Constraints analysis with respect to BFG metadata creation

A thesis submitted to The University of Manchester for the degree of
Master of Science
in the Faculty of Faculty of Engineering and Physical Sciences

2005

Stephen Jonathan Smith

School of Computer Science

Contents

Abstract

The BFG is a tool for the development of coupled models created by the University of Manchester that reads XML documents to obtain configuration data. The XML documents contain complex relationships termed 'constraints' that, if not expressed correctly, can cause errors in the execution of BFG and the coupled models generated by the BFG technology. The research in this thesis is focussed around an analysis of constraints with respect to BFG metadata creation. It shows how current XML validation technology falls short of meeting all the design goals of a BFG metadata constraint checking solution. A new approach is suggested and implemented, leveraging the power of Object Constraint Language. Finally, the new approach is integrated into a graphical user interface for BFG metadata creation also developed by the University of Manchester.

Declaration

No portion of the work in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

I would like to thank my supervisor Dr. Len Freeman for his valuable support and guidance. I would also like to thank Rupert Ford and Graham Riley for their continued help throughout the course of this project. Thanks also to Jamie Rowe and Matthew Tellier for the assistance with integration issues. In addition I would like to thank Rachael Warren and Sante Santos at the Tyndall Centre, University of East Anglia and Allyn Treshansky at the UK Meteorological Office for sharing their opinions and research. I must also note the encouragement provided by Robert Stott and Kate Moyse throughout my career. Finally, I greatly appreciate the help of Katie Johnson, Steven Wright and Andrew Diack who gave help with editing and proof reading.

Chapter 1: Introduction

The research presented in this thesis is focussed around constraint analysis with respect to metadata creation for the Bespoke Framework Generator (BFG). The BFG is a tool that has been developed at the University of Manchester for creating 'wrapper' code in which computational models are embedded [16]. The 'wrapper' code provides a mechanism through which the computational models can execute and communicate, thus facilitating the creation and execution of 'coupled models' [16] [48]. Computational models are computer programs that simulate some scientific behaviour, such as the climate or atmospheric Carbon Dioxide levels, and 'coupling' these models together has many advantages. The 'coupling' can be seen as the passing of variables or data between models. This enables data computed and outputted by one model to be used by other models in the simulation.

The component or modular based approach to scientific modelling increases flexibility, reusability and ultimately reduces the development time of complex scientific simulations. To address computational issues with model coupling, such as how to couple models and allow for data communication between them, a methodology exists, also developed by the University of Manchester, called the Flexible Coupling Approach [4] [16]. The BFG is a prototype implementation of this methodology. These concepts, technologies and ideas will be further discussed in Chapter 2.

To create the wrapper code, BFG uses 'configuration' documents expressed in eXtensible Markup Language (XML) [17]. These XML configuration documents are collectively termed 'BFG metadata'. A short review of XML related technologies is given in Chapter 3. The technologies discussed here will be referenced repeatedly throughout this thesis, but those readers experienced in XML may wish to omit this chapter. A detailed description of the operation of BFG and the use of metadata is given in Chapter 4.

Constraints can be viewed as rules or laws that a system must abide by. For instance, in the BFG, the rule that input cannot connect to input can be seen as a constraint. In the BFG, these constraints relate to structural relationships between elements in the XML documents. The issues concerned with BFG metadata creation are documented in Chapter 5. It is shown how errors in the

metadata can be costly and time consuming, using a use-case scenario as an example.

XML Schemas [46] are a common method of checking the validity of XML documents. However, they do not provide a complete solution when considering more complex scenarios [1] [2] [6] [7] [15]. This position is presented in Chapter 6, along with a detailed review of research in constraint checking of XML documents. The three recognised solutions to this problem are discussed, and a selection of example implementations is reviewed.

A full description of *all* the BFG metadata constraints would be extremely lengthy and is deemed unnecessary for this thesis but in Chapter 7 a selection of example constraints is given, ranging from relatively straightforward to increasingly complex. This chapter gives an idea of the kind of relationships that are present in the BFG metadata and why these relationships exist. It also shows how the constraints can be classified into two groups: 'intra-document' (within the same XML document) and 'inter-document' (across multiple XML documents). Each example constraint is stated in text, described and then expressed in relation to the structures present in the XML metadata documents.

In Chapter 8, the design and implementation of three new solutions for checking BFG metadata constraints are introduced. The most successful of the three has been integrated into a graphical user interface for BFG metadata creation. The general result is shown to be successful, although there are several issues that need addressing.

Finally, Chapter 9 concludes the research presented in this thesis by evaluating the discoveries and proposes areas for further investigation.

To summarise, the aims and objectives of this thesis are as follows:

- To perform an analysis of constraint management with respect to BFG metadata creation.
    - Chapters 2 through 5 describe what constraint management is (with respect to BFG metadata creation) and identify why it is important and needed.
- To suggest a system of constraint checking for BFG metadata.

    - Chapter 6 reviews methods of constraint checking with respect to the BFG (and XML documents in general) and why there are drawbacks to any existing approaches.
    - A design goal is to allow constraints to be easily updated so a coupled model developer has the flexibility to edit or change the level of checking.
- To show how constraint management may be integrated into a graphical user interface.
    - Chapter 8 suggests a new method of constraint management for the BFG that eliminates (some of) the negative aspects of existing methods, especially the complexity involved.
    - The practicality of integrating constraint management into a prototype GUI for constructing BFG metadata developed by Rowe [10] is investigated.
    - A design goal is to avoid 'entangling' the graphical environment with the proposed constraint management solution.
- To compare and contrast the proposed solution with existing techniques.
    - Chapter 9 concludes with a discussion of whether the proposed solution is more advantageous than existing techniques.
    - It discusses the wider implications of the proposed solution and whether it is a problem-specific solution.
    - Future work in this area is outlined.

The following chapter provides background information to this thesis and introduces some terminology. It describes a methodology termed the 'Flexible Coupling Approach' (FCA) that is at the core of the work presented in latter Chapters.

Chapter 2: Background

This chapter first describes the principles and research into coupled model development. Some terminology is introduced and the Flexible Coupling Approach (FCA) methodology is described.

Scientific modelling using computer simulations is a mature and familiar practice. Recently however, scientific communities have realised that faster development of complex scenarios, more accurate simulations and more accurate data output may be achieved by coupling models together in a component-like fashion [16] [48]. Bespoke, hand-crafted models cannot be easily re-integrated to address new problems nor easily adapted for reuse by other members in the scientific community. Furthermore, the natural tendency to increase simulation accuracy inevitably involves the interaction of accurate and well-established models from other development groups and scientific disciplines [16] [48].

The following three sections attempt to define the key concepts and terminology that are referred to throughout this thesis and the domain of coupled modelling in general.

2.1. What is a model?

Looking at it from a high-level scientific perspective, a *scientific model* can be seen as an algorithm that attempts to simulate some behaviour, usually focussed around capturing and summarising the behaviour of real world scenarios. For instance, high school students are taught (a simplified) water cycle model which is essentially a model or approximation of water movement on the earth's surface from ocean to atmosphere to land and back to ocean. The model does not have to be simulated on a computer for it to work – simply knowing the constituent parts of the model such as ocean and atmosphere and how the various parts of the model interact allows the expression of how things basically operate. However, if a detailed execution of the model is needed (usually to predict behaviour and situations over time, such as sea level in 2010) it makes sense to produce a computational model.

A computational model can be viewed as the expression of the scientific model into some computer program. How this is achieved is a design-detail, as it can

be implemented in any computational language.

In this document we frequently use the terms 'computational model' or just model. In some instances, to clarify the distinction between a coupled model (see section 2.2) and model, the term individual model is used.

A special subset of models, known as 'transformers', categorise models that perform simple operations rather than some complex scientific behaviour. For instance, a transformer may perform a mathematical data aggregation function – taking in 10 data items and outputting the mean. The transformation function itself can be quite complex, but it does not capture a scientific model (see above). This is the distinguishing feature of transformers.

The transformer subset of models facilitates the communication of two models which by themselves would be scientifically or mathematically incompatible. The reader may imagine having a library of standard or common transformations such as mathematical aggregations that are reused when necessary.

The models described in this thesis generally have an outer 'looping' structure that encompasses the main body of the calculations performed. Performing one iteration through this outer loop is termed a timestep.

2.2. What is a coupled model?

A coupled model is the interoperation of individual computational models, consisting of some framework code in which the individual model codes are embedded. The framework code serves as a 'wrapper' facilitating the execution of individual models in different computational environments and acting as a means of communication between models. The models and execution environments may be heterogeneous and a good framework should be flexible enough to handle this situation.

Coupled models have become an alternative to producing large individual models that do not lend themselves well to addressing new problems. A coupled model that can support the removal, addition and replacement of individual models can be easily adapted to simulate new scenarios. Furthermore, a standardised coupled model creation environment allows models to interact across scientific

domains without the need for expensive and time consuming adjustments to computer programs [4] [16].

For instance, let us again consider the water cycle. One way of creating a computational model of the water cycle would be to create a single, and probably quite large and bespoke computer program. However, if we were to consider using a composition of smaller, self-contained computational models (ocean, atmosphere and land respectively) that interact across well defined interfaces, interchanging one ocean model for another, possibly belonging to another research institute, becomes a possibility. It would also be plausible to replace the single atmospheric model with a lower-atmospheric model and upper-atmospheric model if this is deemed a more accurate way to view the behaviour. Having smaller coupled computational models enables us to easily adapt to suit the enhanced or adapted scientific model.

2.3. What is Metadata?

Metadata can be described as "data about data" or "data that describes data". For the purposes of this thesis the term is used frequently to denote data describing the individual models, coupled models and configuration of the coupling framework environment. The use and format of BFG metadata is described in more detail in Chapter 4.

2.4. Issues with coupled model development

To couple models together two major issues need to be addressed:

1) How to ensure that the models are mathematically and scientifically compatible.
2) How the coupling is achieved at a computational level.

This dissertation is primarily concerned with the latter issue and in section 2.5 research into this problem is described. In particular, the General Coupling Framework (GCF) developed by the Centre for Novel Computing (CNC) at Manchester University [4] is discussed.

The GCF is best described as an approach that enables the composition and deployment of coupled applications in a flexible manner. For this reason, the technology has recently been re-titled 'the Flexible Coupling Approach' (FCA) [4]. For the remainder of this document this technology will be referred to as FCA but citations in older documents may still refer to it as GCF.

The ideas and research of the FCA are reflected in an implementation called the Bespoke Framework Generator (BFG) also in development at the University of Manchester [3]. The BFG is at the core of this thesis and a general understanding of the FCA and BFG technology is necessary for latter chapters. In the following section an overview of the FCA is given, while the BFG is discussed is greater detail in Chapter 4.

2.5. The Flexible Coupling Approach

For all coupling frameworks it is first necessary to ensure that the models meet certain requirements and are compatible within the framework. Many existing frameworks make this a time consuming task. In contrast, the FCA imposes just a few requirements, termed 'compliance rules', on the underlying model implementation [16]. The compliance rules, described below in more detail, are intended to promote model portability and form the basis of the FCA general extensibility.

- A model must not rely on knowledge from other models or code. While it does interact with other models, it must essentially entirely encapsulate the science of some simulation (or part of a simulation) and must not call out to other libraries or model code. It is expressed as a subprogram unit commonly known as a function, subroutine or method in Fortran, C and Java respectively.

- It must have one entry point, one exit point and cannot self-terminate.

- To simplify the scientific behaviour and make the coupling process easier, the model must only perform one computational timestep.

- All input and output to and from the model must be achieved using the FCA communication Application Programmers Interface (API).

Models that conform to these compliance rules can be used with other FCA-compliant models irrespective of their internal details. The scientific compatibility between two or more models is still a major issue but it is out of the scope of this thesis to discuss this additional problem. Many existing models may fail compliance at the first hurdle – relying on code from other models or libraries. However, it is possible that dependencies such as external libraries of code may also be made FCA compliant and included in the coupled model so that other models can make use of the code, reinforcing the underlying themes of reusability, extensibility and flexibility.

2.5.1. Describe-Compose-Deploy

In the FCA methodology, three phases of activity are involved in coupled model development and they are termed the Describe-Compose-Deploy (DCD) process [16]. At each phase different concerns are addressed: the definition and expression of FCA compliant models, the composition of FCA compliant models to form a coupled model and finally the deployment configuration for the executable coupled model.

2.5.2. Describe

The description phase involves defining the FCA compliant model in three distinct ways. Firstly, a high-level view of the model or, more accurately, its interface is defined. The interface forms a straightforward scientific view of the model, capturing basic data input and output requirements such as *'field1 is temperature data going out' or 'field 2 is volume of CO2 coming in'*. Secondly, more detailed information is defined regarding details of the model implementation such as input/output data formats, model language and hardware/compilation compatibilities. Finally, the executable expression of the model is described detailing the location of the binary file, the model timestep rate and additional compiler options such as datatype sizes. The three different views of the model enable changes to be easily assimilated. For example, a model expressed in both Fortran and C may have different source descriptions, but share a common interface description. Likewise, compile time options can be replaced without affecting the description of the model interface or source code description. In Chapter 4, it is shown how this flexibility is reflected in the BFG implementation through the use of separate metadata documents.

2.5.3. Compose

The composition phase is, in essence, the connecting of model inputs and outputs. These are referred to as **`fields`** in the general literature [16]. Each model involved in the composition may give data to or take data from some other model. In the literature these operations are known as **`put()`** and **`get()`** respectively. The capability for a model to **`put()`** and **`get()`** is defined by its interface developed at the *define* phase. Other, more complicated, scientific factors are also involved at the composition stage such as differing timestep rates between the provider and receiver models. Discrepancies between model timestep rates (and other such differences) may be overcome using the appropriate transformers that aggregate the data.

2.5.4. Deploy

The final deployment phase describes how the coupled model is to be executed. It enables the flexibility to run different models on different architectures and under different environments. The communication type can be specified so that data values are transferred in a particular manner, using MPI, secure shell or possibly over a Globus grid [43], to give a few examples. Ultimately, the separate deployment phase enables the coupled model to be re-deployed onto different architectures and hardware without the need to change the composition.

2.5.5. DCD Cycle

The FCA DCD methodology is a 'separation of concerns' [9] [16] [48] in that it first separates the different aspects of coupled model development into phases. Secondly, it attempts to address the issue of separating the science of simulation from the computer science involved in achieving the simulation in a modular fashion. The DCD methodology does not have to be followed in a strictly linear manner (i.e. D-then-C-then-D). Instead the intention is that users following the methodology will jump back and forth between phases until they have achieved their goal.

The following chapter gives a brief background in XML technologies referenced frequently throughout this thesis. Chapter 4 then introduces an implementation of the FCA which makes use of the XML technologies described in Chapter 3.

Chapter 3: Review of XML, XSLT and XPath

Due to frequent references to XML, XSLT and XPath in this document, it is essential to provide at least a basic understanding of these technologies. Readers already familiar with the operation of these systems may omit this chapter completely. For readers unfamiliar with the concepts, a brief overview is given below. In particular, features of each technology that are frequently used or referred to in this document will be emphasised.

3.1. XML

Extensible Markup Language (XML) is a technology that allows documents and data to be described in a flexible yet computer-readable format. It built upon an older technology (SGML [27]) and is now an official W3C standard [17].

XML documents consist of elements, tags and data values. An element is comprised of tags and data values and may be made up of zero or more child elements. The following example shows how name and age data about a person might be captured. Tags are the values between < and > characters which start and end an element. In this example, the data values ('Jones' and '25') lie between the start and end tags of the **name** and **age** elements respectively. The **person** element has only child elements (**name** and **age**) and no data values of its own.

```
<person>
   <name>
      Jones
   </name>
   <age>
      25
   </age>
</person>
```

Elements may also include attributes which appear after the element tag and contain a data value enclosed in quote marks, for example:

```
<person name="Jones">
   <age>
      25
   </age>
</person>
```

The decision to use attributes or child elements to encapsulate data values is at the programmer's discretion and there are no real advantages or disadvantages to each technique.

3.2. XPath

XPath uses path expressions to select nodes or node-sets in an XML document. XPath is a major element in the W3C's XSLT standard and is now an official W3C recommendation in its own right [18].

As well as defining the syntax for finding information in an XML document, XPath also includes a library of simple functions such as mathematical operations, date/time comparisons and string manipulations.

XML documents are treated as a tree-like structure of nodes. Each node can be one of the following types: element, attribute, text, namespace, processing-instruction, comment, or document (root) nodes. The XPath expression is analogous to specifying a certain 'branch' of the tree – and the data that is returned is everything we find at the end of the branch – which maybe a further sub-tree of nodes or just a single data value.

XPath is best understood with an example. The following code extract is a simple XML document:

```
<person>
   <name>Jones</name>
   <age>25</age>
   <address>
      <name>Holly House</name>
      <housenumber>10</housenumber>
      <postcode>M145XY</postcode>
   </address>
</person>
```

XPath expressions are used to select certain parts of the document. In this example, /person selects all matching nodes from the root of the document. As everything in this code extract is a child of the person element, using the /person Xpath expression would return the entire document.

Using a more specific Xpath expression, such as **`person/name`**, selects all matching nodes throughout the document and in this example would return:

```
<name>Jones</name>
<name>Holly House</name>
```

If we just wanted the data values from the above expression we would specify **`person/name/text()`** which would return:

```
Jones
Holly House
```

XPath expressions can become extremely complex and a complete description of the syntax is well beyond the scope of this thesis. The basic use of XPath as a powerful and recognised way to extract information from an XML document has been highlighted. For further information please see [18].

3.3. XSLT

Extensible Stylesheet Language Transformations (XSLT) is a language that allows users to transform XML documents into XML, HTML, XHTML, or plain text documents. A transformation in the XSLT language is expressed using an XML document. XSLT is an official recommendation by the W3C [19].

XSLT heavily relies on XPath. XPath expressions are used inside the XSLT file to determine which parts of the XML document are to be transformed. When an XSLT processor finds a 'match' it generates the corresponding output as specified inside the XSLT document.

XSLT and XPath allow XML documents to be transformed into other forms. A simple example is shown below. The XSLT stylesheet (below) specifies that the presence of an **`<option><java>`** structure inside the XML document should produce the output "System.out.println("foobar");". Notice that the XPath expression has been embedded in the match attribute of the template element:

```
<stylesheet version="1.0"
   xmlns="http://www.w3.org/1999/XSL/Transform">
   <output method="text"/>

   <template match="/option/java">
      System.out.println("foobar");
   </template>

</stylesheet>
```

By applying this stylesheet to the XML document:

```
<option>
     <java></java>
<option>
```

The result is the following output:

```
System.out.println(“foobar”);
```

This is an extremely simple example. XPath and XSLT technologies are enormously flexible and have the potential to be used in far more complex scenarios.

Chapter 4 introduces an implementation of the FCA called the Bespoke Framework Generator (BFG). The BFG implementation and other Chapters of this thesis are heavily based on the XML related technologies described in the chapter above.

Chapter 4: The Bespoke Framework Generator

The Bespoke Framework Generator (BFG) [3] is a prototype implementation of the Flexible Coupling Approach (FCA) developed at Manchester University. The following section gives a brief summary of how BFG operates. In particular it shows how BFG uses metadata captured in XML and then XSLT processing to produce wrapper code in which the components can be executed. In the description below 'framework' is synonymous with 'wrapper code'.

The BFG follows the DCD methodology outlined in the FCA (section 2.5). For this reason, it can be described as a framework-creation environment rather than a framework itself. Armstrong makes this important distinction in his MSc thesis [5]. He notes that:

> *"…the framework code output from the DCD environment acts as structure within which each deployment unit may execute and through which the models may communicate."*

When Armstrong wrote his thesis [5], the implementation of the BFG was in its infancy. While it has since been developed further, the basic operation of wrapper code creation is the same, and is described below.

- Step 1) The user produces XML metadata describing FCA-compliant models as well as the composition and deployment of these models using the FCA methodology.

- Step 2) Checks are performed to ensure the scientific and computational compatibility of the proposed coupling. Armstrong proposes that this can be achieved by validating XML against Schemas or by employing some other validation mechanism. At this stage, checks to 'validate' the composition as a whole (e.g. enforce any restrictions that would otherwise break the coupling or its execution) can be performed.

- Step 3) Wrapper code is generated based on the parameters in the metadata.

For the purposes of this thesis, steps 1 and 2 are most relevant and will be described in much greater detail in sections 4.1 - 4.8 and Chapter 5. However information about step 3 is also important, and will be briefly described below.
Originally, BFG used a purely Java-based solution to create wrapper code but in later releases this aspect was replaced by XSLT processing. A full description of this technology is well beyond the scope of this thesis, but the general principle is relatively straightforward.

An XSLT document instance defines a set of transformations that can be made to an XML document. Inside the XSLT document, a user can specify certain parts of an XML document to look for. If a certain structure or value is encountered, a specified value can be output. This technology is particularly suitable for the BFG. Using the metadata describing the coupling and a library of XSLT documents (defined by the BFG developer), an XSLT processor can observe certain structures, patterns and values in the metadata and output the appropriate wrapper code as specified inside the XSLT document. For instance, if the user specifies that communication should be achieved using MPI rather than simple buffers, the XSLT transform will output communication code that uses MPI.

It must be noted that the XSLT technology is not strictly *writing* the wrapper code – it is merely outputting what it encounters inside the XSLT document. The wrapper code is written by BFG developers who then embed it into the transformation declaration.

Once wrapper code has been auto-generated, the final operation is to compile and link it with the individual model source code. Currently this is a manual process and users can take many approaches such as shell scripts or a custom built environment such as SoftIAM [26]. Finally, the coupled model can be executed.

4.1. BFG metadata summary

This section describes how BFG is configured through the use of XML. Sections 4.2-4.5 first describe the XML needed to express a model and sections 4.6-4.8 then show how model XML metadata is used to express a composition.

For each model a user wishes to include in a composition, four XML documents are required. The following paragraphs describe each one in detail.

4.2. Model Interface

The 'interface document' describes the inputs and outputs made during the model's execution and provides a high level view of the model. While a **<behaviour>** element is available to describe the model's **type** and **compatibility**, the majority of the interface document is concerned with describing the model input and output. This is represented by a repeatable **<fields>** element that contains child elements **<name>**, **<id>** and **<direction>**.

The 'interface document' is intended to be an abstraction of the underlying model it represents. The format of the inputs and outputs (e.g. array or scalar, float or double) is not considered in the interface. A single interface document may be applicable to many models, particularly different implementations of the same underlying science. For example, model x and model y may both take as input a single value and output a single value. The science behind each model is to compute the square root of the input value and output the result. Inside the models' source codes this may be achieved in very different ways – but the basic operation or high-level view of each model is essentially the same – an input, a computation of square root and an output. Therefore one interface document can be used to express both model x and model y. The same principal can be applied to models performing more complex computations with numerous inputs and outputs.

4.3. Model Source

The second XML document describes the source code of the model. It details the language of implementation, for example Fortran77 or Java, as well as information about the author and author's department. The majority of the source definition is concerned with describing the inputs and outputs of the model source code, as seen in the interface definition. However the source definition is a more low-level view and therefore the fields are described in greater detail. In addition to providing an **id** and a **direction** users may describe the field units of the communication (e.g. m^3 or Watts) and whether the

data is an **array** or **scalar** type. If it is an array further details are required about each dimension such as **size**, **units** and **variable** type. This detailed information is used by BFG to construct the correct implementation of the **put()** and **get()** calls, such as allocating correct buffer sizes and variable types.

4.4. Model Executable

The third XML document required describes the execution details of the model. In this document the user may define the timestep rate of the model (e.g. 30 seconds or 6 hours). BFG uses the timestep rate for each model in the composition together with the run duration of the coupled model to auto-generate the control loops and determine how many times a model code is called during execution of the coupled model. A location element describes where the model is located with respect to machine name, directory and filename (e.g. cronus.mcc.ac.uk, /home/user/modelname, model.f).

4.5. Model Runtime

Finally, a fourth XML document can be provided to describe run-time behaviour of the model. In the current implementation of BFG this document is not used or processed and for the purposes of this project this document can be safely ignored.

The sections above describe the XML metadata needed to express an individual model. Sections 4.6-4.8 describe how these are combined to form the metadata of a coupled model.

4.6. Composition Document

To begin with, a user must specify which models are connected together. A composition document is used to specify which model requires the data and which model provides the data. As two models may exchange more than one piece of information, for every **<connect>** element between two models, a user may define multiple **<fieldconnects>** sub-elements. A **<runDuration>** element is used to denote the total simulation time. For example, if each model in the composition has a timestep of one minute and the **<runDuration>** is one hour, then each model code will be called 60 times. Specifying a 'suitable'

<runDuration> value can be a problematic task as it is related to the timestep of each model in the composition. Currently, the user must manually find a common denominator between all timestep rates.

4.7. Deployment Document

Once a composition is complete, the user must specify how it is to be deployed. Deployment metadata describes which models in the composition are to be executed on a certain piece of hardware. Multiple **<deploymentUnit>** elements can be defined if the user wishes to execute models on different machines. A machine is specified using a machine name (e.g. cronus.mcc.ac.uk). For each model a user wishes to execute on the deployment unit, a **<model>** child element is defined. In addition to defining the deployment unit configuration, a framework element (consisting of 3 child elements: **fabric**, **language** and **comms**) is also needed. The framework describes how the execution is to proceed (single machine or multiple machines), the programming language of the auto-generated control code (wrapper code) and the way in which models are to communicate (e.g. simple buffers or message passing). A discussion of the combinations of **framework** child elements is given in Chapter 7.

4.8. Coupling Document

Finally, to bring all the other documents together, a root or 'coupling' document is defined which contains the paths to all the documents described above. A repeatable **component** element defines each model in the coupling (the paths to each of the interface, source and executable documents) and **compose** and **deploy** elements specify the paths to the composition and deployment documents respectively. The path of the coupling document forms the parameter that is passed to BFG and it is from this document that BFG reads the paths of the rest of the metadata.

Chapter 5 details the problems associated with BFG metadata creation and shows how errors can be time consuming and costly.

Chapter 5: BFG metadata errors

In chapters two and four, the methodology of the Flexible Coupling Approach (FCA) and an implementation of this technology called The Bespoke Framework Generator have been described. It has been shown how BFG uses XML model metadata to generate framework code which can then be compiled and linked to form an executable coupled model.

Although on the surface it would seem an easy task to generate model and coupled model metadata, the following section describes the intricacies and inherent difficulties involved in defining the BFG metadata. Additionally, this section will highlight the nature of the XML metadata problem which forms the basis of the research described in this thesis.

5.1. Issues with BFG XML metadata

There are two main issues with regard to BFG XML metadata: (1) errors that prevent BFG execution and (2) errors that propagate into a coupled model execution. These will be described below.

5.1.1. Errors that prevent BFG execution

XML metadata is currently written out by hand using a text editor or an XML editor such as XMLSpy [44]. As a human factor is involved, it is possible that the XML will contain errors, for example missing elements, extra elements or invalid syntax.

As discussed in Chapter 4, for any particular coupled model, four XML documents are required for each individual model and a further four documents to describe the composition, deployment and coupling are required for the BFG to generate wrapper code. So, to generate wrapper code for a coupled model involving the interaction of 4 models, the user is thus required to write 20 individual XML documents; four for each individual model and four to describe the coupling/deployment. This inevitably leads to mistakes, especially if the user is unfamiliar with XML and XML syntax.

The consequence of invalid XML is that at BFG runtime it is likely that the BFG-processor will not be able parse the documents. This will prevent framework

creation until the errors in the metadata have been fixed. Currently the only solution to this problem is a trial and error approach; if the BFG executes then the user need not worry, however if BFG does not execute the user must search through the XML manually and try to identify where the error occurs.

5.1.2. Errors that propagate

The second main issue, which relates to step 2 of the framework creation procedure outlined in Chapter 4, is the scientific and computational compatibility of models and in particular, the exchange of data between them.

The BFG relies solely on the parameters expressed in the metadata to auto-generate wrapper code. Therefore it is important that the user has correctly specified which inputs and outputs are connected between models. The scientific compatibility between models checks whether it is valid 'science' to pass a variable or set of variables from one model to another. Ensuring this compatibility is extremely difficult and to some extent is beyond the scope of this thesis. For the purposes of this research there is more interest in, for example, ensuring the connections specified by the user are computationally valid. For instance, detecting situations where the user has specified the connection of input to input.

It is not just the connections between models that can be at fault. Other factors such as deployment unit specification and the model definition documents (source, interface, executable) all have intricate relationships that, from a hand-crafted XML point of view, are easy to make mistakes with. As a simple example, it would be sensible to only allow a model to be assigned to one deployment unit. However, it is easy for a user to make this mistake as writing XML in a text editor does not alert them to the fault. There are many more such rules that should be obeyed; however this is covered in Chapter 7.

The consequence of badly-formed metadata is more serious than if the XML simply contains structural or syntactic errors. Syntactic errors (errors that prevent BFG execution) can be easily recognised using Schemas[1] and an XML tool such as XMLSpy [44]. However, if we assume that the badly-formed metadata does not prevent the execution of the BFG (i.e. it is still valid XML)

[1] Schemas are discussed in Chapter six.

then the resulting coupled model may still execute. The badly-formed coupled model could produce erroneous behaviour that hinders or stops the execution or produces erroneous scientific results. This can waste valuable time on expensive computational resources and in trying to discover why the coupled model produced invalid output. More seriously, the user may not even realise the execution was erroneous and base scientific research on the results from a badly-formed coupled model.

5.2. Tyndall Centre: Use-case scenario

In the Tyndall Centre for Climate Change Research at the University of East Anglia [25], researchers have integrated the BFG into a software environment named SoftIAM [26]. SoftIAM is described as:

> *"…a community integrated assessment modelling system which hosts many scientific modules coupled into various integrated assessment models." [26]*

Fundamentally, SoftIAM attempts to combine the principles of coupled modelling with a user-friendly way to carry out scientific modelling experiments. For instance, it attempts to abstract some of the low-level computational issues such as compiling, linking, deploying and executing away from the users of the SoftIAM system through the use of a web-based "SoftIAM Portal".

As mentioned above, SoftIAM adopts principles from the Flexible Coupling Approach and uses BFG as a core technology. Models intended for use in the SoftIAM environment, which are made SoftIAM compliant, are therefore also BFG compliant (or vice versa). This includes defining metadata for each model that is compliant with the BFG metadata documents as described in a previous section.

Currently SoftIAM is in its infancy and as such, a large user base is not yet established. However Tyndall Centre researches are themselves using the system to produce model couplings that require the development of SoftIAM/BFG compliant models and metadata. Currently this is a manual process involving a technically-literate user who constructs model metadata documents using a simple text editor.

To bring focus back to the topic of this thesis it is possible to observe that SoftIAM is heavily reliant on BFG metadata and, as the XML is currently produced by hand by a Tyndall researcher, it exposes SoftIAM to the issues discussed in section 5.1.

To illustrate the possible complexity of the metadata involved, the Tyndall Centre SoftIAM development team supplied example coupled model metadata. The coupling that the metadata describes is used in an active research project to simulate the global effects on average earth surface temperature and average sea-level, based on carbon dioxide emission data [28].

The simulation involves two models; 'ESM' which provides the CO2 emission data and 'MAGICC' – a simple climate model. In total, 127 fields are defined in the source metadata (and repeated in the interface metadata). The coupling itself consists of 21 connections between 42 fields.

With so much XML metadata, all coded by hand, errors may easily creep in. The effects of errors are further magnified, in this particular scenario at least, by the fact that BFG has been integrated into the SoftIAM environment. Errors in the BFG metadata may in turn lead to malfunctions in SoftIAM. A metadata error detection mechanism is therefore highly desirable.

In Chapter 6 a detailed review of related work in detecting errors within XML documents is given. It investigates the types of errors that may be present and proposes solutions to solving the problem. In Chapter 7 specific examples of relationships that exist in the BFG metadata are presented and in Chapter 8 the principles discovered in Chapter 6 are applied to the BFG metadata.

Chapter 6: Related work in XML constraint management

In Chapter 4, the operation of the BFG was described and in Chapter 5 the importance of error-free BFG metadata was highlighted. It showed why errors in the metadata may be the result of syntactically incorrect XML or due to the user making mistakes with relationships between values in the metadata.

In section 6.1 a review of XML Document Type Definitions (DTDs) [45] and XML Schemas [47] is presented. XML Schemas provide a way to ensure that XML is syntactically valid but they are limited in other aspects. In section 6.2 a categorisation of typical (XML) document checks is given and it is shown how Schemas cannot be used to validate all categories. In section 6.3, three types of solution to handling XML document validation are presented and in section 6.4 example implementations are described. This chapter concludes with a discussion of an implementation proposed by Allyn Treshansky [8] as part of the UK Metrological Office 'FLUME' project [9].

6.1. XML DTDs and Schemas

One of the biggest, if not the biggest, design features of XML is its extensibility. It was developed from older technologies as a way to express documents and data in a computer readable, universal format. XML1.0 is the W3C specification that defines the tags and attributes that may appear in an XML document [17]. Any XML document which strictly obeys the XML1.0 specification is termed as a *well-formed* document instance.

However, merely checking that an XML document instance is well-formed is not sufficient to ensure that the document is free from error; in fact this barely scratches the surface. As a typical commercial example, a document may be missing a crucial **<InvoiceNumber>** element but because the rest of the document remains well-formed it is considered to be error free.

To address this issue, developers initially supplemented XML with another technology known as Document Type Definition (DTD) [45] – a legacy throwback that formed part of SGML (which may be considered the precursor to XML). A DTD defines the legal building blocks or collections of elements needed to create a correctly structured XML document instance. For example, a DTD can be

defined that specifies that **<InvoiceNumber>** *must* be present – any XML instance document without this element does not conform to the DTD and will flag an error accordingly. Despite the advantages that DTDs give, they do have limitations. Foremost is the fact that DTDs utilise their own syntax, which is not well-formed XML. Consequently, older XML processors not only had to parse well-formed XML, but the DTD syntax as well. Furthermore, DTDs also lack support for complex datatypes and thus became more unsuitable as XML use in more advanced applications (such as databases) grew.

In response to the limitations that DTDs posed, the W3C founded the XML Schema working group [46]. The goal was to replace DTD technology with a new specification called XML Schemas [47] and eliminate the restrictions DTDs imposed.

The most notable difference between DTDs and Schemas is that Schemas are well-formed XML. Applications or processors which can parse well-formed XML can also parse Schemas. Unlike DTDs, XML Schemas also offer the use of datatypes, which can add a great deal of control to the validation process. For example, with an XML instance document for an inventory, a user might want to limit the **<quantity>** element to containing only positive integers. This would be impossible to accomplish with a DTD, but can easily be done with an XML Schema.

By supplementing XML technology with DTDs or their more advanced counterpart, Schemas, the concept of *validity* is introduced. An XML document instance is described as valid if it can be successfully shown to follow the structure outlined in a corresponding DTD or Schema. Additionally, the concepts of a document being both well-formed *and* valid can be combined, and only then is such a document said to be 'correct'.

A full description of the advantages of using XML Schema is well beyond the scope of this thesis; however it is important to note the most significant benefits. It is shown how BFG uses XML Schemas to describe the structure of the metadata.

The first main advantage is the ability to check for structural validity. A Schema can specify that certain elements *must* exist and can also ensure that elements

appear in a strict order. It may also specify that elements contain specific child elements, which themselves are then subject to certain Schema conditions, or that an element may only occur a certain number of times (otherwise known as multiplicity validation).

The Schema that specifies the structure of BFG **`source`** XML metadata uses structural validity to ensure that certain elements must exist in the metadata. For example, if a user has omitted to include the **`<language>`** element, the metadata is not considered valid. This is useful as the BFG needs this information to successfully generate the correct wrapper code.

The second main advantage is the ability to check for data and datatype validity. For each element a user defines in the Schema, it is possible to specify the type of data that should appear in the element in any XML instance document. As with most programming languages, there are in-built primitive types like strings and integers. However, one of the strengths of Schemas is the ability to declare complex data types such that a user can specify their own datatype. Furthermore, it is possible to derive a new complex type by inheriting from an existing complex type. Consequently, the derived complex type can either add more declarations to the base complex type (using extension) or can restrict the declarations (using restriction). It is also possible to define the format of primitive data type elements such as strings and integers so that, for example, a string must obey some simple pattern or an integer must be greater than zero. Likewise, a user can express an enumeration for a particular element so that any instance document must contain only a value from those specified in the Schema.

It is possible to observe both data and datatype validity examples in the BFG source Schema. For instance, a non-negative integer type is declared to ensure that elements of that type contain integers greater than or equal to zero. Similarly, the **`directionType`** element is restricted to a value from the enumeration **`{in,out}`**.

From the examples given above, it is immediately possible to recognise the extra power of supplementing XML with Schemas. It is easy to supplement XML documents with Schemas and any validating parser can ensure documents are well-formed and valid. Developers of the BFG have taken advantage of this

technology and issued BFG Schemas for all metadata documents required for successful execution. Therefore, a user can use the Schemas for reference during metadata creation and BFG itself can use the Schemas for validating the metadata.

6.2. XML Constraint Classification

It initially seems as though Schemas solve many of the structural and data validation issues involved when representing data in XML. However, the following section will show how Schemas really only solve very simple cases of validation. In response, several technologies have emerged that attempt to address this deficiency.

As will be seen in the following sections, validation of XML documents takes more than just ensuring structural and data type integrity. For this reason, literature refers to the concept of validation in XML documents as *constraint checking* or *content constraining*. Constraint checking encompasses the concept of validation but extends it to include other checks that can be made, as described below. For the remainder of this thesis, the terminology 'constraint checking' is adopted.

In the work described by Jacinto et al from the University of Minho [1], four categories of constraint checking are outlined. The following table, taken directly from this work, gives a small description of each constraint category.

1	Domain range checking	*"This is the most common constraint. We need this type of constraint when we want a certain content/value to be between a pair of values (inside a certain domain). Normally, this kind of constraint is used when data is of type date or numeric."[1]*
2	Dependencies between two elements or attributes	*"We have cases where an attribute value depends on the value of another element or attribute located in a different branch of the document tree. These are clearly context dependent constraints."[1]*
3	Pattern matching against a Regular Expression	*"Sometimes we need to guarantee that content follows a certain format (as in the case of dates: there are more than 100 formats, or telephone numbers)."[1]*

4	Complex constraints	*"We call this last kind of constraints complex because we group here all the remaining constraints, usually weird: they require a nested loop behaviour or complex nested calculations." [1]*

Similarly, in work by Hu and Tao from Pace University [2], content constraints are categorised into seven classes.

1	Well-formedness constraints	*"Those imposed by the definition of XML itself such as the rules for the use of the < and > characters and the rules for proper nesting of elements."[2]*
2	Document structure constraints	*"How an XML document is structured starting from the root of a document all the way to each individual sub element and/or attribute."[2]*
3	Data type/format constraints	*"Those applied to the value of an attribute or a simple element."[2]*
4	Value constraints	*"The value (range) of an element/attribute that cannot be specified by a DTD or XML Schema document; such constraints could be either static or dynamic."[2]*
5	Presence constraints of attributes and/or elements	*"The presence of an attribute or element and the number of occurrences of an element, which could be either static or dynamic."[2]*
6	Inter-relationship constraints between elements and/or attributes	*"The presence or value of an element/attribute depends on the presence or value of another element/attribute."[2]*
7	Consistency constraints	*"Corresponding elements/attributes in multiple documents have consistent values."[2]*

Hu and Tao have termed constraints 1 and 2 as syntactic constraints, whilst 3 to 7 are termed semantic constraints. The constraints in categories 1 through 3 can be specified by DTD or Schema documents and validated with an XML validating parser as discussed in previous sections. However, Schemas are somewhat limited to these simple syntactic constraints and thus constraints which fall into categories 4 to 7 cannot be dealt with by Schemas or DTDs alone. Similarly, the "complex constraints" as identified by Jacinto et al are also impossible to express with Schemas and DTDs as they involve calculations.

Hu and Tao make further categorisations by differentiating between static and dynamic constraints and by the way that the constraint is expressed:

> *"If an XML semantic constraint is conditional to its environment, we say it is dynamic; otherwise we say it is static. A dynamic constraint may impose different limitations on an element or attribute for different instance documents defined by the same Schema." [2]*
>
> *"A constraint can be expressed in the form of an assertion (true/false statement) or a conditional rule (if-then) with embedded assertions. While in theory the constraints could be all expressed as assertions, rule-based constraints allow for more natural and concise specification of many types of constraints.*
>
> *For an assertion-based constraint, it can be simple or composite depending on whether it involves one element/attribute or more.*
>
> *For a rule-based constraint, it can be simple if it is of an if-then structure; or composite if it contains an else clause or nested rule-based constraints." [2]*

McLaughlin [6], makes only two distinctions in checking constraints, classifying them as either 'coarse-grained validations' or 'fine-grained validations'.

> *"Coarse-grained validation is the process of ensuring that data meet the typing criteria for further action. Here, "typing criteria" means basic data constraints such as data type, range, and allowed values. These constraints are independent of other data, and do not require access to business logic. An example of coarse-grained validation is making sure that shoe sizes are positive numbers, smaller than 20, and either whole numbers or half sizes.*
>
> *Fine-grained validation is the process of applying business logic to values. It typically occurs after coarse-grained validation, and is the final step of preparation, before one either returns results to the user or passes derived values to other application components. An example of fine-grained validation is ensuring that the requested size (already in the correct format because of coarse-grained validation) is valid for the requested brand. V-Form inline skates are only available in whole sizes, so a request for a size 10 1/2 should cause an error. Because that requires interaction with some form of data store and business logic, it is fine-grained validation.*
>
> *The fine-grained validation process is always application-specific and is not a reusable component, so it is beyond the scope of this series. However, coarse-grained validation can be utilized in all applications, and involves applying simple rules (data typing, range checking, and so on) to values." [6]*

McLaughlin has simplified the issue by classifying simple validation checks as "coarse-grained" and everything else is "fine-grained". This is similar to Jacinto et al. whose "complex" category is used to classify anything that doesn't fall into the first three of their categories.

Dodds [15] ignores all simple cases of validation and classifies exceptions to this as "difficult" [15] constraints He gives three examples of types of these difficult constraints which are "hard, or impossible to express with other schema languages." [15]:

> *"Where attribute X has a value, attribute Y is also required."*
>
> *Where the parent of element A is element B, it must have an attribute Y, otherwise an attribute Z."*
>
> *The value of element P must be either "foo", "bar" or "baz"."[15]*

Because Schemas often fall short of expressing all the types of constraints that users are typically interested in, several technologies have emerged that attempt to 'fill the gap'. In section 6.3 three types of solution are described and in section 6.4 current example technologies or implementations which fall under each solution type are presented.

6.3. XML Constraint Solution Classification

Hu and Tao [2] and Costello [7] propose three options for supplementing or extending Schemas to express constraints that are otherwise inexpressible by Schemas alone:

- Supplementing XML Schema with another XML constraint language
- Writing program code to express semantic constraints
- Expressing semantic constraints with an XSLT/XPath stylesheet

6.3.1. Supplementing XML Schema with another XML constraint language

Supplementing XML Schema with another XML constraint language involves using a pre-existing constraint language/tool built specifically for constraint checking and validation in XML documents. As we shall see later, there are already a few tools which provide this functionality which gives an advantage to

this approach. The biggest disadvantage is that the user must become familiar with both the tool and language, though this is true of most solutions to a problem. Another disadvantage is that none of the tools are official W3C recommendations so they all differ in approach and implementation. This is a difficulty because it is unclear whether the tool can solve the user's particular constraint problem until it is tried and tested. Finally, as many (if not all) of the tools are non-commercial solutions built by individuals, documentation and tutorials are very scarce.

6.3.2. Writing program code to express semantic constraints

The biggest advantage of this approach is that with a single programming language any constraint is expressible, regardless of complexity. The solution can be tailor made for the problem and once it is written, can be updated by the developer if the need arises.

The biggest disadvantage to this approach is the fact that it is a bespoke solution suitable only for the problem it originally addresses (i.e. some instance or instances of XML that contain specific data). If the underlying structure of the XML changes, then the bespoke solution will need either updating or rewriting. This requires a developer familiar with the language to rewrite and perhaps recompile the program code.

6.3.3. Expressing semantic constraints with an XSLT/XPath stylesheet

In Chapter 3 the principles behind XSLT were briefly discussed. The BFG uses XSLT to transform the parameters in the metadata to framework code. Using the same principle it is also possible to express logical statements and conditional rules that check the content of XML documents. By expressing semantic constraints with an XSLT/XPath stylesheet, this approach leverages the power of XSLT technology and avoids the need for another programming language. However, stylesheet documents are not particularly human-oriented or reusable and creating complex stylesheets can be a great challenge.

6.4. Current Solution Approaches

In the previous section three different approaches for constraint specification in XML documents were defined and advantages and disadvantages to each approach were highlighted. The following sections contain specific implementation examples of each approach.

6.4.1. Supplementing XML Schema with another XML constraint language

During the last few years a number of different XML schema languages have appeared as suggested replacements for the ageing Document Type Definition (DTD) and to supplement XML Schemas. The major XML constraint languages in the literature are Schematron [11], XML Constraint Specification Language (XCSL) [12], XincaML [13], eXtensible Constraint Markup Language (XCML) [2] and Xlinkit [14]. These technologies are extremely similar in their approach and implementation, so this thesis will only focus on one of them: Schematron.

Schematron is a supplementary XML schema language designed and implemented by Rick Jelliffe at the Academia Sinica Computing Centre, Taiwan [11]. It combines powerful validation capabilities with a simple syntax and implementation framework. Jelliffe's main aim was to produce an easy-to-use technology through which a user could validate "complex" constraints (see section 6.2) but also diagnose failures (as opposed to merely outputting a binary success/failure result). While it does have other uses (which are beyond the scope of this thesis), the general purpose of Schematron is to essentially 'fill the gap' that DTDs and Schemas leave behind with regard to XML document validation (see section 6.1 and 6.2).

A meta-stylesheet is an XSLT document that generates other stylesheets. Schematron uses this principle to '*map*' complex XSLT expressions into simpler "core Schematron elements". The user writes a document conforming to the Schematron language which is translated, using the official Schematron meta-stylesheet, and outputs further validation XSLT stylesheets. The XSLT output is then applied to the original XML document to produce a validation report that details any validation failures, together with the reason for failure.

The basic building blocks of the Schematron language are the **assert** and **report** elements. Constraints are expressed as assertions (Boolean tests) about certain patterns in the XML document. The patterns are defined using XPath expressions and are grouped together to form **rules**. Constraints are applied within a certain context so that a **rule** element has a **context** attribute that defines the candidate nodes to which the constraint is to be applied. Rules are collected to form **patterns** and finally patterns are grouped to create the overall Schematron Schema.

The concepts above are best understood using a simple example. The following code extract shows a portion of an example Schematron Schema.

```
<pattern name="Invoice Checks">
   <rule context="//invoice">
      <assert test="count(productID) >0">
         An invoice must have at least one product ID
      </assert>
      <assert test="customerID">
         An ivoice must have a customer reference
      </assert>
   </rule>
   <!-- additional rules -->
</pattern>
<pattern name="Other Constraints">
   <!-- other rules -->
</pattern>
```

The above portion of code shows how two assertion tests have been grouped under the rule-context XPath expression of **//invoice**. The first test ensures that there is at least one productID as part of the invoice element. The second rule tests for the existence of a child **customerID** element. It is also possible to see that a string can be embedded before the closing **assert** tag. This text forms part of the validation report allowing the user to determine which constraint has been violated. As described above, the rules are grouped into pattern elements and we can define multiple pattern elements to perform different constraint checks as indicated by the pattern **name** attribute.

Once the constraints have been expressed in the Schematron Assertion Language they are translated using the Schematron meta-stylesheet into pure XSLT expressions. For example, the following **assert** core Schematron element:

```
<sch:assert test="...">...</sch:assert>
```

…is mapped to the XSLT below:

```
<xsl:choose>
<xsl:when test="..."/>
<xsl:otherwise>...</xsl:otherwise>
</xsl:choose>
```

The resulting XSLT is then applied to the XML document and the output is captured in the validation report.

To avoid having both a Schematron Schema *and* a regular XML Schema as separate documents, Schematron assertions may be included inside a regular XML Schema by inserting them into the **<xsd:annotation>** element. This consolidates the validation and constraint checking into just one document (the Schema itself) and makes it simpler for the user to express all checks in just one file.

6.4.2. Expressing semantic constraints with an XSLT/XPath stylesheet

The best example of this approach is one which is packaged with BFG itself. Within the BFG release is a directory called 'constraints' in which a selection of XSLT files can be found. Each of these is responsible for checking a certain constraint. For example, 'ConnectedGets.xsl' ensures that for each model in the composition, all input fields are connected to an output field from another model.

To run the constraint check, a batch file or shell script is provided for Windows and Unix users respectively. The batch file takes the path to the coupled.xml file (see Chapter 4) and calls a Java XSLT processor XALAN [49] for each XSLT constraint file. The XSLT transform is applied and the output is sent to the command prompt. The intention of this development is to allow users to check the correctness of their composition before attempting an execution of BFG. The script is invoked using a simple command line operation relative to the BFG home directory:

```
./constraints/runtests.sh <path_to_coupling_document>
```

Below is the output from the XSLT transform applied to an example model that is packaged with the BFG release:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<info>Checking array dimension id's.</info>
<info>Checking complete.</info>
<?xml version="1.0" encoding="UTF-8"?>
<info>Checking that model names are consistent.</info>
<info>Checking complete.</info>
<?xml version="1.0" encoding="UTF-8"?>
<info>Checking that compose model names are valid.</info>
<info>Checking complete.</info>
<?xml version="1.0" encoding="UTF-8"?>
<info>Checking that the source and sink of all connections have the same
datatype.</info>
<info>Checking complete.</info>
<?xml version="1.0" encoding="UTF-8"?>
<info>Checking that the source and sink of all connections have
the same number of data elements.</info>
<info>Checking complete.</info>
<?xml version="1.0" encoding="UTF-8"?>
<info>Checking that all gets are connected.</info>
<info>Checking complete.</info>
<?xml version="1.0" encoding="UTF-8"?>
<info>Checking that each get is not connected more than once.</info>
<info>Checking complete.</info>
```

From the script output above, it is possible to observe that the model metadata has successfully passed all of the constraints. To highlight how the output changes if errors are encountered, the coupled model metadata was intentionally broken by resetting the direction of a single field in the source document from 'in' to 'out'. For clarity, the constraints that were successful have not been included as the output is the same as shown above.

```
<?xml version="1.0" encoding="UTF-8"?>
<info>Checking that the source and sink of all connections have the same
datatype.</info>
<error>from "sam" id "1" datatype "real" to "average" id "1" datatype ""
Datatypes [real] and [] do not match</error>
<info>Checking complete.</info>
<?xml version="1.0" encoding="UTF-8"?>
<info>Checking that the source and sink of all connections have
the same number of data elements.</info>
<error>from "sam" id "1" n elements "2" to "average" id "1" n elements ""
number of data elements [2] and [] do not match</error>
<info>Checking complete.</info>
```

From the output it can be seen that by changing the direction of just one field, multiple constraints have been violated. This serves as an illustrative reinforcement of an earlier observation that there are complex relationships inherent in the structure and values of the metadata that are not instantly obvious and go beyond simple validation checks.

This particular implementation of a solution to the XML metadata constraints problem is susceptible to 'corrupt' XML. For instance, a single XML file (from the example model comprising 20 files) was intentionally corrupted by the insertion of just one character in an invalid manner. After rerunning the testing script the output became spurious and reported numerous errors in the metadata – there

was no indication that there was simply an invalid character in one of the files. A user producing XML by hand has a high likelihood of inserting the occasional invalid character and without an indication of where this has occurred, mistakes can become very time consuming.

A further observation is that although the output is not necessarily easily read by humans, it is well-formed XML and thus a secondary processing application could possibly be used to mark-up the output for better human intelligibility.

The number of constraints that are checked by this XSLT solution is far from complete. In reality, it merely serves as an example of how constraints may be addressed and would need extending to include the full complement of constraint checks to be more useful. However, the nature of the implementation lends itself reasonably well to this task. Firstly, the XSLT expressions are written in a text document format and no specialist tools are required for editing other than a basic text editor. Secondly, XSLT does not require complicated compilation procedures as the XALAN parser simply reads the XSLT files, applies the specified transforms and then outputs the results. Thirdly, this implementation has separated the constraints management away from the BFG architecture and thus an extension to the rules requires no knowledge of the BFG internal operations.

The biggest disadvantage to this implementation is that a good knowledge of XSLT and XPath is very important. XSLT and XPath are not typically very 'user-friendly' languages and can pose an extremely steep learning curve. This is particularly true when attempting to express some of the more complicated constraints which involve multiple XML documents combined with logical expressions and nested looping behaviours. The result is a need to write large amounts of XSLT to perform even simple checks. For example, to ensure the source, interface and executable documents have consistent model names and versions, 114 lines of XSLT is required. XSLT can be difficult to debug and does not provide an easy way to update the constraints. Lengthy XPath expressions also add to the complexity of this solution, as the code extract below shows:

```
<xsl:if
test="not(boolean(document($root/bfg:coupled/bfg:component/bfg:model/bfg:source
,$root)/bfg:source[bfg:name=$ProvidesModelName and
bfg:version=$ProvidesModelVersion]))">
<error>
<xsl:text>provides model name "</xsl:text>
```

```
<xsl:value-of select="$ProvidesModelName"/>
<xsl:text>" version "</xsl:text>
<xsl:value-of select="$ProvidesModelVersion"/>
<xsl:text>" in compose document is not found in any model document.</xsl:text>
</error>
</xsl:if>

<xsl:if
test="not(boolean(document($root/bfg:coupled/bfg:component/bfg:model/bfg:source
,$root)/bfg:source[bfg:name=$RequiresModelName and
bfg:version=$RequiresModelVersion]))">
<error>
<xsl:text>requires model name "</xsl:text>
<xsl:value-of select="$RequiresModelName"/>
<xsl:text>" version "</xsl:text>
<xsl:value-of select="$RequiresModelVersion"/>
<xsl:text>" in compose document is not found in any model document.</xsl:text>
</error>
</xsl:if>
```

The full transform can be found in appendix B.

6.4.3. Writing program code to express semantic constraints

Although no specific implementation of this type of solution actually exists, we can discuss the general principles and technology that would be involved.

A simple solution that immediately lends itself to this type of problem is to use some kind of high-level language such as Java or C++. Using Java, it would be possible to leverage the power and familiarity of the language itself alongside free and well-developed libraries of code that provide XML parsing facilities.

The first task would involve reading the XML documents into the Java environment. Several technologies such as Castor [29], Xerces2 [36] and Jakarta Digester [42] already exist to facilitate this operation. Each one differs slightly in it's approach to handling XML but essentially they all allow access to XML documents within the Java environment. A more detailed description of Castor is provided in Chapter 8.

Once the XML representation has been read into the programming environment, it is possible to write method calls which look for values and element structures inside the new representation. At this point, the XML-level detail has been abstracted into the Java environment and thus the full power of the language is at our disposal. Logical expressions and nested loop behaviours are relatively straight forward, especially when used in conjunction with 'helper' method calls that can abstract the search for specific variables or perform aggregations and calculations.

Once the evaluation of constraints is complete, the resulting output can be as verbose as required given that the code has the flexibility to pinpoint the exact nature of any violations that are found. The option of outputting the results in any format is also in the hands of the designer; from raw text at the command line to styled HTML.

As discussed in section 6.3.2, the biggest disadvantage of such a solution is the fact that to edit, delete or extend the constraints a recompilation of the source code is needed. Additionally it is likely the constraints will be distributed throughout the source code, implemented as various method calls and complicated procedures; therefore it would be necessary to become familiar with the code to make intricate modifications.

6.4.4. The hybrid approach

The hybrid approach, so called for the purposes of this thesis, involves certain aspects from each of the three types of solution outlined above. This, currently unpublished solution has been proposed by Allyn Treshansky [8] as part of the UK Metrological Office 'FLUME' project [9].

Treshansky proposes the use of an XML logical language to describe relationships between elements and values. He has written an XML schema which defines the syntax and grammar of the language. A user can express constraints on any XML document by writing an XML document that conforms to the constraints schema specified by Treshansky.

An expression is built from 'operators' and 'tokens'. Tokens represent values to operate against and are either entered into the XML document explicitly e.g. '1' or '2' or via a reference. If a reference is used, the user is able to specify other parts of the XML document (or even another XML document) using XPath expressions. The XPath expression is set as an attribute to the **<reference>** element. Operators are either unary or binary with the current implementation supporting {NOT} in the former group and {AND, OR, LESS THAN, GREATER THAN, LESS THAN OR EQUAL TO, GREATER THAN OR EQUAL TO, EQUAL} in the latter group.

Each constraint expression is formed in a way similar to a high-level language IF statement but without the ELSE clause. If the expression evaluates TRUE, a user-specified action occurs. In the current implementation, textual data indicating what constraint has failed is output, although Treshansky proposes the action "ought to be something a bit more complicated" [8] such as suggesting corrective measures.

To illustrate the ideas discussed above, a typical expression has is presented below. The following expression is valid to Treshansky's logical Schema language and tests the constraint that a CONNECTION must consist of just two FIELD elements with directions IN and OUT respectively.

```
<constraint level>1</constraint level>
 <switch>
  <case>
   <if>
    <expression>
     <unary operator op code="NOT">
      <binary operator op code="AND">
       <binary operator op code="EQ">
        <token>1</token>
        <reference
xpath expression="count(//connection/fieldRef[@direction='in'])"
file path="myComposition.xml"/>
       </binary operator>
       <binary_operator op_code="EQ">
        <token>1</token>
        <reference
xpath expression="count(//connection/fieldRef[@direction='out'])"
file path="myComposition.xml"/>
       </binary operator>
      </binary_operator>
     </unary_operator>
    </expression>
   </if>
   <then>
    <action>invalid connection</action>
   </then>
  </case>
 </switch>
</constraint>
```

Another interesting aspect of Treshansky's design is the use of a **<constraint_level>** element to specify the 'severity of the constraint'. Analysis of the Treshansky Schema shows us that this value is an unbounded non-negative integer so the user seemingly has the freedom to set the severity of a constraint on some arbitrary scale.

To execute the constraints, Treshansky proposes to implement the following steps:

- Step 1) Use Perl to make an XSL transform to the constraint XML, which transforms each constraint expression into XSL

- Step 2) Apply the XSL output from step 1 to the model metadata XML documents to produce Perl expressions

- Step 3) Evaluate the resulting Perl expression from step 2 to produce a TRUE or FALSE value

- Step 4) Depending on the outcome of step 3, take the appropriate action as specified by the original constraint XML

This process seems rather complicated partly because an extra transformation is needed (step 1) to 'extract' the XPath expressions. As we observed earlier, the constraint XML has XPath expressions embedded into element attributes and therefore we first need to translate this XML into a true XSL document. This can be seen more clearly by using an analogy of marking up Java statements inside an XML document. The XML document itself is not directly parseable by a Java Virtual Machine but translating the XML and pulling out the Java expressions would potentially end up with something that *is* parseable. Similarly, the output from step 1 above would produce the following parseable XSL:

```
<xsl:template match="/">
  ( (
  <xsl:value-of select="count(//connection/fieldRef[@direction='in'])"/>
  ) eq ( 1 ) ) and ( (
 <xsl:value-of select="count(//connection/fieldRef[@direction='out])"/>
  ) eq ( 1 ) )
</xsl:template>
```

The XSL output above is then applied to the composition metadata (step 2) which outputs Perl code as shown below:

```
$expression = ((1)eq(1))and((1)eq(1));
```

Perl is then used to execute the above expression and the Boolean result value determines if the action specified in the original constraint XML is performed. A detailed explanation of how this occurs is not given by Treshansky.

This has been termed a 'hybrid approach' because it adopts aspects from the main three approaches detailed earlier. Firstly Treshansky has specified a new

language for logical expression of constraints in any XML document meaning that this is not just a BFG centred solution. Secondly he leverages the power of XSLT, XPath and meta-stylesheets by performing transformations to output additional XSL and Perl code. Meta-stylesheets are used in Schematron and using standard XSLT transforms is similar to the pure XSLT solution above. Finally, to produce the transformations and perform evaluation of the resulting Boolean expression, Perl is used. Although it is not strictly an entirely Perl based solution it does play a huge role in the operation of the approach. Treshansky's approach, though not fully implemented, seems a viable way of constraining XML documents.

However, the complicated multi-step, multi-technology nature of Treshansky's approach has discouraged its use within the FLUME project. Treshansky comments that "I think that it is the complexity of files like [the constraints XML] and my attempts to explain them that caused people to balk at this way of handling constraints… the constraint problem was recognised, discussed, and then shelved because people couldn't agree." [8].

The previous chapter showed how the issue of constraint management in XML documents has been approached in several ways. The following chapter presents specific examples of constraints that are present in the structure of the BFG metadata. In Chapter 8 the principles outlined above are applied in an attempt to detect errors in the metadata.

Chapter 7: BFG Constraints

In previous chapters, the architecture and principles behind the FCA has been presented and it was also discussed why it is a benefit to the scientific community. An implementation of this approach called the Bespoke Framework Generator was also described. The BFG is configured through the use of XML metadata that contains more complicated relationships between structures and data than those checked by a Schema alone. In various literatures these relationships are termed 'constraints' [1] [2] [12]. Chapter 4 presented the consequences of errors occurring in the metadata, classifying them as 'errors that prevent BFG execution' and 'errors that propagate'. Removing errors from the BFG XML metadata has the potential to save a lot of user time and money. It was shown in Chapter 6 that there are three general types of solution to checking constraints in XML documents and advantages and disadvantages were highlighted for each approach. Finally we discussed state of the art example technologies for solving the XML constraint problem, giving an overview of design principles and where possible, implementation details.

In the following chapters a more detailed description of the constraints *specific* to the BFG is given. The intention is to show why the constraints are needed and that a majority of them fall under the general umbrella classification of 'complex'.

7.1. Inter/Intra Constraints

The constraints on the BFG metadata can be classified as *inter* or *intra* document constraints. Intra-document constraints are those which are concerned with relationships *within* the same XML document. For example, a successful BFG execution requires 4n+4 metadata documents, where n is the number of models in the coupling. Each document has its own internal constraints that must be adhered to in order to provide a successful BFG execution. Inter-document constraints are those in which relationships 'cross the boundary' to include other XML documents, for instance relationships between fields in each **`source.xml`** for models which are connected together by **`compose.xml`.**

In the following sections we document some of the typical constraints that need to be evaluated to ensure a successful BFG execution and coupled model

execution. A description of the full complement of constraints is lengthy and somewhat unnecessary so instead the reader is provided with a 'flavour' of the rules, some of which are fairly simple and some that are quite complex. Each constraint is described by the metadata document/s that are involved, a brief textual description, it's relation to the XML and then a summary of why it is needed.

7.2. Intra-document constraints

Document context: source.xml
Textual description: Array dimension indexes are 1,2,3…n.
Relation to XML: For each **<fields>** element that contains **<dimension>** elements, the **<dimension>** elements must have **<index>** elements whose values are 1,2,3…n.
Reason for constraint: This constraint ensures that data communicated between models is stored and retrieved from the correct array dimensions. If two array dimensions have the same index number it is possible that a data value may be put (or retrieved) from the wrong place during execution.

Document context: souce.xml
Textual description: Models only communicate datatypes supported by the language they are implemented in.
Relation to XML: The value of the **<vartype>** element which is a child of **<fields><type><array><dimension>** and **<fields><type><scalar>** must be within the domain of the values specified for the language specified by the value of the **<language>** element.
Reason for constraint: This constraint ensures that the datatype of fields is a valid datatype for the language of the source code of the model. For instance Fortran77 does not support 'float' so a model implemented in this language cannot support a field of this datatype. This is further complicated by the fact that the domain of acceptable **<vartype>** values for each language is currently not captured by the metadata i.e. there is no way to tell (from the metadata) if 'int' or 'float' are acceptable for language 'Fortran77'.

Document context: source.xml
Textual description: The **<vartype>** is the same for all array dimensions

Relation to XML: The value of the **<vartype>** element which is a child of **<fields><type><array><dimension>** must be equal for all instances of the value that are part of the same array i.e. the parent of their parent is the same array element.

Reason for constraint: On the whole, programming languages do not allow an array to contain different datatypes in each dimension – an array is declared to be just one datatype for every dimension. Therefore we must ensure every field has a consistent vartype value for every dimension.

Document context: deploy.xml

Textual description: The combination of fabric, language, comms, number of deployment units and model languages is supported

Relation to XML: The combination of the values of the **<fabric>**, **<language>** and **<comms>** elements of the parent **<framework>** element must be within the domain of allowable combinations. The count of **<deploymentUnit>** elements is also related to the combination of the values of the fabric, language, comms elements.

Reason for constraint: One strength of the BFG is the ability to 'redeploy' a coupled model to different communication configurations and platforms by simply changing a few options at the deployment phase. Therefore the deployment options have intricate relationships with each other. For instance, a coupled model which is configured to run certain parts on physically different machines cannot support a 'shared buffer' data communication mechanism – the code is simply not running in the same physical memory space. The following table summarises the valid combinations:

Fabric	Language	Comms	Deployment Units	Model languages
single-machine	Fortran77	sequential	1	Fortran77
single-machine	Fortran77	mpi|MPI	1	Fortran77
globus2.0	Fortran77	mpi|MPI	Any mapping	Fortran77
tdt|TDT	Fortran77	TSOCKETS| tdtsockets	1 per model	Fortran77
tdt|TDT	Fortran77	TDTSSH|	1 per model	Fortran77

		tdtssh		
oasis3	Fortran90	mpi|MPI	1 per model	Fortran90
webservices	Java	SOAP	1 per model	Fortran77 | C

This is a complex constraint to enforce but also vital for a successful coupled model execution. It involves dependencies between the values of elements combined with the count of the occurrence of the **<deploymentUnit>** element. To strictly enforce the rule it becomes an inter-document constraint relating to how many models we have in the coupled model, how many deployment units are defined and whether the language of each model is suitable for the current deployment configuration. Additionally, the metadata does not capture the 'allowable' combinations – the table above was supplied by the BFG developers. Finally, to complicate matters even further, as more functionality is added to the BFG implementation the combinations of valid values will change. For instance, the BFG developers could possibly add a new communication mechanism to the communication library. The new choice of communication type will have its own relationships between other options such as fabric and language types.

7.3. Inter-document constraints

Document context: source.xml, interface.xml, executable.xml
Textual description: Model name and version are the same for a models xml documents (interface, source, executable).
Relation to XML: The value of element **<executable><name>** is equal to the value of element **<source><name>** and the value of element **<interface><name>** and the value of element **<executable><version>** is equal to the value of element **<source><version>** and the value of element **<interface><version>**.
Reason for constraint: As described in earlier chapters, an FCA-compliant/BFG-compatible model is described by three metadata documents. Each of these documents must contain elements depicting the name and version of the model they are describing. To ensure a user has not made a mistake such as using the wrong interface document, a simple check is needed to check that the model name and version are consistent across its metadata documents.

Document context: source.xml, executable.xml

Textual description: All source vartypes have corresponding executable datasize e.g. real is 4, double precision 8

Relation to XML: Let x be the bag of values of the **<source><fields><type><array><dimension><vartype>** and **<source><fields><type> <scalar><vartype>** elements. For each value v in the bag x there exists an element **<executable><datasize><type>** with value v.

Reason for constraint: Some programming languages support compile-time options to specify the size of datatypes in the compiled executable, for example Fortran77 supports the option to compile all REAL variables as 4 bytes or 8 bytes. BFG reflects this flexibility by allowing a user to declare the size of datatypes in the executable document. Therefore for each vartype found in the source document (describing the datatype of the parent field) a corresponding datasize element must exist in the executable document detailing the intended size of the variables of that datatype in the executable.

Document context: compose.xml, source.xml, interface.xml, executable.xml

Textual description: Connected fields between models are 'compatible'.

Relation to XML: Too complex to intelligibly summarise.

Reason for constraint: This constraint is possibly the most important to validate as it ensures that the communication between models is viable. It is also the most complex involving every XML document in the metadata except for the deployment document. It is important to the successful execution of a coupled model because the exchange of data between models has a high risk of halting the execution or producing bad scientific behaviour. For instance, if a data value is passed between two models sent as an 8 byte real but received as a 4 byte real, it is easy to see that some buffer will probably overflow. In the event that the execution does not halt, it is still at risk from producing strange results because the data value used by the receiving model is essentially corrupt.

To validate this constraint we can break it down into smaller, more manageable rules as detailed in the following list:

For each connection of two fields we must ensure:

- Directions are opposite, IN to OUT or OUT to IN.
- All model inputs/gets() are connected.

- Each input/get() is connected only once.
- Field units are the same.
- Field types are the same, ARRAY to ARRAY or SCALAR to SCALAR.
- If field types are array:
    - The count of dimensions is equal.
    - The size of each dimension is equal.
    - The product of dimension sizes is equal.
    - The unit of corresponding dimension is equal.
    - The vartype of corresponding dimension is equal.
    - The vartype size of corresponding dimension is equal.
- If field types are scalar:
    - The units are equal.
    - The vartypes are equal.
    - The vartype sizes are equal.

The above constraint is considerably more complex than other constraints documented in this chapter but essentially it can be viewed as the logical conjunction of smaller, more manageable rules. Despite this, some of the smaller rules still constitute more than a simple domain range check or presence check and therefore are relatively complex in their own right.

In this chapter it has been shown how the relationships between elements and values in the BFG metadata are quite complex. If these relationships are not expressed correctly by the user it can lead to errors that propagate into the execution of the coupled model. Chapter 6 presented solutions to this problem and in the following chapter these solutions are implemented, tested and evaluated.

Chapter 8: BFG constraint management implementation

In the previous chapter a selection of some typical constraints in the BFG metadata was given. For each example given, it was also shown why the constraint is important to a successful execution of the BFG and the coupled model. In earlier chapters, the prevention of errors in the metadata was discussed as an important time and cost saving measure. In the following chapter a detailed description of attempted implementations to solving the BFG constraints problem is given.

8.1. Design Considerations

The aims and objectives of this project were as follows:

- (i) If possible, suggest a new method of constraint management for the BFG that eliminates the negative aspects of existing methods, especially the complexity involved.
- (ii) If possible, allow constraints to be easily updated so a coupled model developer has the flexibility to 'relax' or 'tighten' the level of checking.

In Chapter 6, state of the art methods of XML constraint management found in the literature were presented. An incomplete but working version of constraint checking, written specifically for the BFG, was detailed. Also reviewed was a method labelled in this thesis as the 'hybrid approach' which was proposed and partially implemented by Treshansky.

Design goal (i) above is concerned with improving existing methods of constraint management within the BFG. At present, there is no *complete* constraint management solution for the BFG. The most comprehensive solution is that which has been implemented by the BFG developers. This approach consists of a library of XSLT transforms managed by a simple shell script. While the solution is appropriate for some situations, it was considered incomplete because it only caters for approximately 25% of BFG metadata constraints.

Possibly the most obvious solution to the BFG constraints issue is to build upon the existing method of constraint management by writing XSLT documents to fulfil the entire complement of identified constraints. However, as documented in

an earlier chapter, this solution suffers from three fundamental problems that make it unattractive as a solution candidate:

- During evaluation tests, corrupt/ill-formed XML was not handled gracefully.
- XSLT and XPath technologies have a steep learning curve.
- The expression of each constraint involves many lines of XSLT (e.g. more than 100 lines are needed to express the (relatively simple) constraint that the value of an element is consistent across three XML documents). This can introduce further mistakes.

While the first issue above can be solved using a well-formed document check before applying any XSLT transformations, the latter two issues, despite being rather speculative in nature, are more pressing reasons to find a better solution. The expression of XML constraints using pure XSLT is recognised as being a laborious task and is a major reason why XML constraint checking languages such as Schematron, XCML and XSML are in development. To complicate matters further, the programming paradigm of XSLT is very different from standard procedural languages and writing XSLT requires more knowledge than just mapping to a new syntax. As the constraints for BFG metadata constitute more than straightforward domain range checks (or similar), the result is that constraints expressed in XSLT are hard to understand at source-code level, time consuming to code and require a knowledge of XSLT and XPath to edit or manipulate.

The intention of design goal (ii) is to allow the constraints solution to be flexible enough to cope with a) changes in the BFG metadata and b) users who wish to edit the constraints. The former becomes an issue if, for example, new functionality is added to the BFG implementation. For instance, if new deployment configuration options are implemented that change the relationships of existing constraints or generate new relationships that need enforcing. Ideally, a simpler and less time consuming way for BFG developers to reflect the new changes in the constraints solution would be useful. The latter issue arises partly because of the flexibility of BFG itself. If every constraint were to be strictly enforced, there would be no room for a coupled model developer to 'break' the rule if they deem it necessary for the purposes of their coupled model. For instance, in Chapter 7 the constraints needed to ensure a valid connection between two models were documented. However, in a real situation it could be

possible for the science being modelled to work even though the models are not communicating in the same field units.

Similarly, the user may wish to create a new rule that is important to the domain of the coupled models they are developing and hasn't been included in the standard complement of constraints. In an ideal situation, the constraints should be expressed in such a way that editing or manipulating them does not require the need for complicated recompilation procedures. This gives them the flexibility to edit the rules at will, keeping them focussed on the science of model coupling and avoiding computer science related issues.

In summary, the design goals that emerge out of our initial objectives are as follows:

- Provide a more concise way to express the constraints, making the rules easier to comprehend and interpret and taking less time to code.
- Enforce the full complement of BFG constraints.
- Avoid, as much as possible, the need for recompilation or other computer science related procedures when editing and manipulating the rules.
- Provide a reasonable amount of extensibility.

In the following sections three attempted solutions to the BFG constraint management problem is given. We first investigate Schematron, a state of the art XML constraint language. After this, other avenues of approach are implemented, discussed and evaluated.

8.2. Constraining with Schematron[2]

As documented in Chapter 6, Schematron is a new approach that allows users to express constraints on XML documents that are otherwise impossible to express with standard W3C Schemas. The following section shows how Schematron can be used to validate the constraints in the BFG system.

Schematron is built on top of XSLT and XPath technology, so all that is needed to perform validation using Schematron is an XSLT processor. XSLT output from the

[2] A description of the Schematron principles can be found in chapter six.

Schematron meta-stylesheet is then applied to the instance documents to produce the validation results.

There are several Schematron-based projects which combine the entire Schematron validation procedure into one simple-to-use tool. Two of the most notable implementations are the <oXygen/> XML Editor & XSLT Debugger [20] and the Topologi Schematron Validator [21]. For users wishing to take a more API-based approach, a small number of implementations exist, including Schematron.NET developed by Daniel Cazzulino [22] and Jing developed by Thai Open Source Software Center Ltd [23]. The former is a set of classes implemented in C# and the latter is implemented in Java. Other Schematron based projects are documented by Jelliffe [24].

Schematron was chosen over other existing solutions such as XCML primarily because it is referred to throughout the literature, well recognised as an XML constraint approach, is in a mature development state and has tools and APIs freely available for use that implement the validation procedure.

As the <oXygen/> XML Editor is not free to use, Topologi Schematron Validator (TSV) was chosen to investigate the use of Schematron as a method of constraint management.

To evaluate Schematron, two Schematron meta-stylesheets were developed to express some typical BFG constraints. The first meta-stylesheet expressed two constraints against the deployment metadata for a coupled model. The second meta-stylesheet expressed four constraints against the source metadata for an individual model and is shown below for ease of reference:

```
<schema  xmlns="http://www.ascc.net/xml/schematron">
<title>Schema for BFG Source documents</title>
<pattern name="field dimension checks">

   <rule context="fields/type/array">
      <assert test="count(dimension) > 0" icon="bug 11.gif">
         An array must have at least one dimension
      </assert>

      <assert test="(dimension/vartype) = (dimension[1]/vartype) "
icon="bug_11.gif">
         array contains different varitypes
      </assert>
   </rule>

   <rule context = "dimension">
      <assert test="../../../../language = 'fortran77' and ((vartype) =
'integer' or (vartype) = 'real')"   icon="bug_11.gif">
```

```
         not language vartype
      </assert>
   </rule>

   <rule context="fields">
       <assert test="sum(type/array/dimension/index) =
(type/array/dimension[last()]/index div 2)*(type/array/dimension[last()]/index
+ 1)"   icon="bug 11.gif">
          Array dimension NOT 1..n
      </assert>
   </rule>

</pattern>
</schema>
```

The first assertion ensures that all array elements have at least one dimension defined.

The second assertion ensures that the vartype for each array dimension is equal to the first dimension vartype for that array (i.e. they are all the same).

The third assertion makes sure that the vartype of each dimension is in the domain of allowable vartypes for the specified language of the model. In this instance we are only allowing vartype 'real and 'int' as permissible, though this rule could be extended by adding others to the domain of possible values. The allowable vartypes for a given language are not captured by the metadata, so rather than comparing them against some other XML file this rule has been 'hardcoded'.

The fourth assertion is an attempt to test for uniqueness of the values of the index elements for each array dimension as well as ensuring that they follow the sequence of positive integers from 1,2,3...n (where n is the number of dimensions for the array in context). As Schematron is built on top of XSLT and XPath technology it is limited to the set of functions provided by XSLT and XPath. XPath does not provide a function to test for uniqueness nor test for a dynamically bounded sequence of numbers. This highlights a weakness of the Schematron approach.

In the example given, a work-around was devised using the Guassian formula n*((n+1)/2), where n is the value of the last index element. By calculating the sum of the index values and comparing it to the Guassian sum it is possible to determine if we have a sequence from 1,2,3...n. This is not a perfect solution as there are other situations where a sequence of integers will produce a Guassian sum but it was an improvement on performing no check at all.

Given the limitation imposed by the XSLT/XPath base, it would seem natural, for computer scientists at least, to think about devising some form of sub-routine or function to provide uniqueness and sequence checking. However, fundamentally, Schematron was not built for providing this type of functionality. Only XPath expressions can be inserted into the assertion test attribute and as such it was not possible to divert the flow of control and perform sub-routine calls.

Using TSV, the above Schematron Schema is combined with the XML instance metadata from a sample model and a validation report is given back to the user. The following screenshots illustrate this process. Figure 8-1 shows how, after the Schematron meta-stylesheet was defined, it was specified to be used against an XML instance document.

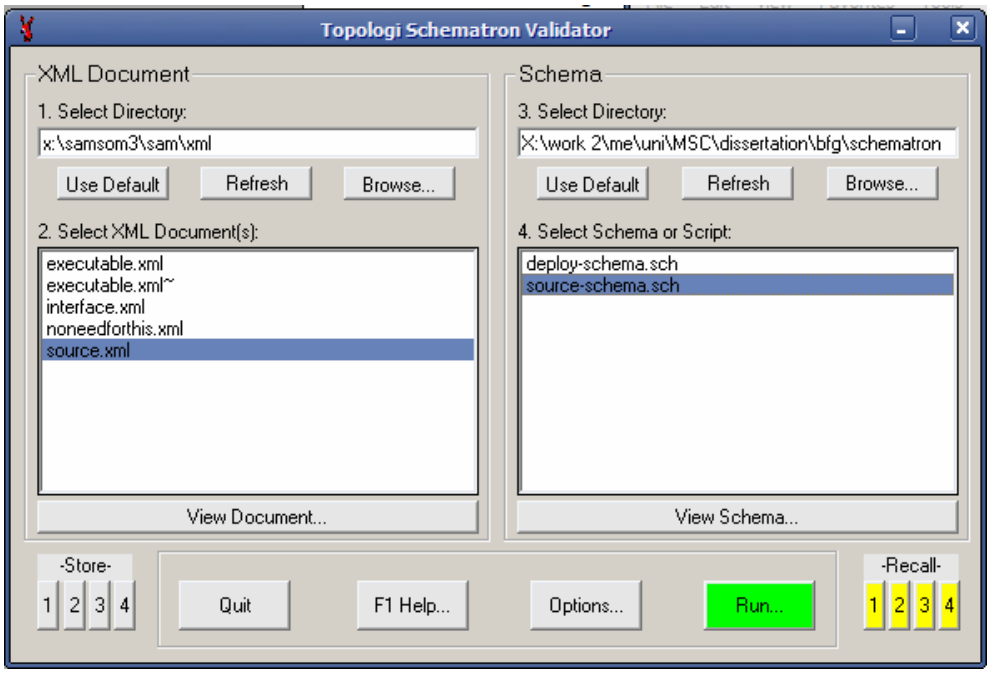


Figure 8-1, TSV configured for execution

After running TSV, the validation report indicates that the model source XML document contains a field where the array dimension index elements do not have values 1,2,3...n. The validation report as output by TSV can be seen in figure 8-2 below.

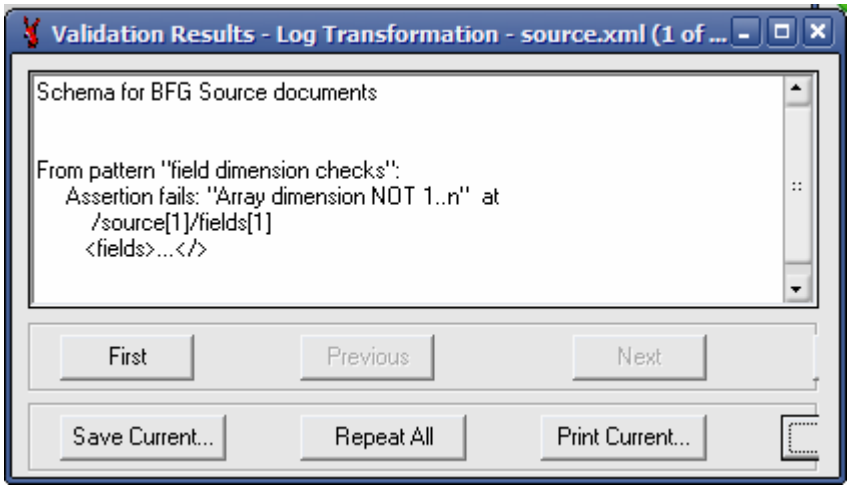


Figure 8-2, TSV validation results

The output from the Schematron validation process is formatted as plain text (as shown in figure 8-2) and includes the description of what has failed, along with the XPath expression of the element that has violated the constraint. This is useful for determining exactly where the problem is located.

Initially, Schematron seemed a likely solution for the problem of XML constraints checking and to some extent it is. As we have shown, performing simple constraint checks such as

```
count(dimension) > 0
```

or

```
(dimension/vartype) = (dimension[1]/vartype)
```

is trivial and the validation results are easy to interpret as a human observer. In fact, for intra-document constraint checking of simple 'business logic' (e.g. *Iis there an invoice number?*, *is the total order price the sum of the items + VAT?*), Schematron is very suitable and is well received throughout the literature.

Complications with using Schematron become apparent when trying to express rules that are more complicated and more specific to the document structure of the BFG metadata. For instance, although the above examples achieve the task, they are not strictly *complete* representations of the actual rule. In reality the XML instance documents are defined as file paths inside a 'master' coupling document (as described in Chapter 4) and as such, the Schematron rules should be written in the context of elements contained in the main master coupling document. The Schematron validation process should then be initiated with just

one argument linked to its execution - the file location of the master 'coupling' document. This behaviour can be seen in the existing, incomplete constraints solution as published with the latest BFG release.

The above aspect of the metadata increases the complexity of the rules considerably as it inevitably involves using extremely long XPath expressions to reference elements in the 'secondary' documents defined by the master coupling document. Combined with the facts that some constraints are simply not straight forward and there is no interactive environment to assist in the creation of the rules (i.e. our definition tool is a simple text editor), it is easy to see how Schematron's initial elegance and advantages can get lost. The constraints become less concise and incredibly difficult to debug. The design goal to '*provide a more concise way to express the constraints, making the rules easier to comprehend and interpret and taking less time to code*' then becomes impossible to achieve.

8.3. Constraining with Java

Another way of approaching the constraints checking problem is to use a programming code such as Java, C++ or Perl. In this instance, an application was built using a pure Java environment to check the metadata constraints that need enforcing for a successful coupled model execution. The following section details the operation of the Java program and highlights the interesting discoveries that the approach uncovered.

As mentioned in a section 6.3.2, the main advantage of using a programming language to implement constraints checking is that it gives tremendous flexibility to express the constraints. A high-level language such as Java supports method calls that can perform operations such as testing for uniqueness and can be used to express concepts that are otherwise unavailable with languages such as XSLT or XML Schemas (such as sequences).

The main architecture of the Java constraints program is as follows:

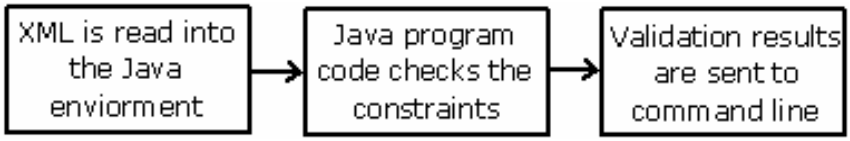


Figure 8-3, a conceptual view of the Java program,

To translate the XML into the Java environment three techniques were considered - a data binding tool, a DOM tree processor and a technology that can be considered a mix between data binding and DOM tree traversal called a digester.

Marshalling is the term used for the process of writing an XML document and *unmarshalling* is the process of reading it. Data binding is the process of transparently marshalling and unmarshalling XML to true Java objects. Before this can occur, however, an API is used to create the Java classes used for object instantiation. For example a **<deploymentUnit>** element in an XML document will result in the creation of a **DeploymentUnit** Java class. This is a one-off operation - the Java classes are created from XML Schemas at design-time. At runtime, the data binding API is used to unmarshall XML files to instantiations of Java objects which can then be used in the Java environment in the usual manner. Objects instantiated from the classes created by the data binding API, support the facility to marshall their contents back to XML files. The XML files created are well-formed and valid to the Schema that defined the classes. There are several data binding tools available for the Java environment including Castor [29] and JAXB [30].

Using Castor, a 'library' of Java classes was automatically created representing the elements found in the BFG metadata. The ease with which data can be read from XML and written back to XML was a crucial reason for this choice of technology.

Once the XML metadata has been marshalled into Java objects, the process of validation becomes simply a task of writing Java code to express the various constraints across the Java objects. For example, the following code extract from the prototype Java program ensures that array dimension indexes are unique and sequential (using method calls to abstract some of the detail):

```
private void checkArrayDimensionIndexes(int arrayID){
```

```
     int[] indexes = getArrayDimensionIndexIDs(arrayID);
     boolean result = areUniqueAndSequential(indexes);

     if (!result){
             textreport.add("Array dimension index IDs are not unique and/or
sequential for array " + arrayID);
             xmlreport.add("
<report>
     <desc>Array dimension index IDs are not unique and/or sequential
     </desc>
     <item>"+arrayID+"</item>
</report>");
     }
}
```

Eventually the rules become more and more abstract and become a collection of method calls as the following code extract illustrates:

```
private void checkConstraints(){
     //check definition
     checkArrays();
     checkConsistencyOfModelNames();

     //check composition
     checkInputsOutputs();

     //check deployment
     checkDeploymentConfiguration();
}
```

The most important thing that was learnt from taking this approach was the recognition that the abstraction of the constraints to simpler expressions of 'business rules' or logic was far more important than how those rules were actually implemented. For instance, the Java program achieves exactly the same objective as the XSLT and Schematron implementations - they all check the constraints and 'correctness' of the metadata and report back any errors found. How the constraints are computationally evaluated is really not a big issue as long as they are abstracted in such a way that there is a clear separation between the *expression* and *evaluation*. In the Java approach the constraints unintentionally evolved into a list of method calls with the function name (or a preceding comment) describing the nature of what the call is going to check for. In the XSLT approach packaged with BFG, the constraints are abstracted to the level of a batch file.

The observation allowed for an alternative investigation to be pursued. Whereas the focus before was on performing constraint checking at a metadata level (i.e. against the XML) the possibility that this was too 'low level' became apparent. The expressions required to ensure constraints are not violated (at the metadata level) are, based on experimental evidence, unavoidably complex, 'non-concise'

and inevitably involve the amalgamation of various pieces of program logic to perform the operation. In the following section a shift of focus to a more logical perspective circumvents some of the problems that have been encountered.

8.4. Object Constraint Language

In previous sections and chapters, several approaches and implementations at expressing constraints on BFG metadata have been documented. Each approach can be seen to have two things in common:

- Constraints are directly expressed on the structures and data found in the metadata at a low-level i.e. on the XML itself.
- Despite trying a variety of technologies, expressing the constraints inevitably results in solutions that are as time-consuming to express and hard to interpret as the original XSLT solution being improved upon.

In section 8.3 it was suggested that performing constraint checking at the metadata level could possibly be the cause of a lot of the complications. The following section moves away from XML-level constraint checking and instead shows how mapping to a 'logical' level or higher-level perspective allows for the expression of constraints that, due to their logical nature, are ultimately more concise, easier to interpret and easier to express.

This section begins with a short review of constraint technologies that do not strictly deal directly with XML but in the Java environment instead. A relatively new area of research termed 'Object Constraint Language' (OCL) [37] [38] [39] is then introduced. The section concludes with an implementation and evaluation of an OCL-based implementation to the BFG constraint issue.

Constraints ensure the integrity of data. Currently there is no native support to express constraints in the Java environment. Due to its popularity as a tool for implementing business applications (which ultimately rely on the integrity of the data they process) several mechanisms for expressing constraints in Java applications now exist. A full description of them all is beyond the scope of this thesis but the following is a summary of the most important features.

From the work proposed in section 8.3 questions arise as to why Java would need a constraint management system at all when, as shown, there is the flexibility to check the integrity of data (or objects) with code. However, rather than implementing bespoke constraints on Java classes every time a new application is developed, there is a more standard and less time consuming way to constrain data in the Java environment.

For example, an **Address** class may require that the **street** private member variable should not exceed 20 characters in length. A **Setter()** method may be used to access the variable and in this method a check to make sure the value being set does not exceed the constraint may be inserted.

```
private class Address{
     private string street;
     private Occupant[] occupants;
     …
     …
     public setStreet(string value){
             if (value.length > 20)
                     logger.log("constraint violated");
             else
                     street = value;
     }
}
```

Each time there is a new requirement for a string value not to exceed a certain length, the above code could be copied and pasted where needed. However, it is easy to see that if the constraint proves to be more complicated (perhaps dependent on the value of other class variables), copying and pasting the same code becomes tiresome and is also considered bad programming practice. Furthermore, the validation report produced for the user is handled in an ad-hoc manner and may not really indicate where or what the actual problem is.

One technique for performing constraint checking in a Java environment is to perform byte code instrumentation. In this method, a tool is used to modify the compiled Java classes at build time. Instructions or 'declarations' of constraints can be made in the Javadoc comments, in the Java methods or in an external text file to the class file. After compiling the Java application in the usual manner, the instrumentation tool reads the constraint declarations and modifies the compiled Java byte code accordingly. A good example of this approach is jContractor proposed by Karaorman, Hölzle, and Bruno [31]. The biggest advantage to byte code level instrumentation is that the source code remains, for the most part, unchanged – the extra code needed for validating the

constraints is 'injected' into the byte code. The biggest disadvantage to this technique is the need for more complicated compilation procedures and in some cases modifications to the Java Virtual Machine.

A similar technique is source code instrumentation. In this method, the instrumentation tool modifies the Java source code prior to compilation based on constraint declarations either in the Javadoc, in separate files or by using the API to define supplementary Java source code. Before compilation is performed, the instrumentation tool is executed against the source code and constraint handling code is inserted automatically based on the declarations found. The source code is then compiled in the usual manner to produce an executable application. Two implementations that take this approach are iContract, first proposed by Kramer [32], and the Dresden OCL Toolkit built and maintained by the Software Engineering Group at Technische Universität Dresden [33]. The greatest advantage to using source code instrumentation is that the developer retains far more control over the application. The modifications made to the program are easily observable (rather than obfuscated into the byte code). However, this then means that the source code can become bloated and confusing as various methods are overridden, replaced and renamed by the instrumentation tool.

An interesting aspect common to the approaches specified above is the separation of constraint declaration from constraint verification process. Each approach supports a means to express the constraint against the Java source that is independent of the way in which the declaration is processed, interpreted and verified. Each approach essentially uses a 'constraint language' through which the constraints may be declared. The declarations are interpreted and subsequently mapped to supplementary source or byte code which is then instrumented directly into the application.

Though not all the approaches use the same constraint language (for instance some use their own propriety language [31] [34] [35]) a few of the technologies use the Object Constraint Language [37].

The OCL is a notational language for analysis and design of software systems. It enables the description of expressions and constraints on object-oriented models and other object modelling artefacts and is being accepted as part of the Object Modelling Group (OMG) Unified Modelling Language (UML) 2.0 standard [38].

OCL is included as part of the UML 2.0 standard as a formal way to express semantics about a UML model. There are three main terms of the OCL. Firstly, an *expression* is an indication or specification of a value. Secondly, a *constraint* is a restriction on one or more values of (part of) an object-oriented model or system. Finally, an *association* can be viewed as a relationship between two classes.

As mentioned above, technologies such as iContract and the Dresden Toolkit leverage the power of OCL by mapping OCL notations embedded in either Javadoc comments or external files into Java source or byte code. For instance, consider the address example given previously:

```
private class Address{
     private string street;
     private Occupant[] occupants;
     …
     …
     public setStreet(string value){
             if (value.length > 20)
                     logger.log("constraint violated");
             else
                     street = value;
     }
}
```

Using OCL notation embedded in a Javadoc comment it is now possible to express the same constraint in a more 'standardised' manner i.e. true OCL syntax:

```
     //@OCL
     //inv: street.length <= 20

private class Address{

     private string street;
     private Occupant[] occupants;
     …
     …
     public setStreet(string value){
             street = value;
     }
}
```

As can be seen in the code extract above, the OCL expression replaces the code in the setStreet() method for a simple, one line declaration that appears in a comment at the top of the class file:

```
street.length <= 20
```

The OCL expression above represents the statement that *'the length of the street attribute must be less than or equal to 20 and this must always be true'*.

The true power and expressiveness of OCL becomes apparent if we consider a more complicated example. For instance, it may be true that the occupants of the address must all have the same surname.

Conventionally, a programmer might enforce this restriction with cumbersome code during **`get()`** and **`set()`** routines. However using OCL associations it is possible to define the expression:

```
//@OCL
//inv: street.length <=20
//association livedInBy between
//   Address[1] role homeOf Occupant[0..*] role //livesIn
//inv: livesIn.surname->asSet()->size()=1
```

In the above example an OCL association termed 'livesIn' captures the relationship between the **`Occupant`** objects and the **`Address`** objects. Using the association it is possible to specify that the set of all surnames of occupants for any address must be of size 1 i.e. all the occupants have the same surname:

```
inv: livesIn.surname->asSet()->size()=1
```

The above example highlights another useful aspect of OCL: the inclusion of powerful mathematical concepts such as set theory and lower order predicate calculus. The combination of business domain modelling techniques (such as relationships) and powerful mathematical expressions leads to an extremely expressive notation useful for expressing constraints in a concise and logical manner. A full description of the OCL features and syntax is available in [37] and [39].

8.5. BFG and OCL

To express constraints on BFG metadata the following architecture was considered:

- Use a data binding tool to auto-generate Java classes from BFG XML Schemas.

- Supplement the Java classes with concise, logical OCL statements embedded in the JavaDoc.
- Use an instrumentation tool to automatically 'inject' the verification mechanism based on the OCL statements defined.
- Run the Java program and unmarshall the XML instance documents to the Java environment and observe the report of the constraint verification process.

However, this suggested architecture does not lend itself particularly well to one of our initial design goals. Using this architecture, editing, adding or removing a constraint would require the original source code to be edited and then 'reinstrumented' and recompiled to reflect the change. Thus it involves a lot of computer science related issues to reflect even simple changes. An alternative architecture was therefore considered, which is described below.

A project called 'USE' is a system for the specification of information systems developed by Richters et al from the University of Bremen [40]. A USE specification contains a textual description of a model using features found in UML class diagrams (classes, associations, etc.). Expressions written in the Object Constraint Language (OCL) are used to specify additional integrity constraints on the model.

To utilise the USE environment, a specification of the system is needed. This is a text document containing the classes and association between classes. Each class can have attributes (class member variables), and operations (methods) defined. Attributes and operations can be typed to specify the data type of the variables. This is very similar to the object oriented paradigm with the additional support of extra declarations to define associations between classes and constraints for each class.

Once the system model specification has been declared, the USE environment can be instructed to load it through a command line argument pre-runtime or from the USE 'command console' during runtime. This action effectively 'configures' the USE environment to the given specification. After the environment has been configured, state manipulation commands can instantiate objects of the given classes.

The state manipulation commands allow the USE environment to represent a given state of the object oriented architecture that is being modelled. For example, the following code extract shows how a **`Person`** object can be instantiated, assuming the USE environment has been preconfigured with the relevant specification. With each command the system moves from one state to another.

```
!create Person 0 : Person
!set Person 0.name := 'bob'
!set Person_0.age := 25
```

The following diagram (taken from [40]) illustrates the concepts described above:

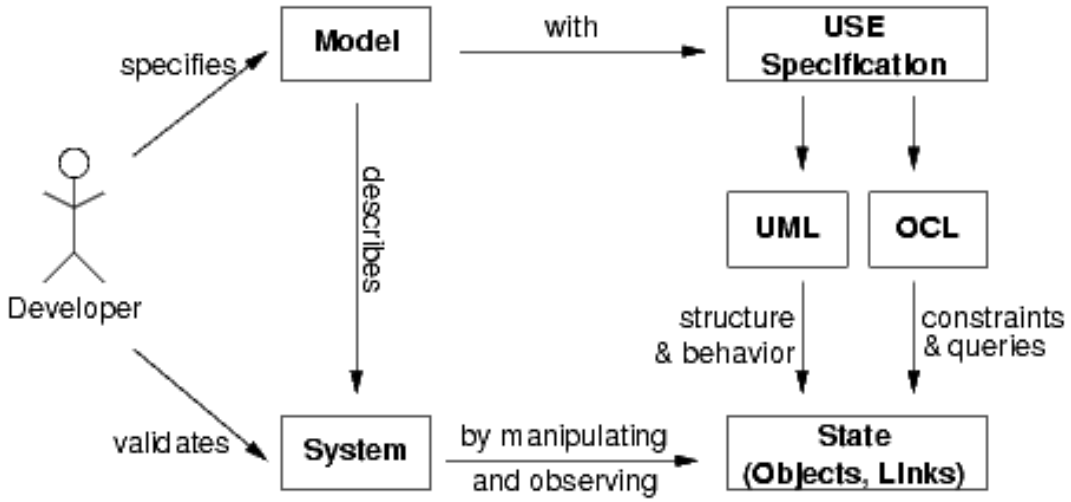


Figure 8-4, illustrating the 'USE' process

To take advantage of the USE environment and OCL for the purposes of the BFG constraint problem, the same principles can be applied. The following section gives a detailed description of the implementation.

Firstly, a specification of the objects involved in the BFG metadata was created. As no official UML model for the BFG is available this was done by considering the major objects modelled by the metadata. These include 'Model', 'Connection', 'Field' and 'DeploymentUnit' classes amongst others. For each class, attributes were assigned representing certain aspects of the object such as 'name', 'language' and 'timestep' in the context of a Model object and 'direction', 'units' and 'size' in the context of a Field object.

Secondly, relationships or associations between the classes were specified. For example a Connection class has a 'connects' relationship between two fields and a field has a 'fieldOf' relationship to a parent connection. Likewise, a model is 'deployedOn' a DeploymentUnit.

Thirdly, OCL constraints were expressed about the classes so that, for instance, it was ensured that a DeploymentUnit had at least one model deployed in it:

```
context DeploymentUnit
inv hasAtLeastOneModelInDeploymentUnit : self.modelDeployedOn->size() >= 1
```

or that the field IDs for a Model object were unique:

```
context Model
inv fieldsOfInputTypeHaveUniqueIDs :
self.owner->select(f:Field|f.direction='in')->isUnique(id)
```

The system specification was saved in a plain text file 'BFG.use' and loaded into the USE environment using a command line option. The full BFG USE specification can be found in appendix A.

Subsequent to defining and loading the system specification, and before the constraints check can be applied, the classes must be instantiated as objects. To achieve this, a simple USE 'command' file can be specified containing state manipulation commands. The command file is constructed from the XML metadata using a Java program. The Java program uses data binding to unmarshall the XML to Java objects. A **`map()`** method goes through the Java objects outputting the corresponding state manipulation statements. For instance, for each Java 'Field' object encountered, a state manipulation command is outputted to instantiate a USE 'Field' object. The state manipulation commands are outputted to a 'temp.cmd' file.

Once the USE command file is written to disk it can be executed against the USE environment using a console command. A further command can be issued to perform constraint checking which evaluates the system state and outputs a report detailing any objects that have violated constraints placed on them. The validation report is bespoke to the USE environment and as such, the format of the output is not configurable.

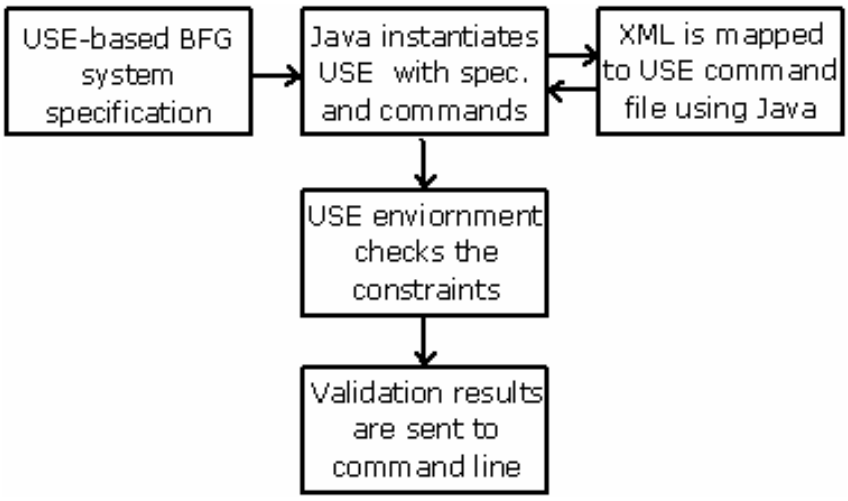


Figure 8-5, conceptual view of system architecture

Figure 8-5 above shows how the architecture is structured. The constraints are located in the USE-based BFG system specification. A Java front-end handles the adhoc mapping of the XML to a USE command file and then instantiates USE with the command line parameters pointing to the specification and command files. The Java program is executed with the file path location to the master XML coupling document and USE-based BFG system specification. Finally, the validation results are collected from USE by the Java program, reformatted and then outputted to the command line.

It is possible to see that the constraint expressions are now separated from the rest of the system as they are included in the USE-based BFG system specification. This allows them to be edited without the need to recompile the Java code. The OCL notation was a successful way to express all the BFG metadata constraints in a concise manner. The following section documents the results that were observed from a selection of sample coupled model metadata.

8.6. Testing

To test the BFG USE-based constraint mechanism, some sample coupled model metadata was used. The first three coupled models are available on the BFG webpage [3] and are called 'SAMSOM', 'SAMSOM2' & 'SAMSOM3'. A fourth coupled model called 'hybridMD' was provided by the University of Manchester. A fifth coupled model called 'e3mg-esm_magicc-scm' was provided by Tyndall Centre for Climate Change Research at the University of East Anglia. The coupled models are named after the individual models that are involved in the coupling

for example ‘SAMSOM’; Simple Atmospheric Model – Simple Ocean Model and ‘e3mg-esm_magicc-scm’; Emission Scenario Model – Simple Climate Model.

As mentioned in the previous section the Java program accepts the location of the master coupling document and the location to the ‘BFG.use’ system specification as a command line parameter to the execution.

```
java ConstraintManager –f ~/models/SAMSOM/XML/coupled.xml –u ~/BFG/BFG.use
```

Inside the Java program, the configuration and instantiation of the USE environment is performed automatically after which the constraint validation output is captured and relayed back to the command prompt. As the USE validation report cannot be configured explicitly, the output is reformatted by the Java program prior to reaching the command line. The following output was observed after running the program on the SAMSOM example coupled model metadata. Some ‘status’ information used for testing purposes is printed at the beginning and end of the output.

```
started new
loaded model : X:\Applications\Java
Stuff\eclipse\workspace\BFGUI\example.models\samsom\samsom\xml\..\sam\xml
loaded model : X:\Applications\Java
Stuff\eclipse\workspace\BFGUI\example.models\samsom\samsom\xml\..\som\xml
success~Connection~N/A~fieldsHaveEqualProductOfDimensionSize
success~Connection~N/A~fieldsHaveEqualTimesteps
success~Connection~N/A~fieldsHaveOneIn
success~Connection~N/A~fieldsHaveOneOut
success~Connection~N/A~fieldsHaveOppositeDirections
success~Connection~N/A~fieldsHaveSameFieldUnitsUnlessOneIsAny
success~Connection~N/A~fieldsNotOwnedBySameModel
success~Connection~N/A~onlyOwnsTwoFields
success~DeploymentUnit~N/A~hasAtLeastOneModelInDeploymentUnit
success~DeploymentUnit~N/A~modelsInDeploymentAreSameLanguageType
success~Field~N/A~arrayDimensionsAre1toN
success~Field~N/A~arrayDimensionsAreAllSameVartype
success~Field~N/A~connectedToFieldWithSameArrayDimensionSizes
success~Field~N/A~connectedToFieldWithSameNumberOfArrayDimensions
success~Field~N/A~fieldIsOwnedByOneModel
success~Field~N/A~inputFieldIsConnectedOnce
success~Model~N/A~allInputFieldsAreConnectedToSomething
success~Model~N/A~anyModelConnectedToHasSameTimestepRate
success~Model~N/A~dataSizesAreNotDefinedTwice
success~Model~N/A~fieldsOfInputTypeHaveUniqueIDs
success~Model~N/A~fieldsOfOutputTypeHaveUniqueIDs
success~Model~N/A~inOnlyOneDeploymentUnit
success~Model~N/A~isInDeploymentIfThereIsAtLeastOneDeploymentDefined
success~Model~N/A~languageMatchesDeploymentUnitLanguage
success~Model~N/A~locationIsSameAsDeploymentUnitName
success~Model~N/A~namesAreTheSameInIfaceSourceandExec
success~Model~N/A~timestepLessThanCompositionRunduration
success~Model~N/A~vartypesOfFieldsAreDefinedInDatasize
success~Model~N/A~versionsAreTheSameInIfaceSourceandExec
ended
```

The output above shows that the SAMSOM example coupled model metadata adheres to all the constraints specified. Each line of the USE output has been reformatted and structured as tilde separated values (due to complications with commas). The output is represented by the following format:

```
Constraint validation result (success or failure) ~
Class of object ~
Object identifier violating a constraint (N/A if 'successful') ~
Name of constraint being processed
```

Success for any constraint indicates that *all* objects of the class satisfied the constraint. Therefore, there are no objects at fault and 'N/A' is logged accordingly as the object identifier, as can be seen above.

In the event of a constraint failure, the object identifier indicates which object is at fault. A constraint may be violated by one or more of the objects in the class. For each violation a failure report is printed. For example, below is the output from the SAMSOM2 example coupled model metadata..

```
started new
loaded model : X:\Applications\Java
Stuff\eclipse\workspace\BFGUI\example.models\samsom2\samsom2\xml\..\sam\xml
loaded model : X:\Applications\Java
Stuff\eclipse\workspace\BFGUI\example.models\samsom2\samsom2\xml\..\som\xml
failure~Model~sam~locationIsSameAsDeploymentUnitName
failure~Model~som~locationIsSameAsDeploymentUnitName
failure~Model~sam~vartypesOfFieldsAreDefinedInDatasize
failure~Model~som~vartypesOfFieldsAreDefinedInDatasize
success~Connection~N/A~fieldsHaveEqualProductOfDimensionSize
success~Connection~N/A~fieldsHaveEqualTimesteps
…
…
…
```

To prevent repetition a majority of the successful constraints have been removed from the output, but we can see that four failures have been discovered; two failures of two constraints. The constraints are described in the positive context. The first two failures indicate that both individual models ('sam' & 'som') have failed the constraint that the model 'location' (the machine the model binary is located on) is the same as the name of the unit it has been deployed on. The second two failures indicate that both models have field 'vartypes' defined that are not defined in a corresponding 'datasize'.

SAMSOM3 is a coupling of four models; the original sam and som models and additional 'average' and 'rsum' models. The additional models are examples of

transformers. The following is the output from the SAMSOM3 example coupled model metadata:

```
loaded model : X:\Applications\Java
Stuff\eclipse\workspace\BFGUI\example.models\samsom3\samsom3\xml\..\sam\xml
loaded model : X:\Applications\Java
Stuff\eclipse\workspace\BFGUI\example.models\samsom3\samsom3\xml\..\average\xml
loaded model : X:\Applications\Java
Stuff\eclipse\workspace\BFGUI\example.models\samsom3\samsom3\xml\..\rsum\xml
loaded model : X:\Applications\Java
Stuff\eclipse\workspace\BFGUI\example.models\samsom3\samsom3\xml\..\som\xml
failure~Connection~average,1.0,realout,2,out,fieldtype,som,1.0,temperature,1,in
,fieldtype~fieldsHaveEqualTimesteps
failure~Connection~rsum,1.0,realout,2,out,fieldtype,som,1.0,energyflux,2,in,fie
ldtype~fieldsHaveEqualTimesteps
failure~Model~average~anyModelConnectedToHasSameTimestepRate
failure~Model~rsum~anyModelConnectedToHasSameTimestepRate
failure~Model~som~anyModelConnectedToHasSameTimestepRate
failure~Model~average~locationIsSameAsDeploymentUnitName
failure~Model~rsum~locationIsSameAsDeploymentUnitName
failure~Model~sam~locationIsSameAsDeploymentUnitName
failure~Model~som~locationIsSameAsDeploymentUnitName
success~Connection~N/A~fieldsHaveEqualProductOfDimensionSize
success~Connection~N/A~fieldsHaveOneIn
…
…
```

In the output above, nine failures were reported. Three failures are from the constraint 'anyModelConnectedToHasSameTimestepRate' corresponding to the models 'average', 'rsum' and 'som'. This test is a good example of a situation where the rules may need to be relaxed or edited. For instance, the coupled model developer may have included transformer models to solve a timestep incompatibility issue between models. Therefore the discrepancy found in timestep values would have been included for a purpose. The coupled model developer may therefore wish to remove this constraint altogether or adapt it to suit the modelling scenario. A discussion of the flexibility of this solution is detailed further in the next section.

The above output also illustrates another interesting aspect to this approach. As mentioned above, it is possible to see that there are models with timestep incompatibilities but it is unclear which models the incompatibility is *between*. However on further inspection it is possible to observe that two Connection objects have violated the constraint 'fieldsHaveEqualTimesteps'.

```
failure~Connection~average,1.0,realout,2,out,fieldtype,som,1.0,temperature,1,in
,fieldtype~fieldsHaveEqualTimesteps

failure~Connection~rsum,1.0,realout,2,out,fieldtype,som,1.0,energyflux,2,in,fie
ldtype~fieldsHaveEqualTimesteps
```

Unlike models, connections don't have specific names, so the object identifier for each connection is a comma separated string value of several variables representing:

```
[OUTPUT MODEL]
model name, model version, field name, field id, field direction, field type,

concatenated with…

[INPUT MODEL]
model name, model version, field name, field id, field direction, field type
```

The connection object identifier is constructed in this way because there is no unique connection identifier provided in the metadata. It could be argued that it would have been better to use a simple integer however this would not have provided such verbose feedback to the user.

The key point of interest in the above example is how constraints can be perceived from the *context* of different objects. In the first instance the timestep discrepancy is observed on Model objects i.e. 'average', 'rsum' and 'som'. In the second instance it is possible to see the individual Connection objects that are at fault. This property is fundamentally the result of the logical nature of the USE approach; objects are intrinsically related to each other through the expression of associations and therefore a semantically equivalent constraint can be expressed in the context of different objects and associations. The rule 'anyModelConnectedToHasSameTimestepRate' in the context of a Model object is more general than the rule 'fieldsHaveEqualTimesteps' in the context of a Connection object. Connection objects are related to Model objects through a mutual relationship to Field objects. This provides tremendous flexibility and power as it allows for the evaluation of logical propositions about the state of the world being modelled from the context of different objects.

The above section documents the results from testing the USE-based constraint solution on the SAMSOM example model metadata. In the following section an evaluation of the solution is given, highlighting strengths and weaknesses and how successfully it reflects the design goals stated in the introduction to this chapter.

8.7. Evaluation

The USE-based approach to evaluating BFG metadata constraints outlined in the previous section was relatively successful. Firstly all BFG metadata constraints are expressible in the OCL syntax or at least their 'counterparts' after mapping to a UML-based USE system specification. The approach was tested against five example coupled model metadata instances – the three detailed in the previous section and the hybridMD and e3mg-esm_magicc-scm models, which achieved similar results. The only example coupled model metadata that did not contain constraint violations was SAMSOM. However, violations in the others mainly consisted of repeated violations of the same constraint by different object instances. For instance, the hybridMD coupled model contained 19 violations of three constraints (12 from just one rule). The e3mg-esm_magicc-scm coupled model contained 145 violations of 6 constraints (118 from just one rule). This result can be explained by the fact that developers of the metadata are probably aware of the detected violations and ignore it either because the constraint does not apply to their particular modelling scenario and will not affect the outcome, or because they are circumventing some other issue. The reader is reminded of the intentional flexibility of BFG, the 'open source' nature of the approach and hence the modelling capability. The constraints used in this thesis are relatively generic and as such may not apply to all users so the desire to adapt, edit, remove or add constraints in an 'easy' manner has been a design goal.

The USE-based approach lends itself particularly well to the design goal of flexibility. As the system specification containing the constraints is essentially a text file specified as a command line parameter to the execution, modifications can be made using a simple text editor, saved and then re-executed. There is no need for recompilation of any component because the USE environment interprets the contents of the specification file. Furthermore, it is also feasible for a user to have different specification documents for different modelling scenarios with a variety or selection of constraints expressed in each.

The issue of expressing the constraints in a less complex and comprehensive manner is hard to gauge. Merely measuring this factor on 'number of lines of code' is not at all scientific. However, it is possible to observe that due to the logical foundation of the notation and the inclusion of concepts such as simple

set theory and 'helper' functions such as **isUnique()** and **sum()**, the constraints are kept concise.

The syntax of the notation and the use of associations between objects may pose the largest hurdle but a further advantage to the USE-based approach is the environment provided by the USE tool. In the solution proposed USE was instantiated inside the Java program but it is also possible to run USE as a standalone application with a command line or graphical user interface. The advantage this gives is the ability to manipulate the system state directly and express logical statements for evaluation. For instance, the following command returns a list of all Connection objects where fields are **not** 'out' to 'in' or vice versa.

```
-- all connections where fields are not 'out' to 'in' or vice versa
?Connection.allInstances
->select(c:Connection|c.fieldOf->collect(direction)<>Bag{'in','out'})
```

The command line environment is extremely useful for constructing constraints, as statements can be created through a trial and error approach. The statement starts off simple but can be improved piece by piece until it eventually reflects the desired logical statement. It is through this method that the constraints expressed as part of this thesis were formed and is extremely useful for validating the syntax.

Chapter 7 documented how the metadata constraints can be classified into 'inter' and 'intra' document constraints. It also showed that this can increase complexity considerably when expressing constraints at the XML level. In the USE-based solution this issue was circumvented by mapping the metadata to the USE environment by using a Java program. The Java program handled the loading of XML documents and extracted the values contained within. As coupled model metadata is composed of several XML document instances the Java program was in actual fact 'collecting' the values from all the documents and converting the XML into a USE specification. This raised a new concern: the flexibility of the proposed approach relies on the Java program – a fault in this component would then require recompilation. A simple solution to this issue would be to perform the mapping using XSLT, a technology built purposefully for transforming XML into other formats. Unfortunately this area of further research falls outside the timescale for this thesis, but it would be a valuable avenue to pursue.

Another disadvantage to the adopted approach is that the validation report does not strictly report the location of the constraint violation in terms of the XML metadata. As the Java program performs the mapping between XML and USE in a 'black box' scenario, the relationship between objects in the USE environment and the structures in the XML metadata is unclear. For instance, the validation report may indicate that a **`Connection`** object has a discrepancy between timestep rates but cannot inform the user where exactly in the XML metadata the problem arises. At present there is no support for this issue and the task of interpreting the validation report falls to the user. This problem, however, is no different to that encountered in the XSLT approach.

8.8. Integration with Graphical User Interface

In this thesis the work presented has mainly focussed on tools or solutions that present textual reports of constraints violated in BFG metadata. To some extent this can be considered a reasonable solution but in the following section it is shown how this work is related to the work of Rowe [10] and how better results may be achieved through reflecting metadata constraint violations in a graphical user interface.

As discussed in Chapter 2, BFG XML metadata is currently produced manually in hand-crafted text documents using a simple text editor. Rowe [10] proposes a new approach to BFG metadata creation with the use of a Java based graphical user interface (GUI). He argues that a GUI would provide faster development time and a less error-prone approach to BFG XML metadata creation. His approach is termed 'BFGUI' for the purposes of this thesis. Research into the BFGUI was performed separately and in parallel to this thesis and as such could not be considered thoroughly until the latter stages of this work.

BFGUI attempts to capture the Define-Compose-Deploy methodology proposed by Ford et al in [4] by allowing the user to produce XML metadata for individual models (define), create a coupled model through the composition of individual models (compose) and specify the deployment configuration for the coupled model (deploy). The main function of BFGUI is to provide a faster and less error prone approach to BFG metadata creation.

In the BFGUI the definition phase is performed using form based input. The forms provide the functionality to produce the 'source', 'executable' and 'interface' XML documents for an individual model. The form based input mechanism aids metadata creation by reflecting the structure of the XML Schema documents. This means that the user need only be concerned with inputting values to the fields of the form and less about writing XML from scratch.

The composition and deployment phases use a slightly different approach to the form-based definition stage. Instead, a 'canvas' is used in which models are represented as circular icons and connections between models are visualised as arcs between models. A user can import individual models to the canvas (by loading XML metadata) and then graphically specify or remove connections between models using the toolbar buttons. The composition can be saved at any point and XML metadata representing the coupled model is either written to disk or to a repository developed by Tellier [41]. Similarly, the canvas can be switched to the 'Deployment' phase and models in the composition can be assigned to or removed from deployment units through toolbar buttons. Metadata representing the deployment configuration can then be saved with the composition metadata.

The graphical approach to BFG metadata creation lends itself particularly well to the USE-based constraint validation mechanism developed as part of this work. For instance, no longer is the user strictly concerned with XML and Schemas – the point of the BFGUI is to abstract this low-level detail through the use of forms, icons and toolbars. In a similar fashion, the USE-based constraint mechanism does not indicate exactly where in the XML the fault was found but instead deals with higher-level objects. Furthermore, other than some basic functionality, the work presented by Rowe in [10] lacks the ability to check the metadata for correctness with respect to the constraints discussed as part of this work. Integrating the systems would potentially provide this functionality.

To investigate the possibility of integrating the constraints checker proposed in this thesis and the GUI for metadata creation as proposed by Rowe [10], a small amount of collaborative research was conducted. This work encompassed a facility through which to interface the two solutions. The following section details how this was achieved. A clear separation of research is present and

collaboration was only necessary to agree on a mechanism through which the two systems could communicate.

As the USE-based constraint management mechanism has already been written with what is essentially a Java based front end, the integration into a Java based GUI, from a technological perspective at least, was simple.

The Java method that maps XML to a USE system specification was adapted to accept a **`CoupledModel`** object from the graphical user interface. A **`CoupledModel`** object, as the name suggests, essentially encapsulates the entire Coupled Model. It includes **`getter()`** and **`setter()`** methods to retrieve data regarding the coupling such as individual model information, connections between models and deployment data. Using the same principle, it was possible to map the **`CoupledModel`** object to a USE system specification, instantiate the USE environment, evaluate the constraints and return the results back to the GUI. The entire process was captured in a **`ConstraintManager`** object that the BFGUI can instantiate (passing the **`CoupledModel`** object to the **`getConstraintViolations()`** method) and finally, retrieve the validation report. The class diagram in figure 8-6 below shows the new **`ConstraintManager`** class with overridden methods:

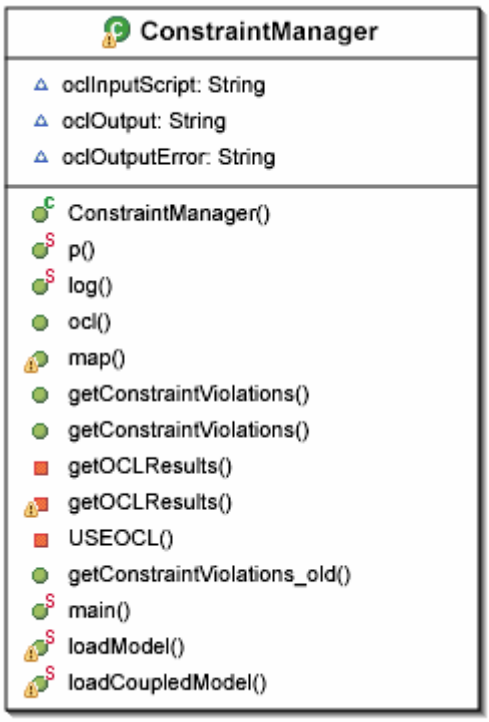


Figure 8-6, revised ConstraintManager class diagram

At each stage in the DCD process (in the BFGUI) a 'check constraints' toolbar button is available. When a user wishes to check the correctness of the coupled model being developed, the button activates the process described above.

The results from the **`ConstraintManager`** validation report are represented in the graphical environment in two ways. Firstly, the validation report successes and failures are entered into a 'Constraint Log' window that can be made visible using a menu option. Rowe has split this view using a tabbed pane so that successes and failures can be viewed separately, making it easier to read. Secondly, if certain objects are determined to have failed a constraint, a corresponding icon on the Compose and Deployment canvases changes colour to signify there is a fault. For instance, if a connection between two models is determined to have incompatible timestep rates, the arc representing it turns red. Similarly, if a model violates some deployment constraint the circular canvas icon representing it also turns red.

The following screenshots illustrate the process described above. In figure 8-7 the example coupled model 'hybridMD' has been imported to the composition window.

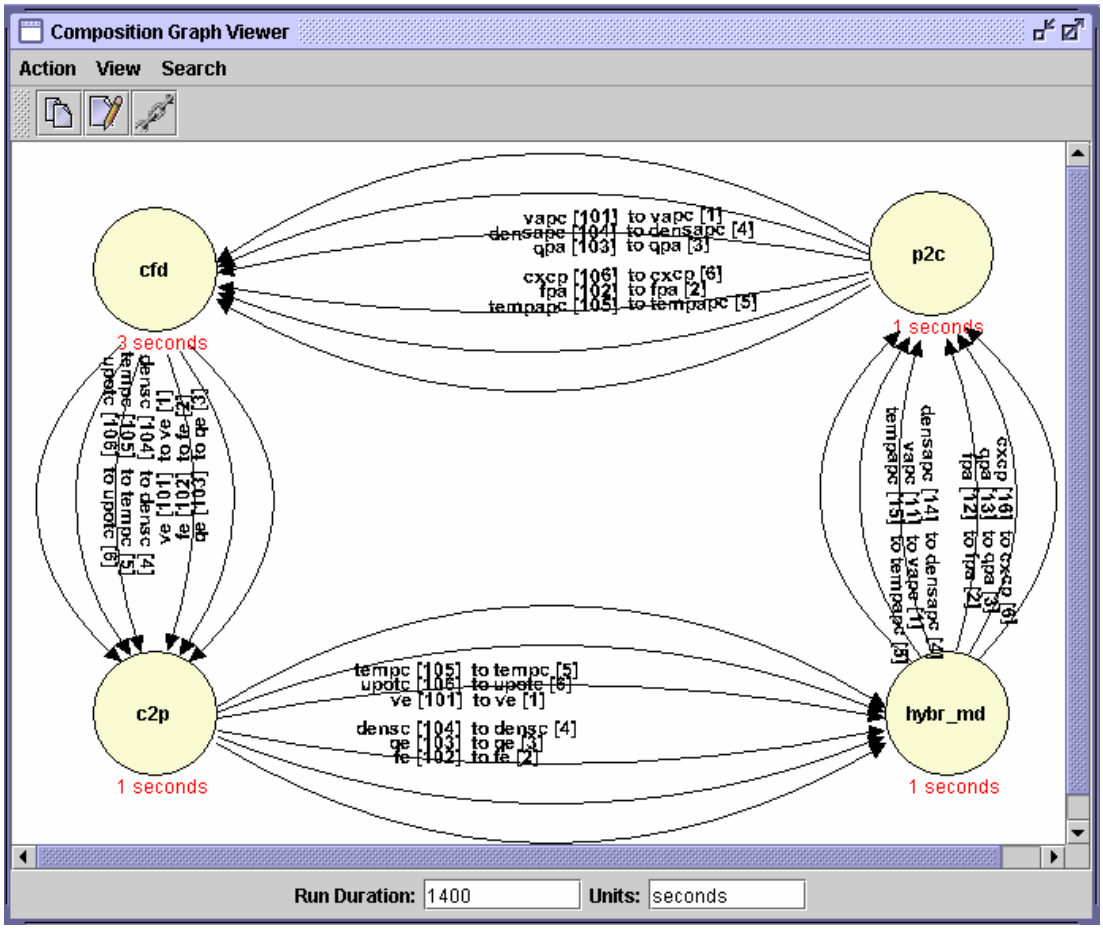


Figure 8-7, BFGUI compose canvas after importing 'HybridMD'

The screenshot below (figure 8-8) shows the composition canvas after the 'Check Constraints' toolbar button (far right) has been pressed. 12 of the 24 connections and 3 of the 4 models have been coloured red indicating a constraint violation has been discovered with the component. By viewing the 'Constraints Log', as shown in figure 8-9, it is possible to observe how the validation report from the USE-based constraint verification mechanism has been de-tokenised and inserted into a tabular format. The constraints log shows that the reason for constraint failures is a discrepancy between timestep rates of connected models. From the screenshot above (figure 8-7), model 'cfd' has a timestep rate of 3 seconds but is connected to models 'p2c' and 'c2p' that have a timestep rate of 1 second.

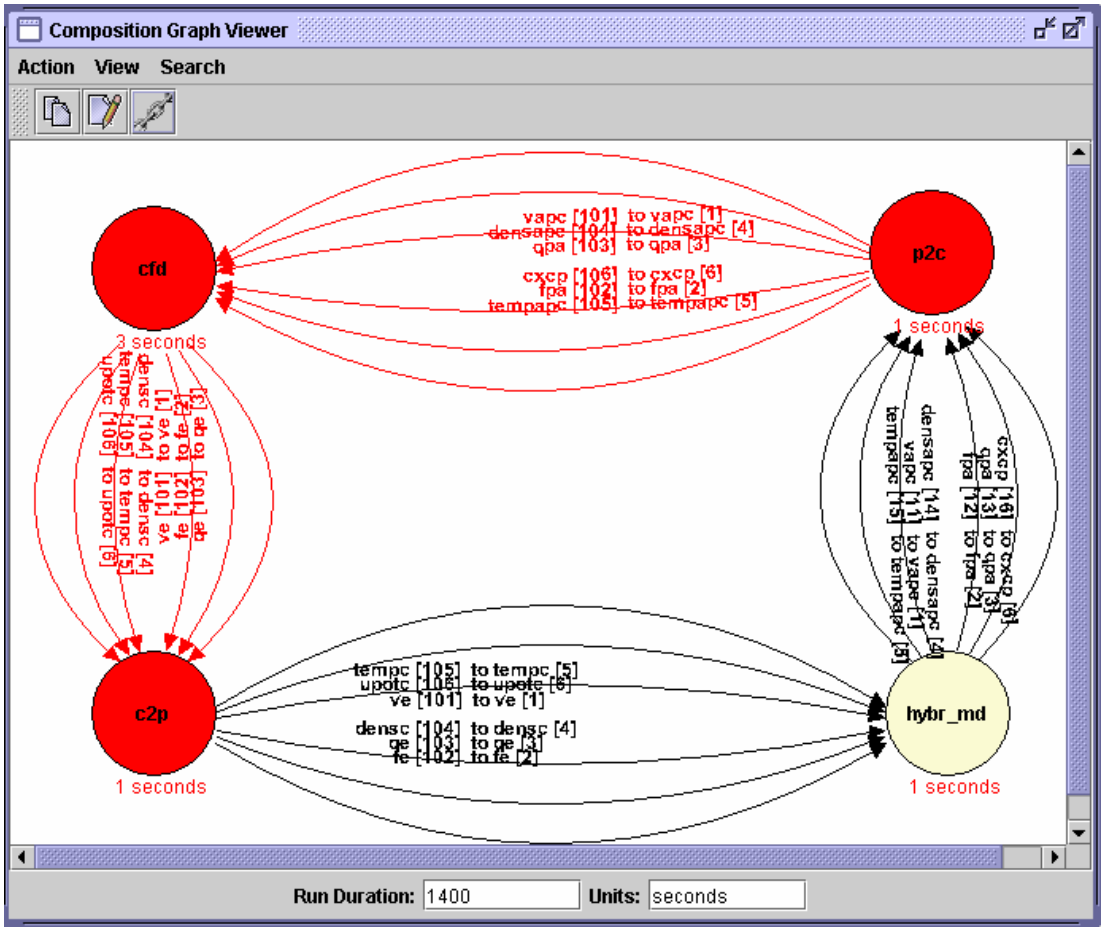
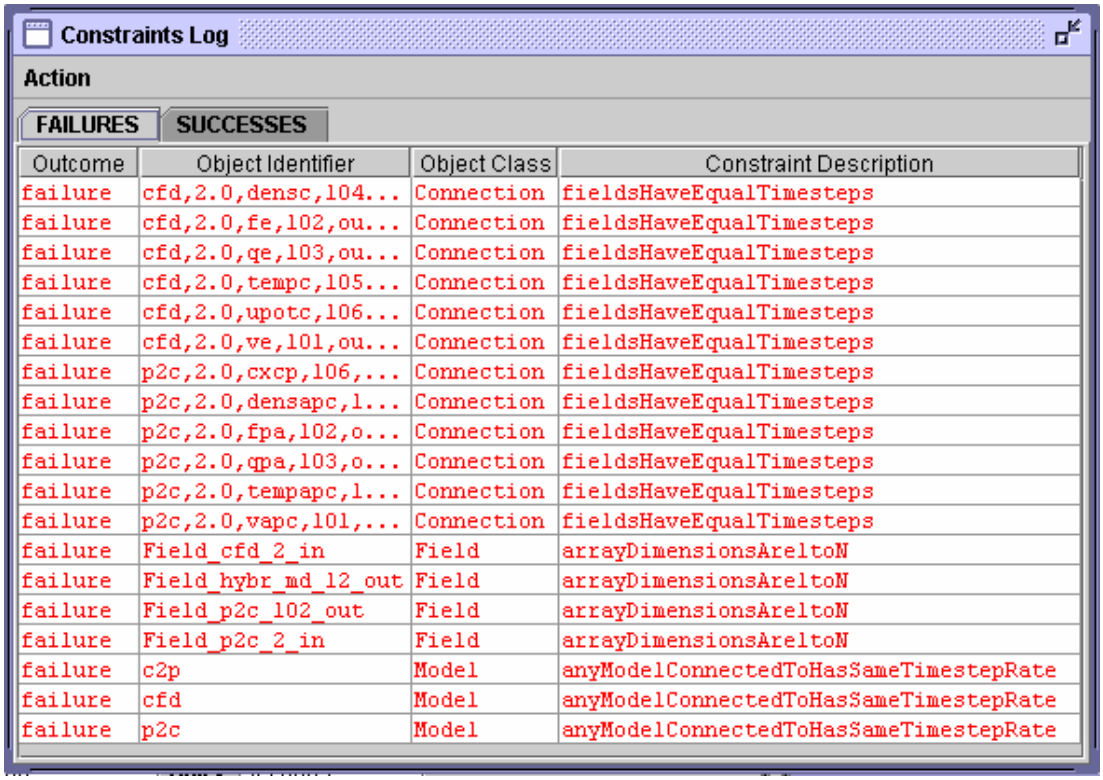


Figure 8-8, BFGUI compose canvas after clicking 'check constraints' button



Outcome	Object Identifier	Object Class	Constraint Description
failure	cfd,2.0,densc,104...	Connection	fieldsHaveEqualTimesteps
failure	cfd,2.0,fe,102,ou...	Connection	fieldsHaveEqualTimesteps
failure	cfd,2.0,qe,103,ou...	Connection	fieldsHaveEqualTimesteps
failure	cfd,2.0,tempc,105...	Connection	fieldsHaveEqualTimesteps
failure	cfd,2.0,upotc,106...	Connection	fieldsHaveEqualTimesteps
failure	cfd,2.0,ve,101,ou...	Connection	fieldsHaveEqualTimesteps
failure	p2c,2.0,cxcp,106,...	Connection	fieldsHaveEqualTimesteps
failure	p2c,2.0,densapc,1...	Connection	fieldsHaveEqualTimesteps
failure	p2c,2.0,fpa,102,o...	Connection	fieldsHaveEqualTimesteps
failure	p2c,2.0,qpa,103,o...	Connection	fieldsHaveEqualTimesteps
failure	p2c,2.0,tempapc,1...	Connection	fieldsHaveEqualTimesteps
failure	p2c,2.0,vapc,101,...	Connection	fieldsHaveEqualTimesteps
failure	Field_cfd_2_in	Field	arrayDimensionsAreltoN
failure	Field_hybr_md_12_out	Field	arrayDimensionsAreltoN
failure	Field_p2c_102_out	Field	arrayDimensionsAreltoN
failure	Field_p2c_2_in	Field	arrayDimensionsAreltoN
failure	c2p	Model	anyModelConnectedToHasSameTimestepRate
failure	cfd	Model	anyModelConnectedToHasSameTimestepRate
failure	p2c	Model	anyModelConnectedToHasSameTimestepRate

Figure 8-9, BFGUI Constraints Log window after clicking 'check constraints' button

8.9. Evaluation of visual constraints solution

Section 8.8 described how the research from this thesis was integrated into a GUI for metadata creation. In the following section a short evaluation of the product of this integration is given.

There are three important issues that need addressing. The first is an 'object identification' issue. In the USE-based constraint checker, XML metadata is mapped to a number of objects. Each object is given a unique identifier based primarily on some variable about that object. For instance, **`Model`** objects are identified by the model name and Connection objects as a combination of variables as described in section 8.6. Currently, BFGUI uses the object identifier in a simple lookup procedure to determine which graphical component to change colour. However, if the format of object identifiers in the constraint mechanism was to change this 'lookup' would fail. A more robust system for object identification is needed.

A second major issue is that at the USE level there are more 'objects' than at the GUI level. For instance, at the USE level a 'Field' object exists to represent a

model field. In the graphical environment no such object has been visually represented therefore a constraint violation of a Field object has no bearing on the components displayed in the canvas. This has occurred because at the USE level certain constraints were inexpressible without creating 'extra' objects and relationships. In the graphical environment these objects have not been represented because they are not needed for the modelling process.

Thirdly, at present there is no provision for checking the constraints individually. Currently the user has to manually press the 'check constraints' button which then checks the entire coupled model and returns everything that is at fault. In an ideal scenario, rather than informing the user that there is something wrong, the faults should be prevented from occurring in the first place. A system in which there is a tighter relationship between action, constraint and result is desirable. This issue has occurred primarily because of the independent nature of the research and the way in which the solutions were built as 'stand-alone' applications that don't rely on each other to operate.

Despite the disadvantages, the visual environment for BFG metadata creation provides a good mechanism through which to report constraint violations. The colour coding of invalid objects visibly indicates that something is wrong with the coupled model. The log window provides an easier way to view the success/failure results from the validation report that would otherwise have been printed out to the command line.

This chapter presented three implementations to checking BFG metadata for errors, one of which was developed fully and integrated into a graphical environment. In the graphical environment constraints violations are represented visually by changing the colour of interactive components. Chapter 9 summarises the findings presented in this thesis and suggests areas for further investigation.

Chapter 9: Conclusion

Throughout this thesis, an attempt has been made to outline the FCA and the BFG approach to coupled modelling while highlighting errors that may occur by defining the coupled models using XML. Solutions were presented that inform the user of errors detected in the BFG metadata. In the following section a review of the work completed in this thesis is given. In section 9.2, areas for further investigation are noted.

9.1. Summary

In Chapter 2, the advantages of using coupled models and a coupled model creation framework were presented. The use of coupled models rather than large individual models promotes flexibility and reusability. It also leads the way to the creation of more complex modelling scenarios and for inter-discipline modelling.

A methodology called the Flexible Coupling Approach developed at the University of Manchester attempts to address computational issues with model coupling such as achieving communication between models and deployment of models in heterogeneous environments. The FCA methodology employs a three phase process termed Define-Compose-Deploy (DCD) which allows users to concentrate on one particular phase at a time. It also allows for certain aspects of the coupled model to be changed without the need to consider other phases. For example, a coupled model can be redeployed without the need to consider or change the composition.

As described in Chapter 4, the Bespoke Framework Generator (BFG) is a prototype implementation of the FCA and reflects the DCD methodology by using individual XML documents to describe each phase of the process. These XML documents are collectively termed BFG metadata.

At the definition phase four metadata documents are needed to describe the interface, source, executable and runtime details of an individual model. At the composition phase a ‘compose’ metadata document specifies the connections between individual models in the coupled model. A ‘deployment’ document allows the user to configure deployment options. Finally, a master ‘coupling’

document ties together the individual documents by providing elements through which the paths to the other metadata documents may be given.

Currently, the only way to produce BFG metadata is by using a simple text editor or XML editor. As XML is not strictly a human-oriented notation it is common for errors to occur in the BFG metadata. Chapter 5 shows how errors may occur in two varieties; invalid XML (errors that prevent BFG execution) and badly-formed BFG metadata (errors that propagate). The latter is more serious as errors cannot be found with well-formed document checks or Schemas.

A use-case example notes that the Tyndall Centre for Climate Change Research at the University of East Anglia has integrated BFG into an environment called SoftIAM. Mistakes in the metadata can lead to time consuming and costly malfunctions to the SoftIAM system or erroneous results emerging from the coupled model. This example highlighted the importance of a system to ensure the metadata was free from error.

To prevent errors that propagate, certain rules, termed *constraints*, must be followed. In Chapter 6 a detailed review of constraint management for XML documents was presented. It was shown how XML Schemas are not expressive enough to cater for all constraints. For instance, XML Schemas cannot validate the simple constraint 'if we see value x in element y then we should see value x in element z'. Three types of solution to this problem were described and the advantages and disadvantages of each solution type were noted. Chapter 7 gave specific example constraints from the BFG metadata that cannot be expressed by XML Schemas.

9.2. Future work

Chapter 8 presented three implementations to express constraints on BFG metadata. The first implementation uses a state of the art technology called Schematron. Schematron is a supplementary language to XML Schemas that bridges the gap between Schemas and the constraints that Schemas alone are incapable of expressing. The general principle behind Schematron is an abstraction or mapping of XSLT expressions into core Schematron elements. It also provides a more structured way to express the constraint combined with the feedback to the user should the constraint fail. Two Schematron meta-

stylesheets were developed that expressed a selection of constraints on the BFG metadata.

The results from the Schematron implementation were promising but it is primarily a system for validating constraints within the same XML document, as are other state of the art solutions. The Schematron meta-stylesheet, which can be embedded in a Schema, usually constrains the structure of XML documents specific to the Schema. As a good proportion of BFG metadata constraints involve more than one XML document, this impedes the conciseness and elegance that Schematron provides. An area for further investigation into this type of solution would be to use XSLT to first translate the multiple XML metadata documents into one large 'intermediate' XML file. At the same time it could also perform mappings and aggregations of values. By declaring constraints on the intermediate document, this could potentially simplify the Schematron expressions that are needed.

A second implementation was attempted using a pure Java approach. The metadata documents were unmarshalled so elements in the XML are mapped to objects in the Java environment. Program code was used to express constraints upon the Java objects and any violations were written to a 'log' and reported back to the user. The familiarity and expressiveness of Java allows for all of the metadata constraints to be created with relative ease. In addition the inter/intra document issue was removed as the metadata was mapped to Java objects. However, this implementation suffered from several drawbacks. The main drawback was the fact the constraints are not easily editable by a coupled model metadata developer as recompilation is required. Furthermore, the Java code does not lead to a concise, easily interpreted expression of the constraints. A big discovery from this implementation attempt was that expressing BFG metadata constraints on the XML was perhaps too 'low-level'.

The third implementation used a formal system modelling approach. A system modelling environment called USE leverages the power of OCL as a language through which it can evaluate constraints on the objects in the USE environment. The metadata documents were 'mapped' to objects in the USE environment using a Java class.

A big advantage to this third implementation was its ability to represent the system using a simple text document. No compilation was required as USE interpreted the classes and constraints defined in the textual system specification. This also provided extensibility as the user could potentially define alternative system specifications containing different constraints for different coupled model development scenarios.

The USE-based approach to constraint checking was the most successful of the three implementations. The conciseness of the OCL notation allowed for constraints to be expressed relatively easily and in less code than the partial XSLT solution that currently exists. The USE command line console is a big assistance in the creation of logical expressions concerning the state of the system. The fact that the constraints are expressed in a text document, in concise OCL notation, allows easy editing without the need for complicated recompilation procedures. The implementation was tested on five coupled model metadata samples and four of the five were found to have contained errors.

Despite the positive aspects, three serious drawbacks to the approach emerged. The first was that the Java based mapping mechanism may be a source of error. For example, the OCL statements being output by it may contain syntactic errors. To correct this, the source code would need to be understood, updated and recompiled as the OCL statements are hard-coded into the Java mapping method. However, this could be rectified by performing the mapping using XSLT– a technology built specifically for the translation of XML documents into other formats. This would be a good avenue for further investigation.

The second disadvantage was that this solution does not use true XML-based technologies such as XSLT and XPath. It is tailored specifically for the BFG metadata constraints issue and is not a general solution to constraint checking in XML documents. To check constraints in other XML documents it would be necessary to change the Java mapping function as well as create a new USE system specification. However, a generic XML logical constraint language has already been attempted by Treshansky and received criticism for being overly complicated. Additionally, Schematron, a true XML related technology, was not suitable to meet all the design goals. A simplification of Treshansky's logical language could be an area for future work.

The third disadvantage is that the validation report does not reflect erroneous aspects of the XML. The XML is mapped to a higher-level logical perspective, thus the violation of constraints are reported on objects that exist in the USE environment. In contrast, Schematron outputs an XPath expression indicating the exact element that is at fault. This flaw became less important when the USE-based approach was integrated into a GUI.

At the end of Chapter 8, the USE-based constraint checking solution was integrated into a graphical user interface for BFG metadata creation. The integration was shown to be successful as the mapping principle was easily adapted to take a **`CoupledModel`** object from the BFGUI. The entire constraint management mechanism was encapsulated in a **`ConstraintManager`** class so can be easily replaced without affecting the architecture of the user interface. The validation report is reflected in the user interface in a log window and through changing the colour of graphical components. The need to report precisely where in the XML documents problems are located was unnecessary as in the graphical environment, users deal with a more conceptual view of coupled model creation.

However, the integration of the GUI and the constraint management solution generates other areas for further investigation. Firstly, the issue of object identification needs resolving. As the **`ConstraintManager`** class and the GUI both have different object identification systems, a lookup method in the GUI translates the ID of objects from the constraints system to visual components in the GUI. This lookup did not prove to be robust as changes to the ID mechanism in the **`ConstraintManager`** meant constraints were not reflected correctly in the GUI. Secondly, the **`ConstraintManager`** deals with more classes of object in the logical level than there are visual components in the GUI canvas. Therefore the **`ConstraintManager`** reports constraint violations on objects that aren't represented in the GUI and may cause confusion.

In conclusion, were it not a design goal to make constraints concise and easy to update, the partial XSLT solution packaged with the BFG, could have been extended to check all constraints on BFG metadata. This would have retained a true XML-based technological solution to the problem. Schematron and other state of the art XML constraint solutions provide an easy mechanism through which to check constraints over single XML documents but many BFG metadata

constraints involve multiple documents. Due to these issues a method to express the constraints at a higher, more logical perspective was implemented. Although this method suffers from its own disadvantages, it provides a way for BFG users to check the correctness of the coupled models they are developing while also providing the flexibility to edit and manipulate the constraints relatively easily.

Chapter 10: Bibliography

[1] Jacinto, M. H. , Librelotto, G. R., Ramalho, J. C. L., Henriques, P. R., (2002), *Constraint Specification Languages: comparing XCSL, Schematron and XML-Schemas*. Available at http://www.idealliance.org/papers/xmle02/dx_xmle02/papers/03-03-02/03-03-02.html

[2] Hu, J., Tao, L., (2004), *An Extensible Constraint Markup Language: Specification, Modelling, and Processing*. Available at http://www.idealliance.org/proceedings/xml04/papers/81/xml-2004-hu.html

[3] Ford, R. W., Riley, G. et al., *The Bespoke Framework Generator*. Available at http://www.cs.manchester.ac.uk/cnc/projects/bfg.php

[4] Ford, R. W., Riley, G. et al., *The Flexible Coupling Approach.* Available at http://www.cs.man.ac.uk/cnc-bin/cnc_fca.pl

[5] Armstrong, C. W., (2002), *Frameworks For Coupled Simulation Modelling.* MSc Thesis, University of Manchester

[6] McLaughlin, B., (2002), *Validation with Java and XML Schema Part 1.* Available at http://www.javaworld.com/javaworld/jw-09-2000/jw-0908-validation.html

[7] Costello, R., *Extending XML Schemas.* Available at http://www.xfront.com/ExtendingSchemas.html

[8] Treshansky, A., (2005), Private Communication

[9] Riley, G. and Ford, R., (2001), *The FLUME project high level design document*. Technical report, Manchester University, December 2001.

[10] Rowe, J., (2005), *The Development and Implementation of a GUI for the BFG Coupling System.* MSc Thesis, University of Manchester

[11] Dodds, L., (2001), *Schematron: Validating XML Using XSLT.* Proceedings of XSLT UK Conference, Keble College, Oxford, England, 2001.

[12] Ramalho, J. C. L., (2001), *Constraining Content: Specification and Processing.* XML Europe 2001, Berlin, Germany,

[13] eXtensible Inter-Nodes Constraint Mark-up Language (XincaML) , IBM Corporation, 2002. Available at
http://www.alphaworks.ibm.com/tech/xincaml.

[14] Nentwich, C. et al., (2000), *Xlinkit: A Consistency Checking and Smart Link Service*. Research Note RN/00/66, Dept. of Computer Science, University College London, 2000.

[15] Dodds, L., (2001), *Schematron: Validating XML Using XSLT*. Dodds homepage, Available at
http://www.ldodds.com/papers/schematron_xsltuk.html#c35e2592b5b1

[16] Ford, R. W., Riley, G. D., Bane, M. K., Armstrong, C. W. and Freeman, T. L., (2005), *GCF: a General Coupling Framework*. Concurrency and Computation: Practice and Experience, to appear 2005.

[17] W3C. Extensible Markup Language (XML) 1.0 (second edition). W3C recommendation, October 2000.

[18] W3C XML Path Language (XPath) 1.0. W3C recommendation, November 1999.

[19] W3C XSL Transformation (XSLT) version 1.0. W3C recommendation, November 1999.

[20] oXygen XML Editor & XSLT Debugger, http://www.oxygenxml.com/

[21] Topologi Schematron Validator, Available at
http://www.topologi.com/products/validator/

[22] Schematron.NET, Available at
http://sourceforge.net/projects/dotnetopensrc/

[23] Jing, *A RELAX NG validator in Java.* Available at
http://www.thaiopensource.com/relaxng/jing.html

[24] Jelliffe, R., *Resource Directory (RDDL) for Schematron 1.5*, Available at
http://xml.ascc.net/schematron/

[25] The Tyndall Centre for Climate Change Research, University of East Anglia, http://tyndall.e-collaboration.co.uk/index.shtml

[26] SoftIAM, *A community integrated assessment modelling system.* University of East Anglia, Available at http://beo1.uea.ac.uk:8080/softiam/

[27] SGML, *A brief SGML tutorial*. Available at
http://www.w3.org/TR/WD-html40-970917/intro/sgmltut.html

[28] Warren, R. et al, *Induced Technological Change in the Stabilisation of Carbon Dioxide Concentrations in the Atmosphere : Scenarios using a large-scale econometric model.* Available at
http://www.stabilisation2005.com/50_Dr_Rachel_Warren.pdf

[29] The Castor Project, http://www.castor.org/

[30] Java Architecture for XML Binding (JAXB), http://java.sun.com/xml/jaxb/

[31] Karaorman, M., Hölzle, U., Bruno, J., (1999), *jContractor: a Reflective Java Library to Support Design by Contract*. In Proceedings of Meta-Level Architectures and Reflection, volume 1616 of lncs, July 1999.

[32] Kramer, R., (1998), *iContract - The Java Design by Contract Tool*. In Proceedings of the Technology of Object-Oriented Languages and Systems (August 03 - 07, 1998). TOOLS. IEEE Computer Society, Washington, DC, 295.

[33] Dresden OCL Toolkit, Software Engineering Group at Technische Universität Dresden, Available at http://dresden-ocl.sourceforge.net/index.html

[34] Kbeans, Kbeans*: Semantic Transparency With JavaBeans*. Available at http://www.faw.uni-ulm.de/kbeans/

[35] Hölzle, U., Duncan, A., (1998), *Adding Contracts to Java with Handshake.* Technical Report TRCS98-32, University of California, Santa Barbara, USA.

[36] Xerces2 Java Parser, Available at http://xml.apache.org/xerces2-j/

[37] Richters, M., Gogolla, M., (1998), *Formalizing the UML Object Constraint Language OCL*. Proceedings 17th Int. Conf. Conceptual Modeling (ER '98) Springer LNCS 1507, 449-464, 1998.

[38] Unified Modelling Language 2.0 Superstructure Specification, Object Management Group, (2005), Available at http://www.uml.org/#UML2.0

[39] The Object Constraint Language Sample Syntax. Available at http://www.csci.csusb.edu/dick/samples/ocl.html

[40] Richters, M., *USE: A UML based specification environment*. Available at http://www.db.informatik.uni-bremen.de/projects/USE/

[41] Tellier, M., (2005), *A Repository for Profiling, Publishing and Subscribing to BFG Metadata.* MSc Thesis, University of Manchester

[42] The Jakarta Digester, Available at http://jakarta.apache.org/commons/digester/

[43] The Globus Alliance, Available at http://www.globus.org/

[44] XMLSpy 2005, Available at http://www.altova.com/products_ide.html

[45] W3C. XML specification DTD (DTD) 2.1 W3C recommendation, 1998.

[46] W3C. XML Schema Working Group, Available at http://www.w3.org/XML/Schema

[47] W3C. XML Schema Part 1: Structures Second Edition, W3C recommendation, October 2004.

[48] Ford, R.W., Riley, G.D., Towards the Flexible Composition and Deployment of Coupled Models. In proc. Tenth ECMWF Workshop on the Use of High Performance Computing in Meteorology; Realizing TeraComputing. ECMWF, Reading, England, 4-8 November 2002. World Scientific, pp. 189--195, 2003.

[49] Xalan-Java Version 2.7.0, Available at http://xml.apache.org/xalan-j/

Chapter 11: Appendix A

USE-based BFG System Specification Document, including all BFG metadata constraints.

```
model BFG

-- classes

class DataSize
attributes
   ds_type : String
   ds_value : Integer
   ds_units : String
constraints
end

class Model
attributes
  name : String
  version : String
  timestep : Integer
  language : String
  location : String
  iname : String
  iversion : String
  sname : String
  sversion : String
  ename : String
  eversion : String
constraints
   inv fieldsOfInputTypeHaveUniqueIDs : self.owner->select(f:Field|f.direction='in')->isUnique(id)
   --inv fieldsOfInputTypeHaveUniqueIDs : self.owner-->select(f:Field|f.direction='in')->forAll(f1,f2:Field|f1<>f2 implies f1.id <> f2.id)
   inv fieldsOfOutputTypeHaveUniqueIDs : self.owner->select(f:Field|f.direction='out')->isUnique(id)
   inv allInputFieldsAreConnectedToSomething : self.owner->select(f:Field|f.direction='in').connects->asSet()->size = self.owner-
>select(f:Field|f.direction='in')->size
   inv timestepLessThanCompositionRunduration : self.timestep < self.compositionOf.runduration
   inv inOnlyOneDeploymentUnit : self.deploymentUnitOf->size() <= 1
   inv locationIsSameAsDeploymentUnitName : self.deploymentUnitOf->size() > 0 implies self.location = self.deploymentUnitOf.machinename
   inv languageMatchesDeploymentUnitLanguage : self.deploymentUnitOf->size() > 0 implies self.language = self.deploymentUnitOf.language
```

```
   inv isInDeploymentIfThereIsAtLeastOneDeploymentDefined : self.compositionOf.deploymentUnitOf->size() >0 implies self.deploymentUnitOf-
>size() >0
   inv namesAreTheSameInIfaceSourceandExec : self.iname = self.sname and self.sname = self.ename
   inv versionsAreTheSameInIfaceSourceandExec : self.iversion = self.sversion and self.sversion = self.eversion
   inv anyModelConnectedToHasSameTimestepRate : self.owner.connects.fieldOf.ownedBy.timestep->asSet()->size()=1
   inv vartypesOfFieldsAreDefinedInDatasize : self.owner.dataPutIn.vartype->asSet() = self.dataSizeOf.ds type->asSet()
   inv dataSizesAreNotDefinedTwice : self.dataSizeOf.ds type->asSet()->size() = self.dataSizeOf.ds type->size()
end

class Field
attributes
  direction : String
  fieldunits : String
  id : Integer
  productdimensionsize : Integer
constraints
  inv inputFieldIsConnectedOnce : self.direction='in' implies self.connects->size <=1
  --inv inputFieldsIsConnectedToSomething : self.direction='in' implies self.connects->size >=1
  inv fieldIsOwnedByOneModel : ownedBy->size = 1
  inv arrayDimensionsAreAllSameVartype : self.dataPutIn.vartype->asSet()->size()<=1
  inv connectedToFieldWithSameNumberOfArrayDimensions : self.connects.fieldOf->asSet()->collect(dataPutIn->size())->asSet()->size()<=1
  --inv arrayDimensionsAre1toN : self.dataPutIn.index->sum() = (self.dataPutIn->size()/2)*(self.dataPutIn->size()+1)
  inv arrayDimensionsAre1toN : self.dataPutIn.index->asSequence() = Sequence{1..self.dataPutIn->size}
  inv connectedToFieldWithSameArrayDimensionSizes : self.dataPutIn->collect(size) = self.connects.fieldOf->select(f|f<>self).dataPutIn-
>collect(size)
end

class ArrayDimension
attributes
   index : Integer
   size: Integer
   units: String
   vartype: String
end

class Connection
attributes

constraints
  inv fieldsNotOwnedBySameModel: self.fieldOf.ownedBy->isUnique(name)
  inv fieldsHaveSameFieldUnitsUnlessOneIsAny: self.fieldOf.fieldunits->asSet()->select(f|f<>'any')->size()<=1
  inv fieldsHaveOppositeDirections: self.fieldOf.direction->asSet()->size=2
  inv fieldsHaveOneIn: self.fieldOf.direction->count('in')=1
```

```
  inv fieldsHaveOneOut: self.fieldOf.direction->count('out')=1
  inv fieldsHaveEqualProductOfDimensionSize: self.fieldOf.productdimensionsize->asSet()->size()=1
  inv fieldsHaveEqualTimesteps : self.fieldOf.ownedBy.timestep->asSet()->size()=1
end

class Composition
attributes
   runduration : Integer
constraints
end

class DeploymentUnit
attributes
   machinename : String
   fabric : String
   language : String
   comms : String
constraints
   inv hasAtLeastOneModelInDeploymentUnit : self.modelDeployedOn->size() >= 1
   inv modelsInDeploymentAreSameLanguageType : self.modelDeployedOn.language->asSet()->size() <= 1
end

-- associations

association HasA between
  Model[1..*] role ownedBy
  Field[1..*] role owner
end

association isPartOf between
  Connection[0..*] role connects
  Field[2] role fieldOf
end

association isOfType between
   Field[1] role uses
   ArrayDimension[1..*] role dataPutIn
end

association isModelIn between
   Composition[1] role compositionOf
   Model[1..*] role modelOf
end
```

```
association isDeploymentUnitIn between
   Composition[1] role compositionOf
   DeploymentUnit[0..*] role deploymentUnitOf
end

association isDeployedIn between
   DeploymentUnit[0..1] role deploymentUnitOf
   Model[0..*] role modelDeployedOn
end

association isDataSizeOf between
   Model[1] role modelOf
   DataSize[0..*] role dataSizeOf
end

constraints

context Connection
  inv onlyOwnsTwoFields:
   fieldOf->size = 2
```

Chapter 12: Appendix B

Example XSLT transform to check only the consistency of model names and versions (one constraint).

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:bfg="http://www.cs.man.ac.uk/cnc/schema/gcf"
exclude-result-prefixes="bfg">
<!--RF est 20/04/04-24/04/04-->

<xsl:variable name="root" select="/"/>

<xsl:output method="xml" indent="yes"/>

<xsl:template match="/">

<info>
<xsl:text>Checking that compose model names are valid.</xsl:text>
</info>

<!-- apply the template to each connection specified in the compose document -->
<xsl:apply-templates select="document(bfg:coupled/bfg:compose,bfg:coupled)/bfg:compose/bfg:connect"/>

<info><xsl:text>Checking complete.</xsl:text></info>

</xsl:template>

<!-- template to match a connection -->
<xsl:template match="/bfg:compose/bfg:connect">

<xsl:variable name="ProvidesModelName" select="bfg:providesModel/bfg:name"/>
<xsl:variable name="ProvidesModelVersion" select="bfg:providesModel/bfg:version"/>
<xsl:variable name="RequiresModelName" select="bfg:requiresModel/bfg:name"/>
<xsl:variable name="RequiresModelVersion" select="bfg:requiresModel/bfg:version"/>

<xsl:if test="not(boolean(document($root/bfg:coupled/bfg:component/bfg:model/bfg:source,$root)/bfg:source[bfg:name=$ProvidesModelName and
bfg:version=$ProvidesModelVersion]))">
<error>
<xsl:text>provides model name "</xsl:text>
```

```
<xsl:value-of select="$ProvidesModelName"/>
<xsl:text>" version "</xsl:text>
<xsl:value-of select="$ProvidesModelVersion"/>
<xsl:text>" in compose document is not found in any model document.</xsl:text>
</error>
</xsl:if>

<xsl:if
test="not(boolean(document($root/bfg:coupled/bfg:component/bfg:model/bfg:source,$root)/bfg:source[bfg:name=$RequiresModelName and
bfg:version=$RequiresModelVersion]))">
<error>
<xsl:text>requires model name "</xsl:text>
<xsl:value-of select="$RequiresModelName"/>
<xsl:text>" version "</xsl:text>
<xsl:value-of select="$RequiresModelVersion"/>
<xsl:text>" in compose document is not found in any model document.</xsl:text>
</error>
</xsl:if>

</xsl:template>

</xsl:stylesheet>
```