# Part Time Modular Course : MSc Project

**Constraint Diagram Editor**

## Brighton University

**Part Time Modular Course : MSc Project**

Version 1.0

Author : R.P. Clark - 2003

Robin Clark
68 Vale Avenue,
Brighton,
East Sussex

# Contents

# List of Figures

# Chapter 1

# Guide to document

## 1.1 Abstract

**This MSc Dissertation describes a Java Application to Edit/Create Constraint Diagrams and to convert the Concrete Diagrams to Abstract Mathematical Models. These abstract models are available as XML and as Java runtime Objects.**

## 1.2 Quick Start : Running the Editor

In the root directory on the CD, run the cde.jar file.

## 1.3 Chapters and Software Life Cycle Documents

If this Application were to have been written in a commercial environment, several Documents would have been written corresponding to stages in a software development cycle (see figure 1.1). In this dissertation these are included as separate chapters, with added chapters relating to the work being an MSc submission.

These chapters follow the standard model for an object oriented products development life cycle, beginning with a reason/need for developing the product, a requirements specification, an object oriented design, an implementation proposal and then a document containing special algorithms and methods. These are followed by a Testing Document (which really should not have been written by the author of the software!). Finally the Deviation Description, a document containing all the changed or missing features from the specification.

I would particularly like people to read the chapter containing special algorithms and methods (see Chapter 7). The work in this chapter covers mathematical methods and algorithms to deal with shapes and areas to determine abstract mathematical information from the concrete diagrams. Use of recursive methods to determine enclosure relations, and definition of 'pure intersection' and 'enclosed intersection' relationships allow development of a fast zone discrimination algorithm (see 7.2.2.1).

For an overview of Object Oriented layout of the application Chapter 4 should be read. This describes the initial design process and develops the OO structures used. An index of all the chapters relating to software life cycle, with a short description is included below:

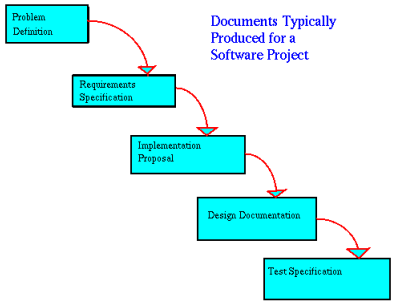



Figure 1.1: Documents Typically produced for a Software Project

- **Scope or Problem Description** A description of the Problem to be solved by the Application. see Chapter 2.
- **Requirements Specification** A requirements list for all features of the Editor, including a description of an Abstract UML model intended for use by third parties. Chapter 3.
- **Object Oriented Design**. From the requirements specification the principal classes and then the packages required to realise the Constraint editor. Chapter 4
- **Editor Behaviour and Implementation Issues** This focuses on how to tackle some of the decisions on how the editors human computer interface should work, with specific reference to building constraint diagrams. Chapter 5
- **Software Implementation** This chapter focuses on the practical details of the design. This should be of use to people extending the project or maintaining it. Chapter 6
- **Special Algorithms**. This chapter mainly focuses on algorithms developed to assist in converting concrete models to the abstract. Chapter 7
- **Future Possibilities**. As a result of developing and testing the software, a discussion on possible future projects or extensions. I have an interest in formal proving of safety critical systems, and recent European standards have made this field more complex. Chapter 8
- **Test Specification**. Tests that should be applied to the constraint editor, and descriptions of Test Harness code segments and their purpose. Chapter 9. This is really a bare outline of a full test specification. For an application such as this a 100 page test spec would be typical.
- **Deviation Description**. A list of variances or missing features in the product compared with the requirements. Chapter 10
- **Conclusion Chapter**. Comments on the project. Chapter 11

# Chapter 2

# The Problem of Accurate Software Specification

```
From a letter to the Daily Telegraph 15th Feb 2003

SIR -  Mathematicians are indeed the best at precise thinking.
A Philosopher, a Scientist and a Mathematician were travelling
by train on their first visit to Scotland.

The Philiosopher looked out of the window and
said: ‘‘Goodness me, Sheep in Scotland are black’’

The Scientist said: ‘‘No, some sheep in Scotland are black’’.
The Mathematician said: ‘‘No, in Scotland there is at least
one field in which there is at least one sheep that is black
on at least one side’’

Lord Grantley, London SW3.
```

## 2.1 Context of Mathematical Software Specification

Most software specifications are written in the style of the Philosopher or the Scientist. The writer knows what he/she means and *mostly* due to the context of the problem, so will the reader.

Typically a requirements specification is written in natural language, and from this the system design and subsequent coding are derived. Even when a UML model is developed, the behaviour of the system and its constraints are generally written in natural language. When using a natural language it can be very difficult to be precise. The take up of formal language is only widespread where it is forced as a project methodology [14]. It is therefore not a mainstream tool for system specification.

## 2.2 Common Faults in Natural Language Software Specifications

### 2.2.1 Ambiguity

Software specifications are generally littered with ambiguities. Often the implementor will guess the meaning in the context of problem for the development work in hand. This does not mean it will be clear

to any of the other users of the document such as the client or the writer of the test specification. In fact because it has been 'interpreted' by the developer, the likelihood of the others concurring completely becomes very remote.

For instance from a sentence from a hypothetical specification of an industrial gas burner control system

*"The Flame scanner is identified by a hexadecimal serial number 8 digits long. The device type is the first 2 hexadecimal digits. It should be displayed on the LCD at commissioning."*

Which 'it', does this refer to ? The 8 digit serial number or the 2 digit device identifier ?

On a more subtle level

Natural language can be very ambiguous. Take another hypothetical phrase ...

*"The Main controller and the flame scanner are both intelligent devices connected via Can Bus 2.0B. On the flame scanner failing it will switch off its fuel inlet valve"*

In a language like English with no reflexive personal pronoun, which *"its"* is this particular sentence referring to ? It could be the main controller (and is most likely to be) but we still cannot be sure.

### 2.2.2 Contradictions and Tautologies

For instance, in a large requirements document, statements scattered throughout it can contain requirements which are at variance with each other. If these requirements are written as mathematical statements they can be automatically checked. It may be found that some statements will erroneously lead to a condition always being true (a tautology) or it may be found that erroneous specification makes a condition both true and false (a contradiction). For instance scattered though-out a large document we may find statements like :-

i . The signals H and L should never be asserted together. If this happens the system state must go to SAFETY_LOCKOUT.

ii . SAFETY_LOCKOUT is a unique state.

iii . At state IGNITION, H and A asserted means skip ignition and purge the combustion chamber of inflammable gas with air.

iv . At Stage IGNITION, L being asserted means the hold switch has been activated during ignition and the system must wait until it is released.

Converting these extracts into Mathematics will lead us to a more precise picture of these requirements.

from statement $i$

$$SAFETY\_LOCKOUT = H \wedge L \tag{2.1}$$

From statement *ii*

$$SAFETY\_LOCKOUT \neq IGNITION \tag{2.2}$$

From statement *iii*

$$SAFETY\_SKIP\_IGNITION = A \wedge H \wedge (STATE = IGNITION) \tag{2.3}$$

From *iii* and *iv* with L asserted we must hold to current state

$$IGNITION = A \wedge H \wedge (STATE = IGNITION) \wedge L \tag{2.4}$$

Clearly from 2.1 the inputs given in 2.4 give SAFETY_LOCKOUT as TRUE as well

$$SAFETY\_LOCKOUT = A \wedge H \wedge (STATE = IGNITION) \wedge L \tag{2.5}$$

Equations 2.4 and 2.5 clearly contradict 2.2

### 2.2.3 Incompleteness

A natural language model of any system will often have some combinations of inputs, which are undefined as to meaning. Most combinations will perhaps only depend on one input, or in context only on germane inputs. These contextual meanings may well be understood by *most* who read the document.

For instance a system may have a state called SAFETY_SHUTDOWN, which is entered on combinations of inputs and states which are perceived to be dangerous.

The natural language description will have typically ignored many of these combinations. A mathematical description of the requirements will be able to highlight undefined combinations.

### 2.2.4 Mixed Levels of Abstraction Vagueness

Natural language is quite good at confusing system goals and high level abstract statements. For instance saying the system should react quickly to error conditions, is a general and high level goal. Stating that, for instance gas flow into a combustion chamber cannot continue for more than one second without a flame being detected is a specific goal, and could be of great importance to the system being designed. These will typically, be mixed together though-out a document, and are therefore easy to miss when compiling a test specification.

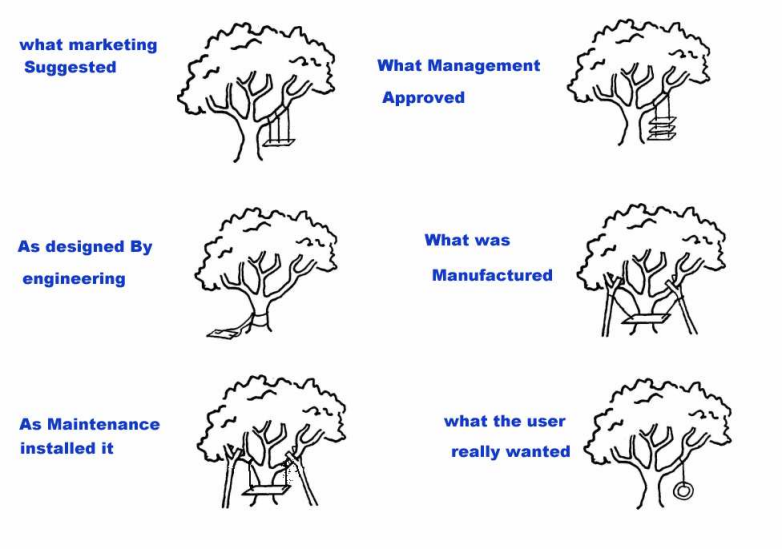



Figure 2.1: Tyre and Tree Specification

## 2.3 The Need for Mathematical Specification

To overcome the lack of precision in natural languages, mathematics using set theory and logic, can be used. The use of mathematics as an aid in software development and specification is generally known as "Formal Methods".

Formal Methods give the advantages of being able to automatically :-

- Prove there are no contradictions or tautologies in the specification
- Provide a firm basis for testing the system
- To accurately follow case studies through at the requirements stage
- Provide a discussion/clarity reference for natural language statements

Mathematical Software specification methods have been available since the 1970s but are rarely used. The OCL language which forms part of UML allows precise specification of systems, but still requires the user to learn the syntax is therefore generally unsuitable to show a client. Another commonly used formal language, 'Z', is even more unsuitable for showing to clients, being comprised of mathematical symbols only used in logic and set theory.

Poor Software and system behaviour definitions are endemic in the Software Engineering Industry. This has led to a 'cultural' feeling summed up by the popular tyre and tree specification in figure 2.1. Or, as seen more recently, "computer humour" Tee Shirts[7] with SQL statements on them such as :

```
$ select  * from  clients  where clue  >  0
$ 0 rows returned
```

One cannot imagine Architects or Electronic Engineers finding humorous the fact that what they had produced was not at all what was expected.

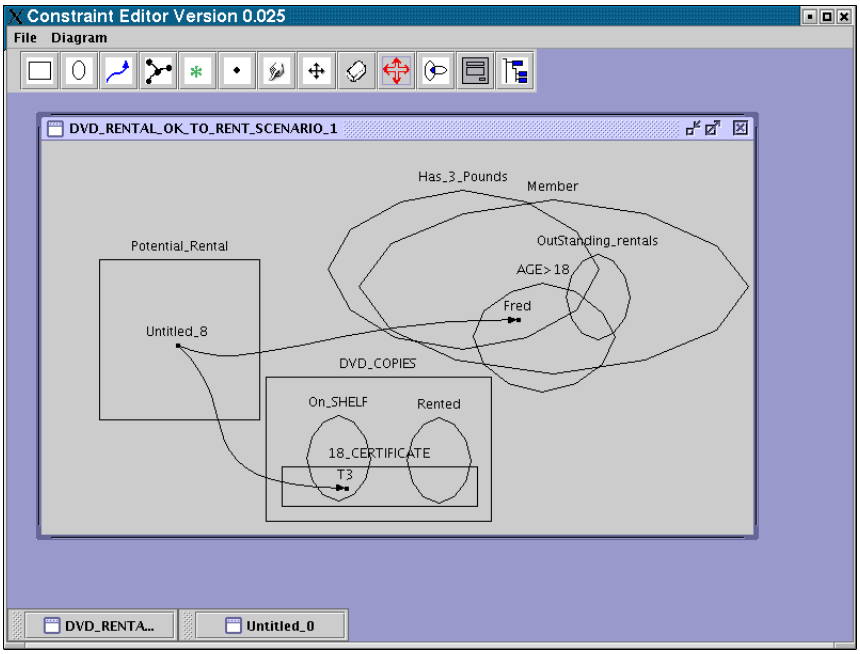



Figure 2.2: DVD Rental as constraint Diagram

### 2.3.1 Constraint Diagrams the Precision of OCL or Z but Understandable by Non Software Engineers

In order to form precise specifications, mathematics must be used to specify software. However the Formal Languages Z and OCL are both notations that require training and are likely to be only understood by Software Engineers anyway. Constraint Diagrams however, are graphical and intuitive. A client can follow the use cases and specifications, including sequences representing actions. A constraint diagram can then be converted into an OCL or Z formal specification, or an intermediate abstract model where mathematical checking for tautologies, contradictions and incompleteness can be applied.

For example the constraint diagram in figure 2.2 describes a situation where a member is allowed to rent a particular DVD from the hire shop. He is a member of the shop, has no outstanding rentals, has three pounds on him and the DVD he wishes to see, a copy of *T3*, is on the shelf. This is all evident in the diagram, and is easily understood by the client, the developer and the writer of a test specification.

However more than this can be extracted from the diagram. There are non-members of the video shop that may exist with 3 pounds on them, but they are not allowed to hire a DVD. While not explicitly stated this is very important ! Also there may be other copies of *T3* on the shelf, but the particular one that Fred has chosen he can rent. The film *T3* may have an 18 certificate, but this is OK as Fred is in the intersection of Members with $AGE > 18$.

The Rented area of $DVD\_COPIES$ is not shaded, therefore there could be other copies of T3 already rented. As the area of OutStandingRentals is not shaded, there could be other members currently with outstanding rentals, who would not be allowed to rent a DVD.

Note that the possible zone $NonMembers$ with $AGE > 18$ and $Has\_3\_Pounds$ is not on this diagram. It is not relevant to a potential hire. It could be relevant in scenarios such as $BecomingAMember$. Even though some things are not explicitly stated in the negative, they are included by definition in this diagram, and leave no ambiguities ! The Constraint Diagram presents a clear mathematical specification of a scenario within a system.

# Chapter 3

# Requirements

The overall rational of this project is to produce an editor which can create and save concrete constraint diagrams and also translate them into the abstract.

This Chapter describes the context and requirements for a Constraint Diagram Editor.

High level general requirements are:-

- Provide a drawing tool for Constraint Diagrams
- Save and restore concrete diagrams in XML
- Provide Conversion of Concrete diagrams to Abstract.
- Provide Abstract models as XML for third party processing.
- Provide facilities for exporting the diagrams in EPS.

## 3.1 Introduction

Tools currently exist to draw Constraint Diagrams which produce output easily included in documents and presentations. These diagrams however have no mathematical consistency i.e. there is no underlying mathematical model ; they are merely drawing aids.

A concrete diagram, is one that is drawn and can be understood and manipulated by the user. An abstract model can be derived from a well formed concrete constraint diagram.

The abstract model is concerned only with the mathematical relationships between the objects, and is not concerned with appearance or positioning on the screen.

This places a requirement on the concrete diagrams. These must maintain or be able to derive relationships between drawn objects in order to convert them to mathematical models.

The Abstract model defined as part of the now abandoned KMF project has been used as a template for the abstract output from this tool (see figure 3.1) [1] This model was to be of a form that can could have been translated and then passed to the (now abandoned) KMF meta modelling system.

This abstract model has been integrated into the ConstraintEditor software and the practical implementation can been seen in figure 6.1.

Notes:

[1] This diagram may be found as a Poseidon format *zargo* file on the CD in the main directory, called *meta_model.zargo*

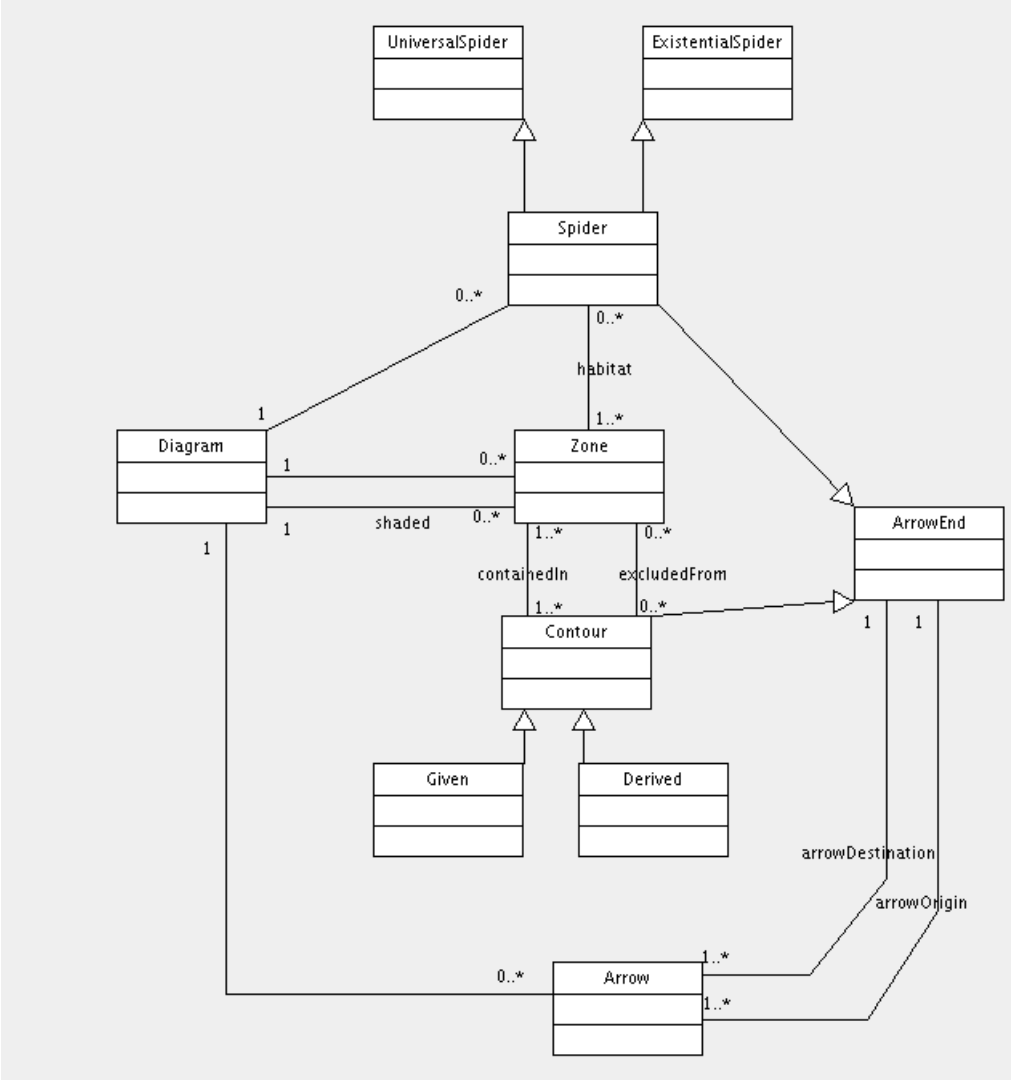



Figure 3.1: 'KMF' UML Mathematical Abstract Realisation of a Concrete Constraint Diagram

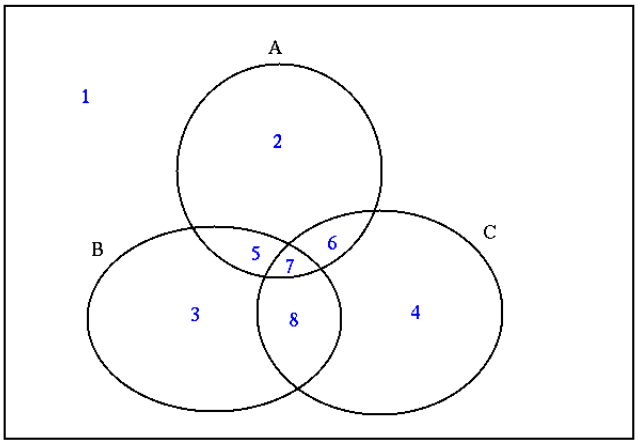



Figure 3.2: Possible Intersections in a Concrete diagram

## 3.2 Requirements derived from the Abstract UML Model

The KMF abstract model (see figure 3.1) is mathematically precise and terse. The associations and the count ranges define very precisely the model and impose constraints upon it. If a picture is worth a thousand words, then a good UML diagram should at least represent as many ! This UML diagram deserves discussion and explanation, as it was carefully peer reviewed and refined as part of the KMF project [23]. However a limitation of UML is that while very good at describing relationships between classes, it cannot easily represent constraints without using OCL. Three OCL constraints are defined to support the diagram

### 3.2.1 UML Abstract Model Walk though

Each diagram holds a collection of Zones, Spiders and Arrows. Spiders are themselves collections of Zones. Zones are comprised of two collections of Contours. Due to inheriting AbstractArrowEnd, Contours and Spiders are valid origin and destination points for Arrows.

### 3.2.2 Zone Definition

Note that for each set included in a concrete diagram (see figure 3.2 ) there are $2^n$ possible zones. Obviously most constraint diagrams will not need all possible combinations of sets in practice. Only those sets defined by intersection (and therefore visible on the diagram) need be recorded.

Any possible zones that are not included are zones not defined in the concrete diagram.

Note that all containment (or enclosure) relations ships can be inferred from a complete list of zones.

The algorithm converting from concrete sets/contours to abstract zones is described in 7.2.2.1.

Note that zones may be Given or derived. In this editor any contour or object which takes a default name beginning with *Untitled_* will be assumed to be “derived”. Any drawable object named differently will be assumed to be “given”.

| *Included* | *Excluded* | *Zone Number* |
|---|---|---|
| { } | { A B C } | 1 |
| { A } | { B C } | 2 |
| { B } | { A C } | 3 |
| { C } | { A B } | 4 |
| { A B } | { C } | 5 |
| { A C } | { B } | 6 |
| { A B C } | { } | 7 |
| { B C } | { A } | 8 |

### 3.2.3 Spider Definition

In meaning a spider simply states that the object may exist in of the zones that it has feet in (or its habitats). From the UML model, it not obvious that from an abstract mathematical point of view this is simply a collection of zones, with the added constraint that a spider can only contain either existential or universal points.

Here a limitation of UML has been highlighted. Because the Spiders are connecting to a base class, the UML, diagrammatically has no way of imposing the non-mixed spiders constraint. Also a Spider cannot have any of its feet in the same zone.

These will have to be specifically written in as OCL guards on the UML diagram.

```
post-condition mixed-spiders-check:
  self.habitat->forAll
      ( sn | sn.zone      instanceof ( EXISTENTIAL )  )
  or
  self.habitat->forAll
      ( sn | sn.zone      instanceof ( UNIVERSAL ) )


post-condition duplicate-zone-check:
     self.habitat->forAll
      ( sn | sn.zone
          self.habitat->forAll
            ( tz |  (tz != sn) or (tz = sn and tz.zone = sn.zone) )
      )
```

The “mixed-spiders-check” ensures that the spider solely consists of EXISTENTIAL, or solely of UNIVERSAL feet. Any spider with mixed feet will fail this post condition.

The “duplicate-zone-check” forces a cross product check of all zones that the spider exists in. If there are any duplicate zones, from different feet within the spider the post condition will fail.

### 3.2.4 Single existential or Universal Points

In an AbstractDiagram these are considered to be spiders with one foot. For a concrete diagram the existential denotes the presence of an object in the zone, and the universal a “for all” condition for the zone it lies in.

### 3.2.5 Contour definition

A contour in an abstract diagram is a set or type in a concrete diagram. In 3.2 A, B and C are abstract contours, and in the concrete sets.

### 3.2.6 Arrow Definition

An arrow denotes a relationship of some sort between drawable objects. This could be a mathematical map or an object oriented navigation. Since the relationship is unchanged in the concrete to abstract model the precise meaning of the arrow is defined by the methodology to which the diagram is being applied.

Note that an Arrow cannot have the same destination and origin. This means an OCL guard must be added to the diagram to represent this constraint.

```
post-condition circular-ref-check:
  self.origin != self.destination
```

## 3.3 Editor Requirements.

In general the editor should be easy to use and intuitive. Anyone understanding constraint diagrams should be able to start creating them quickly. This means using icons to represent the objects to be drawn, and a user friendly GUI.

### 3.3.1 Saving and Loading of Diagram Sequences

Diagram sequences must be saved and loaded back into the editor without information loss, and be readable by other applications. For ease of checking the output should be readable in a text editor or web browser. The XML language is appropriate for this. Saving and Loading must use a GUI for file and directory selection. For the Load functions a Standard XML parser SAX [22] should be used.

### 3.3.2 Saving Diagrams in Graphical Formats

Diagrams produced by the Constraint Editor must be easily included in presentations and documents. Screen grab utilities will of course work with a Java GUI but additional support should be supplied for convenience. The following output formats are proposed :-

#### 3.3.2.1 EPS - Encapsulated Postscript.

This is a standard printer language for representing text and graphics. It is easily included in most word processors and presentation tools [12].

As diagram sequences will be saved, and as diagrams are often included as separate files into documents etc, the diagram sequence will be saved as a zip file containing one compressed file entry per diagram.

#### 3.3.2.2 SVG - Scaled Vector Graphics

This is an XML based language for displaying 2D graphics in a web browser environment [15].

## 3.4 Drawable Objects

This section describes all objects that can be drawn and manipulated by the editor.

### 3.4.1 Object Types Supported

All objects that can be drawn must be named and also contain a comments field.

#### 3.4.1.1 Types

A Type is a form of Set, but usually from a more generic perspective. Represented as a large rectangle, and usually drawn with the intent of containing sub-Sets.

#### 3.4.1.2 Sets

Represented as a Polygon (12 sided), Sets must be able to be manipulated (i.e. changed in shape by mouse drags), and re-sized so as to allow any combination of zones.

#### 3.4.1.3 Derived Sets and Types

This is the set above but uses the default name (beginning with *Untitled_*). A derived set means that its mathematical definition is derived from other elements in the diagram. A non-derived set is a 'given' set.

#### 3.4.1.4 Shaded regions

Regions where sets and/or types intersect must be able to be shaded (indicating no objects are inherently present).

#### 3.4.1.5 Existential

Represented as a small circle/square. Existential points represent the existence of an object in the Diagram. They are also spider 'feet'.

#### 3.4.1.6 Universal

Represented as an asterisk. Universal Points represent the "ForAll" case in a Constraint Diagram. They are also Spider 'feet'.

#### 3.4.1.7 Arrows

Relationships (whether mathematical mappings or object oriented navigations) are represented as Arrows and may be formed between any combinations of Countours (Sets and Types) and Spider Feet (universal and existential points of collections thereof).

#### 3.4.1.8 Spiders

These connect two or more objects. Spiders may not connect to other spiders. Spiders are collections of two types, Universal or Existential. A universal or existential point on its own (i.e. not connected to any others by spider connections) is a singleton spider.

### 3.4.2 Manipulation Types Supported

Diagram elements where appropriate must be :-

- Deletable
- Re-sizable
- Manipulatable (i.e. change shape of polygon)
- Connectable (where mathematically appropriate)
- Movable.
- Attributes viewable in a dialog box.

## 3.5 Implementation Language

The Constraint Diagram Editor was to be able to interface with the KMF Tool which was written in Java. Being written in Java was therefore a requirement for this project.

However Java is a very good choice for the following reasons

- Java is a Modern Object Oriented Language in widespread use
- Java is a compile once and run on any platform language
- Java projects are easy to extend and add to

## 3.6 Interfacing

This tool will produce XML compliant output where appropriate. This will allow third parties to easily examine and use the output from this tool.

## 3.7 Deliverables

### 3.7.1 Complete Source Code and Compiling Instructions

All Source code, Makefile, java files lists and compiling instructions (flags used etc) must be included on a CD in ISO9660 format.

### 3.7.2 All Ancillary Files

All icon graphic files and any other support files must be included on the CD, including any scripts or other programs written to create the software product including All XML Document Type definition Files

### 3.7.3 Java Doc Web Pages

The code must be commented so that the "javadoc" tool can automatically produce web pages describing the classes and the class hierarchy.

### 3.7.4 Design Documentation

Design documentation, describing the key design decisions and explaining the rational behind the procedures and explaining any algorithms used, must be provided. This must include UML overviews and where appropriate formal descriptions of constraints. This will include a deviation description for the original requirements (which this document will lead to).

### 3.7.5 Interpretation of the Diagrams

The editor must be able to interpret the diagrams and convert them from the concrete 'drawn' format to an abstract 'mathematical' format. It should issue appropriate warnings and error messages at the conversion process, allowing the user freedom to draw and move concrete diagrams without being swamped with warning or error messages. This philosophy is similar to letting someone edit a source file, and then showing them the errors and warnings when it is submitted for compilation.

## 3.8 Desirable Features

### 3.8.1 OCL output

OCL representations of the diagrams would be useful for users wishing to cut and paste formal definitions into their UML diagrams.

### 3.8.2 Z Output

This could possibly be in latex format with EPS of the Diagrams, bundled into a zip file.

### 3.8.3 General Desirable Features

Output formats like PNG and GIF. Inclusion of background colours and being able to alter the standard colours of the objects as supplied.

## 3.9 Alternative Data View Possible Formats

### 3.9.1 Hierarchical Tree Display.

This could represent the diagram as a directory tree type structure.

Tree nodes would be represented by Set or Type objects. Any drawable objects enclosed by these would become a leaf node of its encloser.

The advantage of this would be that the items could be selected for manipulation from another viewpoint (i.e. not just on the diagram).

### 3.9.2 Text Description

From a diagram such as :-

A text description of the objects and their relationships. i.e.

```
% Diagram description
% Type(Integers) contains set(OddInts) and Set(EvenInts)
% Set(OddInts) has Relation(times2) to set(EvenInts)
```

## 3.10 Future Extensions - Possibilities - Ideas

### 3.10.1 Mathematical "Operations" Toolbar

Legal mathematical operations (such as the removal of OR conditions) could be offered with an ICON. The Diagram could then highlight all features that the operation could be applied to.

### 3.10.2 Automated Mathematical Reasoning

Operations such as representing some diagram elements in terms of the others in the simplest possible ways or all combinations of ways.

Automated searches for tautologies and contradictions.

### 3.10.3 Parsing of OCL or Z specifications into the Tool

Being able to read OCL/Z and then have a diagram displayed that can then be manipulated under mathematical guidance would be useful for including older specifications into the Tool. It could also aid the understanding of older specifications.

### 3.10.4 Display and Navigation in Three Dimensions

Where a diagram becomes complicated 2D might become too cluttered to follow a constraint diagram. In this case Z coordinates could be added to the objects and Polygons could become 3D shapes. The Java 3D API would support the drawing of these objects.

# Chapter 4

# Object Oriented Design

Given the Requirements Specification the next stage is to produce an Object Oriented Structure as a frame work for starting coding. This structure should not be seen as final but should be close enough for coding to begin without needing major design changes.

In this OO design phase I am employing techniques learnt on an Object Oriented Design Module at Brighton University. As part of the recommended reading, there was a book called "Designing Object Oriented Software" [1]. This described a simple methodology for beginning the OO design process. It started with taking all identifiable objects and making "Object Cards" for them. These cards could then be arranged on a table and moved and grouped with other objects. Where a written specification for the problem existed, significant nouns could be used to name potential classes.

Object cards could then be removed and added until a picture began to form for a working design. I have always found this to be a productive way of collating workable systems quickly. It is especially useful when presented with a complex problem and 'not knowing where to begin'. In the classroom examples, post-it notes and a marker pens were used instead of "Cards". I have used this technique, but use Green post-it for Java Library classes, Blue for interfaces and Yellow for my own.

When the initial design was made, I noted the structures down. From my notes, I have reproduced the OO design process followed at the beginning of this project.

Following The 'Post-it notes' Object Oriented Design Technique all the important and obvious classes are identified and written onto post-it notes (see figure 4.1). These were then be re-organised and brain stormed and optimum configurations and relationships were determined. The following sections go through the OO design process and lead to the object oriented structure used for this Constraint Editor.

## 4.1 Initial Object Oriented Appraisal

From the guidelines in "Designing Object Oriented Software" [1], the initial design process should consist of the steps below.

- Find the Classes in the System
- Determining the operations each class is responsible for and what knowledge it should maintain
- Determine the ways in which objects collaborate with other objects in order to discharge their responsibilities.

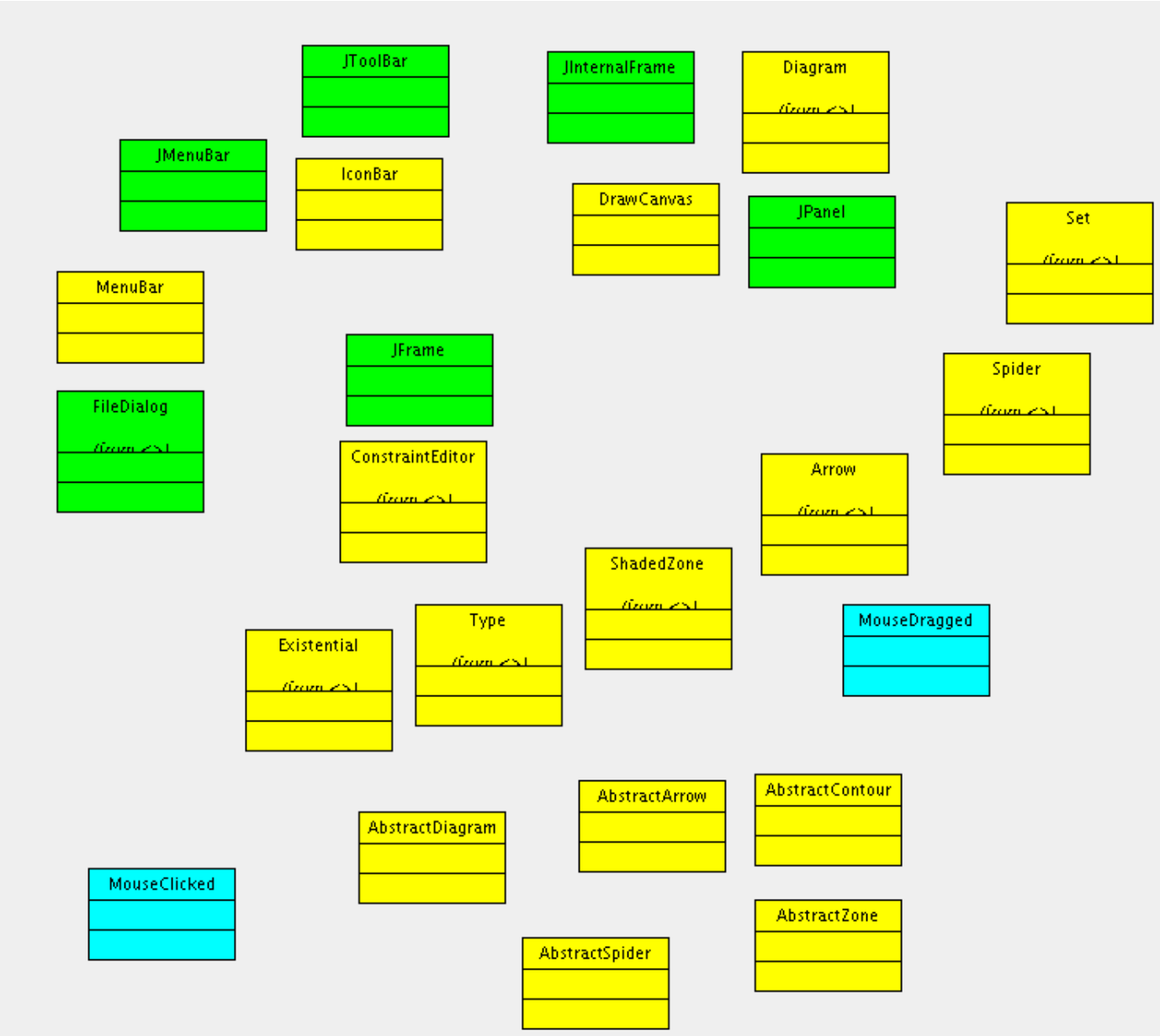



Figure 4.1: Post It Notes : Potential Classes

### 4.1.1 Listing all possible classes - finding the most obvious 'suspects'

The requirements specification (see Chapter 3) lists many 'nouns' that are good candidates for becoming classes in the program. These 'likely candidates' are collected and written onto post-it notes. They are arranged and considered (see figure 4.1).

Obvious 'noun' candidates from the Requirements Specification.

*Constraint Editor, Spider, Arrow, Existential, Universal, Shaded Zone, Diagram, Set and Type.*

*FileDialog, MenuBar, IconBar.*

*Abstract Diagram, Abstract Zone, Abstract Contour, Abstract Spider, Abstract Arrow.*

These are all written onto post-it notes and added to the developing model.

### 4.1.2 More Brainstorming - Data structures and Relationships

The Editor must be able to hold diagram sequences. Each diagram needs to be able to hold a sequence of drawable objects. Some drawable objects need to hold references to other drawable objects (e.g. Spiders, Arrows and shaded sections).

To reduce clutter each diagram in a sequence must be minimisable. The diagrams must also not be constricted as to size and must also be re-sizable. This all points to requiring a JInternalFrame (to give each diagram a sub-window) with a JPanel attached (to hold the actual drawn diagrams). The JPanel should be encapsulated in a class to allow extension, and as the JPanel is being used to draw on, DrawCanvas is an appropriate name.

The JPanel can also intercept mouse click and drag actions made on it, and would give mouse coordinates that would not need translating (i.e. would correspond directly to the diagram $x, y$ coordinates). A DrawCanvas object would be able to intercept these JPanel mouse events. Thus *DrawCanvas, JinternalFrame* and *JPanel* are added to the initial 'post-it' diagram.

The ConstraintEditor itself must be displayed in a window on the users Desktop. A JFrame will perform this function, and so a *JFrame* is added to the Post-it diagram.

### 4.1.3 Classes to Re-Arrange and Contemplate

Figure 4.1 shows the potential classes as 'post-it' notes.

The design process can now proceed to the next stage. The post-it notes can be rearranged and OO data structures can be formed and thought-about/discussed. The next section looks at a logical group of classes that can share a base class.

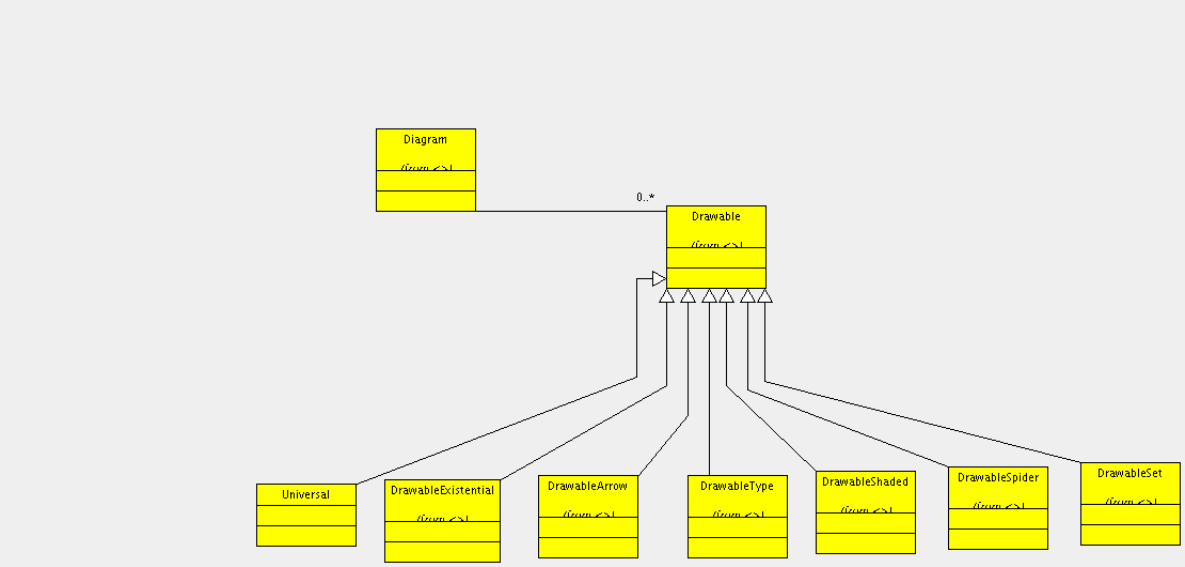



Figure 4.2: Drawable Objects Addition of Base Class

### 4.1.4 Refining Drawable Objects

When looking at the post-it diagram, and considering the way objects collaborate and what data they should maintain it becomes obvious that the drawable Objects have many common features.

- All drawable objects need to maintain information about where the object is physically drawn, and internal dimensions (such as points defining a polygon).
- All drawable objects must have a unique name within the diagram (for identification by other objects, referencing etc)
- All drawable objects must be able to represent themselves as XML (for saving concrete diagrams).
- All drawable objects must be able to represent themselves as Encapsulated Postscript (for printing diagrams and including them in documents)
- All drawable objects need to be held as references by the Concrete Diagram object, and some drawable objects need to reference others.

This means that they are all candidates for a base class. This allows common features to be programmed in the base class (code re-use) and commonly called methods to be called by the virtual mechanism (consistent interfacing).

Re-factoring and adding a “Drawable” base class, the Drawable objects can be now re-arranged to the OO structure shown in figure 4.2.

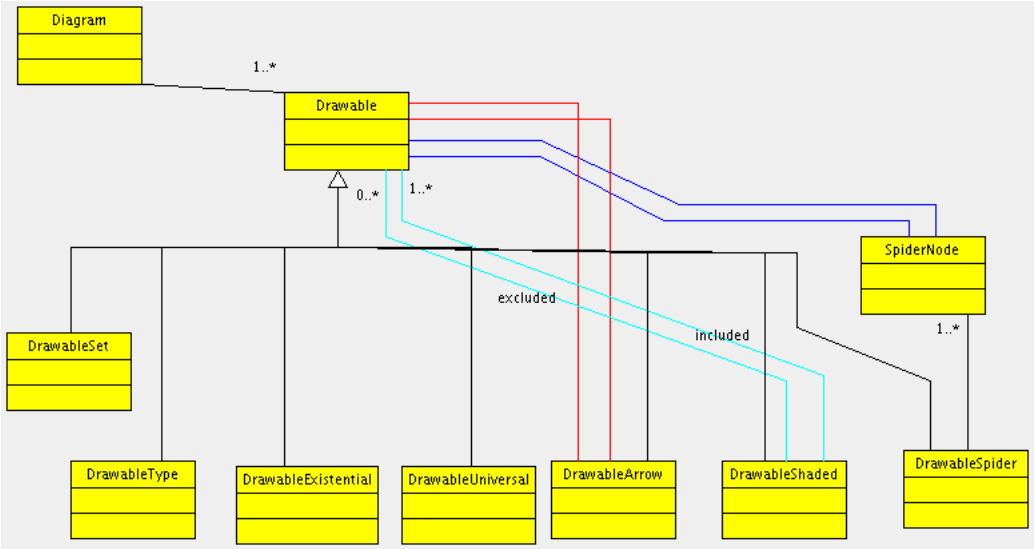



Figure 4.3: Drawable Refinement

The drawable objects are still not complete at this level of design. The spider, Arrow and shaded section classes all have to contain references to other drawable objects.

To list these requirements explicitly:

- Spider Objects Need to hold "one to many" zones. These Zones are defined as Universal or Existential drawable objects and are joined on the diagram by lines. Thus the lines define pairs of Spider feet. The Spiders therefore need a helper SpiderNode class to maintain the data for these drawn connections.
- Arrows need to hold references to their origin and destination objects.
- Shaded sections need a list of included and a list of excluded drawablesets/drawableTypes to define their shape on the diagram.

The above considerations create a new class (the Spider Node) and new relationships for the drawable objects. These additions can be seen in figure 4.3.

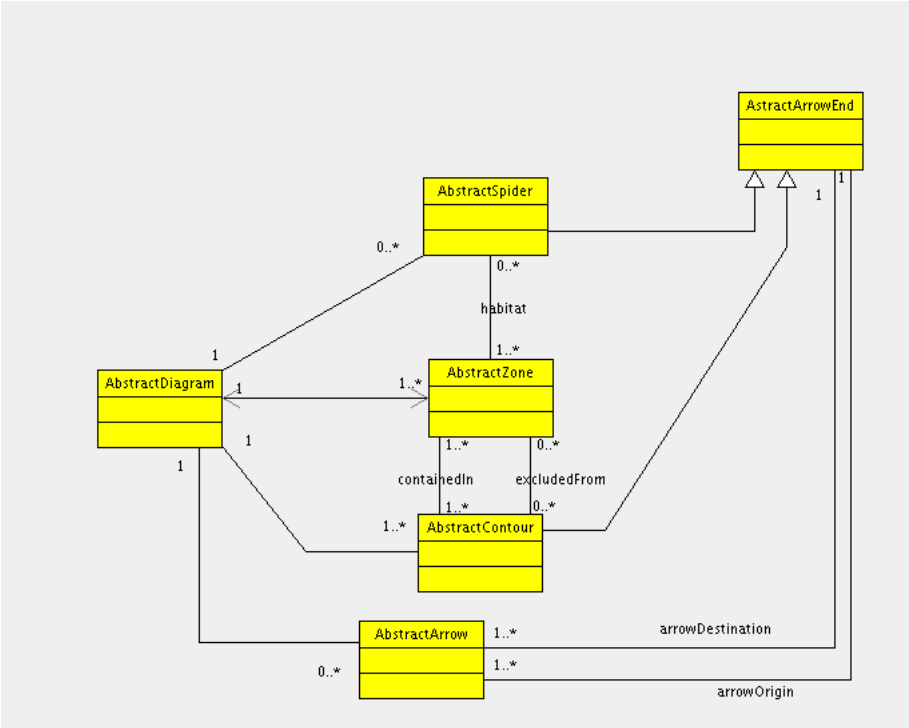



Figure 4.4: Post It Notes : Abstract Refinement on KMF Model

### 4.1.5 Refining the Abstract Model

The Abstract model holds references to Zones Spiders Arrows and Contours (see 3.1) and the relationships between the objects are largely determined by the KMF abstract model.

However rather than have separate classes inheriting the spider types, an attribute (EXISTENTIAL ONLY SPIDER) or (UNIVERSAL ONLY SPIDER) has been designed. Also the name for a Contour determines whether 'Given' or 'Derived' (see 5.1), so two classes inheriting from Contour are not required.

The two references to the (shaded and non-shaded) from AbstractDiagram to the Zones have been replaced with one, and an attribute placed in the Zone to reflect its shaded status. [1]

Notes:

[1]This is easier for saving and restoring abstract XML, but may have to be updated in the future if diagram sequences hold a common model and shading is required to change from diagram to diagram in the sequence

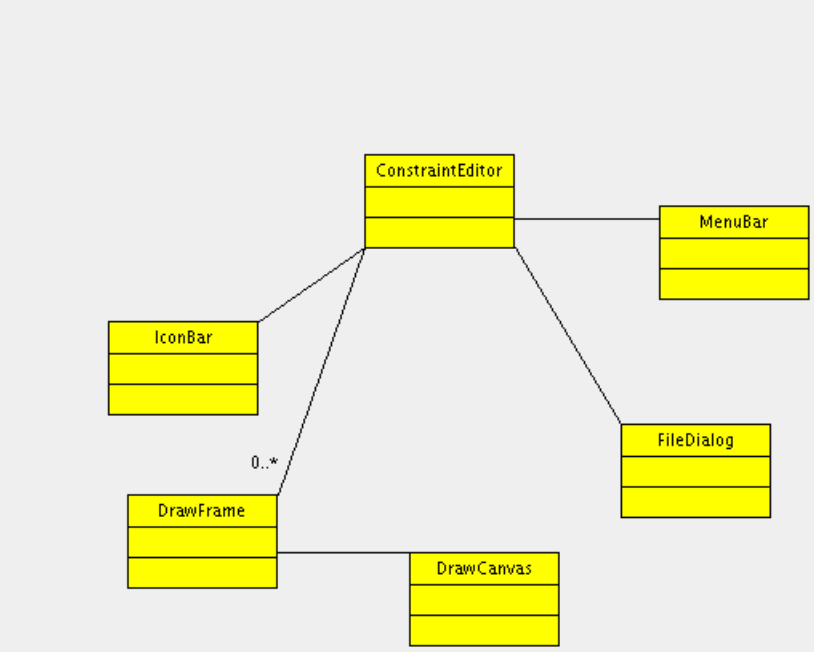



Figure 4.5: Post It Notes : GUI

### 4.1.6 Refining the GUI Model

The GUI is fairly standard Java2 fare, a Menu Bar, a drawing area for Diagram Windows and an Icon bar to select editing actions/draw types, pop-up windows and a FileDialog box.[1]

However, there are GUI design considerations which relate to the choice of Java GUI Library classes to use. Each diagram should be held in a separate window, so that it can be minimised and restored, allowing the user to easily work on several diagrams in the same session. Also clicking on a diagram should automatically select it for drawing operations. Using JInternalFrames to hold JPanels on which the diagrams can be drawn is an obvious solution to this. Simple classes such as 'Alert Boxes' and 'Pop-ups' are not included in this diagram.

Note this initial design diagram (see figure 4.5) shows only what the design will provide to the system. The next section deals with the Java Library classes required to implement these features.

Notes:

[1] Programming details for this type of Java interface can be found in the "Java swing" book [2].

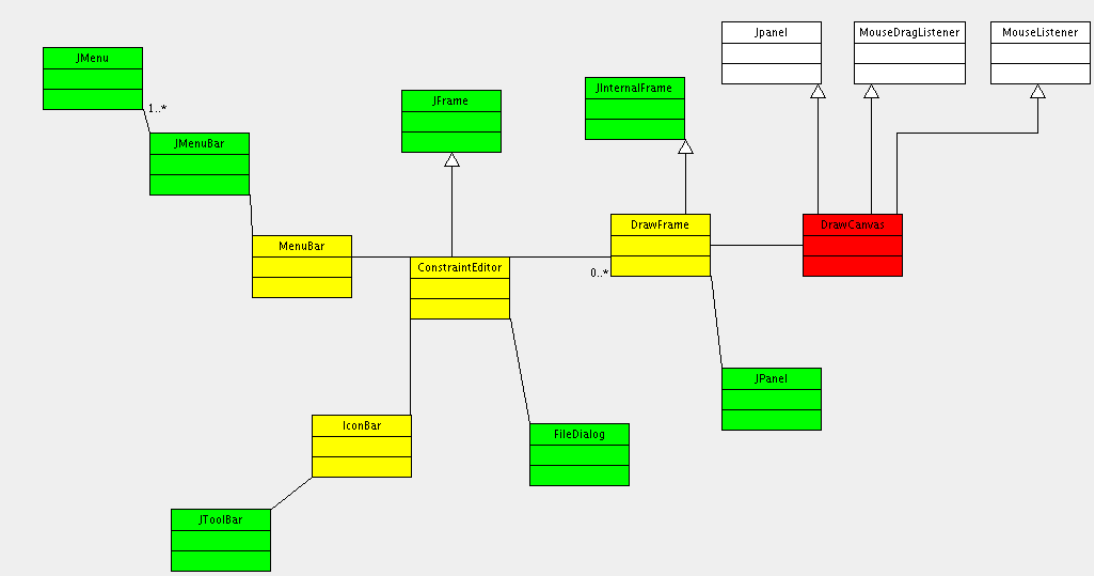



Figure 4.6: Post It Notes : GUI Refined

In order to implement the desired GUI the Java Library classes are required. These, with their functions are listed below.

- For the Main Application Window a JFrame.
- For the MenuBar the JMenuBar will provide drop-down selection from a Bar placed at the top of the Application.
- The file dialog can be used directly by the Constraint Editor.
- The DrawFrame (a window to hold a diagram being drawn) should be a JInternalFrame, so that each diagram in a sequence has its own sub-window, and hold a 'canvas' to draw the Diagram on.
- Each DrawCanvas should be/inherit a JPanel to draw the diagram on and receive mouse events etc.
- The IconBar can directly use a JToolBar.

Figure 4.6 shows the Java Library classes required to implement the user interface. A question still remains about the best location for one of the classes though.

This is The DrawCanvas class (highlighted in red). This class in use lies somewhere between the drawable objects and the GUI. This raises a question as to where it should be modelled, with the GUI or with the Drawable objects. The question of location, brings us to Java Packages. The DrawCanvas class could reside with the drawable classes or the user interface. This will be considered in the next section.

## 4.2 Organising the Classes into Logical Groups/Packages

For large Applications Java provides a grouping of classes called a package which includes a name space environment and a directory structure for the source code.[1].

Taking the classes written on the post-it notes and placing them into logical groups has already pointed to implementing three 'packages'. These three packages 'ui' 'abs' (abstract is a java reserved word [2]. and 'drawable' will be used, and this final stage in the high level design defines relationships between these packages. [3].

### 4.2.1 Re-Factoring DrawCanvas Location in the Packages

The DrawCanvas class must reside in either the 'ui' or the 'drawable' package. In order to place this class in the 'best' location, we can look at its collaborations with other classes and its responsibilities.

These are:

| *Responsibility/Collaboration* | *Drawable* | *User Interface* |
|---|---|---|
| Interpret Mouse Clicks Drags Over diagram | + | |
| Re-draw the diagram after edits/repaint requests | + | |
| Allow access to a diagram by File loading/saving | | + |
| Hold dynamic editing status (hanging edit detection) | + | |
| Display diagram specific pop-up windows | + | |
| GUI Control of sub windows (DrawFrame/JInternalFrame) | + | |

Clearly the "DrawCanvas" class belongs in the drawable package. This means that this must be added to the drawable diagram, thus refining it again (see figure 4.7).

Notes:

[1] Java forces the source code for each package to be held in separate named directories corresponding to the package names

[2] Try importing a package called 'abstract' into a Java program! The error messages themselves may be quite 'abstract'!

[3] Java forces the source code for each package to be held in separate named directories corresponding to the package names. It is thus important to include package factoring into the design process.

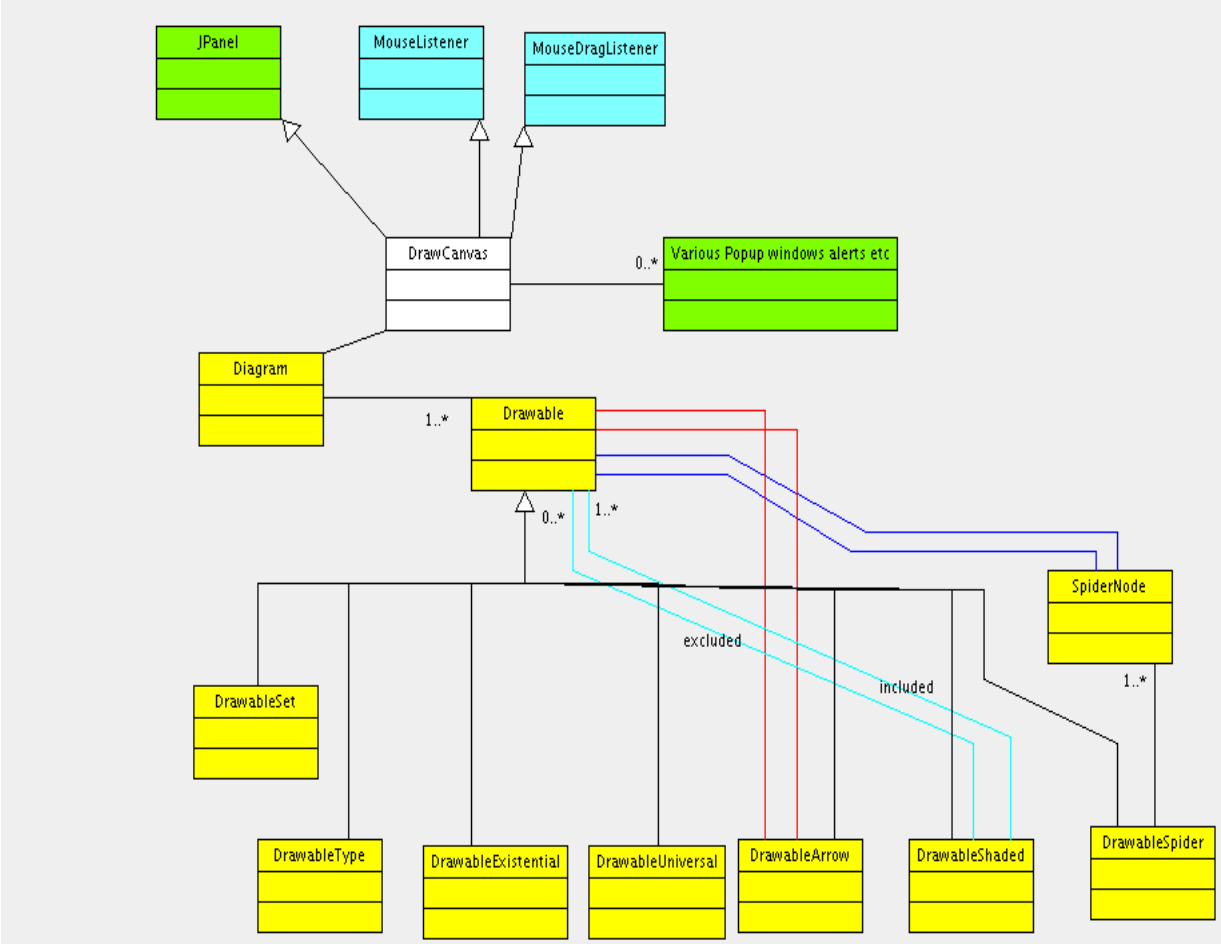



Figure 4.7: Drawable Refinement : Addition of DrawCanvas

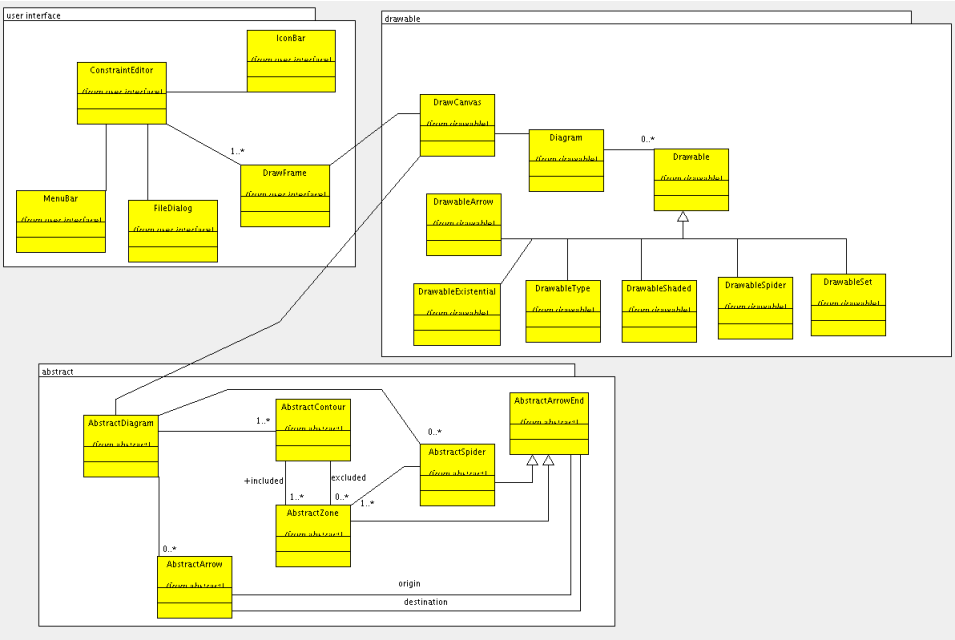



Figure 4.8: Post It Notes : Organising into Packages

### 4.2.2 OO Design of the Constraint Editor with Packages

The result of this design process is not final, but care to create classes that work together naturally will hopefully prevent any major redesigns. The result of this chapter, from initial designs using the 'post-it' method shown in figure 4.8[1] is very close to the design used currently (see figure 6.1).

Notes:

[1]Shown here without interfaces or Java Library classes for clarity

# Chapter 5

# Editor Behaviour and Design : Implementation Issues

This chapter discusses editing behaviour or the human-computer-interface, with reference to specific problems associated with constraint diagrams, and documents the compromises and solutions applied.

In general a philosophy of trying to prevent something happening, rather than issuing warning boxes has been followed. For instance when the user attempts something illegal they could be swamped with annoying pop-up windows. Far better that the impossible actions simply do not react on the diagram.

For instance if you are building an existential spider, you simply cannot connect to a universal point (although the software will have seen the possible connection as the mouse drag coordinates will have 'hit' a connectable object). The dialog box to allow you to confirm a connection will not appear.

Also where possible the user should be able to re-arrange the diagram to make it visually describe the constraints/mathematical meaning. The user should be able to freely drag the elements around, stretch/re-size them etc, without being swamped with warning boxes. Thus a diagram could become 'badly-formed' whilst being changed (for instance a shaded area might temporarily become invisible), to make editing an easier process.

The point at which the user is made aware of problems in the diagram, should be on converting to abstract, and upon saving. A 'Well-formed' check menu operation could also be included.

Factors which define a badly formed concrete constraint diagram are [24] :-

- Derivation circles/ambiguity
- No Removed Shaded Zones
- Absence of Triple Points
- Connectedness of Zones (apparent duplicate zones)
- Non-transverse crossing of Contours
- Spiders with one or more feet in same zone

## 5.1 Naming and Given and Derived Objects

The Editor forces unique names for all drawable objects within a diagram. When a new object is created a pop-up window offers a name beginning with Untitled and pre-fixed with a sequential number. The names of all objects are passed through to the Abstract representation.

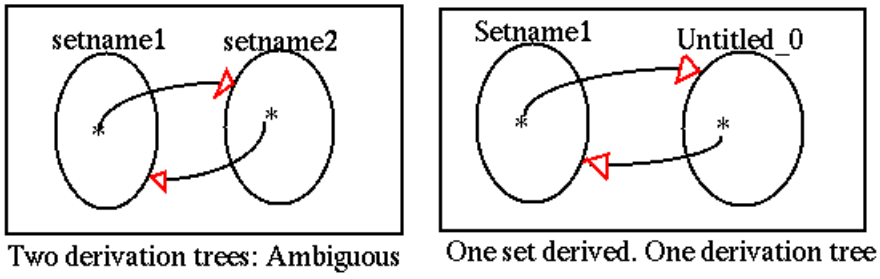



Figure 5.1: Ambiguous Definition by Derivation.

For this Editor it has been defined that names binging with *Untitled_* are derived names. For instance an ambiguity can be seen in Figure 5.1. Here All the elements of a set define by an arrow the elements in another set and visa versa. Any algorithm using the abstract output would be faced with a dilemma, which set to start at ! Sets name *Untitled_* may thus be defined as derived.

Faced with a complicated diagram with several ambiguous derivations, a tree of possible interpretations would develop. A future version may have to assign derivation priorities to contours that are not given.

## 5.2 Spider Creation Problem

A Spider is simply a series on connected Existential OR Universal objects. In the Abstract the spider is defined as a series of zones (or habitats).

Some decisions do have to be made when thinking about how to allow the user to create Spiders on the diagram.

Obviously the Spider is only created by mouse clicking on an object which is valid to connect to a spider. At that moment of creation the Spiders type (Universal or Existential) is set. The Spider is then in a mode where new connections can be made (defining its habitats). The user may come back to a spider later in editing and add another foot. Thus when a new connection (or SpiderNode) is being created it is a two stage process. The new node is created and a hanging foot is created. This hanging foot may then be dragged to the new location.

A problem occurs when a spider is connected to a point already part of another spider. It could mean that the user wants the spiders to merge into one, or it could be a mistake. Currently a Spider cannot connect to another spider, and this solves the dilemma according to the philosophy of not allowing something to happen rather than swamping the user with dialog boxes. If a future version allows the merging of Spiders, it will have to ask the user at the point of merge whether to cancel or to merge, to prevent errors which are difficult to undo. Consider that a 'merge' could have been caused by a faulty mouse drag. A confirmation of a merge would for this reason always be required.

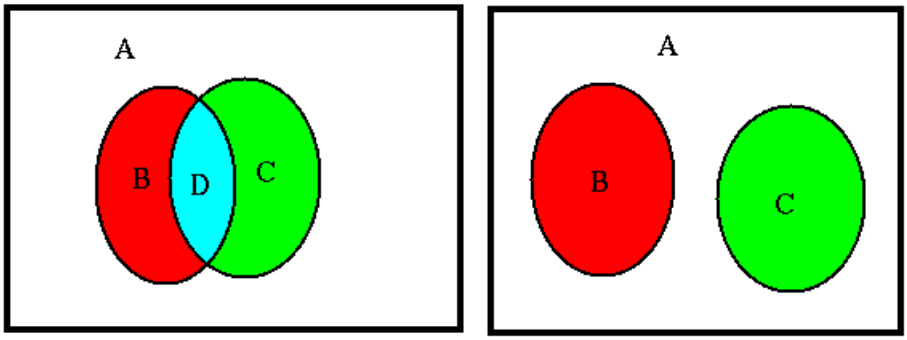



Figure 5.2: Intersection and Zone Creation

## 5.3 Movable Shaded Areas

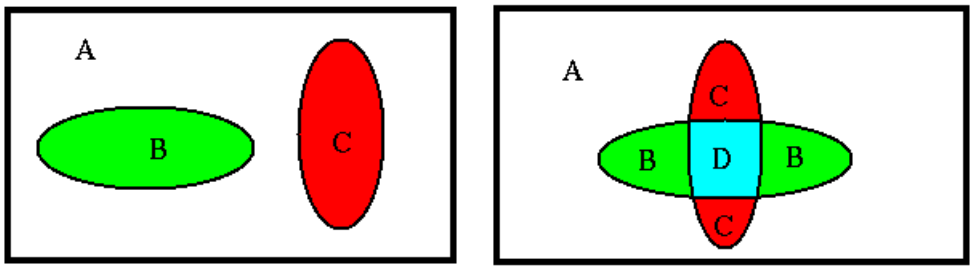
Consider figure 5.2.

The zone $D \equiv B \cap C$ is shaded, and the user then picks up the set B and moves it so that the intersection is no longer visible.

The Editor has a choice to remove the shaded attribute to the intersection, or warn the user, or allow the user to carry on moving the elements about.

The user could be warned that there is an invisible shaded area later (either when saving the concrete diagram or when converting it to an abstract diagram).

A policy decision is required here. I have implemented the warn on save or convert here, because in a complex digram the user could become swamped with warning messages when simply trying to re-arrange the look of a diagram.




instanceof

Figure 5.3: Apparent Duplicate Zones in a Concrete Diagram

## 5.4 Falsely Apparent Duplicate Zones

Consider the following editing sequence.In figure 5.3 we have 3 zones. Now, allow the horizontal ellipse shaped set B to be dragged onto the vertical ellipse. We now have 4 Zones

$$Background\ BC$$

$$B$$

$$C$$

$$D \equiv B \cap C$$

.

However on the diagram 7 zones are visually present. For a trivial example like this it is easy to see the duplicates, but in a more cluttered diagram this will be a serious problem. In the interests of smooth movement of elements on the diagram during constraint diagram construction, warnings will only be issued at save or convert.

A diagram with this confusing shape would be considered 'not well formed', and should cause a warning on save or convert to abstract.

## 5.5 Arrows and Connecting Lines : Mutable Curve

A Mutable Curve was needed to represent the connections in Spiders and Arrows.

Initially arcs (in the style of xpaint [20]) and straight lines were used with an arrow head that could be rotated in steps of 45 degrees. As the test diagrams became larger, these began to add untidiness and inflexibility to the diagrams. The arcs needed to be more flexible and they needed to be bent around objects in the diagram. The arrow heads should join to an object and rotate smoothly, $45^o$ was not fine enough.

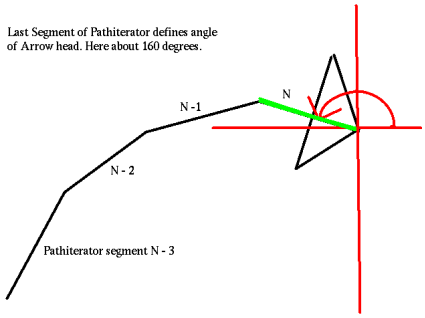



Figure 5.4: Cubic Curve 2D Path Iterator Angle of Last Section

### 5.5.1 Cubic Curve Connectors

Looking in the extensive Java 2 drawing library, a class to suit these requirements was found. *CubicCurve2D.Double* allowed two control points and two smooth curves within one line. The smooth lines were obtained using a path iterator class, which returned line segments (see figure 5.4). I enclosed this Java class in MutableCurve in order to integrate it with the drawable objects. The Mutable curve class allowed the user to drag and drop the control points, once the destination had been decided by the owning class, and thus alter the shape of the lines on the diagram.

### 5.5.2 Rotation of Arrow head

A standard arrow head was made, and from the gradient of the last section (see the *green* last line in figure 5.4) of the cubicCurve2D "PathIterator"[21] output. From this a rotation angle was calculated. The points for the rotated arrow head were then calculated using a standard 2D rotation matrix [5] thus :-

$$\begin{bmatrix} cos\ \theta & -sin\ \theta \\ sin\ \theta & cos\ \theta \end{bmatrix}$$

## 5.6 Editing Environment

### 5.6.1 Main Window

The Main Window needs to hold the drop down menu bar along the top, an Icon bar to select the editing tool functions and a drawing area in which to hold the JInternalFrames in which each diagram lies.

### 5.6.2 Menu Bar

The is a standard drop down menu and has two drop down elements File and Diagram. File contains entries to save and restore concrete diagrams, output EPS zip files and Abstract XML.

### 5.6.3 Icon Bar

This contains Icons representing the editing objects (Set, Type, Spider, Arrow, Universal Qualifier, Existential Qualifier, Shaded) and actions (delete, manipulate, re-size, move, information pop-up). On selection, a variable in the Constraint Editor is set to the 'last selected action'. This is then used by classes receiving mouse clicks/drags.

### 5.6.4 Use of Pop-up and File Dialog Windows

Pop up windows are used to confirm actions (such as object creation and to get the user to name them) and to browse the file system. [2]

## 5.7 Event Sequencing

The event sequencing is mainly handled by the Java GUI Library classes. A mouse click over a diagram for instance will cause the mouseClicked method to be called in the DrawCanvas object, because the DrawCanvas object has implemented the mouseListener interface. The event sequencing for standard Java2 components is described in "Java Swing" [2].

A 'last selected' editing action from the IconBar is held by the constraint editor. A mouseclick/drag is then interpreted by the DrawCanvas receiving it. For instance if the selected action is to create a new set the DrawCanvas object will interpret the mouseclick as a "create Set" event.

Some editing actions are more complicated and require internal states to be held. For instance an arrow must first be created and then dragged to its destination.

### 5.7.1 Mouse Click events on Diagrams

The Class "DrawCanvas" in the package drawable implements the MouseListener and the MouseDragListener. "DrawCanvas" inherits the Java swing class JPanel. Thus all mouse events relate to the Panel on which a diagram is drawn. This has many natural advantages, only the diagram being clicked on receives the event, and because the coordinates are relative to the JPanel, no translation of mouse coordinates from screen position to drawing are required. This also forces the Java GUI engine to do all the work of re-sizing / minimising JInternalFrames and general housekeeping in the windowing system.

Each DrawCanvas Object owns a Diagram in the drawable package. (See figure 6.1). On receiving a mouse click the current action type (a static held in the ConstraintEditor class representing the action to take) is used in conjunction with the mouse coordinates to decide what to do.

#### 5.7.1.1 Set or Type Creation

If the action type is MA_SET or MA_TYPE, (i.e. the action of creating a set on the diagram). A pop-up window is displayed to give the user the opportunity to enter the name of the new set or to cancel the operation. If OK'd a new set will be created at the mouse coordinates.

#### 5.7.1.2 Spider Creation

If the action type is MA_SPIDER, the task is a little more complicated. The Mouse coordinates may not be over an object that a spider can connect to. In this case in accord with the GUI policy (chapter 5) that mouse click will be ignored. If there is a suitable object at the mouse coordinates, it must be examined to see if it is already part of an existing Spider and if so then puts the diagram into a state where a mouse drag will move the spider connector. On a mouse drag end coordinates sitting on a valid Connectable object ( spiders can only connect between existential and universal qualifiers : they cannot connect mixtures of both types) a dialog box is then offered to confirm the connection. Note here there are two states of SPIDER creation. This leads to another problem, that of the hanging edit. If a spider drag is not completed and a new mouse click is made over the DrawCanvas the operation will be left incomplete. This must be detected and the incomplete (unconnected) spider part removed.

#### 5.7.1.3 Arrow Creation

If the action type is MA_ARROW we again are faced with two states, an initial selection of the origin and a drag mode. Again hanging edits, as for the Spider, must be cleared if a new action type is selected.

#### 5.7.1.4 Delete Drawable Object

Action type MA_DELETE. If the Mouse is held over a Drawable object, offer a pop-up window to confirm deletion from the diagram. On delete, the object is removed from the diagrams collection of drawable objects and then a redraw is called.

#### 5.7.1.5 Create Universal or Existential Point

Action type MA_UNIVERSAL or MA_EXISTENTIAL. Display a pop-up window to give the user the option to name the element or cancel the action. If OK'd then the object is placed on to the diagram at the mouse coordinates given to the mouseClicked method.

#### 5.7.1.6 Manipulate

This applies to Sets and Types only. This allows individual points that make up the polygons to be moved under mouse drag control allowing the user to change the shape of the default polygons (a Set is by default an Oval and Type is a Rectangle).

#### 5.7.1.7 Move Drawable Object

This applies differently to Sets/Types and Existential/Universal drawable objects than to Arrows and Spiders. Arrows and Spiders are anchored to Sets/Types and Existential/Universal drawable objects these and move naturally when the points to which they are anchored are dragged.

#### 5.7.1.8 Shade Area

Shade an area on the diagram. Only applies to Zones (set and type intersections). All DrawableSets and Types are examined to see whether they contain the mouse coordinates. Exclusion and inclusion lists are generated as can be seen on the UML model (see figure 6.1). The shaded area will be created as a separate object, and the shaded area to be re-drawn recalculated each time the diagram is painted (see 5.3).

#### 5.7.1.9 Re-Size

This only applies to Drawable Sets and Types. After Selection the mouse drag alters the size of the objects, even letting them be horizontally and vertically reversed.

#### 5.7.1.10 Information

This displays a Pop-up window where information on the object is displayed. Each Drawable Object will allow the label to be shown or hidden and allow comments to be placed with the object (which are saved in the concrete diagram).

Some Drawable Objects may refine this pop-up window, supply more information and allow other procedures to be applied.

### 5.7.2 Creation of a New Diagram

New diagrams are created from a menu bar selection. A pop-up window offers an option to name the new diagram or to cancel. Once the diagram is OK'd it is added to the diagram sequence currently being edited, is opened and displayed ready for the addition of drawable objects.

Each diagram will be displayed in a Java standard JInternalFrame, and comes under the control of the ConstraintEditorMgr class.

### 5.7.3 File Loading and Saving

File Selection for loading and saving will use the standard Java FileDialog class. This ensures ease of use correct operation across platforms for file selection.

## 5.8 Integration with other tools

This editor could have two 'constructors'. One to open the application as stand alone, able to produce concrete and abstract models as XML. The other opens the editor in an internal frame and is intended for use in a larger framework (where it would share the abstract models as run-time Java objects..

# Chapter 6

# Software Implementation

This Chapter is concerned with the code describing the principal classes as implemented.

This chapter concentrates on the overall structure of the Editor software and the interactions with the Java Windowing system and the file systems etc. Most of the Java GUI/Windowing/Menubar code was adapted and expanded upon from the "Java swing" [2] book. Specific details of the fundamental classes were gleaned from the Sun Java Books [3] [4] and the Sun Java SDK API [21].

## 6.1 Packages

There are three Java packages used to build this project:

### 6.1.1 Package ui

This package deals with the user interface, setting up the windows environment (controlling the JInternalFrames which hold the diagrams within a sequence etc) and handling calls to Pop-up windows, File Dialog, Menu Bar and Icon Bar classes.

### 6.1.2 Package drawable

This package holds all individual items that can be drawn on a diagram. It also contains routines to collect, manipulate the drawable objects and also facilities to convert the concrete diagram to an abstract model.

This package can represent its contents as XML via the call to toABS() from the class AbstractDiagram.

### 6.1.3 Package abs

The 'abs' package is a container for the abstract models. It also contains methods to assist in the conversion from concrete to abstract model. This package can represent its contents as XML.

All Major Classes are listed and described in the next section.

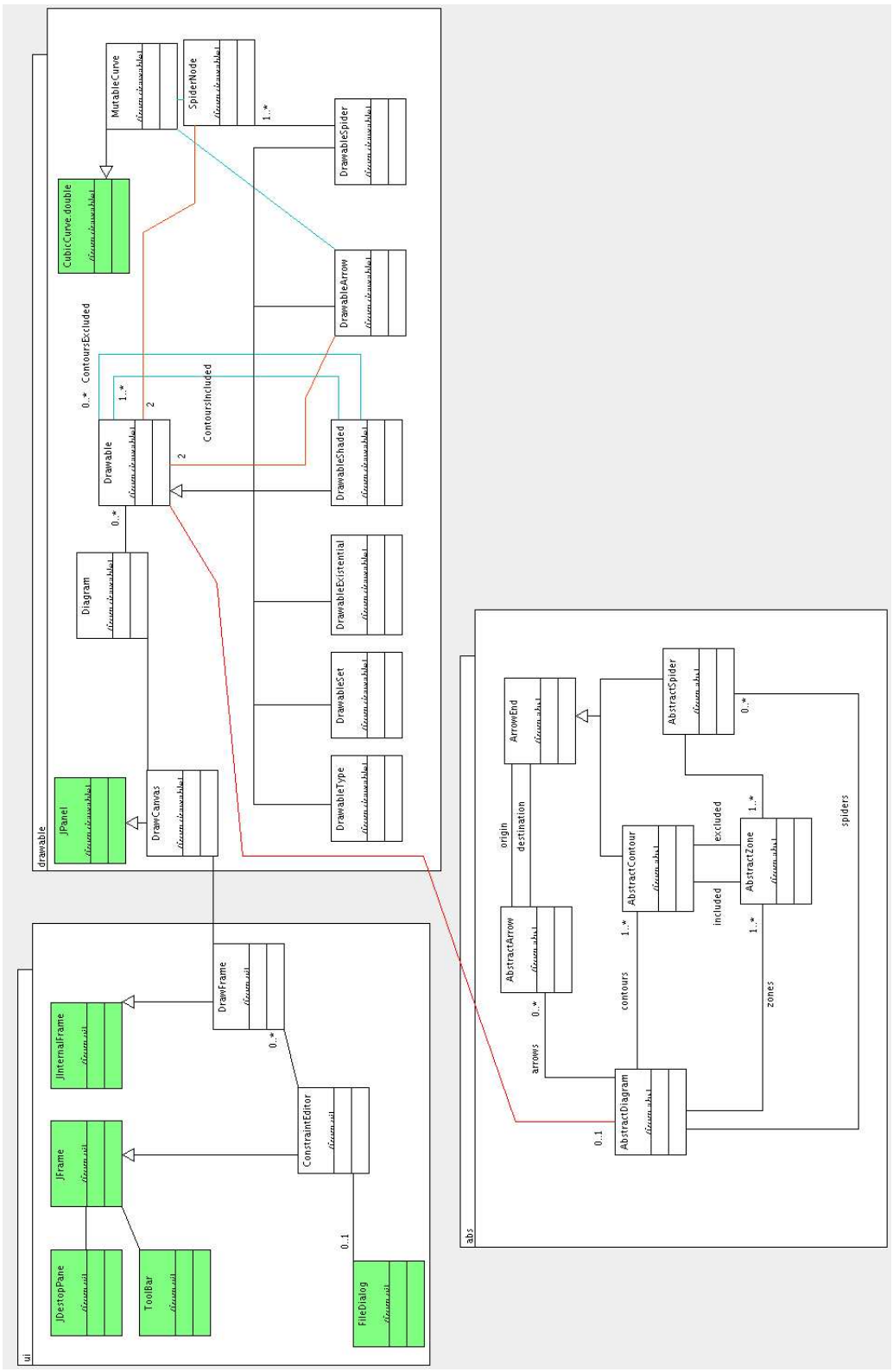



Figure 6.1: UML Description of the Constraint Editor

## 6.2 Directory Structure

The software is implemented in a hierarchical file system. This submitted on a CD complete with all Java Source files, Makefiles[10] and documentation source (latex). The 'abs' directory contains java code for the Abstract model, 'drawable' for the concrete and 'ui' for the user interface classes.

```
$
$ tree -d
.
|-- abs
|-- docs
|   |-- OriginalRequirementsDoc
|   |-- meta_model_spec
|   `-- project_writeup
|-- drawable
|-- examples
|-- images
|-- testing
`-- ui

10 directories
$
$
```

### 6.2.1 JavaDoc - Web Page API documentation

An index.html file exists in the root directory of the CD. This contains the JavaDoc generated documentation for all the classes written.

### 6.2.2 Test Files - testing

All Test Files (see chapter 9) are contained in the "testing" directory.

### 6.2.3 Examples - examples

Examples of concrete diagrams and abstract XML are included in the "examples" directory.

### 6.2.4 Documentation Directory - docs

This has three directories.

- OriginalRequirementsDoc - StarOffice and Word format original proposals for this project. Also various related PDF documents.
- meta_model_spec - KMF documents the original target environment for this project
- project_writeup - Latex source, graphics and zargo[16] files used to produce document.

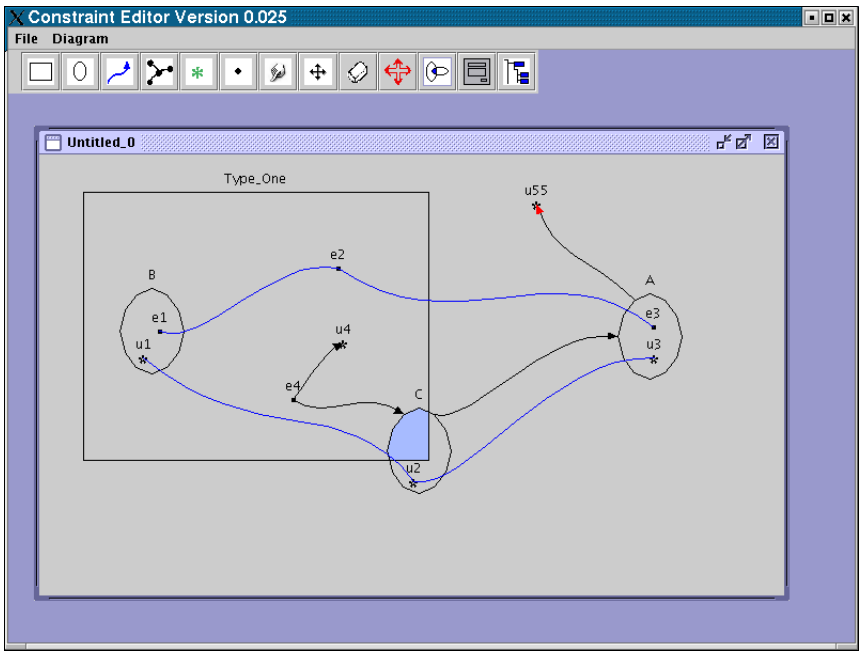



Figure 6.2: Simple Constraint Diagram

## 6.3 Class Descriptions

This section describes the principal classes in detail as implemented. Each Class is described with UML,a short description and then (if appropriate) with the XML that represents the class. All XML examples from Drawable Objects described, have been taken from the concrete XML output from the diagram shown in figure 6.2

The UML gives a theoretical description of the class and how it interacts with other classes, and the XML gives a practical example of the data that a drawable object must contain.

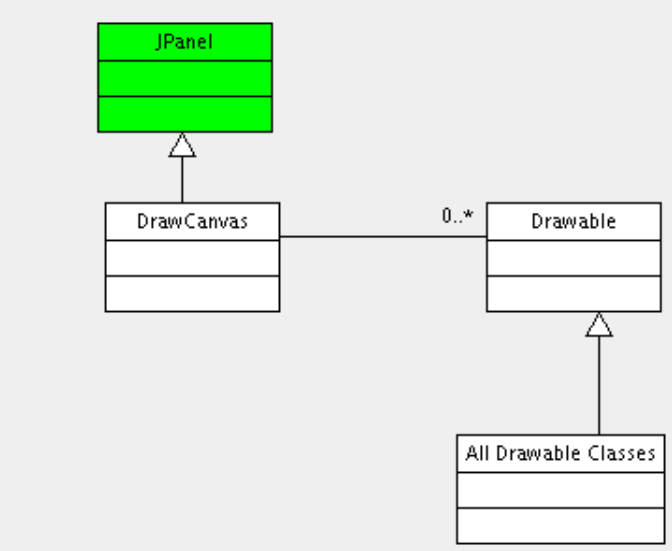



Figure 6.3: UML Drawable Class

### 6.3.1 The "drawable" Package

This package is responsible for the display, manipulation and creation of concrete diagrams. The concrete diagrams are saved by the ConstraintEditor class. However each "Drawable" derived object can represent its self in XML via a "toXML" method.

#### 6.3.1.1 Drawable - Base Class for all Objects drawn in a Concrete Diagram - package drawable

The "Drawable" base class is inherited by all drawable elements, and is used to collect drawable elements by the Diagram class (see figure 6.1) Some Drawable derived classes hold other drawable objects (for instance, Spiders and Arrows).

Some methods are abstract and are used to force a common interface, and some are written to be used or overridden as necessary.

For instance all drawable objects must be able to output themselves as XML strings, EPS Strings and must be able to paint themselves on a Canvas. This class has also been used to hold some miscellaneous methods such as the calculation of Zones Areas and Shapes due to intersection and subtraction. This has no XML representation.

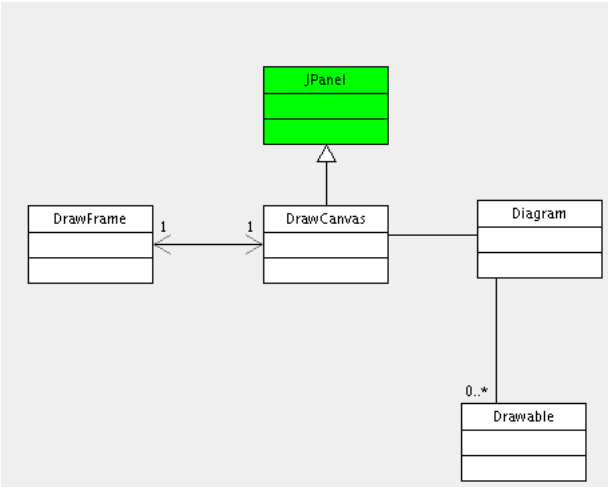



Figure 6.4: UML DrawCanvas Class

#### 6.3.1.2 Drawable - DrawCanvas - package drawable

This Class is necessary to hold the diagram and interpret mouse events. It is held by DrawFrame (which inherits a JInternalFrame) which is the sub window that the editor uses to display each diagram in the editing area. It could be seen as a 'glue class' between a diagram and the user interface.

This Class inherits a JPanel and implements the interfaces for MouseListener and MouseDragListener. since the constraint diagram is drawn on this panel the Coordinates from the mouse events correspond directly to the coordinates on the diagram.

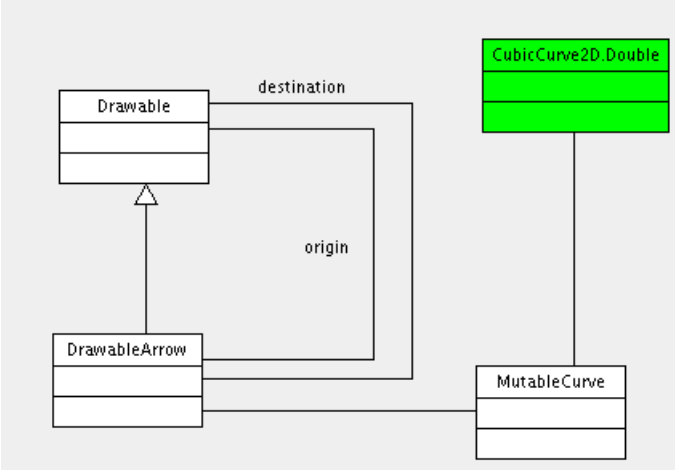



Figure 6.5: UML Drawable Arrow Class

#### 6.3.1.3 DrawableArrow - package drawable

This holds references to two other drawable objects in the diagram, an origin and a destination. It uses the MutableCurve (see 6.3.1.11) class to draw the arrow and draws the arrow head based on the angle of the lines entry to the destination object (see 5.5.2).

The Arrow can only point to and from the following objects *DrawableSet, DrawableType, DrawableExistential and DrawableUniversal*.[1] Also an Arrow cannot point back at its self.

Corresponding OCL constraints below.

```
pre-condition correctType:
      destination = instanceof(DrawableSet)
 or  destination = instanceof(DrawableType)
 or   destination = instanceof(DrawableExistential)
 or   destination = instanceof(DrawableUniversal)

pre-condition circularRef:
   destination != origin
```

XML representation

```
 <arrow name='r1' origin='C'  oindex='5'  destination= 'A'  dindex= '9' >
       <mutablecurve   ctrlx1='409.91'   ctrly1='252.97'
        ctrlx2='465.02000000000004'   ctrly2='162.73999999999998'   />
     <comments>
       An Arrow between two sets
    </comments>
</arrow>
```

Notes:

[1] These correspond to Contours and spider feet in the abstract model.

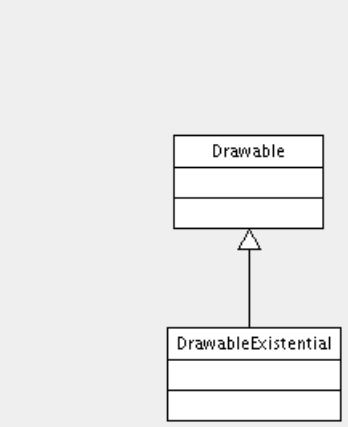



Figure 6.6: UML Drawable Existential

#### 6.3.1.4 DrawableExistential - package drawable

This draws a small filled square on the diagram and can be used as a destination/origin point for Arrows and Spiders.

XML representation

```
<existential name='e4' >
  <x> 237 </x>
  <y> 229 </y>
  <height> 5 </height>
  <width> 5 </width>
    <comments>
    </comments>
</existential>
```

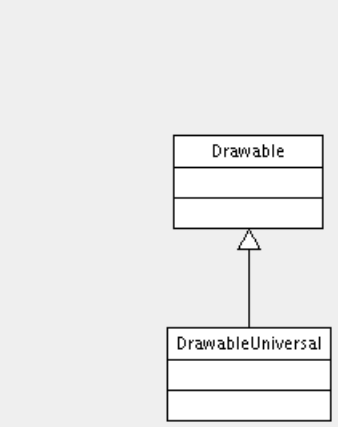



Figure 6.7: UML Drawable Universal

#### 6.3.1.5 DrawableUniversal - package drawable

This draws a small asterisk on the diagram and can be used as a destination/origin point for Arrows and Spiders.

XML representation

```
<universal name='u1' >
  <x> 97 </x>
  <y> 190 </y>
    <comments>
   </comments>
</universal>
```

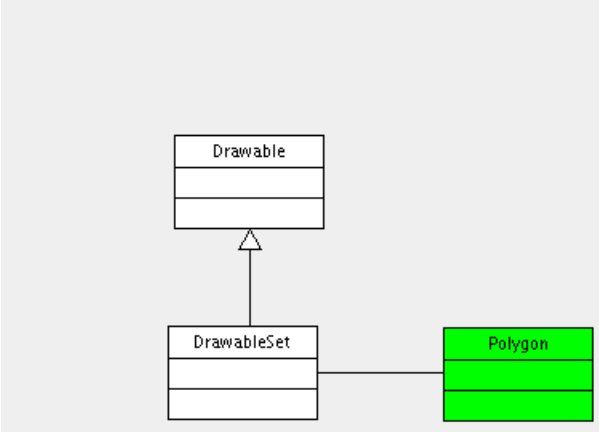



Figure 6.8: UML Drawable Set

#### 6.3.1.6 DrawableSet - package drawable

This draws a 12 sided Polygon in an oval shape on the diagram and can be used as a destination/origin point for Arrows.

XML representation

```
<set name='B'  npoints='12' inherentlyempty='0' >
  <pp pindex='0'  xp='105' yp='205' />
  <pp pindex='1'  xp='119' yp='199' />
  <pp pindex='2'  xp='130' yp='185' />
  <pp pindex='3'  xp='135' yp='165' />
  <pp pindex='4'  xp='130' yp='146' />
  <pp pindex='5'  xp='119' yp='131' />
  <pp pindex='6'  xp='105' yp='125' />
  <pp pindex='7'  xp='91' yp='131' />
  <pp pindex='8'  xp='80' yp='145' />
  <pp pindex='9'  xp='75' yp='165' />
  <pp pindex='10'  xp='80' yp='185' />
  <pp pindex='11'  xp='90' yp='199' />
  <height> 80</height>
  <width> 60</width>
  <x> 105</x>
  <y> 165</y>
    <comments>
   </comments>
 </set>
```

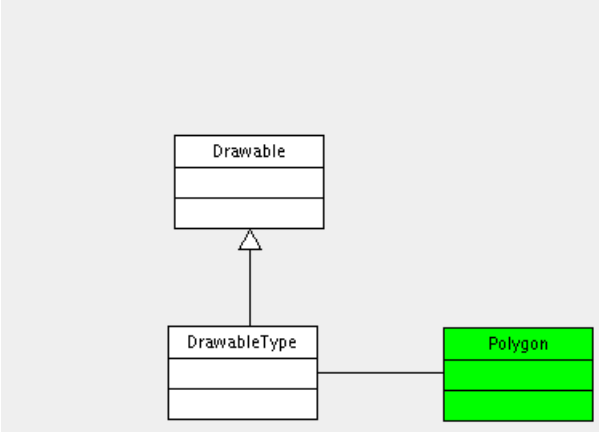



Figure 6.9: UML Drawable Type

#### 6.3.1.7 DrawableType - package drawable

This draws a 4 sided Polygon in a rectangle shape on the diagram and can be used as a destination/origin point for Arrows.

XML representation

```
<diagram name='Untitled_0' >
<type name='Type_One'  npoints='4'  inherentlyempty='0' >
  <pp pindex='0'  xp='41' yp='286' />
  <pp pindex='1'  xp='364' yp='286' />
  <pp pindex='2'  xp='364' yp='35' />
  <pp pindex='3'  xp='41' yp='35' />
  <height> 150</height>
  <width> 150</width>
  <x> 107</x>
  <y> 143</y>
    <comments>
   </comments>
</type>
```

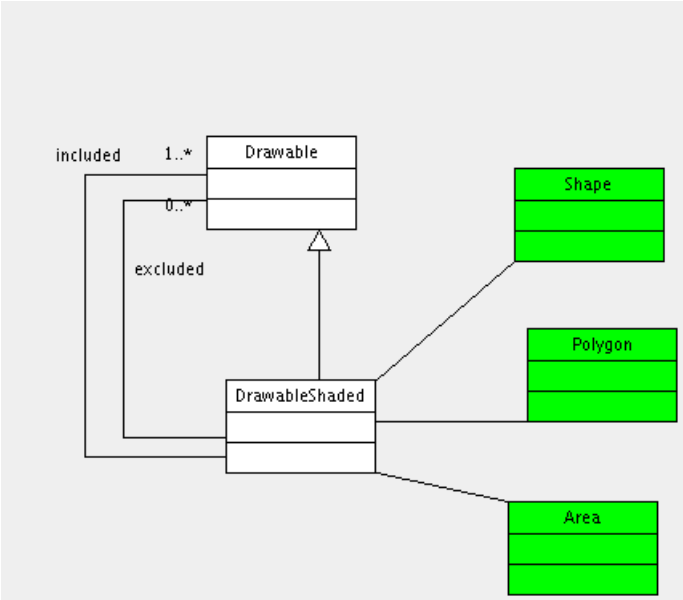



Figure 6.10: UML Drawable Shaded

#### 6.3.1.8 DrawableShaded - package drawable

This class defines a shaded Zone in the concrete diagram. It consists of two vectors, a list of included sets/types and a list of excluded sets/types.

On each repaint of a diagram, the shaded area must re-calculate polygon to fill to represent the shaded area.

It calculates this by converting all the included sets to Java Area objects (via the class Shape) and taking the intersection of them all. It then subtracts all excluded sets again using the Java Area classes. The resultant area is converted back to a Polygon (via the class Shape) and then drawn on the diagram with a fill colour.

The UML diagram shows references to Drawable objects. These drawables can only be of the DrawableType or DrawableSet classes (contours in the abstract diagrams). An OCL constraint is thus required.

```
post-condition:
  forALL->included ( a | a = instanceof(DrawableSet) or a = instanceof(DrawableType) )
  forALL->excluded ( b | b = instanceof(DrawableSet) or b = instanceof(DrawableType) )
```

XML representation

```
<shadedarea>
   <intersectioncomponent>Type_One</intersectioncomponent>
   <intersectioncomponent>C</intersectioncomponent>
  <comments>
  </comments>
</shadedarea>
```

Note that for the Concrete XML, only the intersection components are saved.

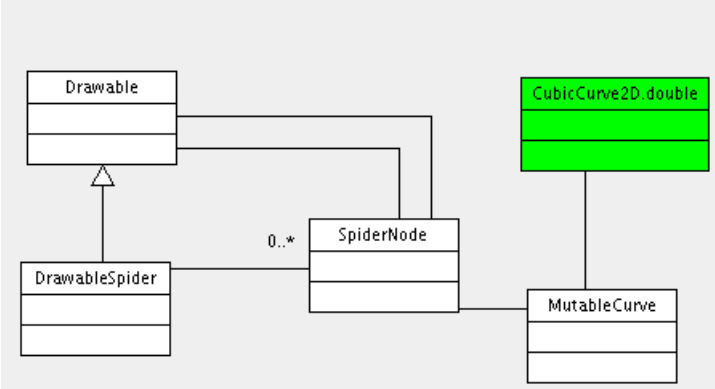



Figure 6.11: UML Drawable Spider

#### 6.3.1.9 DrawableSpider - package drawable

Spiders are connected groups of either Existential or Universal objects on the diagram.

This class contains a Vector of SpiderNodes see 6.3.1.12 which in turn contain MutableCurve objects which define the shape of the connecting lines between the spider feet.

On creation of a Spider, the origin, or first point will either be an EXISTENTIAL or a UNIVERSAL. This sets the spiderType. At the connection of each new spider node, two checks must be performed. That the spider is not connecting to an object already present in the spider and that the object is of the correct type (one of Existential or Universal). These can be represented as OCL pre-conditions.

```
pre-condition check-dest-type:
   spiderType = EXISTENTIAL and (destination instanceof(EXISTENTIAL))
 or
   spiderType = UNIVERSAL   and (destination instanceof(UNIVERSAL))

pre-condition check-dup-zones:
    self->forALL
      ( sn | sn.habitats->forAll( h | h.zone != destination.zone))
```

XML representation

```
 <spider name='ExistentialSpider' >
    <spidernode>
         <mutablecurve   ctrlx1='157.76'   ctrly1='181.25'   ctrlx2='214.52'
            ctrly2='94.29999999999998'   />
       <origin oindex='0'>e1</origin>
       <destination dindex='0' >e2</destination>
    </spidernode>
    <spidernode>
         <mutablecurve   ctrlx1='385.9'   ctrly1='182.48000000000002'
            ctrlx2='481.59999999999997'   ctrly2='84.96000000000001'   />
       <origin oindex='0'>e2</origin>
       <destination dindex='0' >e3</destination>
    </spidernode>
 </spider>
```

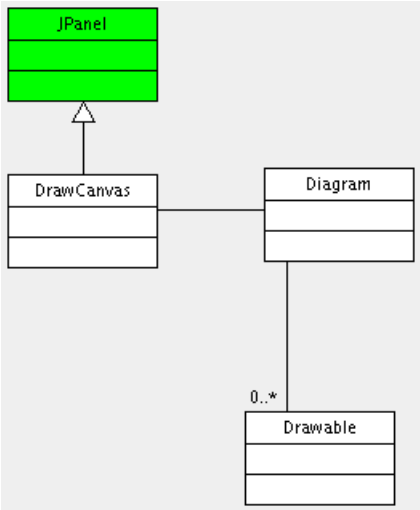



Figure 6.12: UML Drawable Diagram

#### 6.3.1.10 Diagram - package drawable

This class holds a Vector of drawable objects for each diagram. It also contain the functionality to convert a diagram from concrete to abstract.

It can output itself as an XML string, and thus when a diagram sequence is saved, the toXML() method of the diagram class is called which in turn calls the toXML() methods of all drawable objects held by it.

XML representation is in Appendix A.2.

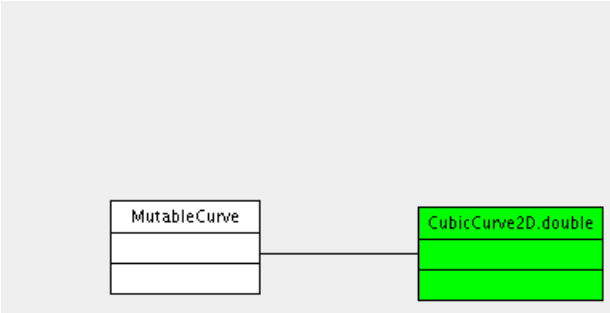



Figure 6.13: UML Drawable Mutable Curve

#### 6.3.1.11 MutableCurve - package drawable

This class allows a line with two curves to be drawn with user manipulatable control points. See 5.5.1.

XML representation

```
<mutablecurve   ctrlx1='385.9'   ctrly1='182.48000000000002'
          ctrlx2='481.59999999999997'   ctrly2='84.96000000000001'   />
```

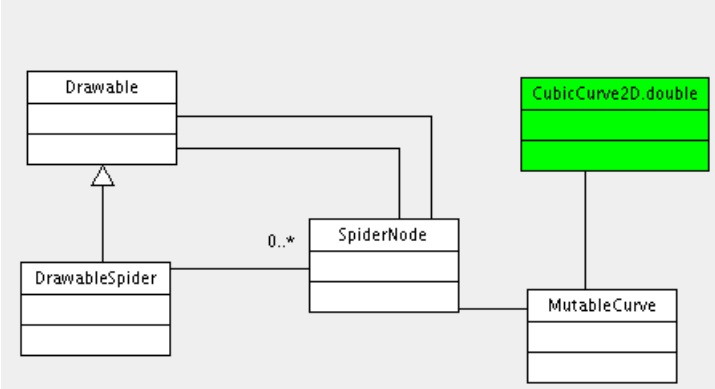



Figure 6.14: UML Drawable Spider Node

#### 6.3.1.12 SpiderNode - package drawable

SpiderNodes contain origin and destination Drawable objects and a Mutable Curve (see 6.3.1.11). This class is defined as an inner class of DrawableSpider.

XML representation

```
<spidernode>
        <mutablecurve   ctrlx1='385.9'                ctrly1='182.48000000000002'
                        ctrlx2='481.59999999999997'   ctrly2='84.96000000000001'   />
      <origin oindex='0'>e2</origin>
      <destination dindex='0' >e3</destination>
   </spidernode>
```

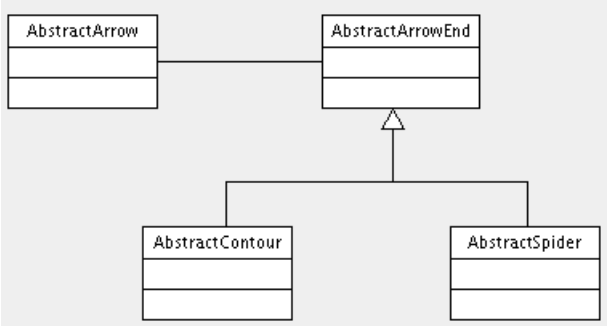



Figure 6.15: UML Abstract Arrow

### 6.3.2 The "abs" Package

This package is used to hold abstract models created from the concrete diagrams. All classes in the abs package, where appropriate, can output themselves as XML.

#### 6.3.2.1 AbstractArrow - package abs

This is a very simple class that contains an origin and destination "ArrowEnd" (see **??**). It converts itself to XML and is aware of the instance type of the origin and destination, which may be an AbstractSpider or an AbstractContour an any of the four combinations possible, thus:

| *Origin* | *Destination* |
|---|---|
| Abstract Contour | Abstract Spider |
| Abstract Spider | Abstract Contour |
| Abstract Contour | Abstract Contour |
| Abstract Spider | Abstract Spider |

Sample XML

```
<abstractarrow>
  <origin type='contour' name='set1'>
  <destination type='spider' name='spider2'>
</abstractarrow>
```

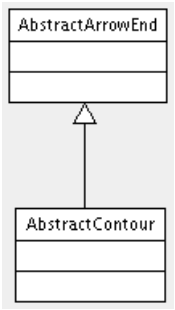



Figure 6.16: UML Abstract Contour

#### 6.3.2.2 AbstractArrowEnd - package abs

This is a very simple and small base class and is used as 'object oriented glue' to allow arrow objects to connect to both Spiders and Contours.

This has no XML representation, as it is in the Java sense of the word, an Abstract Class !

#### 6.3.2.3 AbstractContour - package abs

Again a very simple class, which contains the name of the object from the Concrete diagram (DrawableSet or DrawableType).

Sample XML

```
<contour name='set1'/>
```

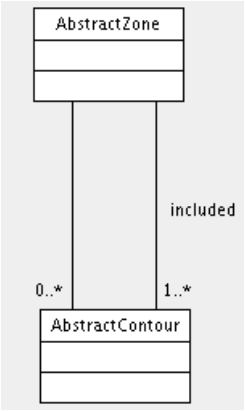



Figure 6.17: UML Abstract Zone

#### 6.3.2.4 AbstractZone - package abs

An Abstract zone is a collection of included and excluded contours. Note that the zones are not named by the creator of the diagram, but they have to be referenced by other objects.

Each zone has been, therefore, assigned a unique integer value which has scope only within its Diagram. Where this is referenced by other objects it is referred to as the *handle*. An abstract Zone may also be shaded. An example of a shaded Abstract Zone

$Type\_one \cup C$

can be seen in figure 6.2.

XML representation of AbstractZone $Type\_one \cup C$.

```
<abstractzone handle='0' shaded='true' >
  <includedcontours>
    <contour name='Type_One'/>
    <contour name='C'/>
  </includedcontours>
  <excludedcontours>
    <contour name='A'/>
    <contour name='B'/>
  </excludedcontours>
</abstractzone>
```

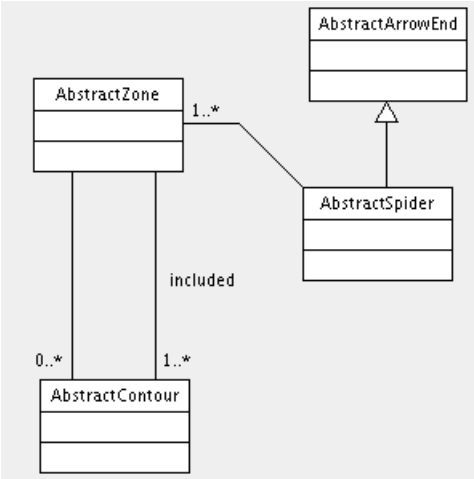



Figure 6.18: UML Abstract Spider

#### 6.3.2.5 AbstractSpider - package abs

A Spider is a collection of Abstract Zones. It has two attributes, its name and its type ( UNIVERSAL or EXISTENTIAL). Its contents are habitats which are Abstract Zones, referenced by their integer handles.

An example of a UNIVERSAL Abstract XML definition corresponding to the Universal Spider in figure 6.2 is given below.

```
<abstractspider name='Untitled_16' type='UNIVERSAL'>
   <habitat type='zone' handle='3' />
   <habitat type='zone' handle='5' />
   <habitat type='zone' handle='2' />
</abstractspider>
```

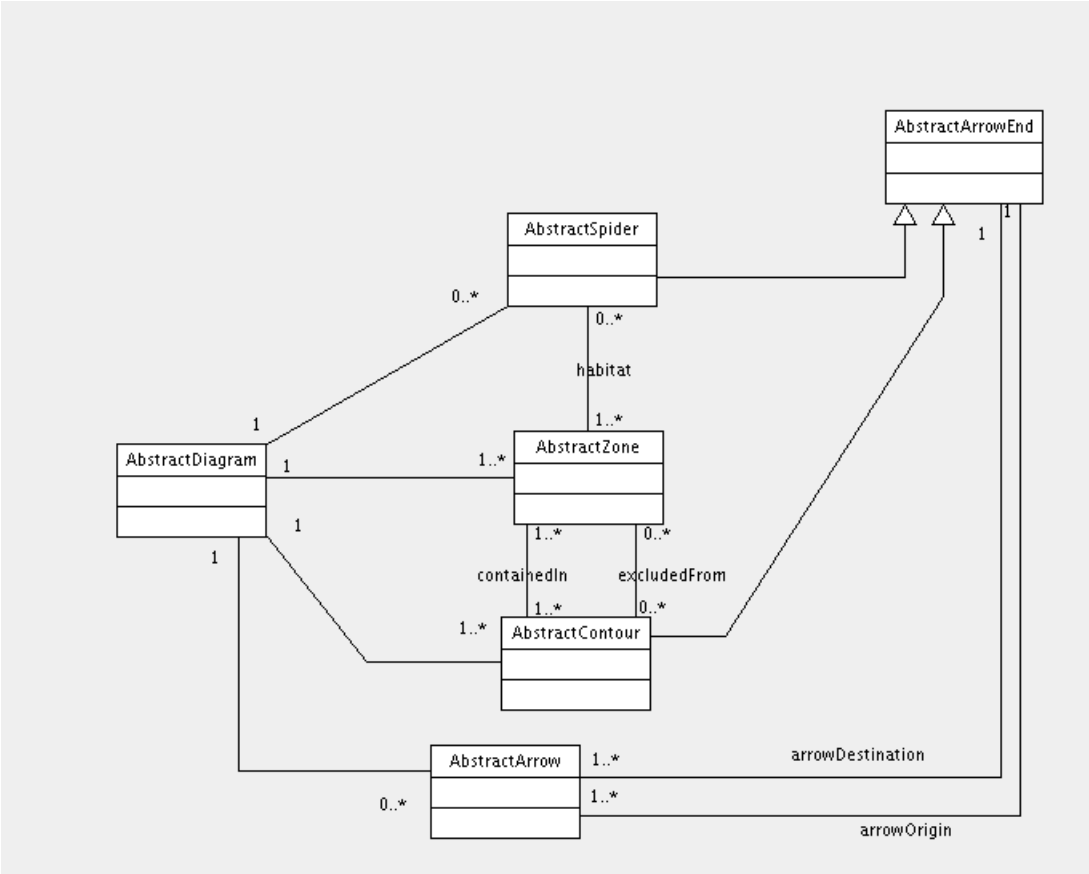



Figure 6.19: UML Abstract Diagram.

#### 6.3.2.6 AbstractDiagram - package abs

The AbstractDiagram class holds one diagram. It is the result of converting a Concrete diagram to Abstract. The Constraint Diagram shown in figure 6.2 is represented as Abstract XML in Appendix A.1.

### 6.3.3 The "ui" Package

This package is concerned with the user interface and file handling functionality.

#### 6.3.3.1 ConstraintEditor - package ui

The class contains the public static main for the standalone version of the editor.

#### 6.3.3.2 ConstraintEditorMgr - package ui

A DesktopManager that keeps its frames inside the desktop. This was slightly modified from examples provided for download from the "Java Swing" Book [2].

#### 6.3.3.3 DiagramTree - package ui

This displays a hierarchical tree of enclosure of the objects present in the diagram sequence. This may be expanded in the future to assist with finding objects to edit.

#### 6.3.3.4 LoadFromFile - package ui

The encapsulates the FileDialog class and links the requested file action to default file extensions and action.

#### 6.3.3.5 TileAction - package ui

This class takes all the JInternalFrames and tiles them across the editing area, opening any minimised windows.

## 6.4 XML and XML Parsing

### 6.4.1 XML : Writing the Concrete Model to Storage

All drawable objects can represent themselves as XML, via the toXML method in the "Drawable" base class. A Diagram object has a toXML method that goes through the entire diagram calling each of its drawable objects methods toXML method. In order to re-create the diagram with a one pass read when loading a diagram, the order in which the objects are written is important. Objects like Arrows and Spiders are anchored to Existential/Universal and Set/Type objects.

Obviously the Existential/Universal and Set/Type objects must be loaded first, since the Arrows and Spiders are defined by them.

The Diagram toXML() method therefore, provides these objects, at the beginning of the XML String it returns.

### 6.4.2 XML Parsing with SAX - Reading the Concrete Model In

There are two common XML parsers available, DOM and SAX. DOM reads in a complete XML file/String and then offers an API to traverse the document structure. The SAX Parser is event based and code based on it is potentially a little less structured.

In order to bring a little more readability into a SAX parser, I treat it as a large state machine with 'inner' states. The first XML tags have a 'state' variable according to the current object being read. To the next innermost enclosed tags I use a 'state1' variable, the next nested tags 'state2' etc.

For instance the following simple XML

```
<PEOPLE name='Fred'>
  <SONG name='Singing in the rain' />
  <SONG name='New Rose' />
  <CAR  name='Vauxhall Viva' />
  <MOTORBIKE name='Suzuki Hyabusa Turbo' >
    <FEATURES BackWheelBHP='340' topspeed='240mph' />
  </MOTORBIKE>
</PEOPLE>
```

The 'state' variable will be set to a PEOPLE object and the 'state1' variable will have SONG CAR or MOTORBIKE values. The name attribute will hold the value that the SAX parser is currently processing. While processing the MOTORBIKE tag, state2 will be set to FEATURES etc.

Without a state machine tied to the hierarchy levels of the XML nesting a SAX parser can become very untidy and difficult to maintain.

### 6.4.3 Abstract XML Production

The Abstract XML is given as a String object by calling the Method toXML() in AbstractDiagram. This method returns first a list of Contours, the list of all abstract zones defined by the concrete diagram, and then the Spiders and Arrows. This order may be important to users of the abstract xml models for the same reasons as the concrete diagram reading procedure (see 6.4.1).

## 6.5 Compiling and Building the Project

This project has produced a multi-platform Java 2 application. It has been developed under Solaris 9 on a Sun Ultra 10 and uses the unix 'make'[10] tool to organise production of different forms of the tool (various test images, runnable java byte code, exexcutable jars etc).

The list of java files is stored in a simple text file "jf". Javac is then called with "javac @jf". This is a platform wide supported compilation mode. Using 'make' in the traditional way is not.

The Makefile accepts parameters which determine the form of the Constraint Editor to build.

Parameters - Forms of the editor to Build.

- all : Compile all files as specified in the text file "jf"
- clean : Remove all old compiled 'class' files.
- jdoc : run Jdoc on all source code in all three directories.
- run : compile an image and run it immediately
- run_abs_t1 : Compile a version with the abs t1 test harness and store the test results in the text file "th1.txt"
- run_abs_t2 : Compile a version with the abs t2 test harness and store the test results in the text file "th2.txt"
- run_drawable_t1 : Compile a version with the drawable t1 test harness and store the test results in the text file "dr_th.txt"
- log : Compile and run with debug flags on. Store all debug output into the newly created text file "dd.log"
- go : Compile and then run the application.
- exjar : Create an executable jar file without source code.
- exjarsrc : Create an executable jar file, with javadoc and source code.

Each Makefile entry could be extracted and converted easily into a batch file for another operating system (such as Windows). For instance the entries that create a Source Executable Jar file are :-

```
exjarsrc: clean jdoc all
echo "Main-Class: ui.ConstraintEditor" > CsEdManifest.mf
jar mcf CsEdManifest.mf ConEdandSRC.jar */*.html ui/*.class ui/*.java abs/*.class abs/*.java drawabl
```

The make file ensures that all 'old' class files are removed, runs javadoc, compiles the project, and then uses the jar command to collate it into the executable jar.

For a batch file commands from the other makefile targets (labels) could be extracted and then included in a batch file.

### 6.5.1 Example - Compiling and Building the Project

Thus in a unix environment, with the current directory set to the top of the source tree typing

```
ultra10:constraint_editor% make exjarsrc
```

Will build an executable jar file including source code.

# Chapter 7

# Implementation Issues and Special Algorithms

This chapter concentrates on algorithms used to solve problems with concrete diagram verification (for well formed-ness) and the concrete to abstract conversion process.

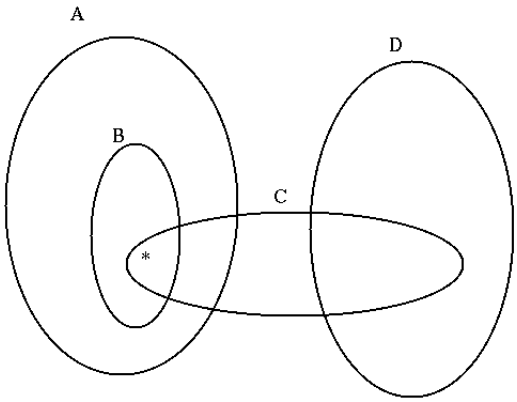



Figure 7.1: Zone Definition for a Point Object

## 7.1 Algorithms to determine Relationships between Sets

### 7.1.1 Determining the Zone of a Point Object

A Point object (such as an existential or universal qualifier) is defined by screen x, y coordinates. By searching through all sets (which all represent themselves as Java Polygons) in the diagram using the Java 'contained'[1] method a list of all included contours can be obtained. All other contours in the diagram are added to an excluded list.

Thus for the universal in diagram 7.1 the Zone it exists in is defined by:

$$A \cup B \cup C$$

In the model and in XML, this will be a list of included and excluded contours thus :

```
<zone>
  <included>
    <contour name=A>
    <contour name=B>
    <contour name=C>
  </included>
  <excluded>
    <contour name=D>
  </excluded>
<zone>
```

Note that the XML has an implied meaning in that the included elements are the intersection of those elements, and the excluded contours are there for completeness.

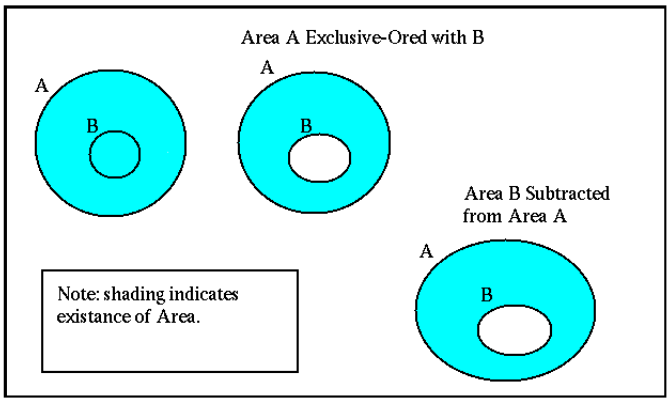



Figure 7.2: Enclosure Algorithm Case where A encloses B

### 7.1.2 Algorithm for Determining Enclosure of Drawable Objects

For a Mathematical discussion on Enclosure see "Nesting in Euler Diagrams"[17].

Java allows a Polygon object to be converted into a Shape[2] and a Shape object to be converted into an Area[3]. The Java Area class is abstract in the sense that it allows high level geometric operations such as intersection, exclusive-or and subtraction of shapes. Often these operations can cause an Area to be split into several Area objects, so care must be taken to deal with them all.

Using the Geometric methods provided by the Area class, it is easy to determine whether a Drawable object encloses another using the following algorithm.

```
-- Pseudo code for determining
-- if one shape encloses another

For all drawable elements a
  For all drawable elements b
    where a != b and a intersects_in_some_way b
      if ( a XOR b == a SUBTRACT b )
addtoContainmentRelationlist(a,b)
```

This results in the shapes shown in figure 7.2. Note that the $A\ XOR\ B$ shape is the same as the $A - B$ shape.

This means that A encloses B.

Notes:

[1]Polygon.contained(x,y) [21]

[2]java.awt.Shape

[3]java.awt.geom.Area

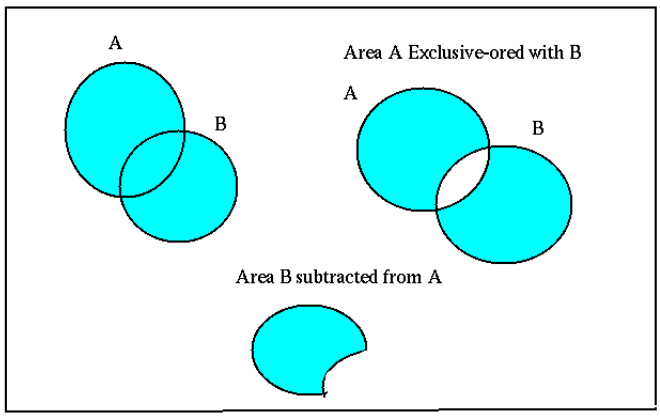



Figure 7.3: Enclosure Algorithm Case where A intersects not encloses B

### 7.1.3 Algorithm for Determining Pure Intersection of Drawable Objects

Pure Intersection, means an intersection where one element does not include the other, but intersects. To determine this the following algorithm is used, again using the Java Area class. See figure 7.3

```
-- Pseudo code for determining
-- if one shape encloses another


For all drawable elements a
  For all drawable elements b
  where a intersects_in_some_way b
      if ( ( a intersection b != b )  AND ( b intersection a != a ) )
        addtoPureIntersectionRelationlist(a,b)
```

This results in the shapes shown in figure 7.3.

Note that these are different and indicate that neither A encloses B nor B encloses A.

### 7.1.4 Algorithm for Enclosure Tree

Using the enclosure relations (see 7.1.2) each contour is parsed and its enclosure path followed in the unordered relations. A tree 'height' is then determined for each element and stored in the Drawable object. For example a contour included within a contour has a height of 0 and a contour not enclosed has a height of 1. The diagram representing 10 zones (see figure 7.5) is shown as a hierarchy tree in figure 7.6.

In the GUI a Tree is then built from this height information using the Java Library JTree class. For zone discrimination only the tree heights are required.

The routine to determine all hierarchical heights uses a recursive helper function which digs down to the bottom of the tree (the most enclosed element) and sets the hierarchical height values in the tail of the recursion. This is best expressed in Pseudo code.

```
BEGIN determineHierarchy( enclosure_relations )
  For All enclosure_relations enc

   // set the Drawable objects Heirarchical Hieght
   //
   enc.origin.HierarchicalHieght =
     MAX (  enc,
            determineHeight ( enc.origin.HierarchicalHieght,
                              enclosure_relations )

  End For All
END

// recursive Helper function
//
BEGIN determineHieght ( enc, enclosure_relations )
  return_value = 0
  For All enclosure_relations enc2
    IF ( enc enclosed by enc2 )
      v = determineHeight (enc2,enclosure_relations)
      return_value = MAX(v,value)
    FI
  END for ALL

  return value + 1
END
```

## 7.2 The Problem of determining All Zones Present In an Abstract Diagram : FZD

In order to solve problems where containment or enclosure of zones is required, it is important to have a complete list of all zones in a diagram. All zones present must be entered into the abstract XML. For clarity all Contours should be listed in the Abstract XML as well.

For instance in 7.4 a universal qualifier in the set A defines by an arrow the contents of set B. Set C is included in set B. This means that all objects in set A are related to all objects in set C, because C is

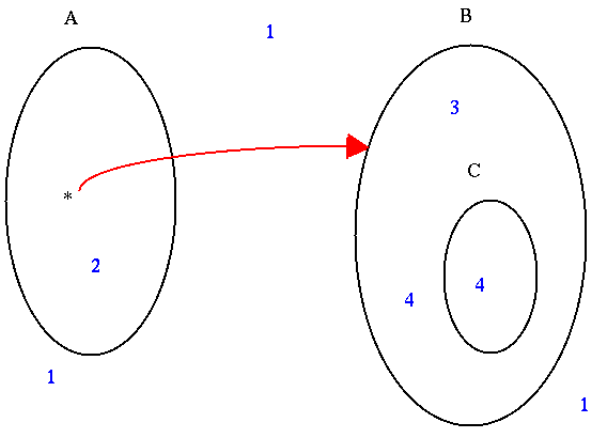



Figure 7.4: Universal defines a Set and its included Set

enclosed by B. This information must be available to the abstract model. Although there are 8 possible zones (see 4.2.2), in this diagram we have less. To list them explicitly :-

| *Included* | *Excluded* | *Zone Number* |
|---|---|---|
| { } | { A B C } | 1 |
| { A } | { B C } | 2 |
| { B } | { A C } | 3 |
| { B C } | { A } | 4 |

Note that by not including the following entries (the other possible zones not present in the diagram from the $2^n$ combinations), it is implied that set C is contained by set B.

| *Included* | *Excluded* | *Zone Number* |
|---|---|---|
| { C } | { A B } | 5 |
| { C } | { A } | 6 |
| { C } | { B } | 7 |
| { C } | { } | 8 |

Thus a complete zone definition (including what is not there) is necessary to correctly interpret an abstract diagram.

### 7.2.1 Exponential number of zones

Because the potential number of zones is $2^n$, it would be impractical to check every possible combination.

Although the Java Area Shape and Polygon classes allow the Java libraries to do the hard work, a large diagram with 32 sets with an average of 16 checks per combination, would take $2^{32}.16 == 68719476736$ shape and area comparisons. This would be impractical. Given this may take 1 ms per check this diagram

would take 2 years to determine all zones existing in the diagram !

A similar problem of processing time exponentially rising with the sample size was seen in frequency analysis. The DFT algorithms processing time was proportional to $2^N$ and the FFT to $2.Log_2N$. As computers were used for frequency analysis routinely the faster but more complicated FFT algorithm became the only one used in practice[14]. Clearly examining for all these zones by brute force is going to be impractical for real world diagrams containing many sets.

A solution to this arises from the the fact that if we have an intersection, which is not enclosed on an object which is contained by another set, we can define the zones arising from this. For example in figure 7.5 the apparent intersection of $W \cap Z$ can be described completely, because it is enclosed by both $Y$ and $X$. Adding these encloser contours, the zone is fully described as:

$$W \cap Z \cap X \cap Y$$

Two more checks per contour are then required. Checking for objects totally enclosed, and then objects not enclosed by any other contours. This determines every zone defined by the concrete diagram without having a ' brute force $2^n$ search ' to check every possible combination. This algorithm will be described in greater detail and with a full worked example in subsequent sections.

I would like to term an intersection that does not feature enclosure by either set as a pure intersection (or *pi* to describe unordered relations sets in the following examples).

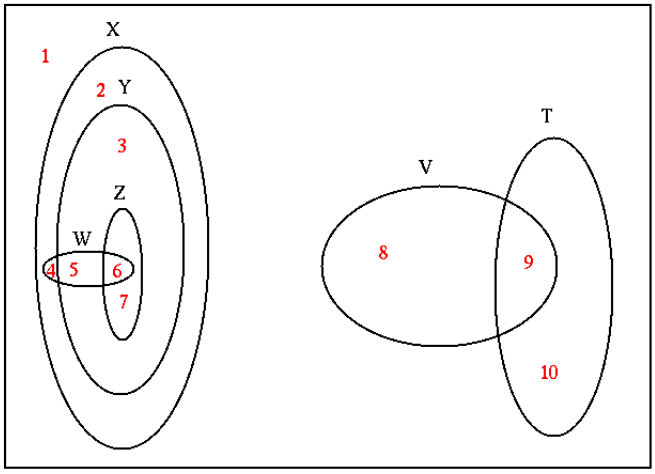



Figure 7.5: Pure Intersection Zone Capture Method

### 7.2.2 Description of Fast Zone Discrimination (FZD) algorithm

In order to follow through the worked example of the algorithm in section 7.2.2.2 a Description of Zones from observation of figure 7.5 is presented here

| *Included* | *Excluded* | *Zone Number* |
|---|---|---|
| { } | { X Y Z W V T } | 1 |
| { X } | { Y Z W V T } | 2 |
| { X Y} | { Z W V T } | 3 |
| { X W} | { Y W V T Z } | 4 |
| { X W Y } | { W V T Z } | 5 |
| { X W Y Z} | { V T } | 6 |
| { X Y Z} | { W V T } | 7 |
| { V } | { W X Y Z T } | 8 |
| { V T } | { W X Y Z } | 9 |
| { T } | { W X Y Z V } | 10 |

Using the algorithm described in 7.1.2 obtain unordered containment relations

$$\begin{aligned} X &\xrightarrow{c} Y \\ X &\xrightarrow{c} Z \\ X &\xrightarrow{c} W \\ Y &\xrightarrow{c} Z \end{aligned}$$

Using the algorithm described in 7.1.3 obtain unordered containment relations

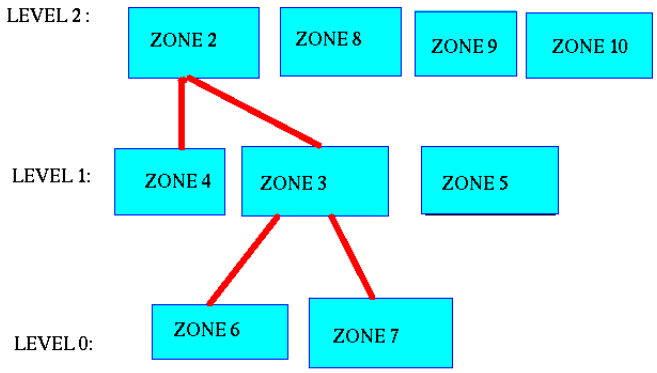



Figure 7.6: Hierarchy of the 10 Zone Diagram

$$
\begin{aligned}
Y &\xrightarrow{pi} W \\
Z &\xrightarrow{pi} W \\
W &\xrightarrow{pi} Y \\
W &\xrightarrow{pi} Z
\end{aligned}
$$

From the containment relations ( $\xrightarrow{c}$ ) a tree is built of containment relationships. The algorithm for building the tree is described in 7.1.4.

$$
\begin{aligned}
X &\xrightarrow{t} Y \xrightarrow{t} Z \\
X &\xrightarrow{t} W
\end{aligned}
$$

In practice, it is not necessary to build an actual tree, as long as the hierarchical heights are available to the zone discrimination routine (see 7.2.2.1).

#### 7.2.2.1 Pseudo code for FZD Algorithm

The Algorithm for determining the zones from the relations and the containment tree is described below in pseudo code.

```
PSEUDO CODE
BEGIN
   FOR ALL Contours

     For All Pure Intersections
       Add Pure Intersection to included list
       Follow through the containment relations
         (using the tree) adding countours to included list
       Add non included contours to Excluded List
       Add Zone to AbstractDiagram

     For All Enclosures (i.e. contours that enclose this)
       Recursively follow back through containment relations
         adding countours to included list.
       Add non included contours to Excluded List
       Add Zone to AbstractDiagram

     If not Enclosed
       Add single contour entry to a Zone
       Add all other zones to excluded
       Add Zone to AbstractDiagram

   END FOR ALL
END
```

#### 7.2.2.2 Worked Example of FZD algorithm

Following the worked example through, the first contour to be processed is X.

PROCESSING CONTOUR X

- This has no pure intersections with other contours.
- It is not enclosed by any contours.
- Therefore make a simple Zone $\{X\}\backslash\{YZWVT\}$. This determines Zone 2.

PROCESSING CONTOUR Y

- Y has a pure intersection with W. A temporary Zone is created initially with $\{WY\}\backslash\{\}$ Y is now examined for containment relationships using the tree. Y is enclosed by X. X is therefore added to the included list. The zone becomes $\{WYX\}\backslash\{\}$. X to be enclosed by another set this would be added to the included list also. All non included contours are now added to the zone which becomes $\{WYX\}\backslash\{ZVT\}$. This defines Zone 5.

- Now looking through at the tree for enclosure relations for Y, the Zone $\{YX\}\backslash\{WZVT\}$ is created. This defines Zone 3.

- As Y is an enclosed contour, no simple one contour zone is made for it.

PROCESSING CONTOUR Z

- Z has a Pure intersection with W. A temporary Zone is created with $\{WZ\}\backslash\{\}$. Z is now examined and found to be contained by Y which in turn is contained by X. These are added to the included list thus $\{WZYX\}\backslash\{\}$. Now the non included contours are added to the excluded list giving the completed zone as $\{WZYX\}\backslash\{VT\}$. This defines Zone 6.

- Looking through for enclosure relationships with Z will give $\{ZYX\}\backslash\{VT\}$. This defines Zone 7.

- As Z is enclosed no simple one contour zone will be made.

PROCESSING CONTOUR W

- W has pure intersections with Z Y and X. However because only W is examined for the containment tree only one duplicate Zone will be attempted (because the containment tree will be searched for W). This will produce $\{WX\}\backslash\{YZVT\}$. This defines Zone 4.

- W is enclosed by X and a duplicate Zone 4 will be offered to the AbstractDiagram class.

- As W is enclosed no simple one contour Zone will be created.

PROCESSING CONTOUR V

- This has a pure intersection with T, and is not in the containment tree. It will therefore produce $\{VT\}\backslash\{XYZW\}$. This defines Zone 9.

- V is not enclosed so no enclosed zone will be created.

- V is not enclosed therefore a simple one contour set will be produced for it $\{V\}\backslash\{XYZT\}$. This defines Zone 8.

PROCESSING CONTOUR T

- This has a pure intersection with V, and is not in the containment tree. It will therefore produce $\{TV\}\backslash\{XYZW\}$. This will be considered a duplicate zone by the AbstractDiagram ( Zone 9 ). class and it will only store one copy.

- T is not enclosed so no enclosed zone will be created.

- T is not enclosed therefore a simple one contour set will be produced for it $\{T\}\backslash\{XYZV\}$. This defines Zone 10.

Thus all zones (save the implicit Zone 1) have been defined by the FZD algorithm, with only two duplicate zones being offered for storage in the AbstractDiagram. For six contours as in this example, a brute force search would have necessitated using the Java area classes on 64 combinations of contour combinations.

For complicated diagrams where there are large numbers of Contours a brute force $2^N$ search will be impractical.

## 7.3 Shading, determining the included and excluded contours

The shading uses the point algorithm described in 7.1.1 to determine which abstract zone has been chosen with the mouse. However, this shaded Zone is dynamic in that the contours that defined it may be moved by the user. This forces a decision, if the shaded zone is made to disappear, should it be removed from the concrete diagram ? Currently as the abstract diagram will not record non existent zones, I have left the definition in the concrete diagram, and in its saved XML. This means that diagrams which later move contours to create the zones, will still find them shaded when they again exist.

# Chapter 8

# Looking Forward, possible extensions

The tool created can produce constraint diagrams and postscript output suitable for inclusion in documentation. The Abstract Diagram in XML can be passed on for further processing with other as yet undefined tools.

## 8.1 Direct Mathematical Output

All the ingredients are here to directly produce mathematical output defining the systems drawn. Java modules could be added which could take the abstract models and calculate mathematically correct statements.

A pre-requisite of conversion to OCL or Z would be that the abstract diagram has been calculated/converted and is well formed. The output could be in the form of a text file, or pop-up windows that appear when a drawable item is selected, describing it in Mathematics or OCL.

## 8.2 Mathematical Operations Toolbar

The Model could be manipulated with standard mathematical operations used in Set theory to derive new diagrams. For instance OR conditions (individual spider habitats could be removed) leaving diagrams with less detail but still 'correct'.

The Book "Software Engineering Mathematics"[8] has rich examples of using mathematical rules to alter and refine schemas. The rules and examples from this could be used as a starting point to choosing the operations to be implemented.

## 8.3 Sequences of Constraint diagrams

Currently this editor holds sequences of diagrams, and saves concrete and abstract files as sequences. However there are no relationships between objects in each diagram. If an object is changed in one diagram it will not affect an element of the same name and type in another drawing.

Sequences could be made that share objects and thus allow editing of an object to change it in all diagrams in the sequence.

## 8.4 Using Constraint Diagrams to model failure mode of components

Consider a system where basic components have given sets of failure modes. These components are combined to create a sub-system. The sub-system will have a set of failure modes which will depend upon combinations of the failure modes from the components.

Consider the following components with sets of failure modes thus:

- Type $\alpha$ having failure modes { $\alpha_{ok}$ $\alpha_A$ }
- Type $\beta$ having failure modes { $\beta_{ok}$ $\beta_A$ $\beta_B$ $\beta_C$ }
- Type $\gamma$ having failure modes { $\gamma_{ok}$ $\gamma_A$ $\gamma_B$ }

Imagine a system $\psi$ that is built from one of each of the above components, that has a set of failure modes, say A, B, C and D.

$$\psi_{states} \triangleq \psi_{ok}\ \psi_A\ \psi_B\ \psi_C\ \psi_D$$

These failure modes are due to combinations component failure modes. If all components are "ok" then the sub-system will have a state "ok".

$$\psi_{ok} \triangleq \alpha_{ok} \wedge \beta_{ok} \wedge \gamma_{ok}$$

Now lets makeup some more rules for these failure modes for this example. Lets say

$$\psi_A \triangleq \alpha_A$$

$$\psi_B \triangleq \alpha_{ok} \wedge ( \beta_A \vee \beta_B ) \wedge \gamma_B$$

$$\psi_C \triangleq \alpha_B \wedge ( \beta_A \vee \beta_B \vee \gamma_B )$$

The definition of $\psi_D$ implies a default state, it could be described as being the default not OK state.

$$\psi_D \triangleq \neg\, \psi_{ok} \wedge \neg ( \psi_A \vee \psi_B \vee \psi_C)$$

This can be more clearly represented by a Constraint/Spider diagram with the addition of a default state/spider, that of $\psi_D$. See figure 8.1. Without the default state/spider all failure modes would have to be represented on the diagram and a very large spider drawn to represent it.

The facility of having a default spider will help in producing practical models of modules. Imagine a power supply for instance. For most combinations of failure modes the power supply will simply not work, but for some selected combinations of failures it will behave in incorrect but well defined modes. Note this does not concern itself with the reliability of the system(i.e. mean time between failure etc) but with possible resultant states of the system.

In tabular form, or as a Karnaugh Map each of the $2^N$ states would have to be represented; in this case $2^9$ (512) entries. The constraint diagram with an added 'default spider' is much easier to understand.

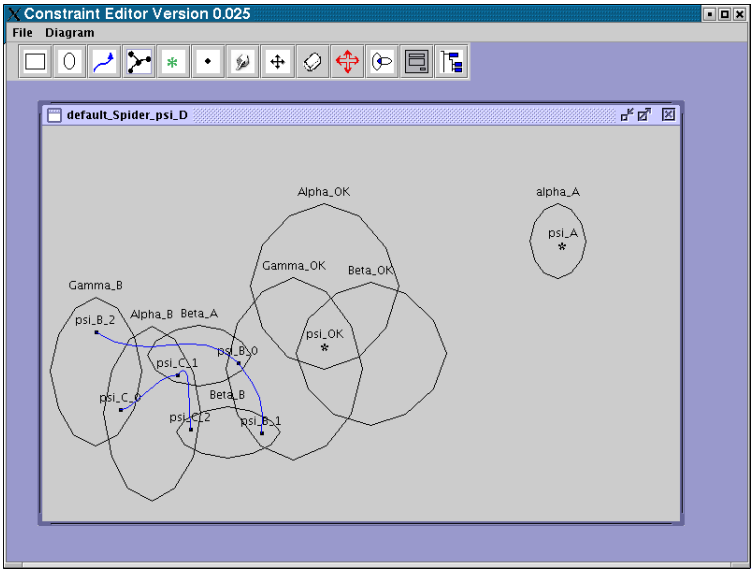



Figure 8.1: Component to Module Failure Mode Diagram

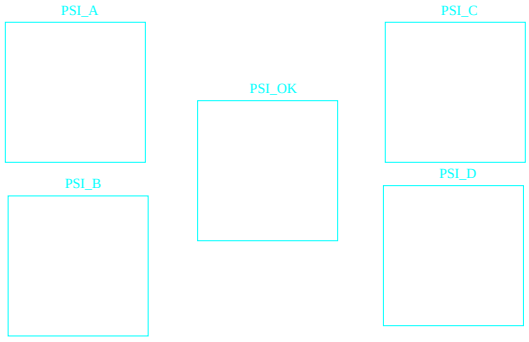



Figure 8.2: Component to Module Failure Mode Diagram

## 8.5 Hierarchical Ordering of Constraint Diagrams

If Constraint diagrams can be defined to have a set of logical outputs, constraint diagrams could be nested. The internal working of each nested object would not be affected by the objects that incorporate them. This would allow separate systems and modules to be developed and then plugged into a higher level diagram. For instance the module developed in 8.4 can be converted into a constraint diagram consisting of sets which represent the failure modes of the module $\psi$ see figure 8.2.

An example using a simple control system is given in section 8.6.

## 8.6 Background - Safety Critical Modelling

### 8.6.1 Components

Currently safety critical control systems are built from components with known failure modes. For instance, resistors can be purchased that only fail by going open circuit.

Electrolytic Capacitors, however, can fail by both shorting and going open circuit.

### 8.6.2 A Typical Safety Measure

To give an idea of measures already in place in safety critical systems such as Gas Burner controllers, consider the independent watchdog function, implemented in hardware.

If a burner controller's micro processor (or the software running on it were to fail) the system could obviously become unsafe. So in the event of microprocessor failure, a watchdog system will (with a tight time window) will wait for a pulse to indicate all is well every say, 50 ms. The watchdog could be a retriggerable monostable or a another microprocessor. Should no pulse arrive, or a pulse arrive outside the time window, the watchdog processor will shutdown the system using relays to main power independent of relays controlled by the main processor. Note that the watchdog processor must also have its own clock. This is because if they both ran from the same clock signal, a single failure, the clock/oscillator, could stop the system from shutting down.

### 8.6.3 Safety Standards

Before EN298 regulations[25] a safety control system for industrial gas burners had to react to one component failure and safely shutdown the system. If we consider a system which has a total of $N$ failure modes this would mean checking a maximum of

$$NumberOfChecks = N(N-1)$$

for individual component failures and their effects on other components when they fail. For a small system with say 1000 failure modes this would demand a potential of one million checks for any automated checking process.

New European legislation[**?**] directs that a system must be able to react to two component failures and not go into a dangerous state even if the system was shutdown. Also that the system must be able to react to two simultaneous components failing.

This raises an interesting problem from the point of view of formal modelling. Here we have a binary cross product of all components. This increases the number of checks greatly. Given that the binary cross product is $(N^2 - N)/2$ and has to be checked against the remaining $(N - 2)$ components.

$$NumberOfchecks = \frac{(N^2 - N)(N-2)}{2}$$

Thus for a 1000 failure mode system, roughly a half billion possible checks would be required for the double simultaneous failure scenario ! Clearly this massive task needs breaking down.

Modelling zone intersections and all possible binary intersections in a graphical tool, could allow the user to break the system down into parts which have a set of given reactions however they fail internally. This should dramatically reduce the number of cross checks necessary.

This would be a process of identifying the sub-systems, and defining the inputs and outputs and thus reduce the number of binary cross products to check. For instance a D.C. regulated power supply module could fail internally in perhaps hundreds of ways, but the results could be apparent normal operation with a failure flag, no power or A.C voltage at a maximum of $V_{max}$ volts. This power supply module can then only apply four conditions to the models that rely on it, rather than hundreds.

This would be an application for constraint diagrams, or a variation thereof. It does seem very likely that some type of software tool to help with these 'double simultaneous component failure scenarios' will become a necessity.

## 8.7 Failure Mode Modular De-Composition

Imagine a complicated system built from components and modules. The modules are combined to form sub systems and the sub-systems combined to form the final system. If each module/sub-system is modelled and has a finite number of failure modes the massive number of checks to be considered is reduced dramatically.

For instance the failure of a capacitor in the power supply producing a high ripple noise level does not need to be considered for every other component. Rather it has become another well defined failure mode of the power supply, which must be considered in relation to how that affects other modules. Most modules (say a heating element) would not consider this failure mode as a fault.

This means that each spider diagram will derive a new constraint diagram, which contains sets for all the spiders defined in the original diagram. This derived constraint diagram can then be used to interact with other modules in the system that depend upon it.

In figure 8.3 three modules $\psi$ $\chi$ *and* $\phi$ are developed from components. Each of these produce a new constraint diagram which simply holds the failure modes. The modules $\psi$ *and* $\chi$ are combined to produce a new constraint diagram $\tau$. This produces a new constraint diagram representing the fault modes of $\tau$ and thus becomes a module/sub-system. The entire system is then defined by combining $\tau$ *and* $\phi$ and the results of that spider diagram create a new constraint diagram consisting of the failure modes of the entire system.

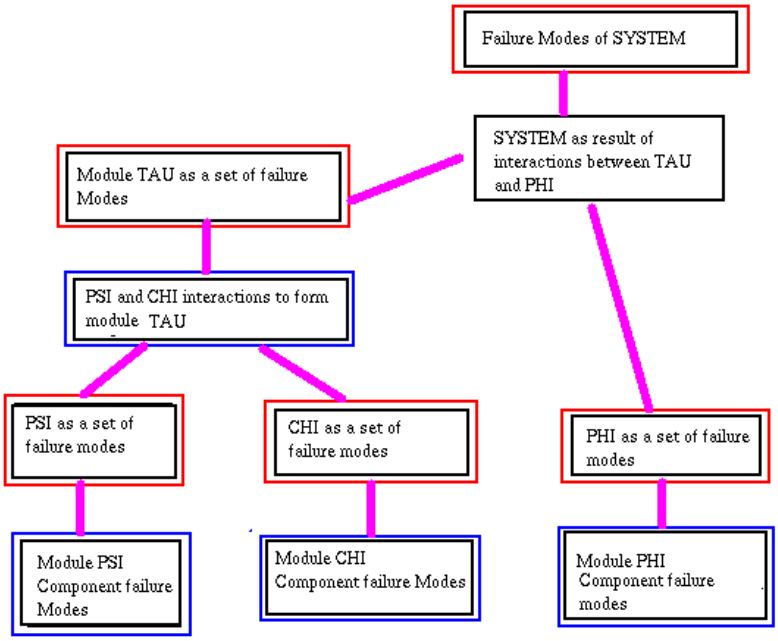



Figure 8.3: Formal Modular Dependency De-Composition

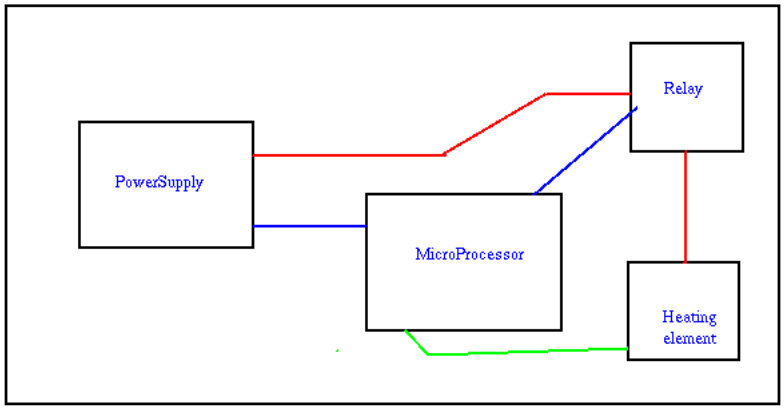



Figure 8.4: Simple Heater Control System

### 8.7.1 Modification to Diagram Sequence Development

Consider a diagram representing say a power supply. After determining all failure modes of all components and how they interact, spiders can be drawn showing the failure modes.

These spiders can then become sets for other diagrams that use the functions of the power supply and react to the failure modes.

In other words a diagram is automatically produced from a constraint diagram, where all spiders in the original diagram become sets for use by dependent systems. The failure modes can then have intersections with the systems that use it and the entire system behaviour can be modularised.

### 8.7.2 Simplified Example of a Hierarchical Constraint Diagram

In order to run through a example to show modularisation of a safety system, consider a very simple heater controller. It has a power supply that provides AC for the heating element and DC for a micro processor which uses an on board ADC to measure the temperature and a relay to turn on/off current to the heater. See figure 8.4.

Taking each of these modules in turn (in a simplified and incomplete way).

#### 8.7.2.1 The Power Supply

This simplified power supply can fail in 4 ways.

- Normal Operation
- Supply No Current (a default condition if no other is true)
- Supply AC but no DC to the microprocessor
- Supply DC but no AC
- Supply the wrong voltage (mains) to the entire circuit

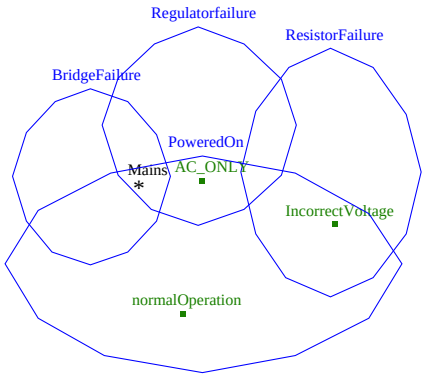



Figure 8.5: Power Supply Components

These are derived from the spider diagram in figure 8.5. Note that "No Current" is the default spider for this diagram.

Each of these failure modes can be used as a set to determine the behaviour of the systems which rely on it. A conversion process could be used to create a diagram to model other circuit elements dependent upon it. The diagram in figure 8.6 shows this after 'conversion'.

The next stage is to take a system which depends upon the power supply. The Microprocessor is the obvious choice. This will function correctly when given DC, and will be damaged by mains voltage. It will not care if the AC voltage (for the heater element via the relay) is present.

The Relay and Heating element depend on both the power supply and the microprocessor, and is dealt with later.

#### 8.7.2.2 The Micro Processor

This requires stable DC in order to operate. Also internally the ROM, RAM, IO or ADC could fail. However, to simplify this example the internals of the microprocessor will not be dealt with.

Combining the diagrams, gives us two microprocessor modes. OFF, and RUNNING. These spiders (see figure 8.7) can now be converted into a constraint diagram where again each spider is represented as a set. This is displayed in figure 8.8.

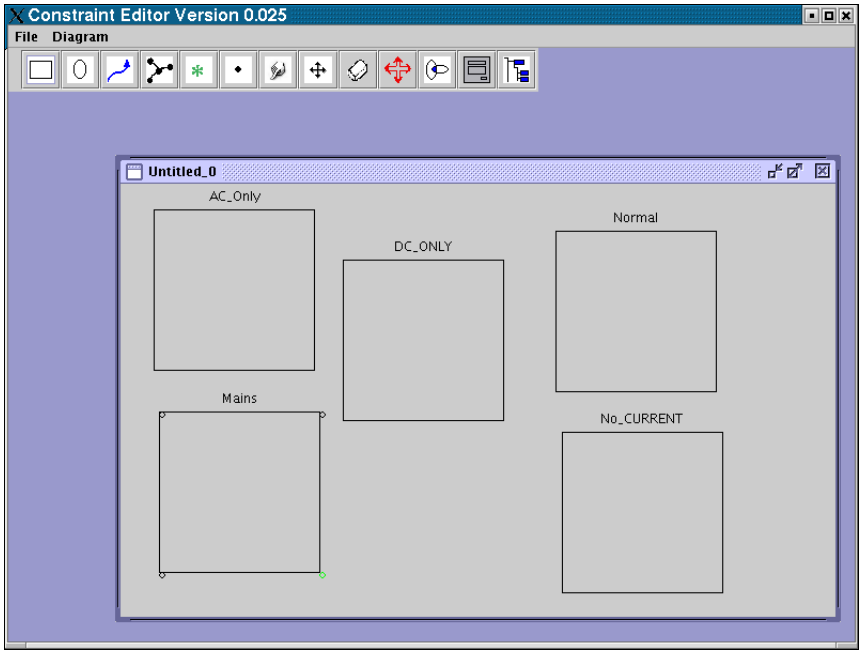



Figure 8.6: Power Supply Spiders Converted into Sets for Further Modelling

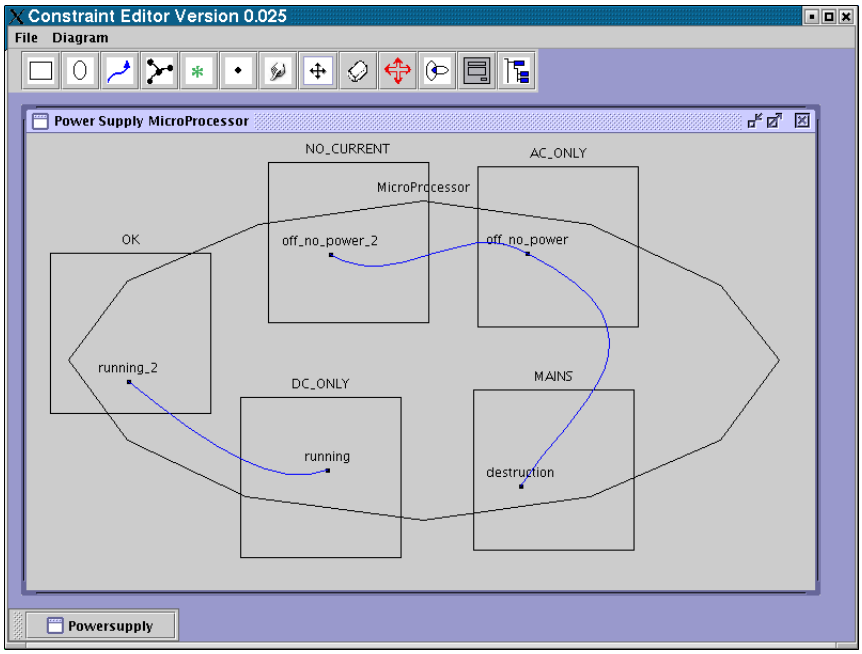



Figure 8.7: Power Supply Failure Modes Combined with Microprocessor Operation

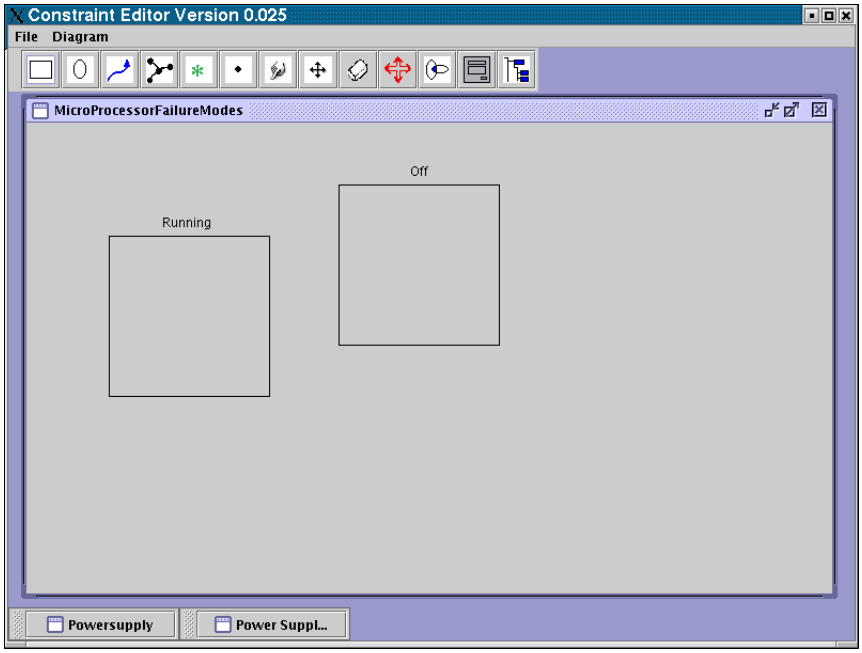



Figure 8.8: MicroProcessor Failure Modes

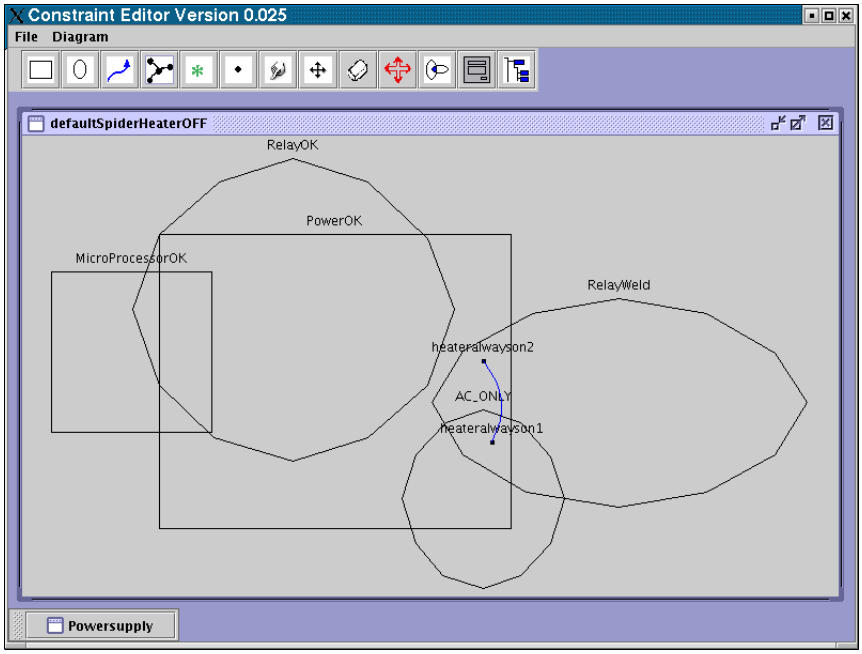



Figure 8.9: Spider Diagram combining Power Supply, Micro Processor and relay/heater

#### 8.7.2.3 The Relay and Heating Element

The Relay can fail in two modes, it can weld itself ON, or it can fail to respond to a TURN ON signal from the microprocessor IO line. The Heating element can go open circuit and thus stop functioning. The relay depends on both the Microprocessor and the power supply (the A.C. part anyway) to function correctly.

A constraint diagram for the relay/heating element combination with a default spider off "OFF" or "NO_HEAT" can thus be constructed. See figure 8.9.

Thus when the derived constraint diagram for the relay and heating element (see figure 8.10) is combined with the Microprocessor and the Powersupply A.C. Part, we will have a failure modes definition for the entire system.

The fact that HeaterAlwaysOn is one of the states possible from a single failure should be seen as quite alarming ! In Mathematics this fault could be described as

$$HTR\_ALWAYS\_ON \triangleq RelayWeld \wedge ( PS\_AC\_ONLY \vee PS\_NORMAL\_OPERATION )$$

The modular process has proved that this combination is possible. How likely it is to occur is another matter and is beyond the scope of this project. The methods described here are designed to comply with the single and double failure of components leading to unsafe conditions, as described in European Legislation for the safety of Gas burner systems. [25].

#### 8.7.2.4 Combining all Diagrams - Comparing with cross product of all components

This idea represents bottom-up modular decomposition of interacting systems reducing the number of cross product checks required. This spider diagram (8.9) shows that one single fault[1] and several obvious double faults can cause this system to fail.

Notes:

[1]Relay Weld, in practice relays are chained and controlled by separate fail safe systems

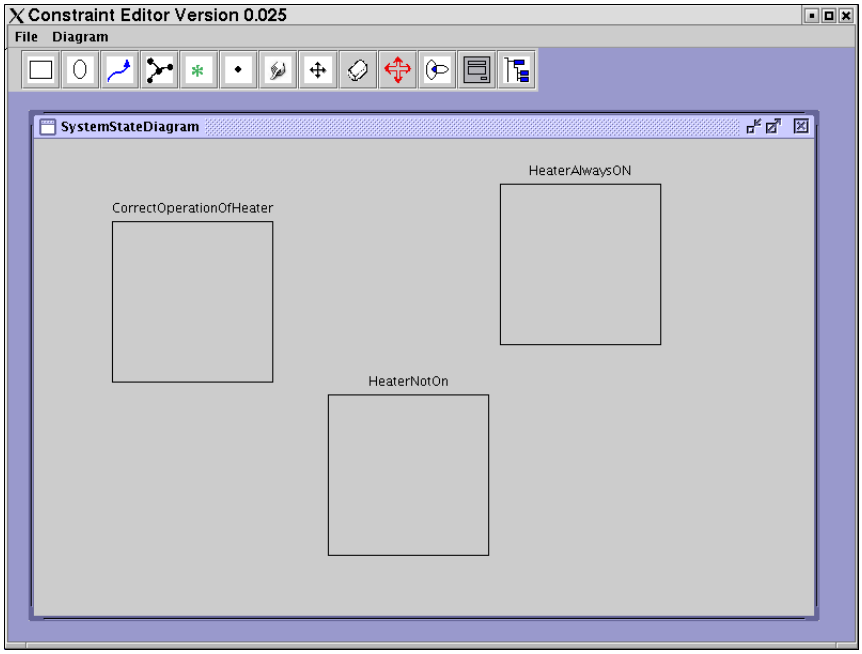



Figure 8.10: Derived Diagram representing possible System States

## 8.8 Extending Failure Mode Modular De-Composition to Double Simultaneous Component Failure Scenarios

For double failure mode the same bottom up technique can be used but checking for double failures within the components. This will probably normally create some extra failure modes per module (i.e. the binary cross product of failure modes will generally be larger than the single failure modes).

The double failure modules will still be separate entities, but more rigorously checked.

These double failure modules will produce derived constraint diagrams, which when combined with other modules, will also undergo double failure mode checking. Obviously this will involve more work and checking than the single failure models, but will not require the astronomical number of checks demanded by a binary cross product of all component failure modes.

# Chapter 9

# Testing and Verification

Test constraint diagrams have been used to test features of the editor and are saved in the */testing/* directory on the CD. Some special "Java Test Harness" classes have been written, and these are placed in their appropriate packages. They are always pre-fixed TH.java or THN.java (where N is an integer) to denote they are test files, and they always include a "public static main" method for invocation, and set the debugLevel to 10000 to ensure output of actions and relevant variables. Makefile entries automatically run and compile them, and in the unix environment leaves test result log files in text format in the current directory.

This chapter lists all, and describes some of the more important tests and test harnesses.

## 9.1 Testing Across Java Enabled Platforms

The executable output of this project is an executable Jar file. This should run correctly on any Java 2 enabled platform. Testing has been performed on three Platforms; Sun OS Solaris 9 on an Ultra 10, Linux RedHat 7.3 on an Athlon based PC, and Windows 98 on A Pentium III based PC. This OS testing tries to ensure the Jar will run correctly on all these platforms. Thus all tests from section 9.2 should be carried out on each platform.

## 9.2 All Drawable Types and Combinations

### 9.2.1 Existential

- Create a Diagram and place an Existential point in it.
- click on the Move Icon. Select the Existential Point. Move it by mouse dragging. It should move on the screen.
- Click on the Info icon on the menu bar. Click on the existential Object. De-Select the "Show Label" check box. Press Apply. The Label on the diagram should disappear.
- Click the delete Icon. Place the mouse over the Existential Point and click. A pop-up window should appear. Confirm the delete action. The Existential point should disappear form the diagram along with the info pop-up.

### 9.2.2 Universal

- Create a Diagram and place a Universal point in it.

- click on the Move Icon. Select the Universal Point. Move it by mouse dragging. It should move on the screen.
- Click on the Info icon on the menu bar. Click on the existential Object. De-Select the "Show Label" check box. Press Apply. The Label on the diagram should disappear.
- Click the delete Icon. Place the mouse over the Universal Point and click. A pop-up window should appear. Confirm the delete action. The Universal point should disappear from the diagram along with the info pop-up.

### 9.2.3 Set

- Create a diagram and choose the Set Icon from the menubar. Create a Set and Name it. A 12 point Oval should appear on the Diagram.
- Select Resize. Click on one of the corners of the oval to select it. Drag the mouse and the Oval should re-size.
- Click the Manipulate Icon on the menu bar. Select a corner of the Oval. Drag it with the mouse. The Shape of the polygon should be dragged to the new shape.
- click on the Move Icon. Select the Set. Move it by mouse dragging. It should move on the screen.
- Click the delete Icon. Place the mouse over the Set. Point and click. A pop-up window should appear. Confirm the delete action. The Set should disappear form the diagram along with the info pop-up

### 9.2.4 Type

- Create a diagram and choose the Type Icon from the menubar. Create a Type and Name it. A 4 point Rectangle should appear on the Diagram.
- Select Resize. Click on one of the corners of the oval to select it. Drag the mouse and the Oval should re-size.
- Click the Manipulate Icon on the menu bar. Select a corner of the Oval. Drag it with the mouse. The Shape of the polygon should be dragged to the new shape.
- click on the Move Icon. Select the Type. Move it by mouse dragging. It should move on the screen.
- Click the delete Icon. Place the mouse over the Type. Point and click. A pop-up window should appear. Confirm the delete action. The Type should disappear form the diagram along with the info pop-up

### 9.2.5 Drawable Shaded Areas

#### 9.2.5.1 Shaded

A shaded section represents a Zone. Thus two Sets/Types must first be placed on the diagram in order to have the possibility of an intersection.

- Create a diagram and choose the Type Icon from the menubar. Create a Type and Name it "T1". A 4 point Rectangle should appear on the Diagram with the Label "T1".
- Choose the Type Icon from the menubar. Create a Type and Name it "T2". A 4 point Rectangle should appear on the Diagram with the Label "T2".

- click on the Move Icon. Select the "T1" Type. Move it by mouse dragging until an intersection is seen between the two types (a common area where they overlap).
- Click the Shade Icon. Place the mouse over the intersection and click. A shaded blue area should appear exactly the same shape as the intersection.
- Select the Move Icon. Select one of the Types and move it. The shaded area should update in shape as the Type is moved.
- Move the Type so that the two Type objects are separate on the diagram. The shaded area should disappear.

Repeat this test for the Set object.

#### 9.2.5.2 Shaded Multi region

A Zone may have several visible regions on the diagram. The program must shade all of these regions correctly.

- Create a diagram and choose the Type Icon from the menubar. Create a Type and Name it "T1". A 4 point Rectangle should appear on the Diagram with the Label "T1".
- Choose the Type Icon from the menubar. Create a Type and Name it "T2". A 4 point Rectangle should appear on the Diagram with the Label "T2".
- click on the Manipulate Icon. Select the "T1" Type. Manipulate it by mouse dragging until an it becoes a thin long horizontal object. Now drag it so that it cuts 'T2' in half.
- Click the Shade Icon. Place the mouse over the an area which is in 'T1' but not in 'T2' and click. Two shaded blue areas representing the Zone.
- Select the Move Icon. Select one of the Types and move it. The shaded area should update in shape as the Type is moved.
- Move the Type so that the two Type objects are separate on the diagram. The shaded area should disappear.

### 9.2.6 Postscript

#### 9.2.6.1 General PostScript Production

- Name of Test : Test PostScript Output.
- How to Run : Create a Diagram with all drawable types, including shaded areas. Save as postscript. Unpack the Zip file and examine the EPS file (in unix "gv"[9] is handy for this). Check that all drawn items are present and that they are in the correct place.

#### 9.2.6.2 Postscript Multi Shaded Region

Follow the instuctions for 9.2.5.2 and when the display shows two shaded regions representing the same zone, save as EPS. Unpack the Zip file and examine the EPS file. Check that all drawn items are present and that they are in the correct place.

### 9.2.7 Arrow

### 9.2.8 Arrow Origin and Destination Combinations

Arrows can connect contours (Sets and Types) and spider feet (Existential and Universal) Objects. Thus each of these four combinations must be tested.

- Create a diagram and choose the Type Icon from the menubar. Create a Type and Name it "T1". A 4 point Rectangle should appear on the Diagram with the Label "T1".
- Choose the Type Icon from the menubar. Create a Type and Name it "T2". A 4 point Rectangle should appear on the Diagram with the Label "T2".
- Choose the Existential icon. Create two Existential points "e1" and "e2".
- Select the Spider Icon. Connect "e1" and "e2" with a Spider. The two elements should be connected on the diagram.
- Save the diagram for future use.

#### 9.2.8.1 Contour to Spider Foot

- restore the diagram from file. The diagram should be displayed.
- Select the Arrow Icon and select "T1". Drag the Arrow head to "e1" or "e2". The Arrow should connect.

#### 9.2.8.2 Spider Foot to Contour

- restore the diagram from file. The diagram should be displayed.
- Select the Arrow Icon and select "e1". Drag the Arrow head to "T1" or "T2". The Arrow should connect.

#### 9.2.8.3 Contour to Contour

- restore the diagram from file. The diagram should be displayed.
- Select the Arrow Icon and select Type "T1". Drag the Arrow head to "T2". The Arrow should connect.

#### 9.2.8.4 Spider Foot to Spider Foot

- restore the diagram from file. The diagram should be displayed.
- Create a Universal Point and label it "U1".
- Select the Arrow Icon and select "e1". Drag the arrow head to "U1". The arrow should connect.

#### 9.2.8.5 Hanging Edit

Because drawing an Arrow is not one simple operation (there is the arrow creation and then any number of mouse drag operations between), tests must be made for edits hanging.

- restore the diagram from file. The diagram should be displayed.
- Select the Arrow Icon and select "e1". Drag the arrow head to an empty area of the screen. The arrow should be left hanging waiting to be connected.
- Select a menu bar option that is not arrow, and click on the screen. The partially completed Arrow should disappear.

### 9.2.9 Spiders

#### 9.2.9.1 Existential Spider

- Create a diagram with two Existential points. Connect them with a Spider. The two elements should be connected on the diagram.
- Create a new spider connection and try to connect to a previously connected element. It should be impossible.
- Now add an Universal Point to the diagram.
- Attempt to connect to the Universal point with a new Spider foot. It should be impossible.
- Now create and add a new Existential point. This should connect and the spider should have 3 elements.

#### 9.2.9.2 Universal Spider

- Create a diagram with two Universal points. Connect them with a Spider. The two elements should be connected on the diagram.
- Create a new spider connection and try to connect to a previously connected element. It should be impossible.
- Now add an Existential Point to the diagram.
- Attempt to connect to the Existential point with a new Spider foot. It should be impossible.
- Now create and add a new Universal point. This should connect and the spider should have 3 elements.

#### 9.2.9.3 Hanging Edit

Because drawing an Spider is not one simple operation (there is the spider node creation and then any number of mouse drag operations between), tests must be made for edits hanging.

- Create a diagram with two Existential points. Connect them with a Spider. The two elements should be connected on the diagram.
- Create a new spider connection and try to connect to a previously connected element. It should be impossible.
- Create a new Spider connection and drag it to empty space. The partially completed spider should be displayed on the diagram with the unconnected node left on the screen.
- Select a menu bar option that is not Spider, and click on the screen. The partially completed Spider Node should disappear.

## 9.3 Save A Diagram

## 9.4 XML output : Well Formedness

Take the XML ".csd" file, rename it to ".xml" and read it into a web browser (I used netscape 7 but any modern brower will parse XML). The web browser will check the xml syntax for well formedness, and will show the file as a collapsible tree.

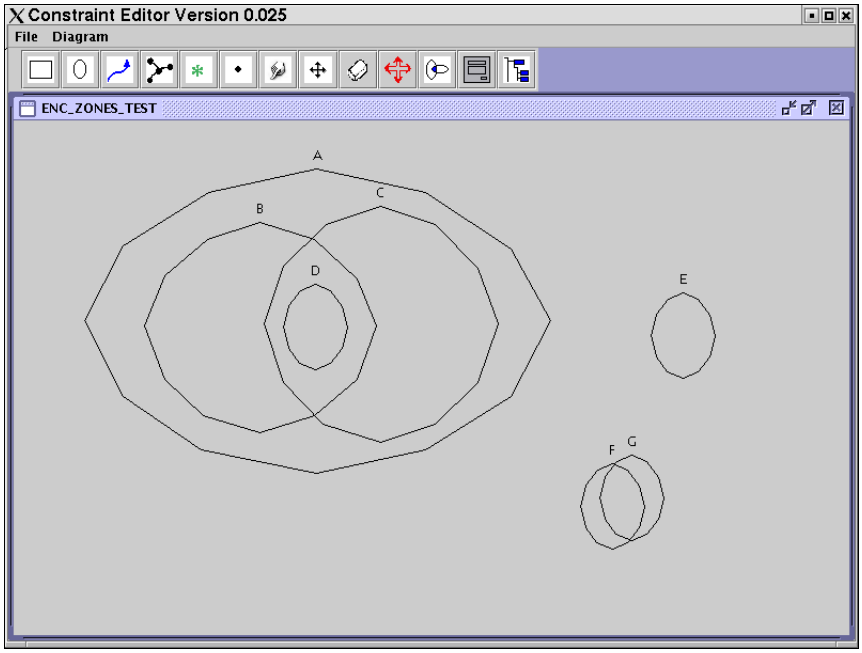



Figure 9.1: Test of Abstract Zone Creation with Enclosure and Pure Intersection

## 9.5 Test Constraint Diagrams

### 9.5.1 Test Of Abstract Zone Creation using FZD 1

#### 9.5.1.1 Enclosed and Pure Intersection Zones

- File Name : *testing/enc_zones.csd*
- To Test : Enclosed and Pure Intersection Zones with Enclosed Pure Intersections
- Result File : *testing/enc_zones.abs*

See Figure 9.1. This test creates zones form all three conditions searched for by the FZD (see 7.2.2.1). It is therefore a white box test (it excersizes all known decisions and paths in the FZD algorithm). The three sources of Zones from the FZD are from pure intersection (which may also be enclosed), being enclosed by another contour and from not being enclosed by any contour.

This diagram should produce the following zones in the abstract XML (note the actual zone numbers/handles may be different in the abstract XML).

| *Included* | *Excluded* | *Zone Number* | *Comment:Algorithm Source* |
|---|---|---|---|
| { } | { A B C D E F G } | 1 | implicit background zone |
| { A } | { B C D E F G } | 2 | Not Enclosed |
| { A B C } | { D E F G } | 3 | Represents a whole where D is |
| { A B } | { C D E F G } | 4 | pure intersection enclosed by A |
| { A C } | { B D E F G } | 5 | pure intersection enclosed by A |
| { A B C D } | { E F G } | 6 | Enclosed by, D has no pure intersections |
| { E } | { A B C D F G } | 7 | Not Enclosed |
| { F G } | { A B C D E } | 8 | Pure Intersection and not enclosed |
| { F } | { A B C D E G } | 9 | Not Enclosed |
| { G } | { A B C D E F } | 10 | Not Enclosed |

#### 9.5.1.2 Enclosed and Pure Intersection Zones with Existential

- File Name : *testing/enc_zones_with_existentials.csd*
- To Test : Existentials in correct Habitat
- Result file : *testing/enc_zones_with_existentials.abs*

Individual existential objects when described in the Abstract XML, are defined by their habitat or zone. In the concrete diagram, they are defined by $(x, y)$ coordinates.

These are based on figure 9.1

#### 9.5.1.3 Enclosed and Pure Intersection Zones with Spiders

- File Name : *testing/enc_zones_with_spiders.csd*
- Name of Test : Spider Habitats in Correct Zones
- Result file : *testing/enc_zones_with_spiders.abs*

Individual spider feet when described in the Abstract XML, are defined by their habitat or zone. In the concrete diagram, they are defined by $(x, y)$ coordinates. Here two spiders are represented, one universal and one existential.

Note that the EXISTENTIAL spider, including the default zone, causes that to be added to the zone list.

These are based on figure 9.1

## 9.6 Test Harnesses

Test harnesses included in the packages can use the methods available to the main program and thus test sections of the full program in a controlled and non interactive way. This allows very selective, precise testing and, with text output, logged checking through all the stages of an operation.

Also a test harness in a simple and open way, shows how the classes work as well as their primary function, verifying the code.

### 9.6.1 Abstract Diagram

- File Name : *abs/AbstractDiagramTH.java*
- Name of Test : Create An Abstract Diagram
- How to Run : Use the make command with the parameter *go_abs_1*

This test populates a complete Abstract diagram object and then outputs it as XML.

### 9.6.2 Abstract Zone comparison

- File Name : *abs/AbstractDiagramTH2.java*
- Name of Test : Test Abstract Zone Comparison
- How to Run : Use the make command with the parameter *go_abs_2*

Abstract Zones are implemented in Java with two Vectors, one holds included contours and the other excluded. Because Abstract diagram building methods may build Zones with the same specification, but different ordering (or even repeated contours) comparison methods are very important in order not to store or to reference duplicate (by specification of included and non included) Zones.

This Test harness explicitly sets up abstract zones with subtly different creation sequences to test the robustness of the comparison and methods.

### 9.6.3 Abstract Spider

- File Name : *drawable/AbstractSpiderTH.java*
- Name of Test : Create an individual Abstract Spider
- How to Run : Use the make command with the parameter *go_d_1*

This test harness creates an Abstract Spider and then displays the result of the object representing its self as XML.

# Chapter 10

# Deviation Description

This chapter lists variations and missing features from the product as referenced to the requirements specification. Any known bugs or other deficiencies are listed here also.

In a commercial environment, this would always be a confidential document !

## 10.1 Missing Features

### 10.1.1 Warnings for Well Formedness

An Error window with warnings for 'well formness' errors in concrete diagrams, to warn a user that the diagram may be ambiguous or confusing has not been implemented, due to time constraints.

### 10.1.2 SVG not implemented

Scaled Vector Graphics, an XML based diagram format has not been implemented. There are however EPS to SVG conversion programs available.

## 10.2 Variances with Specification

This section deals with problems that were solved in the coding and testing phase that caused a change in the requirements specification.

**No Changes to Specification Required**

## 10.3 Known Faults

### 10.3.1 Selection of Control Points

When attempting selection of control points in a cluttered diagram, it can be impossible until other objects are moved out of the way. A solution to this would be to modify the mouse click selection routines by having them return a 'pixels from hit' value. The nearest hit would then be chosen instead of the last one encountered (as it does now).

## 10.4 Known Deficiencies

### 10.4.1 Impossible to Display Label on Spider

Currently the name associated with a Spider cannot be displayed. The strategy for doing this should be to associate the label with the centre of a spider node line, and allow a mouse drag able offset from the centre of the line. Perhaps this should be able to change spider node lines.

### 10.4.2 Impossible to Display Label on Arrow

As with the Spider, currently no label can be displayed for the Arrow. Strategy as for Spider node.

### 10.4.3 No Colour Choices

The user should be able to select colours for all objects to make the diagrams clearer. Due to time constraints this has not been implemented. For encapsulated PostScript output, hand editing could achieve this easily as the EPS output is automatically commented.

### 10.4.4 Tree Diagram selection of diagram Objects

A Tree diagram of the diagrams is available. It was intended that the user could 'select' editable/drawable objects by selecting them in the tree representation. As the JTree object has references to the diagram and objects concerned this should be a simple programming task.

## 10.5 Features Withheld

### 10.5.1 Enclosure Relations

Methods exist to explicitly add enclosure and pure intersection relations, as XML into the abstract XML output. Because the enclosure relations can be derived from complete zone definitions (see 7.2) the information is implicit in the current abstract XML.

Should this information be required explicitly in the abstract XML, calling the toXML() UnorderedRelations object (created and used by the FZD algorithm) and adding this to the XML output, will achieve this.

# Chapter 11

# Conclusion

## 11.1 Original Objective

This project was originally designed to be an editor working within the KMF tool. Unfortunately KMF was abandoned about a third of the way through this project. By the time the KMF project had been dropped this Constraint Editor was producing constraint diagrams and abstract conversion coding had started. Upon the demise of the KMF project I toyed with the idea of producing OCL and/or Z directly from the concrete diagrams. On seeking advice from my supervisors, abstract XML output was chosen..

A requirements specification (in Microsoft word format) submitted for approval by the KMF team is included on the CD at **/docs/OriginalRequirementsDoc/msc_project_spec.doc**. This is essentially chapter 3 with KMF requirements.

## 11.2 Research and Reading

On the ROOM course (see 11.3.2) I was introduced to the CD-Edit[19] Windows based tool. This is a drawing tool for constraint diagrams that outputs postscript diagrams. The version demonstrated only performed drawing and no mathematical relationships could be extracted from the diagrams.

I have not found any books specifically describing Constraint Diagrams, the technique is perhaps a little too new! For background reading on Constraint Diagrams accademic papers [17],[18],[24] were found to be useful. Books on software Engineering Mathematics [11],[13],[8],[14] provided useful background reading.

For specific questions relating to the project I contacted my Supervisor (Jean Flower) either in a meeting, or by emails with diagram attachments.

## 11.3 MSc Modules that Supported the Project

Four modules from the MSc were directly relevant to this project, a fifth showed the usefulness of visualisation and mathematical techniques in software production.

### 11.3.1 Object Oriented Design - Knowledge Applied to Project

One particular OO design technique has been used which was taught on the OOD course.

This is the 'Object Cards' or 'post-it' method[1]

Some Design methodologies seem to become fashionable and then within a few years get quietly forgotten. Others, by their usefulness survive.This allowed one to stand back from the algorithmic problems and concentrate on manipulating the OO structure.

The Object Cards or 'post-it' method[1] is actually useful in breaking down a problem quickly into a sensible Object Model. From an earlier computing 'era', the Yourdon Data flow Diagram (DFD) method, for designing a procedural based programs/algorithms is another technique that that goes beyond being a tool for graphically describing a problem. With that you can take a problem and break down the available data via transforms until a procedural algorithm appears !

The 'object cards methodology' like Yourdon DFD, is one that helps in the design process. UML in itsself is very good at describing an OO system.

This course was directly relevant to the design processes undertaken in this project.

### 11.3.2 Rigorous Object Oriented Modelling - Knowledge Applied to Project

For this Project the ROOM module was most relevant. This was mainly concerned with applying OCL to UML diagrams, and with teaching use of Z schemas. Constraint diagrams were also taught.

The Brighton University ROOM module taught me how to use, and as importantly showed me the potential of constraint diagrams to become a tool for specifying systems with mathematical precision but also being accessible to non Software Engineers.

It also gave me a deeper understanding of Mathematics applied to specification and a deeper understanding of the limitations of UML.

### 11.3.3 Software Engineering Mathematics - Knowledge Applied to Project

This course covered pure mathematical theory and laid a solid foundation for tackling this project.

### 11.3.4 Software Implementation - Knowledge Applied to Project

This module made me learn and use Java seriously for the first time. Before that I had mainly been using C++.

#### 11.3.4.1 Comparing Software Implementation in Java to C++

From the perspective of a C++ programmer, Java at first seemed a little restrictive. It then became apparent that although there were many similarities between Java and C++ the underlying philosophies of the languages are completely different. C and C++ concentrate on code speed, and Java concentrates on portability and safety. With processors doubling speeds every 18 months it is easy see which language of the two will become dominant in the coming years.

The Java application programming interface has another positive effect when compared to C++. In C++ there are a great many ways to solve the same problem. Also there are choices to be made about when to use a function, a structure or a class. In C++ memory allocation and de-allocation is the responsibility of the programmer, along with the complications of looking after pointers. In Java these choices do not exist. Also in Java the supplied classes narrow down the programming choices. This means that software maintenance will be easier. A new comer will be able to understand a Java program better than a C++ program simply because he will recognise (and therefore understand) a greater proportion of the code. Take for instance a typical problem, the Date. In Java a calendar class is supplied. In C++ you could write your own or choose one among the many ready written. A Java programmer encountering

the date handling code will probably already be familiar with the class, a C++ programmer is less likely to have seen it; and will have to learn how the new class works before he can attempt any changes.

Another bonus with Java is the speed of development time. When C++ written with pointers 'goes wrong' the results are not always straight forward. Without specialist tracing tools (which can only be used in a develpoment environment) the best you can expect is a core dump ! The worst when memory can be corrupted and the true fault hidden. With Java mistakes are shown as exceptions and stack trace printouts. Far easier and quicker !

### 11.3.5 Concurrent Systems

Although not directly relevant to this project ( the tools studied were ccs and coloured petri nets) this did show me that tools other than yourdon and UML, were useful in the software development process. The matematical nature of CCS, applying a formal approach to concurrent problems that I had always seen tackled with 'reasoning' and 'possible scenarios' (or, in some cases, plain disabling of interrupts for longer than was probably necessary!), opened my eyes to applying methodologies to concurrent programming problems.

With Coloured petri nets I was fortunate enough to also see in an application at work, where they were used to model real time data extraction for CCD sensors.

Both of these methodologies had one thing in common though. They showed that mathematical methods when applied correctly to a software problem, help define a water tight solution. They provide a firm mathematical framework to build the software around.

In short I had often developed software to implement mathematics, but here mathematics was guiding software implementation !

A Graphical mathematically provable methodology for specification of software is therefore something that I see being potentially very useful. That course has been an inspiration for the Constraint Editor. I hope it, or perhaps derivitives of it are used.

## 11.4 Successes Of the Project

The project has met all important requirements (see Deviation Description chapter 10), and has produced a Constraint Editor that can draw and save/restore constraint diagrams. The diagrams can be mathematically converted to an abstract mathematical form and saved as XML.

This has also given me a Java GUI framework with diagram sequencing, the diagrams containing objects that can relate to each other in an XML saving/restoring environment. In short the base tools for any visualisation tool, to run in the Java2 XML environment.

This has also given me experience with the Java Geometry classes, Java Zip utilities, Postscript generation, Java 2D library and the SAX XML parser.

The Editor has been used for most of the constraint diagrams included in this document. I find it quick and easy to use !

The OO structure (see figure 6.1) and Java coding was carefully designed to allow easy expansion. For instance adding a new drawable type should be relatively straight forward. For instance by inheriting the drawable class, the object is automatically included in the diagrams and has an interface to conform to. On conforming to this interface the new object fits in with the system. For instance, when the system is saving a diagram, it will call each "Diagrams" toXML method. This will in turn call all Drawable Objects "toXML" methods.

I hope that the OO structure and openness of the XML output allow this tool to be adapted and expanded.

## 11.5 Set Backs

I wasted some time creating an 'arc' drawing class (see 5.5). I should have looked into the Java libraries first for a suitable arc or mutable curve class. This wasted about a weeks worth of programming and although it worked correctly, was not flexible enough to use when diagrams became cluttered. The "CubicCurve2D.double" was almost exactly the base class I required for drawing the connecting lines. It draws two curves per line, with two control points. These control points were easy to make into visible dragable objects that the user could manipulate on the screen with the mouse. I would certainly 'look first before writing my own' now.

I missed the need for the abstract output to know of all zones (see 7.2) (even if they were not shaded or inhabited by a spider) and was forced into a last minute re-think on making them all implicitly available. The previous algorithm had only determined a zone when a Spider inhabited it (i.e. the zone was therefore used). This was around an extra weeks work, but produced an interesting algorithm (see 7.2.2.1).

I spent around three weeks of time looking at the KMF project. Part of this was looking at the XMI specification and the output from the Poseidon UML tool (version 1.5) [16]. The KMF project was abandonned before I had anything to contribute to it. The Abstract UML model for abstract representation of a constraint diagram was however, peer reviewed and discussed in detail. This was salvaged from KMF and used to base the abstract XML output on.

## 11.6 Organisational Context of the Project and Progress

Where I was unsure of a mathematical or behavioural issue with the constraint editor, I always emailed my Supervisor (often with constraint diagrams drawn in xpaint) for guidance. Some issues though, such as the need for a complete zone list, were discovered in meetings with my Supervisor where the lastest development stage/feature was demonstrated.

As a part time student, and one who started a new job half way through the project, I had to put the coding down for sometimes weeks at a time. After returning to the project after a long series of distractions, I occasionally found myself writing a method that I had already written (and often with a very similar or the same name).

## 11.7 Constraint Diagram Editor Future

Ideally I would like to extend the constraint diagram editor to work with component failure modelling (see 8.4). This would involve an automated process to create set diagrams from constraint diagrams, and also to maintain a tree of diagrams, the leaves of which would represent modelled components, and further up the tree sub-systems (and sub-systems of sub-systems etc) and finally at the top, the complete system, with all its failure modes (see figure 8.3).

European standards for safety critical equipment are getting more and more complicated and are begining to demand the help of formal/mathematical process that can modularise and separate sub-sytems in a formal way.

# Bibliography


[1] Rubecca Wirfs-Brock et.all *Designing Object-Oriented Software : P T R : ISBN 0-13-629825-7*

[2] Robert Eckstein et. all *Java Swing : O'Reilly : ISBN 1 565592 455 X*

[3] Gosling et. all *The Java Application Programming Interface Vol I : Addisson Wesley : ISBN 0 201 63453 8*

[4] Gosling et. all *The Java Application Programming Interface Vol II : Addisson Wesley : ISBN 0 201 63459 7*

[5] Jack B Kuipers *Quaterions and Rotation Sequences : Princeton. ISBN 0-691-05872-5*

[6] Robet D Stum, Donald E Kirk *Discrete Systems and Digital Signal Processing : Addison-Wesley : ISBN 0-201-09518-1*

[7] ThinkGeek *Think Geek T-Shirts* http://www.thinkgeek.com/tshirts/coder/595d/images/

[8] Jim Woodcock, Martin Loomes *Software Engineering Mathematics : Pitman : ISBN 0-273-02673-9*

[9] Johannes Plass, Tim Theisen *http://wwwthep.physik.uni-mainz.de/ plass/gv/*

[10] Andrew Oram, Steve Talbot *Managing Projects with Make : O'Reilly : ISBN 0-937175-90-0*

[11] Alan Gibbons *Algorithmic Graph Theory : Cambridge University Press : ISBN 0-521-28881-9*

[12] PostScript *Adobe Publishing Web Site* http://www.adobe.com/

[13] R Garnier, J Taylor *Discrete Mathematics for New Technology : IoP : ISBN 0-7503-0135-X*

[14] D C Ince *An Introduction to Discrete Mathematics, Formal System Specification and Z : Oxford : ISBN 0-19-853836-7*

[15] J D Eisenburg *S.V.G. Essentials : O'Reilly ISBN: 0-59-6002-238*

[16] Poseidon / GentleWare *Poseidon UML Editor* http://www.gentleware.com/

[17] J.A. Flower, J Howse, J Taylor *Nesting in Euler Diagrams* http://www.cmis.brighton.ac.uk/Research/vmg/papers/GTVMT02.pdf

[18] Howse J, Stapleton G, Flower J, Taylor J *Corresponding regions in Euler diagrams* http://www.cmis.brighton.ac.uk/Research/vmg/papers/D2K2HSFT.pdf

[19] ED-Edit *Windows Constraint Diagram Editor* http://www.cs.technion.ac.il/Labs/ssdl/research/cdeditor/download/download.html

[20] Xpaint *Xpaint Drawing Program* https://sourceforge.net/projects/sf-xpaint/

[21] Sun *Java SDK API* http://java.sun.com/j2se/1.4.2/docs/api/

[22] SAX XML PARSER *http://www.sax.org/*

[23] Kent Modelling FrameWork *http://www.cs.kent.ac.uk/projects/kmf/*

[24] Jean Flower, John Howse *Generating Euler Diagrams :Proceedings of Diagrams 2002, Callaway Gardens, Georgia, April 2002, Springer Verlag* http://www.it.bton.ac.uk/research/vmg/VisualModellingGroup.html

[25] BSI *European Standard For Gas Burner Safety Systems* Copy included in docs directory as EN298.pdf

# Appendix A

# XML Examples

## A.1 Abstract XML

```
 <abstract_constraint_model ConstraintEditorVersion='0.025'>


<abstractdiagram name='Untitled_0' >

 <contourslist>
      <contour name='Type_One'/>
      <contour name='A'/>
      <contour name='B'/>
      <contour name='C'/>
 </contourslist>


 <zones>

  <abstractzone handle='0' shaded='true' >
    <includedcontours>
      <contour name='Type_One'/>
      <contour name='C'/>
    </includedcontours>
    <excludedcontours>
      <contour name='A'/>
      <contour name='B'/>
    </excludedcontours>
  </abstractzone>

  <abstractzone handle='1' shaded='false' >
    <includedcontours>
      <contour name='Type_One'/>
    </includedcontours>
    <excludedcontours>
      <contour name='A'/>
      <contour name='B'/>
      <contour name='C'/>
    </excludedcontours>
  </abstractzone>
```

```
  <abstractzone handle='2' shaded='false' >
    <includedcontours>
      <contour name='A'/>
    </includedcontours>
    <excludedcontours>
      <contour name='Type_One'/>
      <contour name='B'/>
      <contour name='C'/>
    </excludedcontours>
  </abstractzone>

  <abstractzone handle='3' shaded='false' >
    <includedcontours>
      <contour name='B'/>
      <contour name='Type_One'/>
    </includedcontours>
    <excludedcontours>
      <contour name='A'/>
      <contour name='C'/>
    </excludedcontours>
  </abstractzone>

  <abstractzone handle='5' shaded='false' >
    <includedcontours>
      <contour name='C'/>
    </includedcontours>
    <excludedcontours>
      <contour name='Type_One'/>
      <contour name='A'/>
      <contour name='B'/>
    </excludedcontours>
  </abstractzone>

  <abstractzone handle='2000000021' shaded='false' >
    <includedcontours>
    </includedcontours>
    <excludedcontours>
      <contour name='Type_One'/>
      <contour name='A'/>
      <contour name='B'/>
      <contour name='C'/>
    </excludedcontours>
  </abstractzone>


 </zones>


 <spiders>

  <abstractspider name='ExistentialSpider' type='EXISTENTIAL'>
    <habitat type='zone' handle='3' />
    <habitat type='zone' handle='1' />
    <habitat type='zone' handle='2' />
  </abstractspider>
```

```
  <abstractspider name='Untitled_16' type='UNIVERSAL'>
    <habitat type='zone' handle='3' />
    <habitat type='zone' handle='5' />
    <habitat type='zone' handle='2' />
  </abstractspider>

  <abstractspider name='e4' type='EXISTENTIAL'>
    <habitat type='zone' handle='1' />
  </abstractspider>

  <abstractspider name='u4' type='UNIVERSAL'>
    <habitat type='zone' handle='1' />
  </abstractspider>

  <abstractspider name='u55' type='UNIVERSAL'>
    <habitat type='zone' handle='2000000021' />
  </abstractspider>

 </spiders>


 <arrows>

  <abstractarrow name='r1'>
    <origin type='contour' name='C'/>
    <destination type='contour' name='A'/>
   </abstractarrow>
  <abstractarrow name='r2'>
    <origin type='spider' name='e4' />
    <destination type='spider' name='u4'/>
   </abstractarrow>
  <abstractarrow name='r3'>
    <origin type='spider' name='e4' />
    <destination type='contour' name='C'/>
   </abstractarrow>
  <abstractarrow name='r55'>
    <origin type='contour' name='A'/>
    <destination type='spider' name='u55'/>
   </abstractarrow>

 </arrows>



</abstractdiagram>

</abstract_constraint_model>
```

## A.2 Concrete XML

```
<diagramsequence>

 <diagram name='Untitled_0' >
 <type name='Type_One'  npoints='4'  inherentlyempty='0' >
   <pp pindex='0'  xp='41' yp='286' />
   <pp pindex='1'  xp='364' yp='286' />
```

```
   <pp pindex='2'  xp='364' yp='35' />
   <pp pindex='3'  xp='41' yp='35' />
   <height> 150</height>
   <width> 150</width>
   <x> 107</x>
   <y> 143</y>
     <comments>
    </comments>
 </type>

<set name='A'  npoints='12' inherentlyempty='0' >
   <pp pindex='0'  xp='571' yp='210' />
   <pp pindex='1'  xp='585' yp='204' />
   <pp pindex='2'  xp='596' yp='190' />
   <pp pindex='3'  xp='601' yp='170' />
   <pp pindex='4'  xp='596' yp='151' />
   <pp pindex='5'  xp='585' yp='136' />
   <pp pindex='6'  xp='571' yp='130' />
   <pp pindex='7'  xp='557' yp='136' />
   <pp pindex='8'  xp='546' yp='150' />
   <pp pindex='9'  xp='541' yp='170' />
   <pp pindex='10'  xp='546' yp='190' />
   <pp pindex='11'  xp='556' yp='204' />
   <height> 80</height>
   <width> 60</width>
   <x> 571</x>
   <y> 170</y>
     <comments>
    </comments>
 </set>

<set name='B'  npoints='12' inherentlyempty='0' >
   <pp pindex='0'  xp='105' yp='205' />
   <pp pindex='1'  xp='119' yp='199' />
   <pp pindex='2'  xp='130' yp='185' />
   <pp pindex='3'  xp='135' yp='165' />
   <pp pindex='4'  xp='130' yp='146' />
   <pp pindex='5'  xp='119' yp='131' />
   <pp pindex='6'  xp='105' yp='125' />
   <pp pindex='7'  xp='91' yp='131' />
   <pp pindex='8'  xp='80' yp='145' />
   <pp pindex='9'  xp='75' yp='165' />
   <pp pindex='10'  xp='80' yp='185' />
   <pp pindex='11'  xp='90' yp='199' />
   <height> 80</height>
   <width> 60</width>
   <x> 105</x>
   <y> 165</y>
     <comments>
    </comments>
 </set>

<set name='C'  npoints='12' inherentlyempty='0' >
   <pp pindex='0'  xp='355' yp='317' />
   <pp pindex='1'  xp='369' yp='311' />
   <pp pindex='2'  xp='380' yp='297' />
```

```
  <pp pindex='3'  xp='385' yp='277' />
  <pp pindex='4'  xp='380' yp='258' />
  <pp pindex='5'  xp='369' yp='243' />
  <pp pindex='6'  xp='355' yp='237' />
  <pp pindex='7'  xp='341' yp='243' />
  <pp pindex='8'  xp='330' yp='257' />
  <pp pindex='9'  xp='325' yp='277' />
  <pp pindex='10'  xp='330' yp='297' />
  <pp pindex='11'  xp='340' yp='311' />
  <height> 80</height>
  <width> 60</width>
  <x> 364</x>
  <y> 284</y>
    <comments>
   </comments>
 </set>

<existential name='e1' >
  <x> 112</x>
  <y> 165</y>
  <height> 5</height>
  <width> 5</width>
    <comments>
   </comments>
 </existential>

<existential name='e2' >
  <x> 279</x>
  <y> 106</y>
  <height> 5</height>
  <width> 5</width>
    <comments>
   </comments>
 </existential>

<existential name='e3' >
  <x> 574</x>
  <y> 161</y>
  <height> 5</height>
  <width> 5</width>
    <comments>
   </comments>
 </existential>

<existential name='e4' >
  <x> 237</x>
  <y> 229</y>
  <height> 5</height>
  <width> 5</width>
    <comments>
   </comments>
 </existential>

<universal name='u1' >
  <x> 97</x>
  <y> 190</y>
```

```
    <comments>
   </comments>
 </universal>

<universal name='u2' >
  <x> 350</x>
  <y> 306</y>
    <comments>
   </comments>
 </universal>

<universal name='u3' >
  <x> 575</x>
  <y> 190</y>
    <comments>
   </comments>
 </universal>

<universal name='u4' >
  <x> 284</x>
  <y> 176</y>
    <comments>
   </comments>
 </universal>

<universal name='u55' >
  <x> 465</x>
  <y> 46</y>
    <comments>
   </comments>
 </universal>

<spider name='ExistentialSpider' >
   <spidernode>
        <mutablecurve   ctrlx1='157.76'   ctrly1='181.25'
          ctrlx2='214.52'   ctrly2='94.29999999999998'   />
      <origin oindex='0'>e1</origin>
      <destination dindex='0' >e2</destination>
   </spidernode>
   <spidernode>
        <mutablecurve   ctrlx1='385.9'   ctrly1='182.48000000000002'
          ctrlx2='481.59999999999997'   ctrly2='84.96000000000001'   />
      <origin oindex='0'>e2</origin>
      <destination dindex='0' >e3</destination>
   </spidernode>
</spider>

<spider name='Untitled_16' >
   <spidernode>
        <mutablecurve   ctrlx1='202.64000000000001'   ctrly1='278.87'
                        ctrlx2='284.48'   ctrly2='218.94000000000003'   />
      <origin oindex='0'>u1</origin>
      <destination dindex='0' >u2</destination>
   </spidernode>
   <spidernode>
        <mutablecurve   ctrlx1='402.18'   ctrly1='314.24'
```

```
                        ctrlx2='476.76'   ctrly2='186.88'   />
      <origin oindex='0'>u2</origin>
      <destination dindex='0' >u3</destination>
   </spidernode>
 </spider>

 <arrow name='r1' origin='C'  oindex='5'  destination= 'A'  dindex= '9' >
      <mutablecurve   ctrlx1='409.91'   ctrly1='252.97'
                      ctrlx2='465.02000000000004'   ctrly2='162.73999999999998'   />
    <comments>
    </comments>
 </arrow>

 <arrow name='r2' origin='e4'  oindex='0'  destination= 'u4'  dindex= '0' >
      <mutablecurve   ctrlx1='243.9'   ctrly1='223.16'
                      ctrlx2='260.4'   ctrly2='187.32'   />
    <comments>
    </comments>
</arrow>

 <arrow name='r3' origin='e4'  oindex='0'  destination= 'C'  dindex= '7' >
      <mutablecurve   ctrlx1='270.48'   ctrly1='251.17000000000002'
                      ctrlx2='302.16'   ctrly2='215.74'   />
    <comments>
    </comments>
</arrow>

 <arrow name='r55' origin='A'  oindex='7'  destination= 'u55'  dindex= '0' >
      <mutablecurve   ctrlx1='510.83'   ctrly1='90.48'
                      ctrlx2='481.46'   ctrly2='98.36'   />
    <comments>
    </comments>
</arrow>



   <shadedarea>
      <intersectioncomponent>Type_One</intersectioncomponent>
      <intersectioncomponent>C</intersectioncomponent>
    <comments>
    </comments>
  </shadedarea>


</diagram>
</diagramsequence>
```

R.P. Clark MSc Software Engineering Project Submission 2003.

# FINAL PAGE

# INTENTIONALLY BLANK

Produced by LaTeX on November 29, 2003.