerations for likely extensions to system should be described fully. | Partially |
| DOC4 | SH | Explanation of how existing primitives work and how they should be used. | Passed |

**Table 30: Documentation Requirements**

<table>
<thead>
<tr>
<th>Req ID</th>
<th>Priority</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 31: Future Documentation Requirements**

**Class Diagram**

The Class diagram shows the structure of the Primus tool. Use of the class diagram instead of a module/component diagram makes sense because the tool does not have any large modules and can best be described using the classes used. The components that have been added show the interaction (Or rather usage of) with the plug-ins from MDT.
Use Case

The use case diagram shows the possible interaction of the user with the tool.
Sequence Diagram
The sequence diagrams show some interaction issues and how each class is called during that process.
The sequence diagram situations are quite simple because they are all event driven.

Figure 2: Use Case Diagram

Figure 3: Sequence Diagram – Create Primitive
Figure 4: Sequence Diagram - Validate Primitives

User interaction

The screenshot shows how the user can interact with the tool.

Figure 5: Screenshot Primus
Appendix E

Developer and User Guide

To use Primus you can copy the Eclipse directory that has Primus implemented and make alterations if you wish. To install from scratch follow the steps below. This is useful to know because newer versions of Eclipse may a ‘reinstall’ of Primus. Understanding how Primus was made will also help further development of Primus by other developers.

Installing Eclipse and Required Plug-ins

Go to the Eclipse website. Here all versions of Eclipse can be found. If you want a stable version of Eclipse then go to the downloads section and select Eclipse classic. If you want a new version then on the left hand side there should be a newer version that is still under development. For example, the newest version at the time of writing this document is Ganymede M4.

Setting up the MDT environment

The first step in installing the Primus tool is to make sure that the MDT environment is installed properly. There are three possibilities:

1)

Go to the MDT page (http://www.eclipse.org/mdt) and download the plug-in files. Extract them into the plug-ins directory of your Eclipse install.
List of plugins needed: UML2, OCL, and UML2Tools.
The plug-ins listed requires other plug-ins like EMF and GMF. The list can be found in the download page of the relevant plug-in.
For example, the UML2 plug-in requires: Eclipse and EMF.

2)

In Eclipse there is an update manager that can download and install the necessary files for you. Go to Help and select Software Updates and then Find and Install. From there you can choose modelling tools. Be sure to click the button ‘select required’: this will also install the required plug-ins of the selected plug-ins.
If you want to alter the appearance of the component diagram (as is the case with the current implementation of Primus) then you should install the source code version of UML2Tools. The other plug-ins should be installed using step 1) or 2).

To install source code version of UML2Tools:

Go to the CVS Repository Exploring perspective (This can be found in the top right corner of Eclipse, See Figure 1).

Add a CVS location: location: cvsroot/modelling user: anonymous pw: *blank
Open the location, select uml2tools, open it so that all sub-directories are visible, select all sub-directories, right click and select check out.

The UML2Tools plug-in has now been installed in your workspace and you can alter the files to adapt the component diagram.

A Look at Primus

Primus is a fragment plug-in that extends the component diagram plug-in of the UML2Tools plug-in.

To create a fragment plug-in:

File, select new, select plug-in development and specifically fragment plug-in. Fill in the wizard and select the plug-in that you wish to extend.

When the plug-in has been created there should be a Manifest file open. In the manifest file all the plug-in specific details are visible. The manifest file of Primus is in the Meta-inf directory and is called manifest.mf. This file is actually an extension of the component diagram manifest file.

![Figure 66: Primus manifest file](image)

**Summary:**

So far we have installed Eclipse, installed the necessary plug-ins and created a fragment plug-in.
Interact with Component Diagram

To interact with the component diagram it is possible to add menu items to the context menu. The context menu is opened if the user presses the right mouse button on the component diagram.

The first step is to make an extension. An extension is an extension to an extension point that has been defined by developers of other plug-ins. This is how Eclipse is able to extend plug-ins in a modular way. To extend it you need to know what you want to do and which extension offers that functionality. In this case we want to add to the context menu and thus we require the contributionItemProviders extension.

To make an extension go to the extension tab of the manifest file and click add. Search for the relevant extension and click Finish. Then right click the item that has been added and select pop-up contribution. Add this line to the class field:

```
org.eclipse.gmf.runtime.diagram.ui.providers.DiagramContextMenuProvider
```

You can also copy this code from the Primus fragment.xml tab which has an XML definition of a context menu.

Context menu item:

```xml
<extension
    point="org.eclipse.gmf.runtime.common.ui.services.action.contributionItemProviders">
    <contributionItemProvider
        checkPluginLoaded="true"
        class="action.PrimusActionAddPrimitive">
        <Priority
            name="Medium">
        </Priority>
        <popupContribution
            class="org.eclipse.gmf.runtime.diagram.ui.providers.DiagramContextMenuProvider">
            <popupMenuGroup
                id="PrimusGroup"
                path="/propertiesGroup">
            </popupMenuGroup>
            <popupMenu
                id="primus_action_add_primitive"
                path="/PrimusGroup">
            </popupMenu>
            <popupStructuredContributionCriteria
                objectClass="org.eclipse.gmf.runtime.diagram.ui.editparts.IGraphicalEditPart"
                objectCount="1">
            </popupStructuredContributionCriteria>
        </popupContribution>
    </contributionItemProvider>
</extension>
```
This XML code will define a menu item. The class parameter will tell Eclipse to send the event to the PrimusActionAddPrimitive class. The id parameter should be unique and describe the action.

The code below shows the PrimusActionAddPrimitive class.

```java
package action;

public class PrimusActionAddPrimitive extends AbstractContributionItemProvider implements IProvider {

    public static final String MENU_PRIMUSACTION1 = "primus_action_add_primitive"; //$NON-NLS-1$

    public PrimusActionAddPrimitive() {
        // TODO Auto-generated constructor stub
    }

    public IMenuManager createMenuManager(String menuId, IWorkbenchPartDescriptor partDescriptor) {
        if (!MENU_PRIMUSACTION1.equals(menuId)) {
            return super.createMenuManager(menuId, partDescriptor);
        }

        MenuManager menuManager = new MenuManager("Add Primitive");
        MenuBuilder builder = new MenuBuilder(partDescriptor);
        //XXX: build initial content -- otherwise menu is never shown
        builder.buildMenu(menuManager);

        menuManager.addMenuListener(builder);
        return menuManager;
    }

    The id from the XML code is used in the java code. If the id matches the id (menuId) that has been passed in the createMenuManager method then it will carry out the necessary code which will create another sub-menu or carry out some other code.

Classes that deal with menus:

PrimusActionAddPrimitive
PrimutsActionAddPrimitiveToPackage
PrimusActionPrimitivesOfType
AddComponentToPackageAction
AddPrimitiveAction

Summary:
In this paragraph we have shown how to use an extension and how to adapt the context menu.

Make a Wizard

A wizard is used to create primitives. It can be handy to use if there are a lot of options because checks can be carried out to make sure that the user has selected all the necessary and valid options.

To make a wizard, make a class that extends the Wizard class. From this class pages should be made. These will constitute the pages that can appear during the life of the wizard. All choices made in the various pages of the wizard are stored in a model class. When the wizard is complete it will carry out the action (in the existing case it will make the primitive). The model will provide the parameters for the action.

```java
package action;

/**
 * Wizard class
 */

class AddPrimitiveWizard extends Wizard implements INewWizard{
    public static final String copyright = "(c) Copyright Nick Corporation 2002.";
    // wizard pages
    PrimitiveMainPage primitivemainPage;
    // more pages
    org.eclipse.uml2.uml.Package myPackage;
    boolean canFinish=false;
    // the model
    PrimitiveModel model;
    
    /**
     * Constructor for HolidayMainWizard.
     */
    public AddPrimitiveWizard(org.eclipse.uml2.uml.Package myPackage) {
        super();
        this.myPackage=myPackage;
        model = new PrimitiveModel();
    }

    public void addPages()
    {
        primitivemainPage = new PrimitiveMainPage();
        addPage(primitivemainPage);
    }

    public void addPages()
    {
        primitivemainPage = new PrimitiveMainPage();
        addPage(primitivemainPage);
    }
```
public boolean canFinish()
{
    return canFinish;
}

public boolean performFinish()
{
    CreatePrimitive createPrimitive = new CreatePrimitive(myPackage);
    if(model.primitiveChoice.equals("Callback")){
        createPrimitive.createCallback(model.callerNew, model.callbackNew, model.caller, model.callback);
    }
}

The code above shows the class used for creating a wizard in Primus. Notice primitive.mainPage page, this is a page that is part of the wizard. In the actual implementation there is a large list of pages needed to define a primitive (One page per primitive). The canFinish method returns canFinish Boolean which tells the last page in the wizard that the Finish button can be pressed. The canFinish Boolean should be set to true if all the necessary options have been selected by the user. The performFinish method performs that actual creation of the action. In the case of Primus this involves creating the primitive. This is when the information from the mode is needed. Notice that createPrimitive is called and the parameters come from the model.

Classes that deal with Wizards:

AddPrimitiveWizard
CallbackPage
ConnectorPage
CompositionCascadePage
GroupingPage
IndirectionPage
LayersPage
ShieldPage
TypingPage
VirtualConnectorPage

Adding UML Elements to the Model (MI Level)
First make sure that you have access to the model. This comes from the action class so should be passed along to the creation class. The following will list the commands used to create various UML elements and set various properties:

Create a component
The getUniqueName method can be found in the createPrimitive class. It was added to make sure that names are unique.

```java
Component comp1 = (Component) myPackage.createOwnedType(getUniqueName("Caller",1), UMLPackage.eINSTANCE.getComponent());
```

Create a port
Port p1 = (Port)
```java
comp1.createOwnedPort(getUniqueNameInComponent(comp1,"e",2), null);
```

Create a class
Class class1 =
```java
myPackage.createOwnedClass(getUniqueName("Caller_e",2), true);
```

Set the port type to the class and set the port to composite
```java
p1.setType(class1);
p1.setIsComposite(true);
```

Create an interface
Interface myInterface1 = (Interface)
```java
myPackage.createOwnedType(getUniqueName("ObserveEvent",3), UMLPackage.eINSTANCE.getInterface());
```

Create a realization to the interface (Provided relationship)
```java
InterfaceRealization newElement = UMLFactory.eINSTANCE.createInterfaceRealization();
newElement.setImplementingClassifier(class1);
newElement.setContract(myInterface1);
```

Create a usage to the interface (Required relationship)
```java
myPackage.getPackagedElements().add(newUsage);
newUsage.getClients().add(class1);
newUsage.getSuppliers().add(myInterface2);
```
Apply a stereotype

The `getApplicableStereotypes` retrieves the list of stereotypes that are applicable to the p2 object. In this case a port.

```java
EList<Stereotype> stereotypesPort2 = p2.getApplicableStereotypes();
for (Stereotype stereotype : stereotypesPort2) {
    if (stereotype.getName().equals("CallbackPort")) {
        p2.applyStereotype(stereotype);
    }
}
```

Classes that deal with creating primitives:

`CreatePrimitives`

**Using OCL**

OCL queries are used to constrain and find primitives in the model. The current implementation has a yes/no return value. It either does or does not return the location of a primitive. A future implementation should return what is wrong with a primitive, if indeed there is something wrong (See paragraph 8 of thesis).

To test OCL queries on a model in the Eclipse MDT framework use the OCL interpreter/query editor. This can be installed by adding a new project and selecting the OCL interpreter or by downloading the plug-in from the OCL page. It can be opened by selecting console and then interactive OCL.

![OCL interpreter](image)

**Figure 67: OCL interpreter**

To run OCL queries in code you must have a current element selected. This is selected when the right mouse button is clicked. If a UML element isn’t selected then it will use the package that is viewed. The UML element should be passed along to the relevant class that will run the OCL code. The code that will actually run the query will be given:

```java
public OCLCheckAll(String primitiveType, Element element, TransactionalEditingDomain domain) {
```
```java
myPrimitiveType = primitiveType;
myElement = element;

myOcl = org.eclipse.ocl.uml.OCL.newInstance(domain.getResourceSet());
//myOcl = org.eclipse.ocl.uml.OCL.newInstance();
oclHelper = myOcl.createOCLHelper();

oclHelper.setInstanceContext(element);
oclInv = null;
oclInvNear = null;

oclHelper.setInstanceContext(element);
```

The code above executes the query. The parameter primitiveType is the primitive type that will be queried. For example, a Callback query or an Indirection query.

The element parameter is used as the current element selected which directs the query execution to the model.

The actual execution occurs is achieved using the following command:

```java
oclInv = oclHelper.createQuery(myQuery);
```

The Primus project has two types of OCL queries. One of the query types is used to find and validate primitives (Proper primitive) and the other is used to find primitives that do not have to be validated (Near primitive). The code for the near primitive query is very similar to the proper primitive query however, the ‘constraints’ are not included.

**How to use Primus**

In the ‘how to use Primus’ section we will develop an architecture design that consists of a few primitives. Note that this only shows the process of adding primitives and not making a useful architecture design.
We will start off with a blank project by selecting Menu->New->Component diagram.

The first step is to load the relevant resources which in this case are all the primitive profiles that we are going to use: Layers, Callback, Indirection, Aggregation Cascade and Shield.

Figure 5 shows how to load a resource. Right click the Package and select Load resource.

The next step is to apply the profile to the package (model) that we are going to work with. Figure 6 shows how this is done: Select package, select UML Editor and apply profile to package.
Figure 69: Apply profile to package

Now we can add a primitive to the model. Let’s start with applying a Callback primitive. Right click the component diagram and select Add Primitive->Wizard. This will open the wizard and from there the primitive will be made.

Figure 70: Select Primitive

Figure 71: Select Components
Figure 7 shows the first page of the wizard. The first page selects the primitive that will be created. A connector can also be made between two components using the connector option. Figure 8 shows the second page when Callback is chosen. The first drop down menu determines if the first component (The caller component) should be created or if an existing component should be used. If an existing component is used then the list of components should be used to select which component should be used. The same applies for the second component (The callback component).

![Diagram](image1)

**Figure 72: Callback detailed visualisation**

**Figure 73: Callback basic visualisation**

Figure 9 shows the resulting Callback. Notice that the visualisation is detailed. This can be helpful if only one primitive is shown but can become cluttered if many primitives are viewed in a typical architectural design (Such as Leela). To solve the problem of too much detail we can turn off visual types. For example, the classes are not very important to us as they are only used in conjunction with connectors. It is possible to turn off the visualisation for classes by right clicking a class, select filter and then hide visualtype. It may be of interest to only hide certain elements. For this scenario only hide selection. This way classes that are of interest to the modeller can still be shown. Figure 10 shows a basic view of Callback.

For some reason the menu is also adapted when filters are applied (Filters is the process of hiding visual types as was done above). So to add more primitives we need to turn off the filters. This can be achieved by using an icon in the top tool bar as highlighted by Figure 11.
Now we can add a shield to the model. We will use the Callback component already created as the client of the Shield and create new components for the other shield elements.

![Figure 75: Shield wizard page](image)

Figure 12 shows the options that we need to create a shield using an existing client component and new components for the rest of the shield elements. Notice that the protected component has an ‘Add Protected Component’ option. The protected component will not be created if it is set to ‘No’.

The new group package acts the same as the other options. The only difference is that this will list packages instead of components.
Figure 76: Callback and Shield

Figure 13 shows a Callback and Shield applied. Notice that shieldGroup1 has a link between itself and ProtectedShield1 and AccessPoint1. This indicates that they are a member of that group.

Adding other types of primitives is basically the same as has been shown so we will not discuss any more primitives.

To validate and find primitives right click the component diagram. Select the Validate Primitives of Type option and then select the primitive that you wish to validate/find. Primus will show a pop-up with the results.

Figure 77: Callback validation

Figure 14 shows the results of finding validating/finding the Callback primitive.