YAM: A Framework for Open-Ended Heterogeneous Modelling and Documentation

Sujit Kumar Chakrabarti Y. N. Srikant

IISc-CSA-TR-2005-11
http://archive.csa.iisc.ernet.in/TR/2005/11/

Computer Science and Automation
Indian Institute of Science, India

August 2005

YAM: A Framework for Open-Ended Heterogeneous Modelling and Documentation

Sujit Kumar Chakrabarti Y. N. Srikant

Abstract

Typically, during system development, a voluminous amount of technical documentation is created. This documentation plays a crucial role in enhancing productivity in the latter stages of a project. Therefore, having all technical documents in correct and consistent state at all points in time is widely accepted as an important problem. However, the unstructuredness and heterogeneity of technical documents are two main issues in providing an effective solution to this problem.

In this paper we propose a novel approach for modelling system descriptions based on YAML – a meta model for the definition of the *logical view* of a technical document. This logical view imposes a structure to the document and explicates traceability relations. The expressive representation of the document logical structure through YAML proves very useful in subsequent consistency maintenance in documents – both manually and automatically.

YAM is a modelling framework, that we have developed, based on this concept. We demonstrate how YAM's plugin based architecture supports open ended heterogeneity in modelling languages and tools. This fact also makes it possible to use the power of existing verification tools to carry out automatic consistency checks in technical documents. We cover some of the top level design aspects of YAM that are a direct implementation of YAM's underlying design philosophy: *Open ended heterogeneous modelling using a logical view of the document.*

1 Introduction

High quality documentation has been recognised for long as an essential aspect of successful software projects[4](essay 6 and 10), [16]. Some prime qualities of a high quality documentation system are: expressivity for breadth of modelling, formalism for precision and automation, and consistency. In this paper we propose a method for creating and working with high-quality technical documentation. At the heart of this method is a simple object-oriented metamodel – which we have developed and named YAML – of a technical document. Along with that, we have developed a documentation framework called YAM, which allows creation of technical-documents as instances of the YAML meta-model. Section 2 surveys earlier work which have addressed the issues of heterogeneity and consistency in software artifacts in various forms. Then we introduce our approach in section 3 highlighting its similarities with existing body of work, and its novelties. We describe YAML, the document logical structure meta language in section 4. Then, we introduce YAM, the plugin based modelling framework, by working on small but complete example documentation project in section 5. Section 6 presents the conceptual design of YAM more precisely, and elaborates on YAM's strategy of handling heterogeneity and consistency. We conclude with a few words on the future directions in section 7.

2 Related Work

The fundamental goal of this work is to contribute towards bringing the practice of formal system description closer to the day to day software engineering practice. We have identified two aspects to this problem: development of powerful modelling languages and tools; and consistency maintenance. Here, we discuss the works which seem to aim at the same goals, and have concentrated on some or the other of its aspects mentioned above.

A work close to the spirit of heterogeneous modelling is Espress[5]. Espress methodology allows development of *compound specification documents*. Different modelling techniques, e.g. Statecharts, Z etc are combined into a Z-based notation called μSZ, which is the basis of *semantic integration* of the formalisms. The methodolody (and the tool) works on the concepts of *data integration* through a uniform data representation. *Control integration* is achieved through definition of *adaptors*.

Plugin based tool integration is supported in Eclipse[7]. Eclipse recognises the need for open ended tool integration in a very general setting. The Eclipse IDE can be used to build new plugins which can then be integrated to the IDE itself, thereby continually enriching the power of the Eclipse environment. Eclipse is already a popular platform in projects using plugin based development approach.

Object Management Group's Unified Modelling Language [1], or UML, is an alternative approach for heterogeneous system modelling. It is a huge suite of diagrammatic notations (views) for modelling object oriented systems. Its rich collection of notations is widely popular in industry and academia. This is both for the reasons of expressivity, and because of good tool support. Object Constraint Language[19] is now an integral part of UML specification. It provides formal notations for writing constraints on UML diagram, thereby infusing a greater degree of formalism into UML diagrams. A huge community of researchers have been working on building automatic constraint checking methods on UML. We refer the interested reader to Precise UML group which aims at making UML a precise modelling language[12]. A related work on *live sequence charts* can be found in [6]. Tools employing various strategies of integration of UML model validation are Neptune[2], Use[11], ArgoUML[18].

Another popular branch of research is based on (software) architecture description languages, or ADLs. Heterogeneity is supported by the generic constructs of the languages. ADLs, as opposed to UML, stress more on formalisation. Hence they have stricter semantics, and are almost invariably supported by tools for correctness and consistency checking. A good survey and comparative study of this body of research is provided in [14].

When softwares are described by *architectural styles*, there could be mismatches known as *architectural mismatches.* Gacek[10] talks about software architectural mismatches and their automatic detection. Consistency checking in multiple viewpoint software architecture is dealt with in[10] and [8]. In this a graphical modelling language and a simple constraint language are provided along with a checking algorithm. An early treatment of the idea of consistency in requirement specifications is done in [17]. A more recent related work giving a method of consistency checks between documents is [16]. It uses xlinkit, a lightweight application service that provides rule based link generation and checks the consistency of distributed web content [15].

This brief survey aimed at giving a glimpse of the research work concerning high quality system description by targeting the following aspects:

- Increasing expressivity of system description language by supporting heterogeneity in modelling languages and tools.
- Consistency maintenance through development and use of more formal description languages, thereby creating scope of automatic consistency maintenance.
- Open ended development of methods of system modelling through tool integration frameworks and data interchange formats between case tools.

3 Our Approach

In this section, we briefly introduce our approach to solve the problem of creation and maintenance of high quality system descriptions. Our approach can be summarised through the following points:

- Maximisation of expressivity through heterogeneous modelling.
- Use of formal languages and tools for allowing automatic consistency maintenance.
- Open ended architecture allowing languages and tools to integrate freely.

Two parameters defining the design policy of a modelling environment are:

1. Degree of formalism and automation
2. Degree of open endedness of tool integration

The first parameter makes the modelling environment take a more aggressively formal approach – sticking to a restricted set of formalisms, making full use of verificaton tools for consistency checking etc. The second suggests a less aggressive approach – more heterogeneity, less strict formalism, open endedness prefered to automation etc.

We consider ours as a middle path. In terms of open endedness, our work is placed somewhere between Espress and Eclipse. YAM targets modelling and documentation as Espress does. However, it doesnot require semantic integration to be done inside the YAM framework. While passing the responsibility of automation to plugin verification tools, YAM takes up the task of maintaining and managing a data base of heterogeneous models. The responsibility of invoking the right plugin at the right time is again YAM's responsibility.

However, YAM's completely novel aspect is that it allows creation of system models and documentation over a logical view of the document. YAM then makes use of this logical view to keep track of consistency in the models.

4 YAML – A Metamodel for The Logical Structure of a Design Document

4.1 Introduction

Now we present YAML, the meta language that provides constructs for defining the logical structure of a document. The aim behind defining YAML as

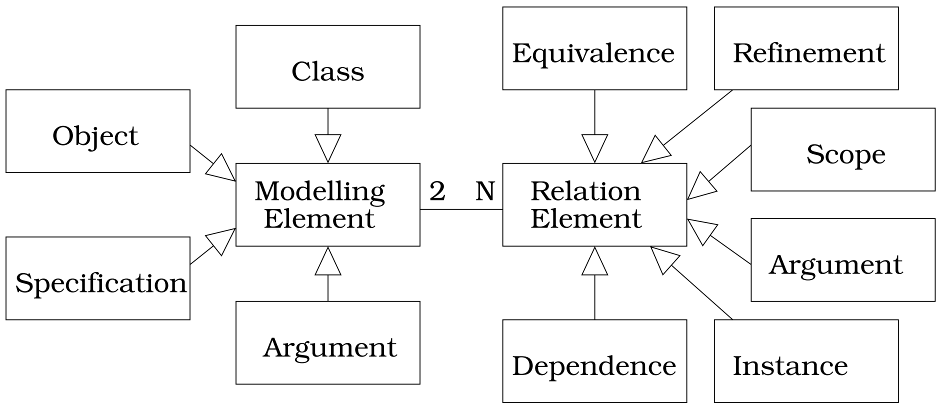


Figure 1: YAML – A Metamodel for The Logical Structure of a Design Document.

it is are the following:

1. identification of the role or type of a particular document fragment based on a finite vocabulary of types.

2. identification of the traceability relation between these fragment, again from a finite set of relation types

The set of **modelling element** types and the set of **relation element** types shared by them constitutes YAML, the language we propose to describe the logical structure of a document.

Figure 1 shows YAM, represented, using UML notations, as the meta model of the document logical structure. A technical document consists of several modelling elements, each corresponding to one or the other *logical fragment*. By a 'logical fragment' we mean a minimal visually contiguous portion of the document that holds a piece of meaningful information. In fact, the complete document is covered by assigning all its contents to some or the other modelling element. As per our current specification of YAML, all modelling elements completely partition the document. A logical fragment of the document belonging to one modelling element cannot belong to any other. Depending on the content of the document fragment, the subtype of the modelling element class that it actually belongs to is defined.

The subtypes of modelling element class, as depicted in Figure 1, are four in number,namely, **class, object, specification** and **argument**. The relationship shared by one document fragment with another can be of one of the six types as shown in Figure 1. These are: **equivalence, refinement, scope, argument, instance** and **dependence**. The multiplicity of the association between modelling element and relation element means that the relation elements are all binary relations connecting two modelling elements; and a modelling element may be associated through any number of relations with other modelling elements. A document can thus be shown diagrammatically as a collection of objects each representing a particular logical fragment of the document, and belonging to one of the four aforementioned types of modelling elements. These objects are connected through links, each belonging to one of the six aforementioned types of relation elements. We call this diagrammatic representation of a document its **logical view**. Please note that relation links between modelling elements, appearing in the logical view of document, are not explicit parts of the **reader's view**, the view of the document corresponding to its ordinary appearance.

We devote the rest of this section to describe YAML in greater detail.

4.2 Modelling Elements

4.2.1 Class

A fragment containing description of a *type* is given the name **class**. The term `class`[1] has been borrowed from the OO terminology, since the sense in which we intend it to be interpreted in our domain is akin to the OO usage. For example, a description of the rules of construction of a regular expressions corresponds to the class of regular expressions. A description of a graph represents the class of all abstract-structures fulfilling this description. Depending on the context, the same description may or may not qualify as a `class`. These differences will eventually get clear with the description of the remaining features of the language.

4.2.2 Object

The term *object* is also borrowed from the object-oriented domain. A document fragment describing an entity in the modelling world is an **object**. Sometimes there may not be a `class` fragment corresponding to the `object` fragment present in the document at all. Although the relation in a document between a `class` and an `object` is the same as in a computer program – i.e. `instance` (described in the next subsection) – there are two important differences. First, unless the document contains the description of an executable model of the system (e.g. source code), there cannot be assumed any

[1]When a keyword of YAML is introduced we use **boldfaced** font. However all subsequent uses of these keywords are in verbatim font. At many places in this paper, terms like 'class' and 'object' are used with interpretations different from as keywords of YAML. All such uses are in normal font.

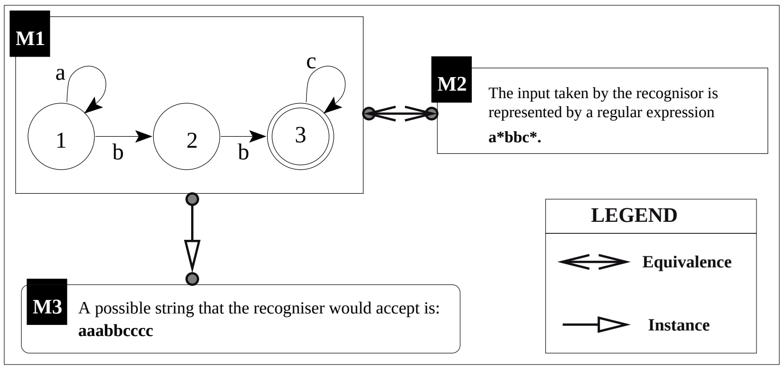

M1
a
c
1
2
3
b
b
M2
The input taken by the recognisor is represented by a regular expression **a*bbc***.
M3
A possible string that the recogniser would accept is: **aaabbcccc**
LEGEND
Equivalence
Instance

Figure 2: A logical view describing some related portions of a typical document.

notion of a runtime. The contents of a document represent the description of a system, not the system itself. Secondly, if an `object` modelling element has been modelled as an instance of some other `class` modelling element, it may include its full description. An implicit assumption that it will reconcile with the specification of that `class` contained in the `class` element doesnot hold here. Hence, it is subject to verification whether indeed the description of an `object` fulfils the specification of its `class`.

Figure 2 shows a partial logical view of a typical document. This portion of the document is describing a language recogniser which accepts a regular expression. Both the finite state automaton (M1), and its equivalent regular expression representation (M2) form a modelling element each. A possible string that the recogniser would accept (M3) forms the third modelling element. While the two elements describing the input language are modelled as two `class`es (shown as ordinary rectangles), the example input string is modelled as an `object` (shown as a rectangle with rounded corners). The two `class`es describe the same language, hence are related to each other through an `equivalence`. The `object`, on the other hand, is an `instance` of the `class`es. The relations thus explicated here indicate the intended relation between these fragments of the document. Whether these relations are actually satisfied decides the consistency of the document: a matter subject to verification – either manual or automatic.

4.2.3 Specification

A fragment describing *verifiable* properties of any fragment – either a `class` or an `object` – is a **specification**. By 'verifiable' we do not mean that a verification algorithm or tool should exist to mechanically check the satisfac-

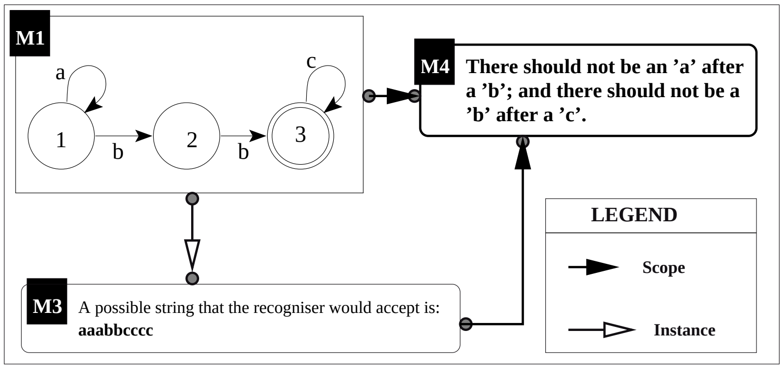

M1
a
c
1
b
2
b
3
M4
There should not be an 'a' after a 'b'; and there should not be a 'b' after a 'c'.
LEGEND
Scope
Instance
M3
A possible string that the recogniser would accept is: aaabbcccc

Figure 3: A logical view showing a specification for a class and an object.

tion of that property. It means that the property stated in a `specification` should be an assertion which is present in the referred fragment, either explicitly or in an implied manner.

Figure 3 shows a partial logical view of the same document, with a `specification` (M4, shown in bold rectangle with rounded corners) for M1 and M3 of figure 2. As it can be seen, the content of the `specification` is a fact, formula or assertion about the related `class` or `object`. Following observations are in place at this moment:

- The consistency of the document is dependent on whether the fact contained in the `specification` actually holds for the related elements.
- An `object` and a `specification` do not differ from each other by the syntactic or semantic structure of their contents. It is the role of that element in the document which decides if it is a `class`, an `object` or a `specification`. An `object` is an added information about some 'thing' in the modelling world. Its removal will cause reduction of the total information about that 'thing'. More formally, each `class` or `object` restricts the set of things in the modelling world that satisfy the properties mentioned in them. On the other hand, a `specification` is an assertion of a property of that 'thing' which has been described in the modelling elements (`class`es or `object`s) describing it. This property should be 'verifiable', i.e. explicitly or implicitly contained in the related elements describing that 'thing'. More formally, the `specification` should not restrict the set of licit interpretations any further than what has been done in the modelling elements `scope`d by it.

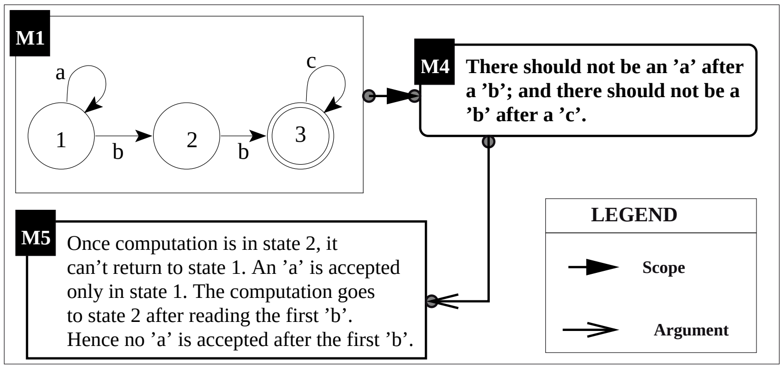

M1
a
c
1
b
2
b
3
M4
There should not be an 'a' after a 'b'; and there should not be a 'b' after a 'c'.
M5
Once computation is in state 2, it can't return to state 1. An 'a' is accepted only in state 1. The computation goes to state 2 after reading the first 'b'. Hence no 'a' is accepted after the first 'b'.
LEGEND
Scope
Argument

Figure 4: A logical view showing a specification and an argument for it.

4.2.4 Argument

In the language of mathematical logic, a proof means a sequence of *formulae* called the *premiss*, followed by another formula which is the *conclusion*. Similar structures are visible in technical document where arguments are given for a design decision based on already known facts. These arguments do form an important part of any technical document. Such a fragment qualifies to be called an **argument**. In spirit it is similar to a premiss part of a formal deductive proof. The role of the conclusion is played by a `specification` in the document.

Figure 4 shows a section of the logical view of a document with a `specification` (M4 of figure 3) and its `argument` (M5, shown in a bold rectangle). The `argument` again looks very similar to the `specification`, with its list of facts, formulae or assertions. But the purpose of its being there in the document is to reinforce the `specification`, and not to independently state properties of the element. The `argument` transitively `scope`s the modelling element that is `scope`d by its `specification`; hence refers to names in it.

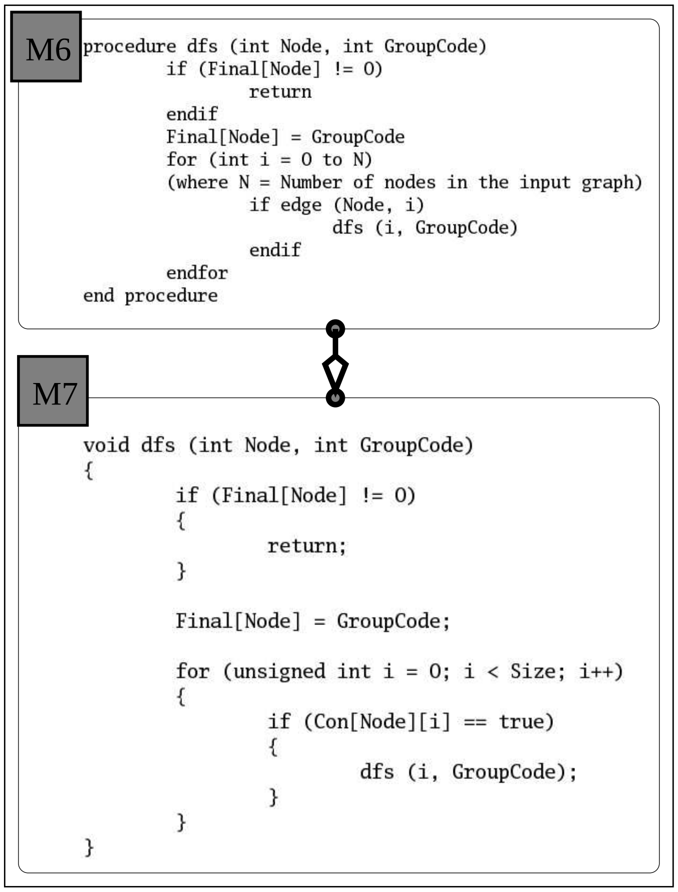
4.3 Relation Elements

4.3.1 Equivalence

When the contents of two modelling elements mean the same thing, they are related to each other through an **equivalence** relation.

4.3.2 Refinement

When one fragment explains in a greater detail, something that another has already described, the former is a **refinement** of the latter.



```
procedure dfs (int Node, int GroupCode)
        if (Final[Node] != 0)
                return
        endif
        Final[Node] = GroupCode
        for (int i = 0 to N)
        (where N = Number of nodes in the input graph)
                if edge (Node, i)
                        dfs (i, GroupCode)
                endif
        endfor
end procedure
```

```
void dfs (int Node, int GroupCode)
{
        if (Final[Node] != 0)
        {
                return;
        }

        Final[Node] = GroupCode;

        for (unsigned int i = 0; i < Size; i++)
        {
                if (Con[Node][i] == true)
                {
                        dfs (i, GroupCode);
                }
        }
}
```

Figure 5: An example of refinement: The source code of a function is a refinement of its pseudo code.

A `refinement` relation is reminiscent of the traditional *horizontal traceability*[13] in requirement engineering usage. This is evident in Figure 5, which shows two `object`s. One is the pseudocode of a version of a depth first search algorithm (M6 in figure 5). The other is the source code written in C programming language (M7 in figure 5). A source code fragment implementing an algorithm written in pseudo code is a `refinement` of it, since it contains language specific implementation details which are absent in the pseudo code. Note that M6 could reside in the *low level design document*, while M7 is a part of the source code implementation. Let us mention here that the logical view of the technical documentation of a complete software project spans across all individual technical documents, including the source code.

4.3.3 Scope

The relation shared by a `class` or `object` fragment with a `specification` is the **scope** relation. A *scope* relation between two modelling elements implies that one specifies a verifiable property of another.

4.3.4 Argument

The relation between an `argument` modelling element with a `specification` modelling element which is its 'conclusion,' is again named **argument**.

4.3.5 Instance

The meaning of this relation is akin to that in the object oriented domain. The relation between an `object` modelling element and its `class` modelling element is called **instance**.

4.3.6 Dependence

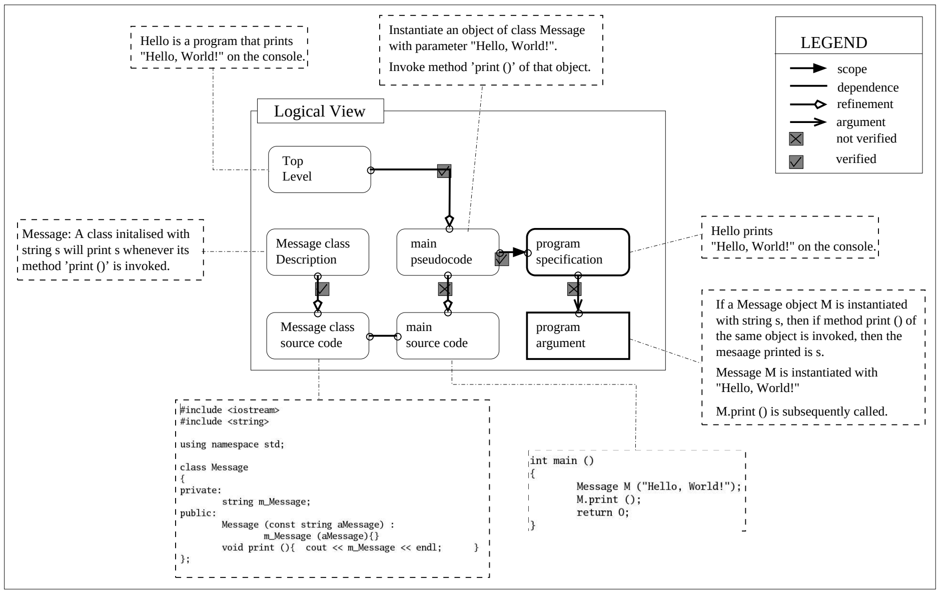
One important motive behind drawing the above five kinds of relations between various modelling elements is to explicate the traceability links between them. The objective of coming up with a taxonomy of different kinds of relations is to characterise these relations in a more fine-grained manner. However, unfortunately, complex relations actually exist between modelling fragments. Many of them cannot be honestly modelled as per the above five relation elements. However, the fact that there exists some relation cannot be denied. In such cases, YAML allows the designer to model these with the **dependence** relation.

4.4 Constraints

The following constraints define the rules of semantic well-formedness of a YAML document. Their accompanying mathematical representations assume the following context:

$\mathbf{M} = \{$ `class`, `object`, `specification`,
`argument` $\}$
$\mathbf{R} = \{$ `equivalence`, `scope`, `refinement`, `argument`, `instance`,
`dependence` $\}$

$A\ R\ B$ where
$A : \mathscr{A},\ B : \mathscr{B}$ where $\mathscr{A}, \mathscr{B} \in \mathbf{M}$
R: $\mathscr{R} \in \mathbf{R}$



Hello is a program that prints "Hello, World!" on the console.

Instantiate an object of class Message with parameter "Hello, World!".

Invoke method 'print ()' of that object.

Message: A class initalised with string s will print s whenever its method 'print ()' is invoked.

Hello prints "Hello, World!" on the console.

If a Message object M is instantiated with string s, then if method print () of the same object is invoked, then the mesaage printed is s.

Message M is instantiated with "Hello, World!"

M.print () is subsequently called.

```
#include <iostream>
#include <string>

using namespace std;

class Message
{
private:
        string m_Message;
public:
        Message (const string aMessage) :
                m_Message (aMessage){}
        void print (){  cout << m_Message << endl;      }
};
```

```
int main ()
{
        Message M ("Hello, World!");
        M.print ();
        return 0;
}
```

Figure 6: The complete documentation of the **'Hello, World!'** project.

means that A is related to b by R where A and B are modelling elements and R is a relation.

1. Two modelling elements can have only one link between them.

2. An `instance` relation can exist only between a `class` and an `object`. The `class` is the source and the `object` is the destination.

$$(R : \texttt{instance}\ ) \Rightarrow (A\text{: }\texttt{class}\ ) \wedge (B\text{: }\texttt{object})$$

3. A `scope` relation can have only a `specification` as its destination. The source end may have either a `class` or an `object`.

$$(R : \texttt{scope}\ ) \Rightarrow (A\text{: }\mathscr{A} \in \{\texttt{class}, \texttt{object}\}\ ) \wedge (B\text{: }\texttt{specification})$$

4. `Equivalence` relation may exist between any type of modelling elements. However, the type of the modelling elements at both ends should be the same for a given instance of `equivalence`.

$$(R : \texttt{equivalence}\ ) \Rightarrow (A\text{: }\mathscr{A}) \wedge (B\text{: }\mathscr{B}) \mid (\mathscr{A} = \mathscr{B})$$

5. `Refinement` relation may exist between any type of modelling elements. However, the type of the modelling elements at both ends should be the same for a given instance of `refinement`.

$$(R : \texttt{refinement}\ ) \Rightarrow (A\!: \mathscr{A}) \wedge (B\!: \mathscr{B}) \mid (\mathscr{A} = \mathscr{B})$$

6. An `argument` relation should have a `specification` as its source and an `argument` modelling element as its destination.

$$(R : \texttt{argument}\ ) \Rightarrow (A\!: \texttt{specification}\ ) \wedge (B\!: \texttt{argument})$$

7. A `dependence` may exist between any two modelling elements.

4.5 More Properties

Now we present some more properties which can act as a representive rule-base, along with the constraints mentioned above, to formally define the condition of logical well formedness of a document. More complex well formedness properties can be derived from these while doing semantic analysis of a YAM document.

1. `Equivalence`, `refinement` and `dependence` are transitive and reflexive relations.
   Quite a few redundant links can be removed taking this property into account.

2. `Equivalence` and `dependence` are symmetric. Therefore, the direction of these relations need not be considered while modelling.

3. `Equivalence` is a type of `refinement`. It is a symmetric `refinement`.
   Hence, a cycle involving all refinements should be replaced by all `equivalences`.If that cannot be done, there is something wrong in the choice of `refinement` as a describing relation between the involved modelling elements.

4. `Scope` is an inverse of `refinement`. Considering them as each other's inverse makes sense for a verification tool writer and the semantic analyser. However it is better to consider their difference as per the definition while actual modelling.

5. If a `specification` is the `scope` of a `class`, it implicitly becomes the `scope` of all the `objects` which are `instances` of the `class`. This

implies that in most cases it doesnot make much sense to add additional `scope` relation $s2$ between a specification S with object O if there already exists `scope` relation s_1 between the `specification` and the `class` C of which O is an `instance`. Consider a scenario where a specification S is `scope` s_1 and s_2 of a `class` C and of an `object` O respectively, and that O is an `instance` of C. Now if a modification is done in S, both s_1 and s_2 get *affected*. However, if s_1 is subsequently *verified* to hold, it implies that s_2 also can be marked as *verified* without directly being verified.

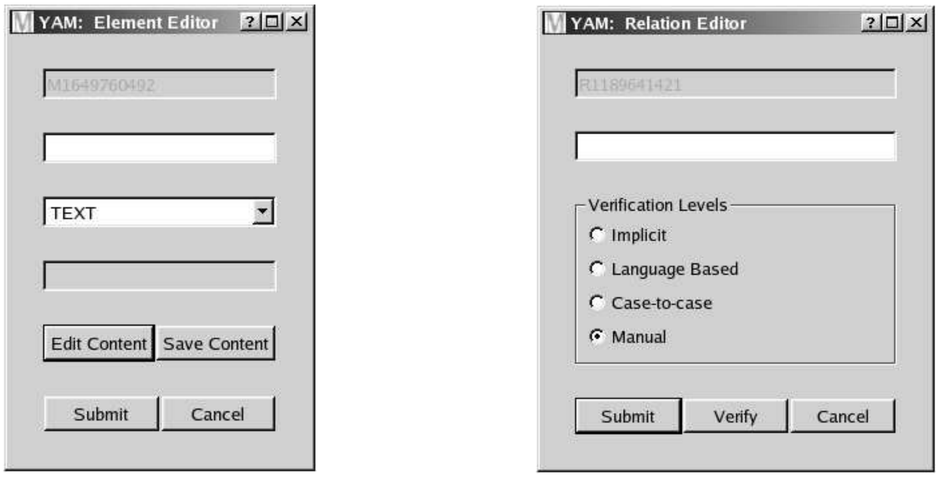
6. If a `specification` is the `scope` of a modelling element, it implicitly becomes the `scope` of all the `refinements` of that element. Therefore, in most cases it doesnot make sense to show a specification S as a `scope` s_2 of a modelling element M_2 if S is already a `scope` s_1 of another modelling element M_1, and M_2 is a `refinement` of M_1. In such a scenario, a modification in S causes both s_1 and s_2 to be marked as *affected*. If verification of s_1 marks it back as *verified*, s_2 too can be marked as *verified*.

Please note that YAM's constructs concern themselves only with the overall logical structure of the document, not with its content. Hence, filling up contents in the modelling elements which agree with its logical role depends on the user. Errors therein would fall in the purview of the external verification tools integrated with YAM as plugins.

5 YAM – A Technical Documentation System

In this section we present a technical documentation system called YAM. YAM is essentially a framework that allows creation of technical documents through a logical view based on YAML constructs. Through its object-oriented architecture YAM provides a simple interface – based on *adaptor*, template, and strategy design patterns [9] – through which an arbitrarily rich set of languages and tools can be integrated into YAM. This makes a very powerful documentation system – rich in its repertoire of languages and tools, and very resilient to inconsistencies. We have implemented a prototype of YAM, and have used it on case-studies proving its usefulness and power.

We describe the basic user interface of YAM by working out a complete example documentation project. The subsequent two subsections explain in some detail the heterogeneous modelling and verification perspectives of YAM.


YAM: Element Editor

M1649760492

TEXT

Edit Content | Save Content

Submit | Cancel

YAM: Relation Editor

R1189641421

Verification Levels
- Implicit
- Language Based
- Case-to-case
- [✓ Manual]

Submit | Verify | Cancel

(a) (b)

Figure 7: (a) The Element Editor; (b) The Relation Editor

5.1 The Smallest Documentation Project – An Example

Let us take an example driven approach to get introduced to YAM's main features. Consider the classic 'Hello, World!' program. Figure 6 shows a complete documentation for this program. The central part contains the complete logical view – introduced in section 4 – of the project. We have maintained the conventions as introduced in section 4. There are two points of difference to notice in the logical view shown here as against examples of the last section. One is that, instead of showing the contents of the modelling elements inside them, we have extracted them out and shown them enclosed in dashed rectangles outside the boundaries of the logical view. The correspondence between a modelling element and its content is shown by dashed links. Though, in figure 6, the contents are shown alongside the logical view, in our current implementation of YAM, the main logical view does not show the contents. To view the contents, the user has to open it in its editor. The other point to note is that each traceability link is tagged with a tick ($\surd$) or a cross ($\times$). They signify the 'verification status' of the relations. A ticked link means that the modelling elements on both ends are *verified* to be related to each other by the given relation. In other words, this link is *verified*. A crossed link means that one of the connected elements has undergone a modification, after which the link has not been successfully verified, i.e. the relation is currently *not verified*. A successful verification of a crossed link would tick it. A document is proven consistent when all the links in it are ticked.

This is how a document would look like in YAM during an advanced state of evolution of a project. We now turn our attention to the steps in the basic process of creation of such a document.

5.1.1 Creation of a Basic Document

A documentation process starts with creation of documentation project in YAM. Let us name our documentation project as 'hello-world.' Then we approximately follow the following steps to arrive at a final state similar to the one shown is figure 6.

1. We create a modelling element of the type `object`[2]. We name it as 'Top Level.' We right click on the element and choose to **edit** the element. The **Element Editor** dialog appears. Its appearance is as shown in figure 7(a). Among other things, we choose an appropriate formalism (in this case `TEXT`) for the content of this modelling element.

2. After *submitting* this information, we press **Edit Content** button on the element editor. The model editor corresponding to the formalism that we had chosen (in this case `TEXT`) opens. In that we fill in the contents of the top level description of the program (in this case: 'Hello is a program that prints "Hello, World!" on the console'). Thereby, we submit the content from within the model editor and exit the editor. We are now back to the **Element Editor** dialog.

3. We press the **Save Content** button. Selecting an appropriate location and file name for the contents of 'Top Level,' we save the content. The first modelling element of 'hello-world' project, along with its contents, is created.

4. We add another `object` named 'main pseudocode' in similar lines, write its content using the `TEXT` editor, and save the contents.

5. The 'main pseudocode' is a `refinement` of the 'Top Level' object. We pick the `refinement` relation from the toolbar and connect these two elements with it – with 'Top Level' as the source, and 'main pseudocode' as the destination.

[2]The reason to choose an object for this element is in accordance to the definitions of the modelling element types as explained in section 4.

6. The `refinement` relation between 'Top Level' and 'main pseudocode' is in 'not verified' state, signified by the cross appearing on it. We right click on the link and choose to **Edit** the link. The **Relation Editor** dialog appears. Its appearance is as shown in figure 7 (b).

7. We choose the verification level as 'Manual'. For the present, we just press the **Verify** button on the **Relation Editor**. YAM asks us to confirm if we consider the given link as 'verified.' We say 'Yes', causing the link to get 'ticked.' Our document now contains two modelling elements, connected through a traceability link (of the type `refinement`) which we have asserted to be 'verified.' The current state of our document is 'consistent.'

8. Other elements and traceability links are subsequently added in a similar way. In the process we also set all the *not verified* links to *verified* state.

This completes the creation of the complete documentation of the 'hello-world' project. Two main features of YAM, which have found mention several times earlier in this paper, are:

1. Heterogeneous Modelling

2. Verification

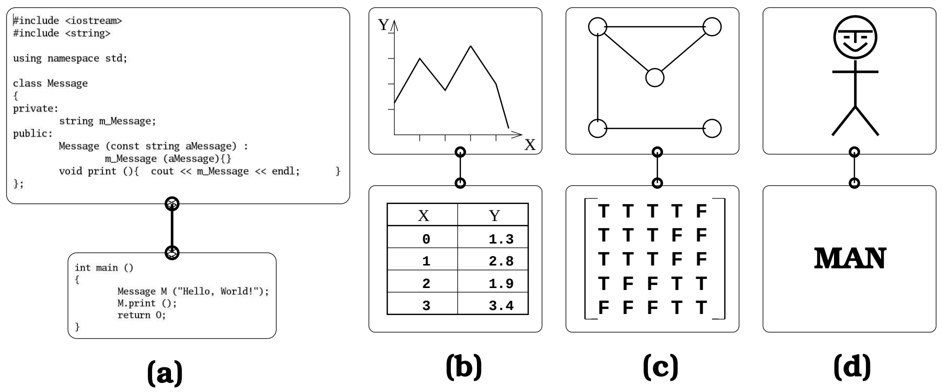
We explain where they figure while using YAM in the next section.

6 Discussion

6.1 Formal Usage Context of YAM

Consider the case when modelling element $A : \mathscr{A} \in \mathbf{M}$ is related to modelling element $B : \mathscr{B} \in \mathbf{M}$ through a relation $R : \mathscr{R} \in \mathbf{R}$, where:

$\mathbf{M} = \{$ `class`, `object`, `specification`, `argument` $\}$
$\mathbf{R} = \{$ `equivalence`, `scope`, `refinement`, `argument`, `instance`, `dependence` $\}$
content(x) denotes the content of a modelling element x.
Tool pool of YAM is nothing but the set of tools that have been plugged into it, and hence are free to be used by the user. We denote this tool pool by
$\mathbf{T} = \{\mathbf{L}, \mathbf{E}, \mathbf{V}\}$
$\mathbf{L}$ is a set of modelling languages whose editors are available as



```
#include <iostream>
#include <string>

using namespace std;

class Message
{
private:
        string m_Message;
public:
        Message (const string aMessage) :
                m_Message (aMessage){}
        void print (){  cout << m_Message << endl;      }
};
```

```
int main ()
{
        Message M ("Hello, World!");
        M.print ();
        return 0;
}
```

X	Y
0	1.3
1	2.8
2	1.9
3	3.4

$$\begin{bmatrix} T & T & T & T & F \\ T & T & T & F & F \\ T & T & T & F & F \\ T & F & F & T & T \\ F & F & F & T & T \end{bmatrix}$$

Figure 8: Verification Levels: (a) Implicit (b) Language Based (c) Case to Case (d) Manual

plugins to YAM.
E is a set of model content editors, i.e.
$\mathbf{E} = \{\mathscr{E} \mid language(\mathscr{E}) \in \mathbf{L}\}$ where $\mathscr{E}$ is an editor that allows modelling in the language $language(\mathscr{E})$.
V is a set of verifiers: i.e.
$\mathbf{V} = \{\mathscr{V} : (\mathscr{L}_1 \times \mathscr{L}_1 \times \mathscr{R}) \mapsto \texttt{boolean}\}$.

Here, '$\mathscr{V} : (\mathscr{L}_1 \times \mathscr{L}_1 \times \mathscr{R}) \mapsto \texttt{boolean}$' denotes a verification tool that can check for the validity of a link $R : \mathscr{R}$ between two modelling elements A and B such that $content(A) \in \mathscr{L}_1$ and $content(B) \in \mathscr{L}_2$. Here,
$\mathscr{L}_1$ and $\mathscr{L}_2$ are two languages (or formalisms, or notations) in which the contents of A and B are modelled. Of course, it is required that $\mathscr{L}_1, \mathscr{L}_2 \in \mathbf{L}$.

6.2 Heterogeneous modelling

In the **Element Editor** dialog, there is a facility to select the **formalism** of the contents of the modelling element we are editing. Each of these formalisms corresponds to a modelling language – formal or informal – whose editor is present as a plugin to YAM. Once a formalism is selected for a modelling element, the corresponding editor is invoked by YAM each time the user presses the **Edit Content** button. Once a content is created and submitted, the user is no more allowed to change the formalism of the modelling element while the modelling element exists. Modelling elements may of course be deleted by the user.

The liberty to select an appropriate formalism, or language, to create the content of a modelling element is what separates the processes of creation of document logical structure, and filling up of its contents, from one another.

This simple principle keeps open the possibility of subsequent plugging in of an arbitrarily large number of formalisms (i.e. essentially the corresponding model editors) into the tool pool of YAM. This realises YAM's promise of open ended support for heterogeneous modelling. The implementation of this feature perhaps doesnot qualify as a research work. But it was certainly an interesting software design exercise which involved a clever use of object oriented design principles[3]. We forego that discussion due to reasons of space constraint.

6.3 Verification

Whenever two modelling elements are connected by creating a link between them, the link is initially in a *not verified* ($\times$) state. The relation can be verified by pressing the **Verify** button on the **Relation Editor** dialogue box. We have observed that the type of verification the user may want to do can be classified into four levels: **implicit**, **language based**, **case to case** and **manual**. Each of these roughly corresponds – in a decreasing order – to a particular level of formalism in the modelling languages used in the linked modelling elements. Below we discuss the meaning of each of these verification levels.

6.3.1 Implicit Verification

Figure 8(a) shows a typical case where the user would select an implicit verification level. When the source code of the main and class Message are compiled, all possible conditions of consistency will be taken care of by the compiler. The designer may still prefer to model them as two separate modelling elements, and add a dependence link between them to elicit the logical relation. However, he leaves the job of consistency check on the compiler. YAM will cross ($\times$) the link if either is modified. But it will be unconditionally ticked ($\surd$) back when the user presses the **Verify** button on the **Relation Editor** dialog.

6.3.2 Language Based Verification

The user would choose to carry out language based verification when $\mathscr{V} : \mathscr{L}_1 \times \mathscr{L}_2 \times \mathscr{R} \mapsto \texttt{boolean}$ exists in YAM's tool pool. This means that the user is not assuming any consistency check to happen outside the modelling purview. Hence, the verification cannot be left implicit. Moreover, there exists an appropriate verification tool in YAM's tool pool to check this consistency. Figure 8(b) shows an example. Here, $\mathscr{L}_1$ is 'graph plotting notation', say `PLOT`, and $\mathscr{L}_2$ is a 'table drawing notation', say `TABLE`. $\mathscr{R}$, in this case, is `equivalence`. In this example case, it so happens that $\mathscr{V} : \texttt{PLOT} \times \texttt{TABLE} \times \texttt{equivalence} \mapsto \texttt{boolean}$ is available in YAM's tool pool. Therefore, the user chooses 'language based' verification level while creating the given link.

6.3.3 Case to Case Verification

A third case arises when no such verification tool is available. It could be just incidental that user has not been able to develop or acquire an appropriate verification tool to verify $A\ R\ B$. Or the reason could be theoretical. For instance, if two models written as as context-free grammars meant to be two descriptions of the same thing, there cannot be a generic tool that will be able to decide their equivalence. Another possible reason of the unavailability of such a tool is that an editor may be used to model something that is not the default interpretation of a model in the used language. For instance, a graph drawing editor may be used to show a finite state automaton. A comma separated list of names could be used to show the trace of a computation with the above FSA. A general verification tool cannot exist in such non-default modelling. A desirable thing to have in such a case is a way to specify the validity condition of this particular link in some way. This is exactly the facility provided in the 'case to case' verification level. Whenever the user chooses this verification, he has to provide externally, the condition of validity of the link. Figure 8(c) illustrates one more example. Here $\mathscr{L}_1$ is a certain 'graph drawing notation', say `GRAPH`. $\mathscr{L}_2$ is a matrix writing notation, say `MATRIX`. In this example, the plugin verifier $\mathscr{V} : \texttt{GRAPH} \times \texttt{MATRIX} \times \texttt{equivalence} \mapsto \texttt{boolean}$ is not there in $\mathbf{T}$, the tool pool of YAM. User chooses 'case to case' verification level and provides his own case to case verifier $\mathscr{V}'(\mathit{content}(A), \mathit{content}(B), R)$. Here, the type of $\mathit{content}(A)$ and $\mathit{content}(B)$ are no more assumed to belong to any language in $\mathbf{L}$. The detailed implementation of this feature is beyond the scope of this paper.

6.3.4 Manual Verification

Only when none of the above three options are enabled for the current case, YAM will ask for help from the user. This means that in that case the user has to establish the consistency through manual inspection. The user expresses his intention to do so by selecting 'manual' verification level. Figure 8(d) shows an example. An image of a man can be drawn in an image drawing notation, say `IMAGE`. Another fragment may contain a textual description of this man in `TEXT`. It is unlikely that such a verification tool $\mathscr{V}$: $\texttt{TEXT} \times \texttt{IMAGE} \times \texttt{equivalence} \mapsto \texttt{boolean}$ would be available. Manual verification would be necessary in this case.

Let us clarify that at any point, the total number of verification tools required so that all default interpreted models can be verified at the 'language based' level is $\#(\mathbf{L})^2 \times \#(\mathbf{R})$. So many verification tools are not possible to be plugged into the framework. First, this number increases as the square of the number of formalisms plugged into YAM. Second, for certain combinations of modelling languages, such tools may not just be possible to be created. Consequently, practically, for many links created in a documentation, the verification level will be 'manual.' Nevertheless, the benifit of traceability will still be there. And as YAM seasons in the hands of its user, the repertoire of verification tools can be expected to grow over time.

7 Conclusions and Future Directions

In this paper we have introduced a meta language YAML to describe the logical structure of a technical document. The objective of defining a logical structure over the contents of a document is, on one hand, to articulate the logical role played by its various fragments; and on the other, to define traceability links between these fragments. There are multiple benefits in doing so. First, it enforces a system designer to articulate his thoughts better before putting them down into the document. This results in better quality. Secondly, it explicates the dependencies between parts of a document, imparting resilience to the document against inconsistencies creeping in at a later point in time. Thirdly, it opens doors to make use of verification technology, wherever applicable, in maintaining consistency in a seamless manner hand in hand with manual verification.

We also described a modelling framework YAM that realises the above principle. In addition, its framework based architecture rests on the philosophy of open-ended plugin based development. YAM would achieve its full power when various developers join hands to provide modelling and verification tools as its plugins. That way, YAM has a potential to become a very powerful high quality documentation and modelling system.

Quite a few interesting tasks are lined up in the agenda of this project.

In the short term, we are exploring the viability of migrating YAM into the Eclipse platform (currently it is a stand alone C++ program). Eclipse has a naturally plugin based architecture; it is open source; and it has a huge developer community. For all these reasons, Eclipse appears to be a suitable development platform for YAM. Second, we are in the process to add a powerful document generation facility to YAM that will demonstrate its user friendliness even to users with a traditional mindset about documentation, modelling and programming.

In the long term we intend to incorporate enhancements to YAML to allow a richer and more accurate description of the document logical structure. It is obvious that, once done, it will also provide wider and clearer openings towards the automation of consistency maintenance by adapting existing verification tools to this purpose. Our long term objective is to refine YAML to a point where the benefits of using it can be demonstrated on an industrial strength project.

The constraints and properties described in section 4 indicate interesting logical properties of a YAM document. We are exploring ways to implement a well-formedness checker that makes use of these properties One important area of extension seems to be in the area of Software Testing. We are actively investigating this direction. We intend to address additional (not any less important) issues of configuration management and distributedness of development through YAM.

One message that has got stated directly or indirectly several times throughout this paper is that creation of documents and models based on a logical view has several advantages. YAML is a new language that provides the model writer with a novel way to do that. This incorporates another level of knowledge about the traceability and consistency as an inherent part of the document. This opens new avenues of consistency maintenance – both manual and automatic. YAM clearly illustrates the feasibility of this.

References

[1] OMG Unified modeling language specification, Sept. 24 2001.

[2] J.-P. Bodeveix, T. Millan, C. Percebois, P. Bazex, and L. Fraud, editors. *NEPTUNE : Method, Checking and documentation generation for UML application.* NEPTUNE Consortium, 2003.

[3] G. Booch. *Object Oriented Analysis and Design.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[4] F. P. Brooks, Jr. The Mythical Man-Month. *ACM SIGPLAN Notices*, 10(6):193–193, June 1975.

[5] R. Bussow, S. Herrmann, W. Heicking, and W. Grieskamp. An open environment for the integration of heterogeneous modelling techniques and tools, Apr. 01 1998.

[6] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[7] Eclipse.org. The Eclipse homepage.

[8] A. Egyed and C. Gacek. Automatically detecting mismatches during component-based and model-based development, Sept. 15 1999.

[9] R. J. J. V. Erich Gamma, Richard Helm. *Design Patterns Elements of REusable Object-Oriented Software*. Addison and Wisley, Boston, 1994.

[10] C. Gacek. Detecting architectural mismatches.

[11] M. Gogolla and M. Richters. Development of UML descriptions with USE. In H. Shafazand and A. M. Tjoa, editors, *Proc. 1st Eurasian Conf. Information and Communication Technology (EURASIA'2002)*, volume 2510 of *LNCS*, pages 228–238. Springer, 2002.

[12] T. P. U. Group. The precise UML group homepage.

[13] M. Lindvall and K. Sandahl. Practical implications of traceability. *Software Practice and Experience*, 26(10):1161–1180, 1996.

[14] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering*, 26(1):70–93, 2000.

[15] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: a consistency checking and smart link generation service. *ACM Transaction on Internet Technology*, May 2002.

[16] C. Nentwich, W. Emmerich, and A. Finkelstein. Flexible consistency checking. *ACM Transactions on Software Engineering and Methodology*, 12(1):28–63, January 2003.

[17] B. Nuseibeh, J. Kramer, and A. Finkelstein. A framework for expressing the relationships between multiple views in requirement specification. October 1994.

[18] D. F. Redmiles and J. E. Robbins. Cognitive support, UML adherence, and XMI interchange in argo/UML, May 31 1999.

[19] R. Software, Microsoft, Hewlett-Packard, Oracle, S. Software, M. Systemhouse, Unisys, I. Computing, IntelliCorp, i Logix, IBM, ObjecTime, P. Technology, Ptech, Taskon, R. Technologies, and Softeam. *Object Constraint Language Specification (version 1.1).* Rational Software Corporation, Sept. 1997.