Luís Gonzaga Martins Ferreira

FORMALIZING MARKUP LANGUAGES FOR USER INTERFACE

Dissertação para Mestrado em Informática

Escola de Engenharia

UNIVERSIDADE DO MINHO

Braga, 2005

Luís Gonzaga Martins Ferreira

FORMALIZING MARKUP LANGUAGES FOR USER INTERFACE

Dissertação submetida à Universidade do Minho para obtenção do grau de Mestre em Informática, área de especialização em Sistemas Distribuídos, Comunicações por Computador e Arquitectura de Computadores, elaborada sob a orientação do Professor Doutor José Nuno de Oliveira, Professor Associado do Departamento de Informática da Universidade do Minho.

Dissertação desenvolvida no âmbito do Projecto EUREKA IKF (E!2235)

Escola de Engenharia

UNIVERSIDADE DO MINHO

Braga, 2005

À Ana, Aninhas e Ritinha

Abstract

This document presents a Dissertation theme, as integral part of Masters Degree in *Distributed Systems, Computers Architecture and Computers Communication*.

The work has as primary objective the application of formal methods in the specification of presentation layer. Even reaching several relevance HCI concerns, the scope focus essentially on the way how formal methods can be explored to specify user interfaces described using markup languages.

The state-of-the-art analysis of user interface markup languages and *UIML* - User Interface Markup Language formal specification are main contributions. Therefore the *tabular* graphical object OLAP main features are formally specified in *VDM-SL* and animated using UIML.

This research should be considered as a contribution towards a definition of a visual component library , with user interfaces components composition and reuse.

Acknowledgements

I would like to thank my supervisor Professor José Nuno Oliveira, member of DI (Department of Informatics, Minho University[1]), who encouraged all formal methods research and initiatives at the University, for his useful support and advice during this work. I am also grateful to Mark Adams and James Helms *Harmonia* members, for their enthusiastic support along this research as well for the availability of their *UIML* supporting tools.

To all which improved the English in this thesis, I am also very grateful.

Thanks also to all my friends which have made contributions to this work.

would like to thank my Ana for her admirable patience and support during this work, and our daughters Aninhas e Ritinha, which were deprived of their father for the best part of these years.

[1]http://www.di.uminho.pt

Table of Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Overview

There will always be old software$_{(Hausi\ Muller)}$

The human brain is able to learn and to reason in vague and imprecise contexts. It can decide upon imprecise and qualitative data. In contrast, formal methods (logical and mathematical) demand accurate and quantitative data.

Since the computer machine appeared, we understood its capabilities, when it was properly used. It is true that the computers have important capabilities, but they still are far away from being a match for the human intelligence. Nevertheless man is making an effort in "instructing" these machines in that sense.

It is one of the great challenges of ours days, that complex data, contained in databases of great dimension and longevity are processed and which are increasing exponentially in size. The Internet provides obvious evidence of all this.

From another perspective, for the actual companies it is crucial the full integration and consistency of all the information which flows or is stored in its databases. The number of specific and different applications, manipulating this information, is necessarily big. Maintenance and support of these applications requires expensive team work.

The (conventional) *Relational Database Management Systems (RDBMS)* no longer can guarantee timely efficiency in the answers to complex queries. The treatment of information in bidimensional format (tables and spreadsheets, for example), the inability of the analysis, transformation and consolidation of such information, restrict the overall application of these systems.

If we look at the young history of Computer Science (almost nothing existed before 30 years), we find orientations more or less objective, always focusing on the principle of trying to reach something that does not exist. A lot of times following typical "azimuthes" with "boomerang" effect, believing that, tuning certain options and returning again to the starting point, will reach something different, innovative or not. Practically similar to the new fashion effect. Something like "déjà vu".

This can be easily seen in several programming language paradigms, which appear everyday. If we carry these thoughts to the User Interface development area in several computer applications, we obtain a not less important question and perhaps, more critical, since progress on it was a little shallow along the last years.

From observing multiple facts, one can observe that large companies involved in software development, invest hardly in user interface quality and associated features. *HCI -Human-Computer Interaction* [SIG, oM03], is a branch of research in *Computer Science*, focused precisely on this kind of problems, where one tries to get near to human behavior, in the way the application functionality is presented to the user. Ergonomic issues, as well as organization and process logic are key points in these works.

The scope of this research work, although it can reach, with some relevance, HCI concerns, focus essentially on the way how formal methods can be explored to specify the user interface graphical objects and associated features.

We focus on an initial and specific interface and its related issues which can then be translated to different platforms. The *Web* is an evidence of this. Several entities are converting their normal applications to be available also in the Internet. The quick Web spread, the *B2B*[1] scenario, the emergence of *EAI*[2] concerns, can justify many of these decisions.

The advantages of formal specifications of complex systems using formal methods are well known. It is known and accepted by a considerable group of persons, as a way to avoid ambiguities and clearly guarantee correctness and consistency of the developed programs. In this context, it is useful to experiment these methods on user interface development, trying some kind of merging process of new methods specifications with existing graphical objects specifications.

$$ProgramSpecification = Specification(Data + Services + Interfaces)$$

As a consequence of this large set of graphical elements, many of them only found in pure commercial application, its formal specification process, towards a *Visual Component Library - VCL*, could becomes complex. This complexity can even increase during the tentative abstraction of real problems.

An interface specification, based on the intended model, i.e., strongly supported by a top-down analysis of existent interface and end user opinions, can conduce to a partial system specification. This is due to the analyzed interfaces inability to represent correctly the all system. In this sense, to get a deep abstraction, we think it is fundamental to consider also the interfaces specification based on operations, properties and services.

1.2 Problem Statement

During all this period of continuous advances in the development of software technologies, the programmer always needed to deal with two complementary challenges:

[1]B2B - Business to Business
[2]EAI - Enterprise Application Integration

from one side he needed to solve problems and from the other side he needed to do it properly (that is, with correction and adequate performance).

Jumping from the interpreted to the compiled programming language paradigm cannot be justified only by quality. We can not say “one language is better than an other” without discussing whether it explores efficiently the capacities of the support technologies, namely the *Hardware*.

Let us in retrospective, review some programming languages/technologies, looking for very simple examples of common problems.

Listing 1.1, presents a piece of *BASIC* source code which implements the mathematical function $F(x) = x^2$, applied to a range of values from *A* to *B*.

Listing 1.1: *Function $F(x) = x^2$ implemented in BASIC*

```
REM TABULATE A FUNCTION F(X) FROM X=A TO X=B IN STEPS DX
REM ENTER THE FUNCTION F(X) IN THE NEXT LINE

DEF FN F(X)=X^2
:
INPUT "FROM X= ";A
INPUT "   TO X= ";B
INPUT "IN STEPS OF ";DX
:
FOR X=A TO B STEP DX
        Y=FN F(X)
        PRINT X,Y
NEXT X
:
END
```

We observe that the whole program (in this case interpreted), constitutes only one source code document. The way the information reaches the end user (at this time, the term User Interface was not common) does not deserve any special attention.

The next program fragment (Listing 1.2), implements a *Stack* structure in *Pascal*, a compiled language. Although some of the source code organization is in *modules*, *functions* or *procedures*, some concepts, related with data calculation, user integration and interaction are merged when reading values and presenting results (see e.g. the *addToStack* function)

Listing 1.2: *A* PASCAL STACK *implementation*

```
{*STACK module - by lufer*}

program exStack1 (input, output);
type
        pcell = ^cell;
        cell = record
                no: integer;
                data: string;
                nxt: pcell;
        end;

var
        top, temp: pcell;
        nserial, ncount, ID: integer;
        ch: char;
        data: string;

procedure addToStack;
```

```
begin
        if top = nil then
        begin
                new(top);
                top^.nxt := nil;
        end
        else
        begin
                temp := top;
                new(top);
                top^.nxt := temp;
        end;
        top^.no := nserial;
        nserial := nserial + 1;
        writeln('Cell of ID ', top^.no : 4, ' was added at
                the top of stack!');
        writeln('Please Input some data');
        readln(top^.data);
        writeln;
end;

procedure removeFromstackTop;
begin
        if top <> nil then
        begin
                temp := top;
                top := top^.nxt;
                writeln('-------------------------------------');
                writeln('Cell ID = ', temp^.no);
                writeln('Data :   ', temp^.data);
                writeln('-------------------------------------');
                writeln;
                writeln('Cell of ID ', temp^.no : 4, ' was removed
                        from top of stack!');
                writeln;
                dispose(temp);
        end
        else
        begin
                writeln('Stack is empty!!');
                writeln;
        end;
end;

procedure ScanStack;
begin
        temp := top;
        ncount := 1;
        if temp = nil then
        begin
                writeln('Stack is empty');
                writeln;
        end
        else
                while temp <> nil do
                begin
                        writeln('---------------------------------------');
                        writeln('This is ', ncount : 4, ' th Cell!');
                        writeln('Cell ID = ', temp^.no);
                        writeln('Data :   ', temp^.data);
                        writeln('---------------------------------------');
                        writeln;
                        temp := temp^.nxt;
                        ncount := ncount + 1;
                end;
end;
```

```
begin
        writeln('This is a Demo for understanding Stack!!');
        nserial := 1;
        repeat
                writeln;
                writeln('Please Select function!');
                writeln('A)ddCellToStack  R)emoveCellFromStack
                        S)canStack  E)nd');
                readln(ch);
                writeln;
                if (ch = 'A') or (ch = 'a') then
                        AddToStack
                else if (ch = 'R') or (ch = 'r') then
                        RemoveFromStackTop
                else if (ch = 'S') or (ch = 's') then
                        ScanStack;
        until (ch = 'E') or (ch = 'e');
        writeln('Program was assigned to end!!');
        writeln('Bye!!');
end.
```

Thinking of (and more real critical) situations, like supporting bank information systems, insurance companies, etc., the scenario is the same. Surely much investment was made on improving and certifying these kind of systems. Not only large hardware equipment had to be acquired but also complex software had to be developed. Having done so, the whole attention was put on ensuring that“nothing wrong” could happen to the system. The way the operations were handled by the users, passed to a secondary plane. In summary, the interface remained the same for a long time even if the systems were frequently upgraded.

In ours days things are different. One can see the natural tendency of users to “accept” new technologies (while the number of end users is increasing), like *Internet*, *wireless*, mobile devices, etc. Maybe because the systems are considered as being stable and enough robust for their intended purposes.

Almost every company puts effort in having their business contents available in these new emerging resources. Many of the existing systems remain as they are and the interfaces need to be frequently transformed and many times created new ones to support the new devices. The main concern nowadays is to ensure interoperability among new software components, applications, etc., on top of existing ones.

However, user interface transformation or adaptation is not yet adequately handled and ensured. If not anarchic, it lacks scientific rigor. Chapter 2 addresses this topic.

An important question is: what is *adaptability* and how can it be, if it can be, measured? We will see later (on section 2.4.1) that adaptation means changing a system to reflect changes in the environment. In the context of user interfaces, where end users can question non functional requirements such as usability, simplicity, etc., it will be interesting to analyze how visual component interfaces reflect them.

1.3 Motivation and Objectives

The work described in this thesis stands from the thoughts of Brad Myers [Bra98]. The area of the UI is an example of the strong influence of academic research in the industrial way of doing things. It is often assumed mistakenly that, in case the universities do not do it, industry will take charge of developing. There are several facts that reflect the erroneous way of this thinking.

Today practically all kinds of interface standards result from strong influences of the research in teaching institutions.

In its essence, this thesis work, intends to apply formal semantics techniques to the user interface development, in order to impose scientific rigor on the whole process, from interface specification and validation to its transformation ("transcoding").

The application of formal methods should never surpass the limits of versatility, trying whenever possible to build the specification through the composition of components previously specified.

This work is targeted at formally specifying visual components, classifying new components and reusing existing ones that are already classified. We refer to visual components used in actual tools for *Graphical User Interface* development (GUI), such as *ListBox*, *Button*, *TextBox*, *Menu* (Figures 1.1,1.2 and 1.3), which lack rigorous specification, both at the functional (semantic) and structural (syntactic) levels.

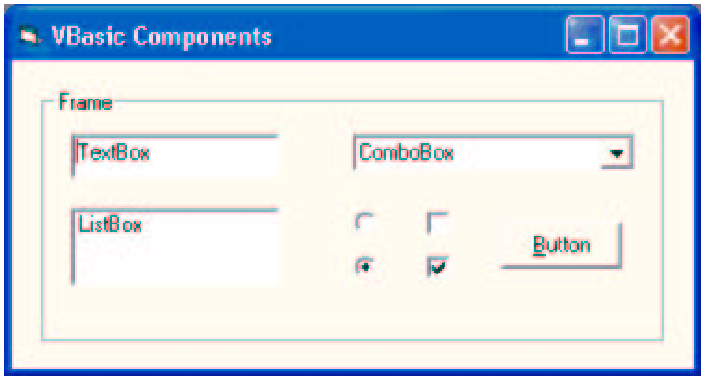


Figure 1.1: *VBasic visual Components*

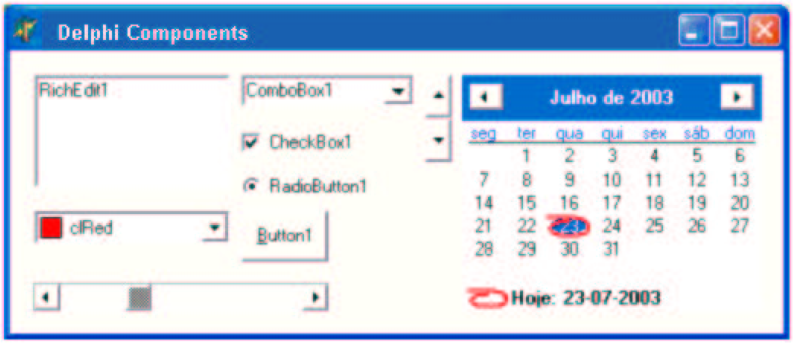


Figure 1.2: *Delphi visual Components*

By applying formal methods, UI development becomes a rigorous discipline with focus on higher abstraction levels (relative to implementation), using visual compo-

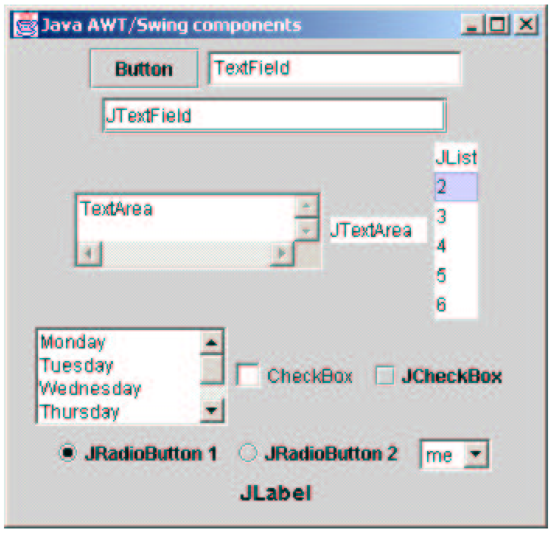


Figure 1.3: *Java AWT/Swing visual Components*

nents and a semantic set of rules which describes the process.

The notation chosen to define components is supported on *Sets Theory* [Oli03]. The semantic specification of each component will resort to semantic models, describing the distinct internal states, associated to execution of each operator. For instance, in a *ListBox* one must specify the effect of *Insert*, *Remove* or *Select* events of some of its items (in this case because a *ListBox* is a composite component).

The traditional process of application development, culminates with the creation of complex interfaces, which facilitate the interaction with the information system.

With the advent of *Application Program Interfaces (APIs)* and using some available interface operators, one can create quickly and globally accepted development methods. However, the process of interface design appears for the developer as something conditioned by the information organization on the database or any other information support system.

The choice of a *ComboBox* or a *ListBox* for instance, is done, knowing that these components will represent some specific information. In this way, any necessary change to an interface, could be a delicate and perhaps not very flexible process.

For all this process of UI development, this thesis suggests the use of formal methods. User interface design should be based on formal specification, before its final representation or implementation on a particular platform.

The process usually starts from a data model, properly (or not) normalized, which describes the intended information system. Using a formal specification language, *VDM-SL* in this case [Hop01], developers will try to specify the data model, by identifying the necessary components of the interface for its information visualization.

It is a typing process that guarantees coherence in the final placement of the components in the interface layout. Also for each component it can be of interest the definition of conditions (invariants) which, in the later phase of layout placement, justify its relative position.

Figure 1.4 depicts the main intervenients int the user interface development process. *C* is the traditional method while *A* and *B* pictures the approach followed in this thesis.

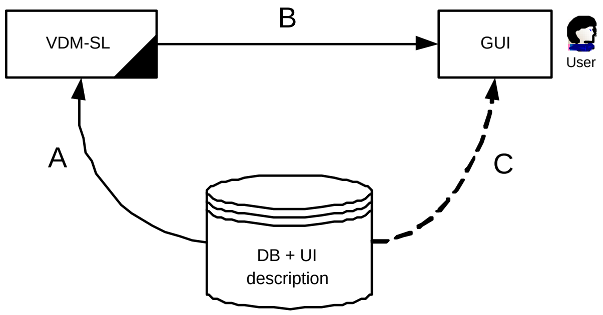


Figure 1.4: *Interfaces development*

In this figure, *A* represents the modelling process in our prototyping platform (*VDM*), of the intended data model. Database tables and their relationships will be represented in *Set* notation.

The *B* process carries out a simplification of the initial structures created by A, by a process of eventual matching of the available visual components for each structure specified. This process should finish with the respective interface presentation, in a specific environment.

By contrast, *C* represents the normal process of interface development. A specific software "translates" the data model in *forms* which will be the application interface.

Our approach will be described in Chapter 5, where all these concepts will be described in detail. Following we summarize this process, using a simple formal modelling process.

Consider the non (normalized) relational structure, depicted in Table 1.1. In this structure *Student* is the primary key and *Mark* is the final calculated mark considering *Lab* and *Exam* marks.

Student	Lab	Exam	Mark
António	10	12	11
Ana	14	14	14

Table 1.1: *Non normalized Relational Structure*

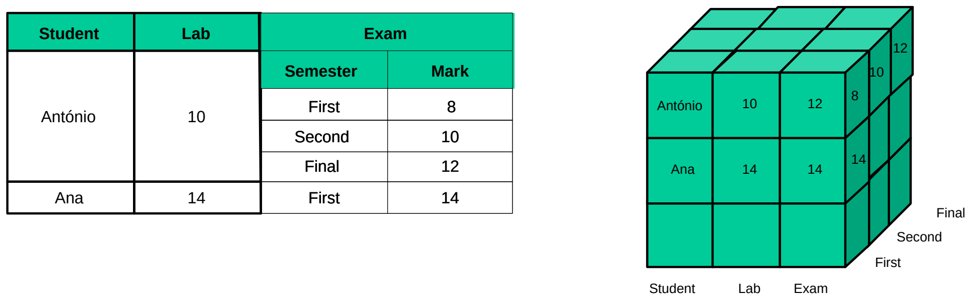
Modelled in *Sets* notation, this structure is written as follows:

$$Student \hookrightarrow Lab \times (Exam \hookrightarrow Mark) \tag{1.1}$$

From this expression, a normalized view can be derived by calculation:

$$(Student \hookrightarrow Lab) \times (Student \times Exam \hookrightarrow Mark) \tag{1.2}$$

We can now try to represent this structure using a multidimensional array (a technology dealt with in Chapter 2), which constitutes one of the great advances of *Multidimensional Analysis* [Nig01]. The result can be represented in the next figure, as follows:



Student	Lab	Exam	
		Semester	Mark
António	10	First	8
		Second	10
		Final	12
Ana	14	First	14

Figure 1.5: *Example of multidimensional representation*

The result of applying some operators associated to this kind of representation (later we will describe operators like *Rotating*, *Ranging*, etc.) could now be easily applied and observed.

In the context of using multiple development technologies, with reengineering and the process of integrating current and legacy systems an important demand and in the context of a continuous investment on distributed and mobile systems with end user interaction, it is essential the presence of a *System Architect* person.

The purpose of this work is to provide guidelines for user interfaces specification and development in computer science, based on formal methods as follows:

- to show the applicability of formal methods in the specification and development of interfaces;
- to show the applicability of formal methods in software reengineering processes;
- to analyze the importance of the actual technologies of Markup languages in integration processes;
- to create in *VDM* the kernel of a visual component library for user interfaces specification support;

Figure 1.6 depicts the four main phases of our process:

- *Phase 1* - *Transcoding* - developing of a mapping process from source code to *VDM-SL* notation;
- *Phase 2* - *Validation* - inverse process which will allows to obtain a source description from a formal VDM-SL specification;
- *Phase 3* - *Abstraction* - progressive abstraction of existent formal specification, using calculation.

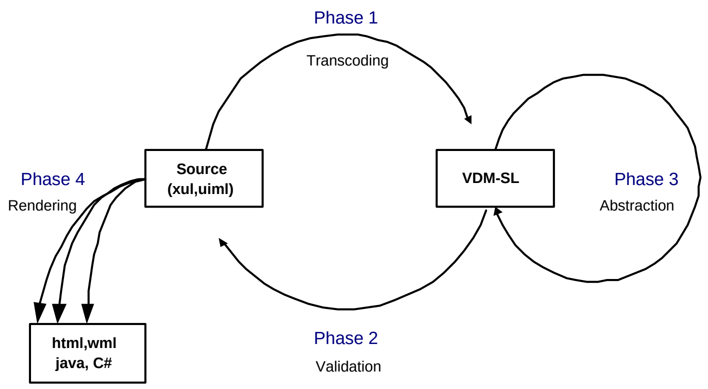


Figure 1.6: *Phases of the formalization process*

- *Phase 4* - *Rendering* - rendering mechanism to analyze the result in a particular platform.

Our work will focus onto the area of the specification through *DSL* - *Domain Specific Languages* [ABB$^+$97] because, just as its name suggests, we will try to focus on components specification regardless or abstracting their implementing details, although continuing to support the creation of prototypes. We will use *VDM-SL* - *The Vienna Development Method Specification Language* [IFA00c] as our specification language. We will avoid following a specific programming language, working with *GPL* - *General-Purpose Languages*.

1.4 Summary of Contributions

The main contributions of this dissertation are as follows:

- A formal specification of a particular Markup Language (UIML)
- A *XML StyleSheet* to convert *UIML* to *VDM-SL*
- A *VDM* render to generate *UIML* from *VDM-SL*
- A formal specification of a graphical Table
- A formal specification of some basic *OLAP* operators

1.5 Structure of the Dissertation

This dissertation puts forward a new strategy for user interface development. Quoting Vijay [Mac96], "*user Interfaces are considered as one of the six core fields of Computer Science and are regarded as the most critical area for organizational impact*". It should be experimented the application of formal methods to the user interfaces development process, trying to give rigor or semantics to such a process not yet sufficiently certified.

This document is organized in two logical parts, preceded by this Chapter 1 - *Introduction*, and followed by several Appendices.

In the first part, constituted by Chapter 2 - *Research Foundations*, we introduce concepts and the most significant developments in areas related to the one of this work. We also describe related work upon which this dissertation is built, review the specification, design and programming methods for describing user interfaces, existent formalisms and models, and give an overview of the existent and new technologies, including XML markup languages, which are now emerging for user interfaces. A brief presentation of tools and dedicated frameworks is also included. Here must be underlined also the description, on the section *Data Description and Manipulation*, of the importance of Markup Languages on this process, mainly represented by *XML* markup language and its mapping scenarios.

The Second part, constituted by the remaining chapters, is reserved to the presentation of all contributions of this work, including the application of formal specifications and the process adopted for reverse engineering. Chapter 3 - *Markup Languages for Interfaces description*, presents the most significant markup languages to describe interfaces, mainly supported by *XML*, namely *XIML*, *XUL* and *UIML*, their syntax and semantic principles, scope and evaluation.

Chapter 4 - *UIML Formal Specification*, presents the *VDM-SL* specification of *UIML* markup language as well as all processes involved, namely the *"Transcoding"* process of *UIML* to *VDM* using a *XML StyleSheet*; its reverse *Validation* process, where *UIML* can be generated from VDM-SL; finally the abstraction obtained over initial *VDM-SL* specification and performed formal calculus.

Chapter 5 - *Case study: Table IO* presents the result of specifying formally, using the *VDM* specification achieved on Chapter 4, a Table, a particular user interface visual component. It also specifies in *VDM* several *OLAP* operations and demonstrates its application to this graphical object.

Chapter 6 - *Prototype and Supporting Tools* presents the prototype which animates the table OLAP features. It describes also all tools developed to manipulate the resulting specifications.

Chapter 7 - *Conclusions and Future work* presents the main conclusions concerning the obtained results and contributions, problems, advantages and perspectives for future work.

Following Chapter 7 comes the *Bibliographic* references used on this research and then *Appendices* which complement this work with other information not included in the main document body. It includes the *VDM* notation, the *UIML* Document Type Definition, sources of generated code and some examples used on this work.

To better analyze this research work, all results use the same practical test case as reference. It is a Stack implementation, with frontend depicted in figures 1.7 and 1.8.

1.6 Document support

This document was written in LaTeX [Lam94, Byn98, AAN03, Vic01], because of its accepted capacity to deal with large scientific documents. All *VDM-SL* source code

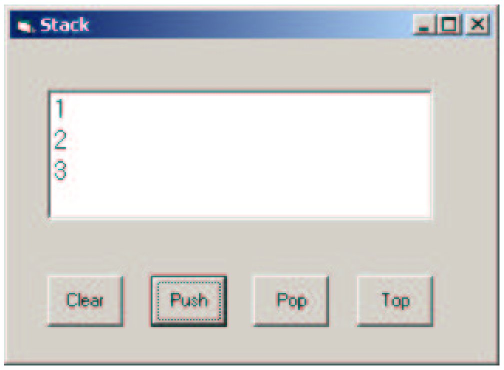


Figure 1.7: *Stack implemented in Visual Basic*

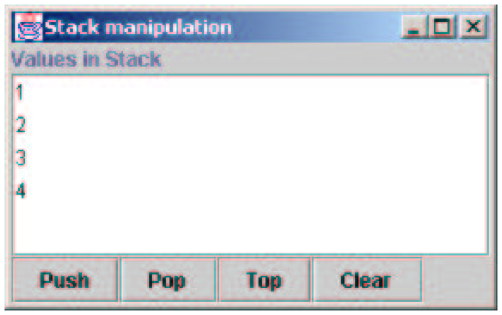


Figure 1.8: UIML *stack rendered to Java*

referred in this document, were formatted with styles of *evdm*, a LaTeX Style [Oli02]. There is also a CD-ROM as complement of this work, which has all the resultant material used in this investigation, including source code and consulted bibliography.

Chapter 2

Research Foundations

2.1 The context

Who gets faster the correct information and uses it properly, will leave winner
Don Keough[1]

The Software Engineering of today places emphasis on the need to systematize and to structure the processes of software development. Adopting more accurate methods (often identified as formal) or more or less structured processes, it is natural to find difficulties in the interpretation of the real need to transmit and to coordinate what must be executed, satisfying the expectations promised by the solution.

This can be compared with the scenario of a new house construction where, even if the projected architecture represents all the things to be built, it can only be certified at the end of the work. The same happens with our expectations in software design!

Referring essentially to the most recent terminology and concerns of the current software development techniques, we can list the following items:

- Fusion and Separation of concerns
- Temporality on restricting interactions
- *MultiModal* Interfaces [LFJ95, Ovi99]
- Contents security
- Data hierarchies
- Hypermedia contents
- Portability for multiple devices and platforms
- Configured Visual Components
- Client/Server and X-Internet[2]

[1] President of American Coca-Cola.

[2] X Internet - The X Internet pretends to be the future of applications, combining the advantages of centralized application deployment with the functional richness of locally installed software

- Web Services
- Portals
- Inter-operability and B2B
- Legacy Systems

This work, being specially focused with data Visualization in application interfaces, it will give priority to the analysis of the existing technologies in the area. In [Pha00], *Constantinos* summarizes the main steps which one can identify in this process and, as the *Seehein model* [Pfa85] advocates, the importance on maintaining a real separation between the presentation layer and the remaining application layers.

Amongst the vast literature on *HCI - Human Computer Interaction* technology, published in the specialty journal - *e.g. Interactions*, several annual conferences - *ACM*[3] *SIGGRAPH Symp.*, *Human Factors in Computing Systems*, etc., the work of *Martins* [Mar95], *Vijay* [Mac96], *Myers* [MHP00] and *Constantinos* [Pha00] is significant.

2.2 User Interface Properties

"The User Interface (UI) is that part of a computer program that handles output to the display and input from the user. The rest of the program is usually called the application" [Mye95].

The complexity of this theme can be justified in many ways. Foremost among them is the difficulty in perfectly understanding tasks and users. One of the recommended solutions for this complex problems is the *interactive design*, although, this process still is, by itself, complex and prone to errors. It could be long and consequently expensive and difficult to identify the correct end of the interaction process. Next we present some desirable user interface [Mac96, Gee00] properties:

- *Functionality*. This refers to $what$ an interface must perform. It must be defined before design and implementation.
- *Usability*. This deals with how good an interface is in satisfying its functionality, like promptness, ergonomic layout, performance, etc.[McE04].
- *Isolation from Application*. This is where the application must be isolated from the program user interface - *Seeheim Model [Pfa85]* - maintaining the inter-operability between the parts.
- *Adaptiveness, Ability*. Respond to different user profiles or program contexts,like the recent *Multimedia Skins* (*http://cs-skins.net*) or CSS[4] [Gee00] .
- *Consistency*. Which facilitates the transfer of skills from one system to another.

[3]http://www.acm.org

[4]CSS - Cascading Style Sheets (*http://http://www.w3.org/Style/CSS/*)

- *Standard*. To assure consistency and portability.

User interfaces and almost everything around them have been called different names over the years. From *UIMS, to Toolkits, User Interface Development Environments, Interface Builder Tools*, etc., clearly merging tools with principles and concepts [Mye96, Cyp93].

The authors of [JBK89] have also explored some system requirements for user interface development. They have concluded that most of the existing systems fulfill only some of those requirements. In Figure 2.1 (adapted from [JBK89]) we try to summarize the components of user interfaces. Although a slightly dated perspective (1989), we will see later on that it is still actual.

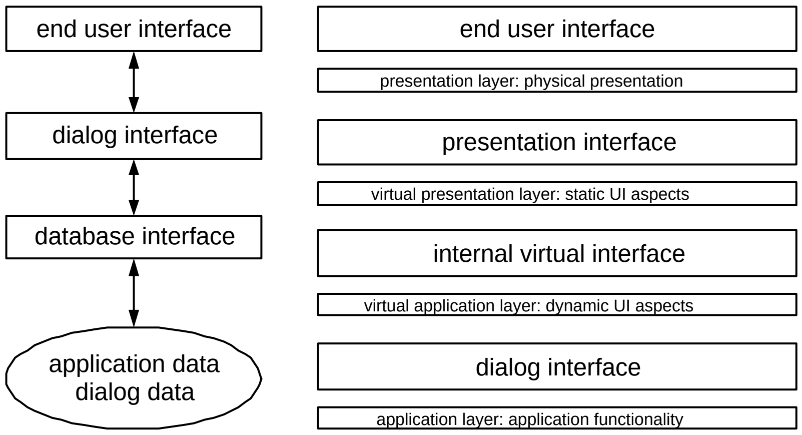

end user interface
dialog interface
database interface
application data
dialog data
end user interface
presentation layer: physical presentation
presentation interface
virtual presentation layer: static UI aspects
internal virtual interface
virtual application layer: dynamic UI aspects
dialog interface
application layer: application functionality

Figure 2.1: *The basic software architecture and the corresponding layered model for Human-Computer Interaction (adapted from [JBK89])*

As we can see in *Seeheim Model* [Pfa85] and in [Mye96, MB86], and later on in recent architecture models (such as *N-Tier* [Mic04b]), there are evident concerns in separating responsibilities between interfaces and the rest of the application.

The functionalities are structured in layers as follows [Sch01]:

- *Presentation layer*: output to the screen; handling of input from the user; toolkit functionalities
- *Virtual presentation layer*: separation of input and output from the dialog; definition of logical devices and virtual terminals; independent device presentation; handles all static aspects of the UI.
- *Virtual application layer*: this contains the semantics of the underlying application; dialog between the user and the application; handles all dynamic aspects of the UI.
- *Application layer*: main application's functionality

We will be back to this theme later in sections 2.3.3 and 2.6.2.

2.3 User Interface Models

Let us go back in time and recall the "black box" architecture of the first software solutions, where the user could only use it and never try to change anything. So, it was difficult to use the same application in a new situation if not impossible. In our days the existing solutions are "broken" in several units, and behave like modules or components allowing modular interfaces.

Several models exist (and continue to be created) that can also be applied to User Interfaces [Mar95, Pha00]. A recent one was explored in *Vadim* thesis [Vad96], experimenting automatic UI generation for applications. Also Markopoulos in [Pan97], describes important particularities for the main interface architectures and models. In the following sections, we will present the most significant ones, trying to demonstrate, by example, their particularities.

2.3.1 "Ancient" HCI models

Several models have been proposed to describe UI and human-computer interaction.

Perhaps the first and the most significant UI model was the one presented at the Seeheim Workshop in Berlin, in 1985 (thus the name *Seeheim Model*). Figure 2.2 depicts its main particularities, where we can see clearly the separation between *User interaction* and *Application logic*, using an UI split into three components. The *Application Interface Model*, which describes how and when methods in the application logic should be used (like semantic), the *Dialog control*, which is responsible for correctly sequencing the dialog events (like syntactic) and the *Presentation*, which should "show" the interface (like lexical) to the user [Pfa85]. This "Compiler" mentality, tried to obtain, from Application Interface Model, a rapid semantic feedback.

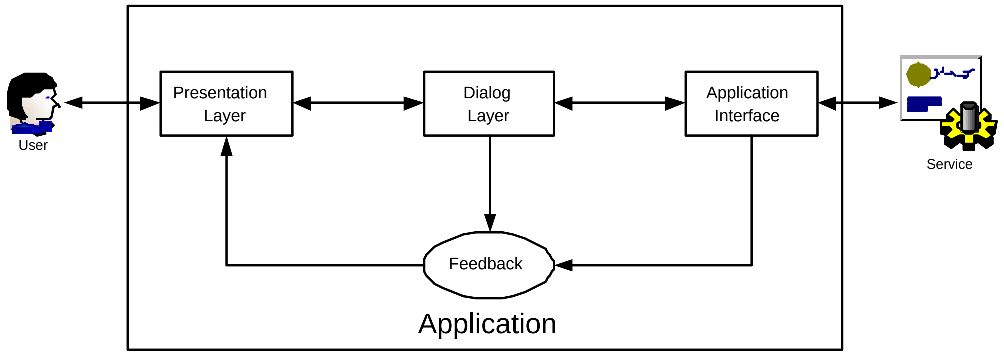


Figure 2.2: *Seeheim Model*

However, this logic and generic model was criticized and considered inadequate for current complex UI, because there are a lot of particularities that it does not take into account. Things like, for example, notations to use and information type to pass between components, are not present.

The same happened with the *Arch* model, specific for the run-time architecture of an interactive system, where, instead of examining the possibility to separate the presentation from other parts, this model analyzes the nature of data that is communicated between the user interface and the other UI components [Mig97]. Figure 2.3 depicts this model base architecture, where five components interact between them.

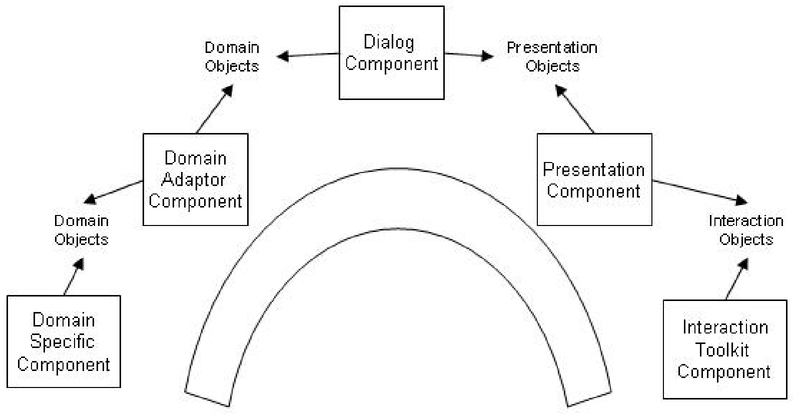


Figure 2.3: *Arch Model*

Those with an object oriented background, defend the idea of interfaces as collection of objects, with a mechanism to pass information between components and related mechanisms. The idea was to minimize the effects of changing technologies, as supported by object oriented characteristics [org92]. It works as a generalized *Arch model* but it does not support adaptive intelligent systems [Mig97].

The *Triple Agent Model of HCI*, with three components: Task Machine (application), User Discourse Machine (interface) and User, reflecting the human point of view over intended tasks, incorporating the strengths of previous models (integrated with *Arch model*), becomes adequate for adaptive intelligent systems, like those supporting IDSS[5], supporting their multimodal interaction and dynamic presentation functionality [Pue93].

Believing that modern interfaces tend to be collections of quasi-independent agents (components as buttons, grids, etc.) hierarchically organized, and that inheritance, composition and aggregation are possible, the *MVC - Model View Controller* [KP88] (Figure 2.4) and *PAC-Presentation-Abstraction-Control* model [Víc96] (Figure 2.5), show how a single application abstraction could be multiply presented.

The *MVC* model divides an interactive application into three components. The model contains the core functionality and data. Views display information to the user. Controllers handle user input. Views and controllers together comprise the user interface. A change-propagation mechanism ensures consistency between the user interface and the model.

The *PAC* model defines a structure for interactive software systems in the form of a hierarchy of cooperating agents. Every agent is responsible for a specific aspect of the application's functionality and consists of three components: presentation, abstraction, and control. This subdivision separates the human-computer interaction aspects of the agent from its functional core and its communication with other agents [Pue93].

Although not the most quoted in current UI development, these models represent

[5]IDSS - Intelligent Decision Support System

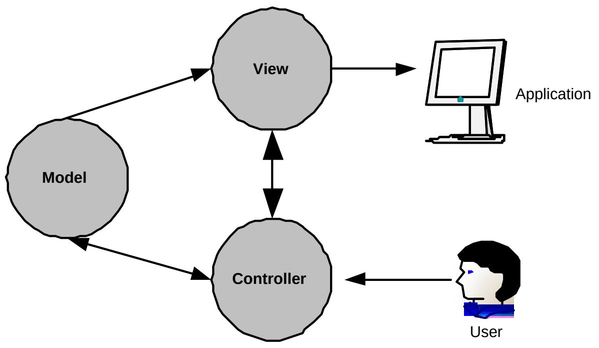


Figure 2.4: *MVC -Model View Controller*

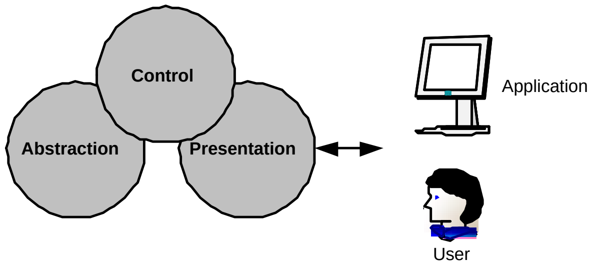


Figure 2.5: *PAC-Presentation-Abstraction-Control*

the main results of previous research and the cornerstone of current mentalities. We shall see the "heritage" of these principles int the (current) models that are analyzed next.

2.3.2 XForms Model

XForms is the next generation of Web Forms [Rec03f]. It is based on *XML* [Rec03c][6] and, being a device independent description, it is intended to work with a variety of standard or proprietary user interfaces.

There is a standard set of visual components, which can be used in other markup languages, such as XHTML [W3C03], allowing interface description and form events handling. The input data is, naturally, represented in XML.

In this model, the interface is organized into three layers: *presentation*, *logic*, and *data*. The *data layer* defines a data model for the forms (e.g., XML Schemas). The *logic layer* defines dependencies between fields. The *presentation layer* describes the recent interface and the mappings to different devices [Pha00].

The main goals of XForms, can be enumerated as follows:

- Support for handheld, television, and desktop browsers, plus printers and scanners;
- Richer user interface to meet the needs of business; consumer and device control

[6]Appendix B summarizes XML technology

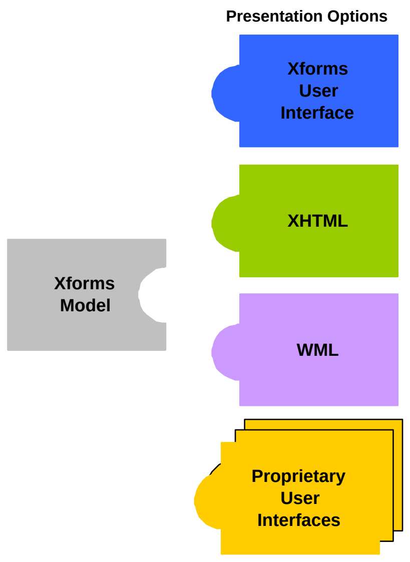


Figure 2.6: *XForms architecture (adapted from [Rec03f])*

applications;

- Decoupled data, logic and presentation;
- Improved internationalization;
- Support for structured data forms;
- Advanced forms logic;
- Multiple forms per page, and pages per form;
- Suspend and resume support;
- Seamless integration with other XML tag sets.

As we can see, XForms is mainly concerned with inter-operability features, integration needs and distributed cooperations. All these issues are associated with Web technology, going towards a standardization of using browsers as interface engines.

This model is particularly important for the work described in this thesis, as it provides a dialect to the widespread XML markup language used on user interface description. We will use XHTML to present our *VDM-SL* specification tests [IFA00c, Jon90].

2.3.3 The N-Tier Model

Although a recent model, the *N-Tier model* can also be considered in the context of the *Seeheim Model*. Their principles are not very innovative and the scope does not offer something particularly new, except for the fact of being suitable to Web technologies and web programming.

The main particularity of this model consists, in fact, on a clear separation of the application in several layers (or Tiers), as follows:

- *Presentation Tier*, typically a client web browser
- *Dynamic Presentation Logic Tier*, usually done in the Web server using many technologies (scripting, XML, *Lets*, etc.)
- *Business Logic Tier*, where all business objects and rules are implemented (using for instance Java Beans [Mic96]).
- *Data Access Tier*, which works as a *wrapper* around data repositories (relational databases, flat files, etc.)
- *Backend System Integration Layer* (often named *Data Tier*) which consists of a distributed set of relational databases, integrated with the middle tier using specific technologies (ex. JDBC [Mic03d]) as well as other legacy systems.

Figure 2.7 represents a distributed N-Tier model and current support technologies and applications.

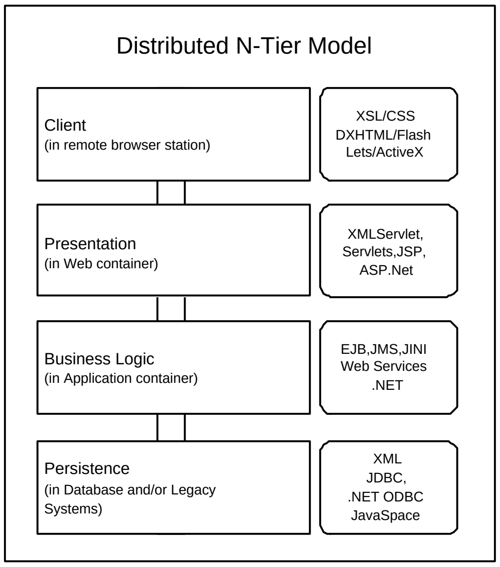


Figure 2.7: *N-Tier Model*

This model will be further explored in section 2.6.2, where we discuss the *N-Tier architecture* used in the *Microsoft .Net architecture*, in the J2EE Sun platform [Mic04c] and in the Intel e-Business [eBC01] platform.

2.3.4 MIM Model

MIM is an abbreviation of *Meta-Interface Model* which extends the level of abstraction of the *Slinky model* [org92]. *MIM* was created with the proper abstractions to describe interfaces that can map into multiple and distinct types of devices [Pha00].

We include the description of this model here, because it is the basis of the markup specification language used in this work to describe user interfaces: the *UIML* [Pha00, Vög03].

MIM divides the interface into three major components: *presentation*, *logic*, and *interface*, the last one being divided into *structure*, *style*, *content*, and *behavior*, as depicted in Figure 2.8.

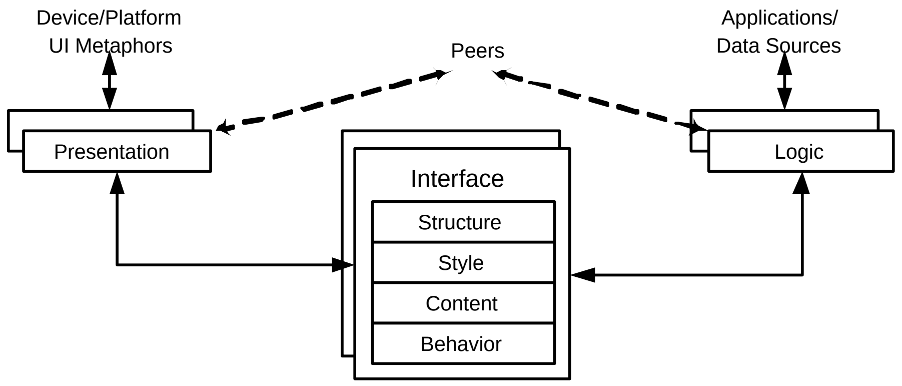


Figure 2.8: *Meta-Interface Model (adapted from [Pha00])*

The *logic* component provides a canonical way for the user interface to communicate with an application, while hiding information about the underlying protocols, data translation, method names, or location of the server machine.

The *presentation* component provides a canonical way for the user interface to render itself, while hiding information about the widgets and their properties and event handling.

The *interface* component describes the dialogue between the user and the application using a set of abstract part, event, and method calls, that are device and application independent.

Based upon the MIM model and able to describe generic interfaces that map into multiple distinct devices connected to a wide range of application technologies, is the *UIML* 2.0 [AH00], markup specification language for generic interfaces (see section 3.4 for a *UIML* description, providing more detailed information on this matter).

2.3.5 Other research in HCI modelling

HCI [Pan97] is a dynamic area of research. Recent studies describe important results which are briefly reviewed below.

[Reh01a] revealed the fact that a lot of these applications violate basic HCI guidelines such as Norman's design principles [Pop01]. More specifically, developers often leave the user with too little control, do not provide appropriate feedback about what the system is doing and fail to show appropriate constraints to the user [KR02, Nun01].

In [Reh01b], Kasim argues that, founded on the principle that "every object has an output", there is the continued possibility to work on new user interfaces models, connecting the real to the virtual world with a *VDU - Visual Display Unit*.

Another interesting point of view was presented in [JFMdM92], where Baar shows the importance of “coupling” application design and user interface design. The main argument is based on spending less time and effort on the development process, avoiding the risk of building similar but not identical specifications during the development process. In his opinion, the data model and user interface design may have a lot in common, including objects, actions and attributes (and quite often, rules).

Recently, Geert in [Gee00] explored some formal modelling techniques in Human-Computer Interaction, trying to evaluate these techniques according to the main UI principles: completeness, applicability, validity and usability.

A recurrent theme is the possibility of automatically generating some user interface parts (like direct forms) related to some database relations. There are already some “engines” which work as “extractors” to some different representations (for example XML) of all data model information, notably the *Database-to-XML mapper* from Altova (*http://www.altova.com*).

Not completely disagreeing with these perspectives, we hope to contribute within this work area to clarify some nuances and give some input to this cycle of continuing development.

Trying to focus on the main goals of this work, we will approach the three main phases of the user interfaces development process for computer applications, involving models structured or not:

- *The Specification*, which should provide the definition of what one intends to do
- *The Design*, which should provide the representation of intended results
- *The Interface Programming* which should provide the execution of the final work, the Interface.

2.4 User Interfaces Analysis and Specification

”User Interfaces are considered as one of the six core fields of Computer Science and are regarded as the most critical area for organizational impact” [Mac96]

Any programmer does, even as a reflex of a memory exercise, a specification of what he intends to execute. So, to specify can simply mean to opt.

Supported in more or less certified methods, rebounding or not the user interaction during analysis, the verification or not of application interfaces, often referred to as Legacy Systems, it will result in something that will support later phases of the development process. This phase is usually called *Specification*, *Prototyping*, *SRS - Sheet of Requirements Specification*, etc.

Problems arise in this initial phase because programmers have to cope with the complete definition of a final application, including user interfaces and everything to support it (e.g. storage information, integration rules, etc.). Any rigidness applied to these processes can be harmful. Even more, development teams are often constituted by only one person.

References [Mye96, Mac96], describe several options to archive UI specifications. Here we refer two of them which are actually very common and essentially related with the main goal of this work: *Application Frameworks* and *Interactive Specification*.

- *Application Frameworks*, like the older X-Windows API or the recent *Visual Studio .Net* [Mic04b] and *J2EE* [Mic04c], allow programmers to develop UI by abstracting from the underlying platform. This means to work on program interfaces (and other parts) and have the same aspect in multiple platforms, allowing some customization features.

- *Interactive Specifications*, often called *Direct Manipulation* [Shn97] (we will go back to this later on section 2.4.2), allow programmers to develop UI by manipulating objects on screen, using pointing devices. In this group we can include *prototyping tools*, like the recent visual tools (PowerDesigner from Sybase, CodeWarrior from MetroWorks, etc.), *Wizard tools*, etc., which allow assisted intervention on developing particular interfaces (based on *forms*, *grids*, etc.), like recent Visual Studio .Net in C# or Visual Basic .Net languages, and *Graphical Editors* which enable, in a way, to develop an interface from a library of existing components (like DLL, OCX, Java Beans, etc.) and, in another way, some important particular features used on debugging processes, data analysis, etc. We should refer here the Microsoft OLAP Cube component which offers, for instance, *Data Warehousing* features.

In the recent technology evolution, Remote Process and *Peer-to-Peer*[7] solutions are crucial elements in the development process. So a new complement to describe Interactive Specification needs to be considered. We also need to consider new important features for the programmers, allowing them to have their widgets library, some of those remote components which should "work" in his computer. We are talking about *Web Services*[8], for instance.

Proceeding with our review, it is now important to describe more accurately some common technologies or processes to carry out UI development. As mentioned when listing the main goals of this work, it is important to focus on formal methods and related technologies in this process, thus preparing for its exploration in the context of this work.

2.4.1 Adaptable Interfaces

We already know that the User Interface (UI) allows the user of the software system to interact with it. Being itself a kind of software system, with similar development processes as other application types, it has a set of attributes or requirements - called *Non Functional Requirements* (*NFR* in [Lec99, SC03], such as usability, reliability, simplicity, ergonomics, etc. [Mye95], very hard to measure or to classify). Adaptability is perhaps the most worrying property, due to the recent need to support several emerging new devices.

[7]http://www.openp2p.com
[8]http://www.w3.org/2002/ws/

Adaptation can be seen as the capacity of a system to react against changes in its environment, which, as in almost all software systems, will be possible only if its support architecture is also adaptable.

There are several studies in this area and some application solutions which deal with this. We focus our attention on two of them. The first one is the *XXL/XIBuild* [Lec96], an interactive Interface Builder based on Visual Programming. It is supported by a particular specification language and allows for automated building of a graphical user interface, easily converted to C/C++ code. It can represent all the interface in text view (source code), graph view (abstract iconic in Figure 2.9) or widget view (graphical user interface), where all user interaction will take place. We are talking about *Motif* widgets[9] and all adjacent rules and conditions[10].

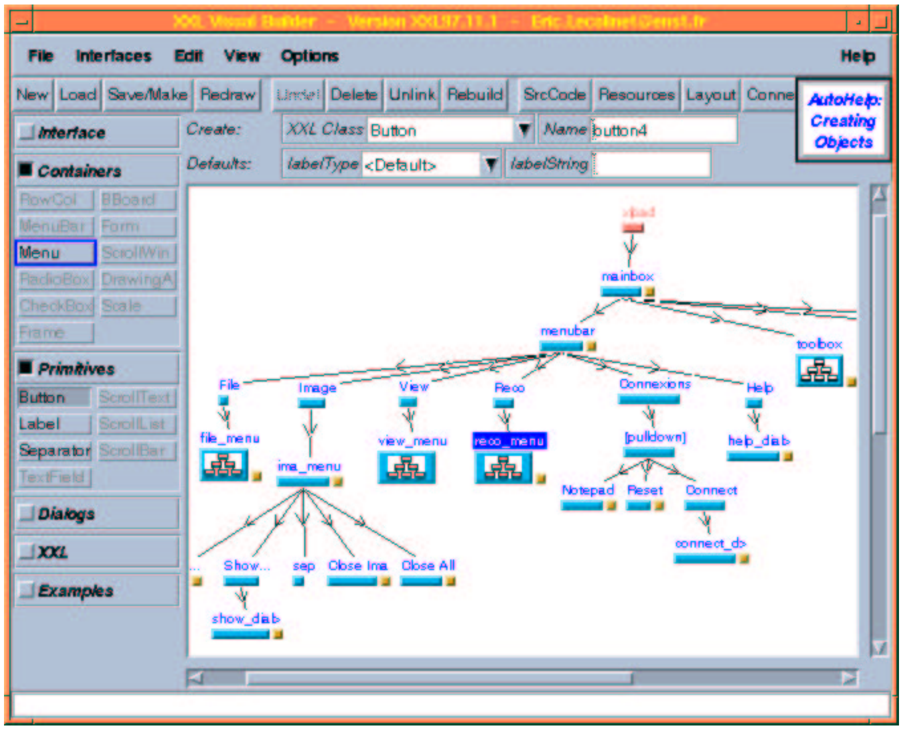


Figure 2.9: *XXL/XiBuilder Visual Builder Graph View*

The second one that deserves our attention is SA3 [SC03], where these concepts are well explored, having resulted in a definition of adaptability as described in the following paragraph:

"adaptation of a system (S) is caused by change (δ_E) from an old environment (E) to a new environment (E'), and results in a new system (S') that ideally meets the needs of its environment (E')" [SC03]

So, mathematically Adaptation is defined by the following finite function:

$$Adaptation: \ E \times E' \times S \rightarrow S', \ where \ meets(S', need(E'))$$

[9] http://www.motifdeveloper.com/widgets.html

[10] The most recent experiments with UMLi caused XXL to be almost forgotten

where the Figure 2.10 explains the relationship between the various symbols described above.

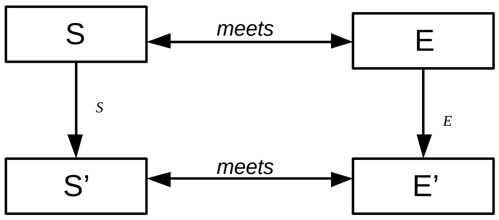


Figure 2.10: *Explanation of symbols in the definition of adaptation (adapted from [SC03])*

From our point of view this behavior of reacting to environment changes reflects the recent orientation of any one who worries with user interfaces. Putting a particular application (legacy or not), available on Web support, on mobile devices via WAP, etc., translates this kind of interaction over existent interfaces. We are also convinced that several adaptations on user interfaces of recent applications are made from the scratch or result from traditional *ad hoc* development, without any *model-based approaches* [FF93, BVE02, VBS01].

Later in this thesis we will show that these adaptations can be done in a systematic way following rigorous methods and mechanisms. We must always be sensitive to the fact that “to find the best component to use in a particular situation” is be a delicate process.

2.4.2 Formal Methods and Specification

“Formal techniques have a defined syntax and semantics and therefore ambiguity is completely eliminated” Andreas Gerstinger”[Ger01]

Among many definitions in the literature of a “formal method”, we chose the one provided by the largest professional organization promoting the use of formal methods - *FME - Formal Methods Europe* [FME03]:

“Formal Methods are mathematical approaches to software and system development which support the rigorous specification, design and verification of computer systems.”

2.4.2.1 Why are they not used more widely?

Andreas Gerstinger presents (in Chapter 2 of [Ger01]) an interesting overview about formal methods, considering their foundations, classification and application criteria, as well as the main particularities of common formal languages. This contributes to highlight their numerous potential benefits.

However, there are a lot of assumptions which “delay” the application of this kind of methods. Anthony Hall, in ”*Seven myths of Formal Methods*” [Hal90], on the application of a formal method (Z) to a large real problem, puts forward seven unjustified stereotypes and refutes each one of them:

1. Perfect software results from formal methods
2. Formal methods mean program proving
3. Formal methods can only be justified for safety-critical systems
4. Formal methods are for mathematicians
5. Formal methods increase development costs
6. Clients cannot understand formal specifications
7. Formal methods have only been used for trivial systems.

On the one hand, Hall moderates overly optimistic “myths”, such as “formal methods can guarantee that programs are correct”. On the other hand, he argues that formal methods do not involve complex mathematics, do not increase the cost of development and are not incomprehensible to clients [Ste99].

Working on this cause, Brad Myers in [MHP00] comments on the unaccomplished application of formal languages to UI development tools. As many other (formal or informal) approaches like *Transition Diagrams* (mainly on *Structure Analyzes*), *Parsers* for *Context Free Grammars*, etc, they looked very promising at first, but they did not catch on for several reasons. Most of them are pointed out as a topic for future work.

There seems to be a lost opportunity, because the excellent work on using formal languages for *dialog* based interfaces, was surmounted by interface styles based on *Direct Manipulation* [Shn97], which have emerged meanwhile. Another reason could be related to the difficulty to represent them using unordered operations (like *Set* operations), by contrast, they can easily be represented using sequences. So, the interfaces should naturally appear with a rigid sequence of operations.

And last, the need to learn a new programming paradigm, putting the programmer against the need to understand new programming concepts, is a hard obstacle to overcome.

2.4.2.2 Archetype: the necessary proof?

[MO85, Mar95] present an important step in reinforcing the application of formal methods in software development. These works appeared on technological scenarios where more rigorous mechanisms on defining entities, processes and their interaction in graphical systems (and, in consequence, in user interfaces) became an important demand.

They purport the idea of having, for each system, objects one has to work with, and a corresponding object for its visualization. These visualization objects are like semantic graphical representations [Mar95].

Object construction, a common operation in practically all interface “builders”, induces the creation of standard abstract objects which could represent the same and consequently be transformed in concrete representations, following instantiation processes, like those existing for class attributes, in the object oriented paradigm. These abstract objects are called *archetypes*. These archetypes are formal and abstract representations of geometrical objects.

These construction processes are supported by *mathematic algebra* operators, principles and rules, as well as by inference mechanisms.

We can see this scenario as a presentation model where two main categories of objects are defined. One, the *Concrete Interaction Object - CIO*, and the other, *Abstract Interaction Object - AIO*. CIO represents a real object (e.g., list box, button, etc.), being simple or composed by other simple objects (e.g., a list box). AIO consists in an abstract representation of a particular CIO, from its presentation and behavior viewpoints, independent from any support platform.

Later on, in section 5 of this document, will be presented an abstract representation (in *VDM-SL*) of the *table* concrete object and applied some strategies for mapping that are innate to the reverse engineering process. The sections which follow describe some of these mappings.

2.4.2.3 Formal Specification

> *"As software continues to be used to control critical systems, it is increasingly likely that software will be considered critical in order to achieve mission goals. Consequently, ensuring the correctness of software becomes paramount in order to avoid catastrophic failures like that of the Ariane 5" [GC99b]*

Formal specification is a requirement specification written in a formal notation. Being also applicable to specify interfaces for software applications, it is increasingly a concern for many programmers and designers of software solutions (also called software architects), as well as a need for whoever wants profitability and efficiency in the development processes.

As advocated by FME, formal methods are techniques that embody the use of formal specification languages which have a well defined syntax and semantics, founded on *mathematics*. On the other hand, formal methods use calculation rules to analyze a specification, to validate its consistency and correctness. The mathematical foundations of formal methods prepare the path to automation [GC99a]. Against this concept persists the idea that everything that has to do with mathematics is very "delaying" and "boring".

Due to the fact that several formal notations already exist it becomes difficult, for programmers, to decide which to use in a certain phase of the development. It is not assumed which universal formal notation must be used to specify all the different development phases, especially because, development groups being constituted by people with different technical qualifications, the option will be naturally conditioned.

Playing against preferences, particularities and advantages of known formal notations, the way to maintain the bridge between the "expected to get" and "what was really done”, seems to be to insure the interaction between notations.

This work resorts to a well-known and certified formal methodology *VDM-SL* (The Vienna Development Method, ISO/IEC 13817-1), which has been applied to several academic and industrial projects [IFA03].

2.4.2.4 Formal Reverse Engineering

Reverse or Inverse Engineering [MJS$^+$00], can be understood as an analysis and identification process of all the existent sections and components of a program (and not necessarily the entire program), as well as all interrelation types between them, working towards higher levels of abstraction. The scheme in Figure 2.11 depicts this process over user interfaces.

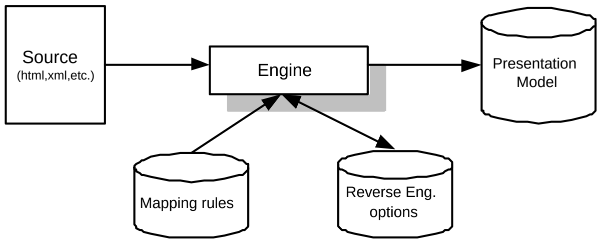


Figure 2.11: *User Interface Reverse Engineering*

Further to this, the *Reengineering* process works towards a different implementation, after the system analysis, perception, knowledge and modification process [GC99a].

Compared with conventional *Engineering*, usually called *Forward Engineering* [MJS$^+$00, oRE03] (where one tries to move from high-level abstractions, logical and implementation-independent designs, to the physical implementation of the system - Figure 2.12), and applying it to *Software Engineering* (where, starting from high abstraction levels and implementation models based on prototypes, one intends to get its real implementation), *Reverse Engineering* must appear supported by certificated description forms of what one intends to analyze.

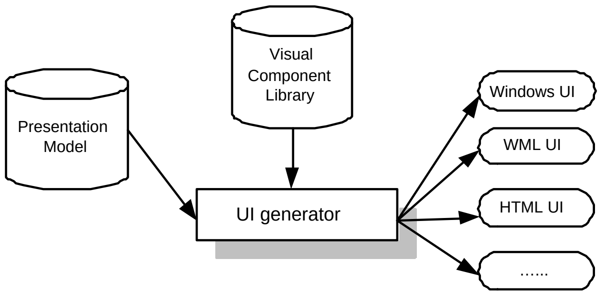


Figure 2.12: *User Interface Forward Engineering*

Let us consider the case of preparing a web site in order to make it accessible from the widest range of computing devices or platforms, including browsers, PDAs[11],

[11]Personal Digital Assistants

SmartDevices, Mobile Phones, etc. One must prepare, not only the information, but also the application front-end or user interface to those distinct devices. Exercises of transforming HTML to WML, VoiceXml, etc. and viceversa, should become usual and necessary.

Moving from one context to another could entail changes on one single or multiple UI, by performing it by *generation*, *conversion* or even *adaptation* (as presented in section 2.4.1) [VBS01]. We will see later (section 3.4) that UIML, after describing the presentation and dialog specification (in a XML-based vocabulary), can be "converted" to different platforms (Windows applications, Telephones, Palm Pilots, etc.), using render code for Java, VoiceXML, WML, etc.

These transformations should be oriented by principles of *consistency vs variability*, where one tries to get an acceptable compromise to respect the constraints of the context and the consistency of UI; *design vs run-time*, allowing the transformation during the development or during the execution; and at last, *partial vs total* transformations, where one tries to support only some parts of the next context [VBS01].

Focusing on formal methods application to Reverse Engineering, we can identify two main phases in its process [GC93]:

- *Phase 1* - Information extraction
- *Phase 2* - Abstract representation of what has been extracted in phase 1

There are opinions which defend the organization of this process in more and different phases. However, we think that they can be modelled by this approximation.

All reasoning applied in this work is based on a Reverse Engineering process where formal methods and mathematical principles will be used towards a rigorous semantic approach to object visualization and manipulation in a User Interface development process.

2.4.3 VDM-SL

This section provides a short overview of *VDM-SL*, abbreviation of *Vienna Development Method Specification Language*.

2.4.3.1 Presentation

> *"It is particularly important to realize that specifying a model in any formal notation does not bring with it the magic confirmation that the model is correct. The algebra may be correct - but it may specify something quite different from that originally intended"[Hop01]*

VDM-SL, is a general purpose formal specification language (as is Z [Bow96] or RSL [UI97]), working with high levels of abstraction and rigour, created at IBM Vienna Research Laboratories - while formally defining the PL/I programming language [IBM03].

By supporting abstraction of data and functionality, *VDM-SL* was fully formal (syntax and semantic) ISO standardized in 1996 [ISO96]. It offers a standard module approach with imports, exports, parameterization and instantiation features.

Since its creation, *VDM-SL* has been refined and enhanced with a formal algebraic model, having actually, like other methods (formal or not) its advantages and disadvantages.

If we look at the actual programming languages scenario, there are not so many formal languages and many of them are specific for a particular area. Having as initial goal the specification of software systems, due to its expressiveness, *VDM-SL* has been used to model a large number of other kinds of systems (not only computer systems) many of them classified as critical systems. The FME homepage[12] contains an application database with the details of application of VDM.

VDM is supported by one of the best tool suites - at time of writing - among formal methods[13]. It is the *IFAD VDM-SL Toolbox* [IFA00a], which has been used in all tests of this thesis.

As any traditional programming language and formal method notation, a *VDM-SL* specification is basically a collection of type and function definitions. As an overview of *VDM-SL*, we are going to describe their main characteristics.

2.4.3.2 VDM-SL characterization

A *VDM-SL* model consists of a *static* and a *dynamic* part. The *static part* contains the *data* and *type* definitions, as well as an optional *state* of the system. The *dynamic part* consists of *functions* which are either explicitly or implicitly defined, and *operations* which can modify the *state*. Figure 2.13 depicts a typical *VDM-SL* module structure.

VDM-SL works with not so many basic data types. More precisely, numbers (*nat*, *nat1*, *int*, *real*), boolean (*bool*), characters (*char*) and tokens (*token*). Constant values are written as identifiers inside angle brackets (e.g. <Black>)

To define a type it is necessary a type symbol and a set of operators to manipulate their values (mainly writing). Each operator must have a signature, listing its input and output types. This operator can be *total* (applied to any domain element) or *partial* (otherwise). A typical operator definition is as follows:

$$op : T_1 \times T_2 \times \cdots \times T_n \rightarrow R$$

Appendix A describes all *VDM-SL* standard data types and associated operators.

There are also type constructors which allow for compound types. These constructors support finite *sets* and *sequences*, *mappings*, *records* and *optional* values. Table 2.1 presents these constructors and respective *VDM-SL* notation.

[12]http://www.fmeurope.org

[13]Others tools are available, such as ”VDM Through Pictures”, from Oxford Science Park

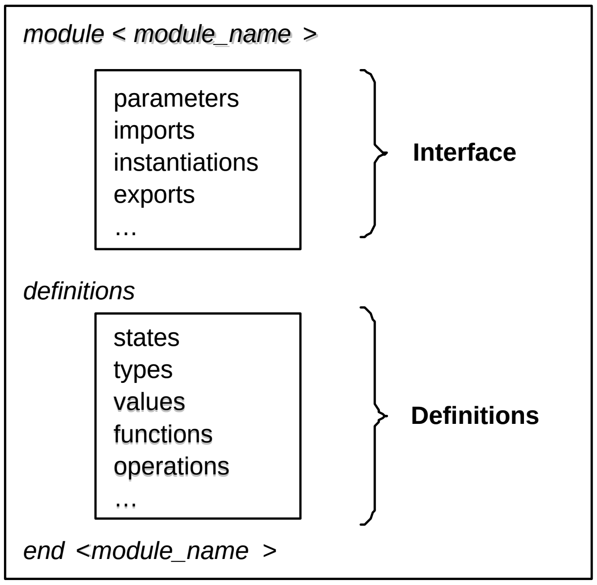


Figure 2.13: VDM-SL *module structure*

Constructor	Description
$set\ of\ \_$	Finite sets
$seqof\ \_$	Finite sequences
$map\ \_\ to\ \_$	Finite mappings
$\_ \mid \_$	Type Union
$[\ \_\ ]$	Optional Type
:: notation	Record Types

Table 2.1: *VDMl-SL constructors*

We will provide several examples of these compound types in our *UIML* specification (described in Chapter 4), as it is the case of *String* and *ID*, defined as follows:

$$String = Seq\ of\ char$$

$$ID :: String;$$

A very important concept in *VDM-SL* is that of a type invariant. Each type definition can be equipped with a boolean expression called *invariant* which constrains the inhabitants of a particular type. For instance, we can define a type *PhoneNumber* as being a 9 digits telephone number. Its definition could be as follows,

```
Digit = char
inv d == d in set {'0','1','2','3','4','5','6','7','8','9'};

PhoneNumber = seq of Digit
inv pn == len pn=9;
```

where the two invariants (*inv*) ensure the length of a phone number composed only by digits. Although these are basic invariants, the ability to define specific invariants is one of the most useful features of *VDM-SL*.

As referred before, there could be also a *state* in a *VDM-SL* specification, which is equivalent to a set of global variables. This state can then be manipulated with specific operations. The following *VDM-SL* code depicts one example of this:

```
state Contact of
    Addresses: map Person to set of Address
end;
```

The dynamic part of a model contains the functions and operations[14] of the specification. The next *VDM-SL* excerpt below describes the implicit definition of the *sqrt* mathematical function with precondition *pre* and postconditon (*post*) definition:

```
sqrt(x: real) r: real
    pre x>=0
    post x = r*r;
```

The precondition limits the input values *x* to positive or zero numbers, and the postcondition states that *sqrt* is the converse of square.

In our specification (to be presented in Chapter 4) we have many explicit functions definitions, in which the way calculations are performed, must be described. The next excerpt of *VDM-SL* code defines the function *length* (from page 148):

```
length : (String |  ID ) -> nat
length(x) ==    cases x:
                    mk_ID(s)->len(s),
                others -> len(x)
                end;
```

There are also important considerations about expressions (for functions) and statements (for operations) which contribute to the high level of expressiveness of the language. A detailed presentation of them can be found in the references [IFA00c, Hop01].

Being not object oriented[15], *VDM-SL* behaves as a model-based method more suited for sequential applications, where it is not possible to implement inheritance. This normally entails a larger number of code lines, as is clear in our specification of *UIML*, whenever we define a new method.

2.5 User Interface Design

One must understand User Interface (UI) design as a process which can describe, graphically or under any textual notation, the particularities and features for the UI. *Vijay* [Mac96] forwards three possible models for the User Interface design:

[14]These differ from functions because of the possibility of changing the global state

[15]VDM++ [IFA00b], is an object oriented extension of VDM, not yet standardized.

- *Demonstrational Interfaces*, where the programmer develops some typical examples of what is intended, continuing the development process with or without a minimal interaction with the user [MCM+91]. This model results from surmounting some difficulties found on the previous *DMI*[16] model, where the end user can manipulate objects directly on the screen, using devices like mouse or keyboard. We can see a simple example of this in *macro* definitions on some well known text editors (like Emacs or vim). Once a *macro* is defined, it can be used repeatedly.

- *Intelligent Interfaces*, where appropriate *artificial intelligence heuristics* are applied to support end user decisions. We can have two possible scenarios: either it accelerates the process, facilitating the selection or, being the wrong option, the process will be delayed or hindered. An example of this kind of interfaces exists in recent Office tools, as, for instance, is the case of Microsoft Word and Excel, where, after writing a text, with some kind of spelling mistake, it can be automatically corrected. The success obviously depends on the correction being valid for the instance at hand.

- *Distributed and Collaborative Interfaces*, (often called *Cooperative*), represent the support to several processes and users. They result, fundamentally, from the emergence of computer networks, the availability of cooperative work and consequent Client-Server applications on a distributed processing environment with multiple kinds of interface, involving distributed components. These interfaces are very common in *Database Applications* where, the database being in a remote computer allows each network computer to access.

With the current scenario of rapid Web proliferation, including mobile devices PDAs, SmartDevices or Pocket PCs, wireless infrastructures, and so on, the ”old” Client-Server applications become crucial, supported by a distributed process, justifying the fusion of companies and their spaces and processes reorganization. We are dealing with a typical process of system integration, inherent to the *EAI*[17] and *IAI*[18] areas, where a possible integration is directly based on interfacing. For instance, it could be a rule, that all company users should use a browser as their main frontend application. In this context, *Nokia*, one of the most popular mobile devices producer, has issued an important message: “*By 2003, more than a billion worldwide mobile phone will be there*”.

Today we know that *Wireless technology*, *IP Telephony*[19], *Internet2*[20], *IUNET*[21], etc., work towards freeing users from physical or space limitations. We are in the infancy of the next generation of communications technology and the UI will be there too.

If we look to the technology state-of-the-art which supports current UI development, this particularity of a new supporting model for interfaces is even clearer. Let us

[16]DMI - Direct Manipulation Interfaces [Shn97]
[17]EAI - Enterprise Application Integration [Lin00]
[18]IAI - Inter-Enterprise Application Integration
[19]*http://www.internet-telephony.org*
[20]*http://www.internet2.edu*
[21]*http://www.iunet.net*

address this topic next.

Having analyzed other alternatives, e.g. *CASE* tools, Structure Analysis with their *DFD- Data Flow Diagrams* and *STD - State Transaction Diagrams*, and even *UML*, we are going to describe a small part of the most significant and interesting languages that help in designing user interfaces. This is the *UMLi* [PP00], a specialization of *UML - Unified Modelling Language* (*http://www.omg.org*). This object oriented methodology, dealing graphically with objects and their relations, can be compared to our methodology to describe interfaces, using markup terminology. This will be explored when we present *UIML*, later in section 3.4.

2.5.1 UMLi

Because we are dealing with user interfaces design, the *UML* [Gro03c], the basis of *UMLi - Unified Modelling Language for Interactive Applications* [dSP00], represents one of the most significant methodologies for software design using object oriented principles.

Although *UML* is a notation for creating software application designs in an object oriented manner, the resultant application models describe few aspects of user application interfaces. Thus, according to *Object Management Group* [OMG02], some difficulties can be identified in the use of UML for modelling interactive applications:

- no UML diagram provides a notion for containment and enactment of interaction objects (widgets);
- no UML diagram provides graphical identification for abstract roles that interaction objects can play in user interfaces (e.g., displaying information to users, receiving information from users, triggering actions, etc.);
- the modelling of some categories of behavior commonly observed in interactive systems (e.g., performing an activity in a repeatable way [repeatable behavior], performing a set of activities in any order assuring that each activity will be performed once [order independent behavior], etc.) is complex in behavioral models of UML (e.g., in sequence diagrams, collaboration diagrams and activity diagrams);
- the use of interaction objects to specify the dataflow between users and non-interaction (domain) objects is difficult to specify and visualize in UML diagrams;
- UML diagrams, in general, become complex even in modelling very simple user interfaces.

However, due to the great spread and acceptance of UML in the programmers community, it makes sense to improve it, so as to be prepared for model interfaces. This is *UMLi* rising [dSP03].

There is another *Model-based User Interface Development Environment*, constituting the *MB-UIDE* [Pin00] technologies group. This is considered a state-of-the-art approach to user interface development, which provides the capability to design and implement user interfaces in a systematic way. However, current *MB-UIDE* technology can only be used to design user interfaces and not applications.

So, *UMLi* aims to address the problem of designing and implementing user interfaces using a combination of *UML* and *MB-UIDE* techniques. Indeed, a *MB-UIDE* that supports user interface models designed with *UML* will provide the necessary integration between user interfaces and the underlying applications.

For instance, by having a single modelling notation, common structures and behaviors of user interfaces and their underlying applications can be shared at design time. Furthermore, user interface developers can benefit from designing user interfaces in a systematic, and almost standard way [PP00].

UMLi is a modelling language that extends UML providing the following additional facilities for user interface design:

- a new diagram for modelling UI presentations called user interface diagram.
- a new set of activity diagram constructors for modelling UI behavior.
- a tool, *ARGOi* [Gro03b], which supports UI presentations and respective behavior modelling.

Figure 2.14 depicts an example of a user interface diagram *ConnectUI* (Connect User Interface in figure), modelled using *UMLi*, where a user connects to the system by providing his/her login and password.

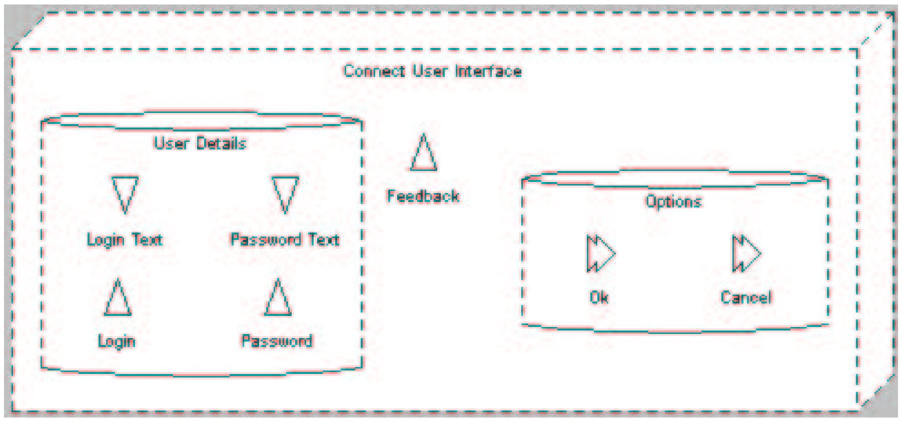


Figure 2.14: *ConnectUI - UMLi example of User Interface Diagram (generated using ARGOi version 0.8.0)*

In the diagram of this figure, $\bigtriangledown$ - (*Inputters*) are responsible for receiving information from users and $\bigtriangleup$ - (*Displayers*) are responsible for sending visual information to users.

User interfaces specified in user interface diagrams are conceptual models of UI presentations since they only specify abstract aspects of user interfaces that may be relevant to understand the UI. For example, the *ConnectUI* container in the figure 2.14 does not say anything about:

- layout (for instance, whether the *Login* displayer is going to be to the left or the right of the *Login Text*, or whether the *User Details* container is going to be at the top, at the left, at the right or at the bottom of the the *ConnectUI*);
- widget selection (for instance, if the *OK* action invoker is going to be a button, a list, a combo box, etc.)
- toolkits and programming language.

In spite off the considerable effort to understand UML and consequently UMLi (due to their large classes of diagrams and particular graphical notations on one hand, and its object oriented rigor to describe structure and behavior, on the other hand), it is accepted as an important alternative to achieve interface design.

2.6 User Interface Programming

Concerning interface design, it is hard to achieve correct forms for describing what one intends to develop [MHP00]. A possible way and perhaps the most advantageous is based on the higher-level tools currently available. This type of tools can provide an important support in the interface design phase, which facilitates the organization of the different components intended for its interface, also often facilitating the animation of its behavior and interaction (e.g. tools of the UIMS[22] group [Bra98, MHP00]).

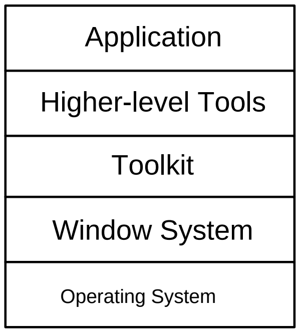


Figure 2.15: *Components of user interface software (adapted from [Mye96])*

Considering the architecture presented in [Mye96] for user interface software components (Figure 2.15 depicts its division into various layers), one concludes that, currently, an equivalent division is present in several user interface programming methodologies. In the following we will try to explain this statement.

2.6.1 UI Programming Methods

Low-level/High-level/Visual Programming . In the beginning of computing, assembly code or even ASCII code was the only way to get some graphical aspect on interfaces. This kind of programming is still there, mainly for handling specific

[22]User Interface Management System

purpose devices, where portability is not an important issue or even appropriate compilers are not available. Currently, there are a lot of frameworks or tools which provide programmers with excellent conditions to program, including user interfaces, debugging and test mechanisms. It is a more high-level work and the need of coding is often supported by direct manipulation of graphical objects on the screen. Quite often, they also work as prototyping tools and even interface builders. For instance the recent *Visual Studio .NET* [Mic04b] (the release of the current Visual Studio), the *Kylix* framework from Borland [Bor03], the QT from *TrollTech* [Tro03], the Tcl/tk from Tcl/Tk consortium [Con03a]. This is why this kind of tools is known as *IDE - Integrated Development Environment*.

Toolkit Programming . Uses object-oriented techniques for abstraction in user interfaces building [Pha00]. As in high-level programming, graphical component libraries are also available, allowing programmers to work independently of the underlined platform abstracting all low-level details. Several code generators are available and so, the programmer can spend more time in different issues, like code documentation or even usability. Using technical terminology, toolkits are the common *API - Application Programming Interfaces*, like *MFC - Microsoft Foundation Classes* or *JFC - Java Foundation Classes*.

Web Programming . Encompasses all programming efforts in the Internet over the Web. It is an important programming area and perhaps the most significant in the next years. As mentioned above, we are dealing with scenarios of enterprise integration, effort cooperation and distributed services. So,the remote access to data and processes, should be, once more, at the core of everything.

Because the Internet communication infrastructure or any other, should support all this, the technology needs to grow continuously, day after day, trying to handle new requests from users. So, there are (and there should always be) a lot of technologies to support this kind of programming, most often resulting in an *hybrid* style of programming [Pha00] where several different kinds of technology may coexist.

Technologies like *Java Applets/Servlets* [Mic03d], *Perl* [Per03] and *Python CGI* [Pyt03], *Scripting* with *Javascript* [Mic03f] or even *Windows Scripting* [VS03], *.Net ASP* or *JavaBeans* [Mic03d], *Embeb* code [Gro03a], *Webservices* [Rec03d], etc., are alternatives that programmers must select at all time. Recently a new technology has appeared, not specifically for programming but to give different features to the programmer: *Markup languages* [Luí03].

New types of applications should also appear, as it is the case of emerging *CMS - Content Management Systems* or even *Fuzzy Search Engines* such as *Google* [23], due to the current Web business concerns. Their *interfaces scalability* and *portability* will be a central question [Fre03].

End-user Programming . This is one of the most promising areas of programming. This statement can be argued whenever we look at the success of spreadsheets.

[23] *http://www.google.com*

The primary reason for this success is that end users can program it (using macros, formulas, etc.). However, this capacity is not present in all applications.

If we look at the current Web scenario, where everyone, without the need to learn so much, can write a web page, we are in the presence of multiple static information, where the consistency and up to date of the information is rather weak. This is caused by the pages not being dynamic and with autonomous behavior. The end user can not do it alone and will need the help of a specialist.

In the future, using scripting, CGI[24] languages like Perl [Per03], the possibility offered to have what the end user wants will be an important step [RSF97]. If the user works only with appropriate interfaces, with customizable capacity to go through its requirements, the complexity associated to the new technologies stop being a problem.

We can see all this by looking at the new mobile devices (like PDAs, SmartDevices, etc.), where display properties and communication capabilities disallow sophisticated user interfaces. Image and voice (pure multimedia data channels) will be the preferred user interfaces.

For instance, voice information does not have a "normal" user interface. *Voice Extensible Markup Language* - *VoiceXml* [Rec04b], only describes audio dialogs that feature synthesized speech.

Markup Languages . We have decided to single out markup languages because, being a significant area of programming concerns in our days, there is a lot to say about it. Markup languages appeared a "long time ago" [Luí03], with SGML [Rec01b] being one of the milestones in its progress.

Its original main goal was to describe (using markup words) and preserve data. Because there are precise rules based in text coding, this kind of languages provides high degrees of portability and consistency of data. A piece of text "goes" faster over the network than does a piece of binary code and, if some part of the information is damaged, the problem should not be so severe as in byte code.

All these factors explain their quick dissemination over the Web and acceptance by almost all type of Internet applications.

Since the Web deals with high capacity user interfaces, XML markup languages are also applied to describe them, allowing for the preservation of its portability.

Almost all new markup focus areas (there are a lot[25]) are applications of the *eXtensible Markup Language* -*XML* [Rec03c]. Following the W3C [Con03b] recommendation, only *XML* syntax rules are defined. The vocabulary used could be created by anyone. This enables the proliferation of several new *XML* specifications, for areas like *Healthcare*, *Insurance*, *EGovernment*, etc.

Markup descriptions can be easily ported to different platforms and are usually device-independent. This new specification, appeared with *XHTML* [W3C03],

[24]CGI - Common Gateway Interface [Dev03]
[25]*http://www.xml.org*

leading programmers to use markup only to describe content, having "visualization" information on separate files, called stylesheets [Rec03a, Rec01a]. All device-dependent information must reside in them.

This kind of formalism [W3C96], is an important step to a new generation of programming techniques. In this work we are going to lean over this and experiment the application of formal methods to specify markup languages.

Recent investments in large, expensive and complex solutions as *CRM* or *ERP* provide evidence of the direction followed by many companies. The most important points for any company that is client of these information systems, are the following:

- Users need fast access to summarized information (essential);
- Data should be supported by an architecture Warehouse (global vision);
- Good performance and maintenance (easiness and speed);
- Network support (distribution);
- Users need a natural vision of the data (perspective of the company);
- Analysis methods supporting forecasts;
- Multidimensional vision, including hierarchies;
- Easy access (e.g. spreadsheet).
- Logical access (following reasoning processes).

Everyday, new methodologies are being developed and tested with the intent of satisfying many of these needs. Terms like *OLAP*, *Fuzzy Logic*, *Data Mining*, *MDDB*, *DSS*, *CRM*, *ERP*, *WebServices*, etc., are common expressions or abbreviations, that represent different forms of information processing, encompassing store mechanisms, user interfaces features, etc.

It is difficult to find a complete description of all technologies and related paradigms that translate the whole panoply of options to follow. Especially because, many of the situations are nothing more than experiments or immature results from research projects. This stops them to be considered a viable alternative.

2.6.2 N-Layer architecture

This section could also be called *Layered Architecture*, *Layered Application* or *Multi-Level Architecture*, as mentioned in *Enterprise Patterns Definitions* [Mic03b]. In another way we can see that there is some confusion on the usage of terms *Tier* and *Layer*. So, *N-Tier* very often appears written as *N-Layer*. Following [Mic04a], a *layer* should represent the *Logical Application Structure* and *tier* the *Physical Distribution/Implementation* onto the server infrastructure, which means that, in the deployment process, application and infrastructure teams should map components to tiers.

If we concentrate on the analysis of methods for developing recent applications, we can clearly witness an "evolution", in programming languages, in the architectures or support models and even in the support tools to its development, predominantly based on a multi-level architecture, a version perhaps more consistent and structured of the program module concept. This can also be seen as a different perspective of the Object Oriented paradigm and the creation of *ComponentWare* [BRSV99, BRSV98] technology (explored in section 2.6.3.1).

In this structure, the basic concept related to the responsibility of "having to do something", should exist between each architecture layer, insuring their inter-operability. In essence, some mechanisms are implemented to guarantee the consistency among several levels.

The diagram of Figure 2.16 depicts a perspective of the evolution of the main architectures available.

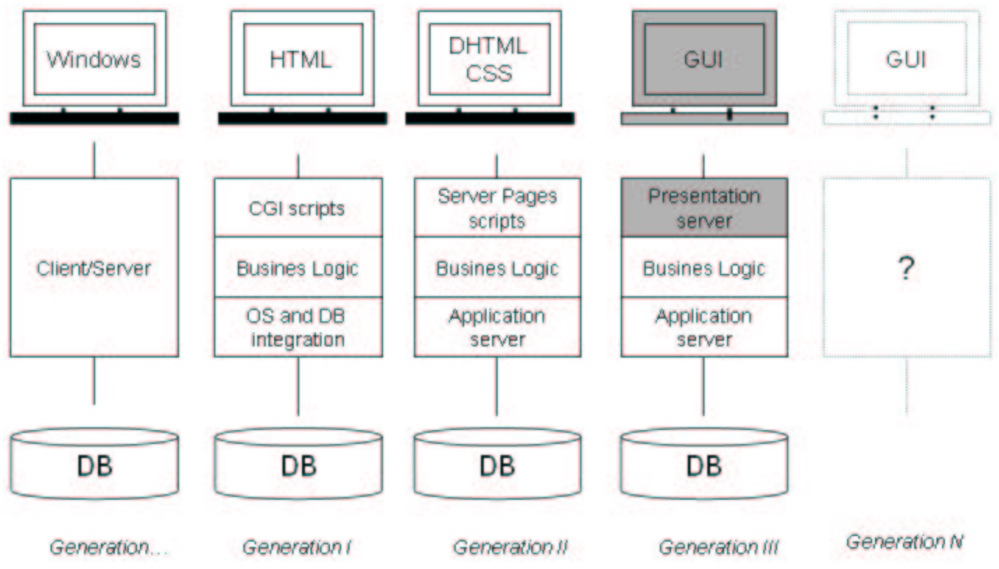


Figure 2.16: *Generations of Architectures*

In our days we are dealing with very complex distributed enterprise applications, where the *Web* executes a kernel job. Being mainly supported by *Web Services* [Rec03d], we can call it as *Layered Services Applications*. This notation is actually treated as a new generation of architectures, named *4-Layered Services Application*, where, in addition to the standard three layers, a set of foundation services (security, management and communications) is set up which all layers can use [Mic03b].

Trying to concentrate on the goals of this study, it is important to analyze the *Presentation Layer - PL*, present in recent architecture generations (identified as *Generation III* in 2.17).

Just as the *Application Server* layer bridges between the Repositories of information (Databases, Data Centers, etc) and the layer where business rules are defined, the - *Business Layer*, the *PL* bridges between the final user interface and the layer which applies these business rules.

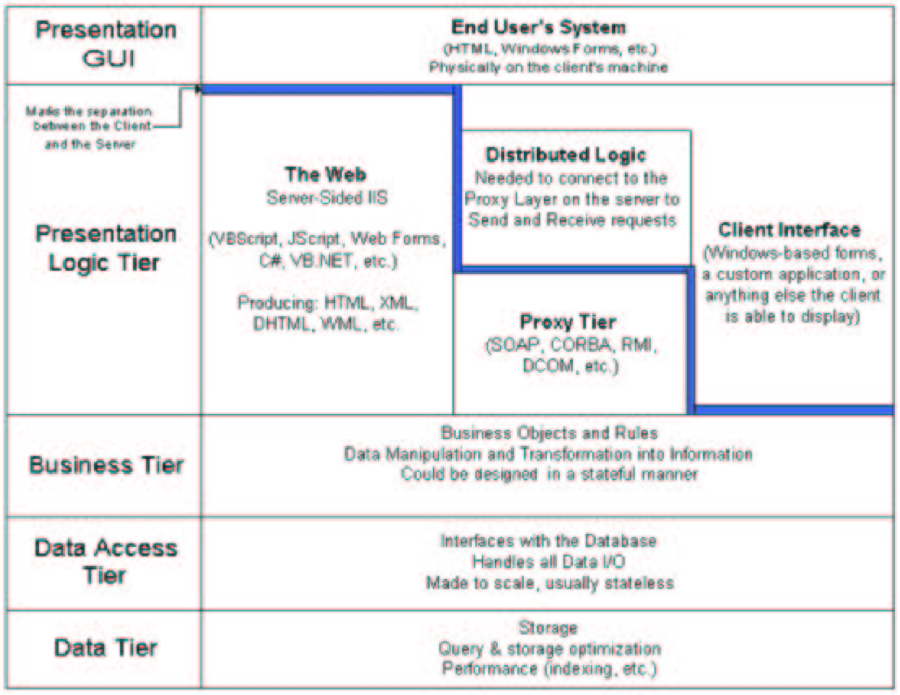

Presentation GUI
End User's System
(HTML, Windows Forms, etc.)
Physically on the client's machine
Marks the separation between the Client and the Server
Presentation Logic Tier
The Web
Server-Sided IIS
(VBScript, JScript, Web Forms, C#, VB.NET, etc.)
Producing: HTML, XML, DHTML, WML, etc.
Distributed Logic
Needed to connect to the Proxy Layer on the server to Send and Receive requests
Proxy Tier
(SOAP, CORBA, RMI, DCOM, etc.)
Client Interface
(Windows-based forms, a custom application, or anything else the client is able to display)
Business Tier
Business Objects and Rules
Data Manipulation and Transformation into Information
Could be designed in a stateful manner
Data Access Tier
Interfaces with the Database
Handles all Data I/O
Made to scale, usually stateless
Data Tier
Storage
Query & storage optimization
Performance (indexing, etc.)

Figure 2.17: *N-Tier Architecture*

Basically, this type of layers can be represented by three essential processes: *Display*, *Communications* and *Data Synchronization*.

The first, answers to the graphic elements and adjacent rules that constitute its interface; the second, answers to the communication between the application (in the server) and the user (the customer). The third, answers to the data update.

The identification of these entities is, in the general case, associated to methods or classes created for their specific operation.

Nowadays, the companies that work with the new Information Technology (IT) are forced to strongly invest in solutions able to answer to new demands, looking to process efficiency or usability issues. The form of the available information and the efficiency in extracting it from the system, is more and more regarded as critical.

The conventional methods of information management have become obsolete and insufficient, forcing some restructuring in storage forms or in their access and manipulation mechanisms. The Relational tables and respective reports are no longer preferred solutions.

2.6.3 Tools and Applications

2.6.3.1 Componentware

The idea of creating applications using dynamic integration of components, made and compiled separately, was demonstrated in the system Andrew [Pal98] of Information and Technology Center of Carnegie Mellon's University. Components, ie, units with processing and interaction (input/output) capacities, become the main building block for software construction. This is known as *Componentware* [BRS$^+$00, BRSV98].

In spite of the existence of several technical concepts and tools for component-oriented software engineering - *CBD - Component Based Software Development* [GeA02], whose rules were inherited from *Civil Architecture*, on one side, and the *Object Oriented* paradigm, on the other, the success of these sources could not be transferred for this process of software development. This is due, fundamentally to the lack of a solid methodology for componentware [BRSV99].

However, this is one of the areas of great expansion and interest in our days. One works harder in the aim of getting certified mechanisms for all intervenients in this process: the ones that develop (*Component Developers*) and those who use the components (*Component Users*).

In [BRSV99, BRSV98], Klaus Bergner suggests that the design of such a methodology, should be centered around some fundamental concepts:

- A *formal system model*, to define, without ambiguity, terms and concepts.
- A *technique to describe components*, according to that formal system model;
- A *model for the whole process*, for the users or for components developers, capable to organize the whole development process;
- *Support Tools*, for description or for supporting the development process.

If we port these concerns to the development of those components responsible for dealing with user interfaces, we will realize that they are also important. For example, a *graphic drawing* placed in a document should be controlled by a *drawing component* whose behavior should be independent of the used component to write the text of that document. This was the idea adopted by the main development entities, namely Microsoft (in OLE, OCX and ActiveX technologies) [Mic03c], Apple (in OpenDoc technology) [IAL03] and Sun (in JavaBeans technology) [Mic96].

The question arises: *How to modularize software in small components, continuing to guarantee the capacity of inter-operability and processing for who wants to use them?*

This work tries to explore the possibility of creating a kind of Visual Components Library (see Chapter 4, on page 69) where a new component could be obtained by composing several other known components. However, the experimentation comprises only the specification phase (see our VDM specifications of Chapter 4)

2.6.3.2 SDL

The emergence of *System Description Languages*, abbreviated to *SDL* [MT97], has extended visual programming with capabilities related to fault tolerance and feasibility.

Mainly used in hardware architectures description, *SDL* is a formal language that allows programmers to model *Finite States Machines* [AF00] using processes, signs and communication channels. Additionally, it is possible to enrich the models with data processing associating state transactions to the changes in variable values. This combines a drawing system with a visual model.

Basically these tools allow for the graphic design of systems, using sequences of visual components. This includes diagrams validation and code generation (usually C). It is possible to animate the designed system, detecting ambiguities and inconsistencies.

The tools based on *SDL* use *MSC* (abbreviation of *Message Sequence Charts*), to define the interactions between components. *MSC* defines sequences of messages exchanged among components, in a clear and readable form, helping in analyzing user requirements.

Summing up, these tools work at a higher level of abstraction and stimulate components reuse. The structure of *SDL* provides a mechanism for easy system documentation and, consequently, has advantages to the process of it maintenance.

We will see later, in our contribution, that using the VDM formal language to specify user interfaces (including methods and properties defined for operator composition), and using its integration with UML and using its Java code generation, we can also simulate a design in a rigorous way.

2.6.3.3 RAD

As seen before, visual programming is still one of the main lines of activity and investment of large institutions working in software development. To get a fast and efficient application development - *RAD*[26] [vBMvR] in the terminology of the software Engineering - it is important to reach and explore the state-of-the-art software development. Tools such as Visual Studio from Microsoft, Delphi from Borland, JEEE from SUN, etc., act in the development process of *GUIs* for applications, properly integrated with support for database manipulation. This kind of applications is often called IDE[27] (or simply *Frameworks*).

Below we present a list of common and important frameworks, tools and support technologies used to generate applications as well as user interfaces.

- Graphical User Interfaces Environment:

  QT/QSA - QT Script Application - *www.trolltech.com*

  GTK+ - Gnu Toolkit - *www.gtk.org*

  XWindows/Motif - *www.motifzone.com* [Her91]

  Delphi/Kylix - Borland for Linux - *www.borland.com.uk/kylix*

  .NET Framework - Microsoft .Net - *www.microsoft.com/net*

  Flash MX - Flash e XML - *www.macromedia.com/software/flash*

  Tcl/tk - *www.tcltk.com*

  Eclipse - *www.eclipse.org*

[26]RAD - Rapid Application Development

[27]IDE - Integrated Development Environment

- **Support technologies**

  FSM - Formal System Modeling

  LIM - Lotus Interactor Model

  ICO - Interactive Cooperative Objects Formalism (based on Petri Nets)

  FP - Functional Programming

  JML - Java Modelling Language - behavior interface specification

- **Markup Languages**

  XwingML - Java Swing XML (read zwing-M-L)

  BeanML - Bean Markup Language

  UIML - User Interface Markup Language (see section 3.4)

2.7 Data description and manipulation

The importance of this area for this work arises from the functionalities which are common to end users of this kind of applications.

As we saw before, in the *End-user programming* item, the decision maker (Managers, Executive operators, CEO, etc.) should have efficient tools to manage its information. This management could imply transformations on visual objects present in the user interface. In current terminology, we are talking about *interaction data* and the *adaptability* requirement analyzed in section 2.4.1.

So, we are going to explore the main features associated to interaction, data representation and manipulation, as a forecast of interface components, often called Interaction Objects - IO. As we know from class definitions (in the *Object Oriented Programming* paradigm [Bja91]), a class instance - *object*, inherits from its class, properties and methods to manipulate it. These concepts will be of significant interest in our reasoning.

We proceed to briefly addressing the *Multidimensional Data Models* [Fer00], important concept applied to *Data Mining*, *Multidimensional Analysis* - *MDA* [MK97, Bus02] and *Online Analytical Processing* - *OLAP* [SCJS01, Nig01] application areas, mainly wherever we try to manipulate interaction data.

2.7.1 Multidimensional Analysis

Multidimensional analysis [Nig01, HM01] offers to users greater levels of perspicacity in capturing the knowledge contained in databases and following the analyst mental model, by reducing confusion and limiting incorrect interpretation. Once a structure of multidimensional support is set up, the processing speed is superior to other structures of databases.

In the case of companies (the users), the information presents a multidimensional feature, strongly related and nested. For example, in forecasting the sales of a product in multiple areas it would not be correct if patterns of purchases of the past, as well as the new products foreseen for each area, were not considered.

This multidimensional information, many called as heterogeneous multidimensional hierarchies or schemas [HM01], resides in legacy systems, in spreadsheets, in relational databases, etc. In this way, they are necessary mechanisms that allow obtaining information from different sources and to offer the work group the capacity of observing them in a convenient perspective. With the intention of respecting existing solutions, the multidimensional analysis system needs to transform the structure of the incoming information of those sources.

In the case of text files (in *flat* format) data analysis requires the presence of meta-information (e.g. Data Dictionaries), so as to know the format and structure of the target data.

On the other hand, in the treatment of *Relational Databases - RDB*, its replication for a Multidimensional Database (hereafter abbreviated to MDDB) is not necessary (nor desirable).

Multidimensional data models categorize data as being either *facts*, which means data values and eventual attributes, or as being *dimensions*, that categorize the facts and are mostly textual [Fer00]. The model of a MDDB is an array of n-dimensions as depicted in Figure 2.18 (often called *Hypercube* or collection of related cubes [TC04]). Although the term “cube” implies three dimensions, a cube can have any number of dimensions.

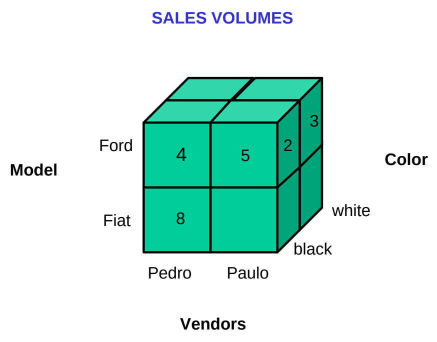


Figure 2.18: *HyperCube*

Each dimension is associated to a hierarchy of consolidation levels of data. For example, a dimension “*time*” can have a hierarchy with levels *day*, *month* or *year*. A dimension works as an index of identification of values in the array. Each array position, corresponding to the intersection of all dimensions, is a *cell*.

The variables (also designated by measures or metrics) in a multidimensional array correspond to the columns of the related tables. The values of each variable correspond to the values of the columns of the same tables, variables being virtual values calculated from values stored in those tables. However, the virtual variables are treated by the system, in same way as all other variables.

The set of meta-information provides the mapping of RDB columns and lines to multidimensional array dimensions and cells. It can also contain rules for data consolidation of hierarchies in each dimension.

For example, some changes in "time" dimension could be,

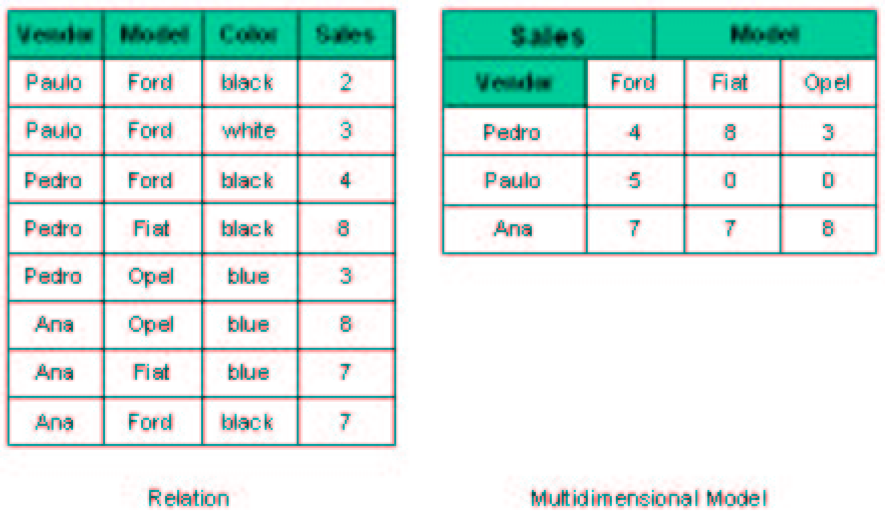
$$\begin{aligned} days \Rightarrow weeks \\ days \Rightarrow months \Rightarrow year \end{aligned} \tag{2.1}$$

Data consolidation in each of these levels can take place.

Once a data dictionary defines a data source, its dimensions, its attributes and hierarchies, all variables and virtual variables calculation formulae, become crucial to all this process of information translation. On the other hand, the previous definition of location and format of the data through meta-information is another important feature.

Multidimensional analysis, as it is the case of the Multidimensional Databases [MK97], should provide a group of operations, namely *Multidimensional Views (Queries), Rotation, Ranging, Rolling-Up, Drilling-Down*[28], *Hierarchy, Reach Through*. Besides these operations, multi-user support is required in a client-server architecture.

With the purpose of illustrating all these concepts and operators, we will give an example of a particular *Car Sales Database* with three dimensions of data. Figure 2.19 depicts the database structure (relational and multidimensional) which supports this problem.



Vendor	Model	Color	Sales
Paulo	Ford	black	2
Paulo	Ford	white	3
Pedro	Ford	black	4
Pedro	Fiat	black	8
Pedro	Opel	blue	3
Ana	Opel	blue	8
Ana	Fiat	blue	7
Ana	Ford	black	7

Relation

Sales	Model		
Vendor	Ford	Fiat	Opel
Pedro	4	8	3
Paulo	5	0	0
Ana	7	7	8

Multidimensional Model

Figure 2.19: *Relation versus Multidimensional Model*

2.7.1.1 Multidimensional Views: Queries

High levels of information structuring in a multidimensional array translate into simple and intuitive queries. For example, in our case, to know the volume of sales for model, for each vendor, such a query would be:

$$PRINT\ total(Sales\ KEEP\ Model\ Vendor)$$

The result could be presented by the following two dimensional table (matrix):

[28]some of the bibliography denote the same operators with *Roll-up* and *Drill-down*

	Vendor		
Model	**Paulo**	**Pedro**	**An a**
Ford	5	4	7
Fiat	0	8	7
Opel	0	3	8

Table 2.2: *A two dimensional table*

Trends and comparisons can easily be made from this kind of output. To get the same result with a SQL query over a relational database, we would have to type:

```
select model, vendor, sum(sales)
from volume_vendas
group by model, vendor
order by model, vendor
```

and the result would be:

Model	**Vendor**	**Sum(Sales)**
Ford	Paulo	5
Ford	Pedro	4
Ford	Ana	7
Fiat	Paulo	0
Fiat	Pedro	8
Fiat	Ana	7
Opel	Paulo	0
Opel	Pedro	3
Opel	Ana	8

Table 2.3: *Relational SQL query result*

As we can see, the results are less intuitive and more extensive. No trends emerge.

2.7.1.2 Rotation

Looking to a multidimensional array is similar to observing a spreadsheet. After an operation takes place, users often wish to see the outcome from a different viewpoint. For example, if the result of the initial operation gets *Sales* of *Models* for automobile *Color*, to get *Sales* for *Color* and for *Model*, the user would have to repeat the initial query.

In a multidimensional array this functionality is obtained in an easy and fast way through dimension *Rotation*. Figure 2.20 depicts the result of applying this operation.

In case the number of dimensions increases, the number of possible views also increases, being equal to *n*(n-1)*, for *n* the number of dimensions.

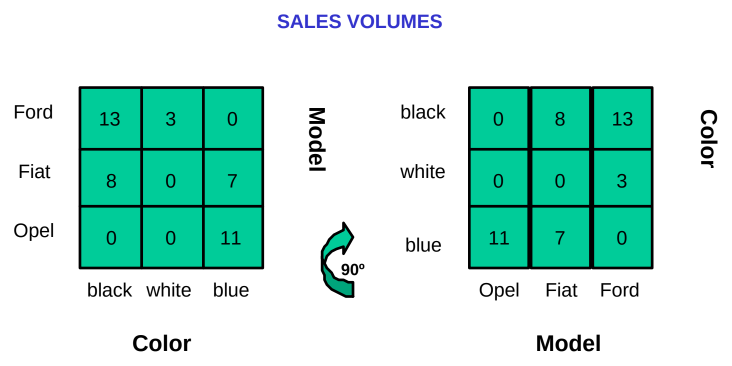


Figure 2.20: *Multidimensional Rotation*

2.7.1.3 Ranging

The essence of ranging consists on selecting a part of the existent elements in each one of the dimensions. In our example, some instances of data ranging could be:

- select Ford and Fiat on Model dimension
- select Pedro and Paulo on Vendor dimension
- select Black and White on Color dimension

Figure 2.21 depicts these operations.

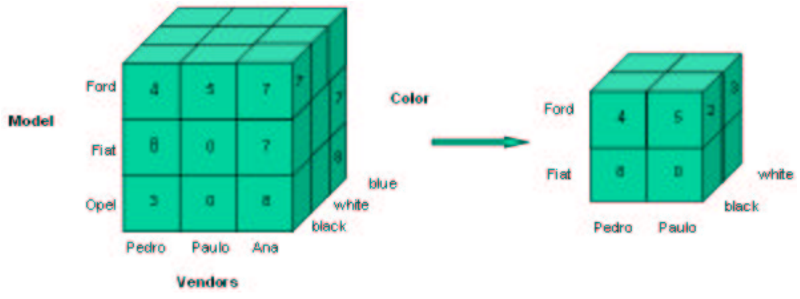


Figure 2.21: *Ranging operation*

Once ranging takes place, the resulting array can be manipulated as if it was a new one. All possible operations can be applied again. This operation is often called as *data dicing*, because smaller subsets of the original data are created.

2.7.1.4 Hierarchies, Rolling-Up and Drilling-Down

As it was said above, the need to view the same data or results from different perspectives is a natural one. Differences can be found either in the ordering of the different dimensions or in the level of information detail that one intends to see.

Back to our example, for instance, it can be interesting to see the volume of sales at District level. In case the result is not sufficiently clear, it can be interesting to analyze the same values now associated to each Vendor which operates in that District. So, a relationship exists between the volume of sales at the *Vendor* and the volume of sales at *District* level. In this case the volume of sales at District level corresponds to the *sum* of all volumes of sales for each *Vendor* on that *District*.

The technology of multidimensional databases is especially prepared to work with this type of natural relationships. Two independent dimensions can be created (one for Vendor and another for District). More efficiently, two related aggregations can be defined over the same dimension (for example, dimension *Sales Region*). The illustration in Figure 2.22 depicts this perspective, showing a schema and instance hierarchy [HM01], inside of the *Sales Region* dimension.

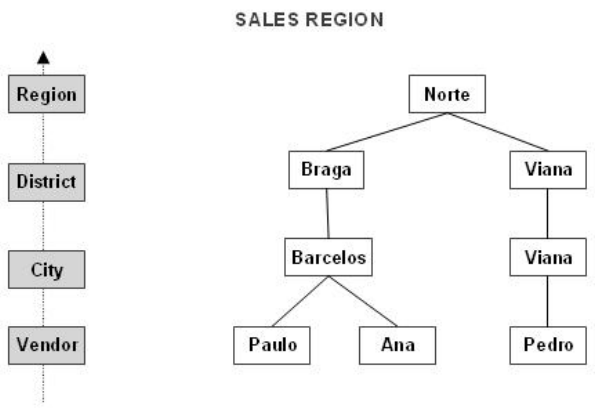


Figure 2.22: *Schema and instance for the* Sales Region *dimension*

Hierarchies definition allows for detailed analysis at different levels inside each dimension. The process which allows to go down or to rise in hierarchy levels of a particular dimension is called *Rolling-Up* and *Drilling-Down*, respectively. Figure 2.23 illustrates these two concepts.

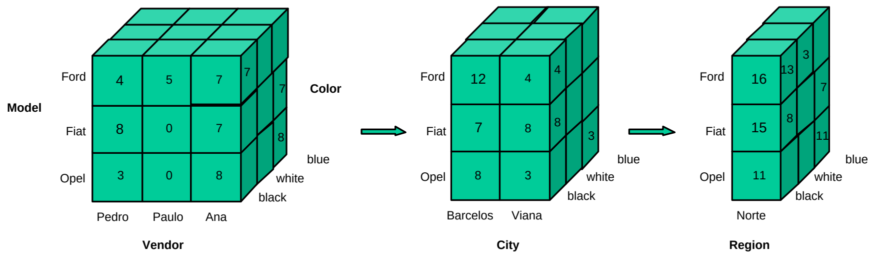


Figure 2.23: *Rolling-Up and Drilling-down*

Although Figure 2.23 depicts directly the *Rolling-up* operator across the hierarchy *Sales* ($Vendor \rightarrow City \rightarrow Region$), on the reverse direction we get the result of the *Drilling-down* operator.

Each view, defined by the levels chosen for each dimension, can be rotated or "filtered" (with *Ranging*) just as we described previously. As similar process on relational context would demand rebuilding the queries for each new view. In Figure 2.22 we show the importance of hierarchies in data representation. A summarizing process is associated to each "slice" of information structure. These summarizing processes are supported by distributive aggregate functions. A distributive aggregate function can be computed on a whole set or on each its disjoint subsets [GCB$^+$97]. SUM, MAX and COUNT are examples of distributive functions.

Assuming *S* a set and S_i parts of *S*, the expression (2.2) illustrates the distributive behavior of the SUM function.

$$\begin{aligned} S &\equiv S_1 \cup S_2 \cup .. \cup S_n \\ SUM(s) &\cong SUM(SUM(s_1), SUM(s_2), ..., SUM(s_n)) \end{aligned} \tag{2.2}$$

2.7.1.5 Reach-through

This feature consists in obtaining the information detail which generated a particular result. That is, the possibility of, after a summarizing operation or aggregation, showing the details (to any level) that could justify that such result exists. This operation can be performed by interacting with the *Drilling-Down* mechanism.

2.7.2 OLAP

Information technology producers are dealing with the new challenge of building systems which can offer strategic and tactical decision procedures to those who must manage and decide (managers, supervisors, etc.), based on global and real data.

This kind of *Decision Support Systems - DSS* [Ma98], are frequently referred to as *OLAP* systems [Nig01], the abbreviation of *On-Line Analytical Processing*, which offers intuitive, fast and flexible manipulation of information.

In general, *OLAP* systems should: (1) support complex user requirement analysis, (2) analyze the data from different perspectives (dimensions), and (3) support complex analysis on large data sets (atomic data).

There are two basic *OLAP* system architectures: *Multidimensional OLAP - MDOLAP* and *Relacional OLAP- ROLAP* [Nig01]. The former uses multidimensional databases as support, while the second manipulates data directly on relational databases. Both architectures implement analysis features of similar levels. However, *MDOLAP* implements this capability on top of previous access to metrics or stored values. In *ROLAP*, measures only exist if they are stored, otherwise they will have to be calculated.

Other OLAP attributes, such as *Dimension* (number of dimensions) and the *Atomicity* (atomic data volume) are significant to the applicability of each one of the architectures.

OLAP systems are mainly related with reading and aggregating large heterogeneous data sets, trying to unveil relationships or patterns (patterns) from data. Such dynamic multidimensional analysis of consolidated data supports all features presented earlier in section 2.7.1 - Multidimensional Analysis.

To conclude this description of methods for knowledge discovery in information systems, one must say that, although the goals are the same, *DataMining* should not ne confused with *OLAP* and *Decision Support*, because these technologies, as we have presented, use a deductive reasoning. They need the user intervention to generate queries or formulate hypotheses. By contrast, *DataMining* techniques use induction, working in a way similar to Artificial Intelligence, where the user looks for a model to discover previous patterns and foresee future behaviors.

2.8 User Interfaces Evaluation

User interfaces evaluation and feedback are final steps which should analyze the final outcome of the interface design. As we can see in UI properties (section 2.2), this phase must guarantee that properties are present and must measure them qualitatively. This can be a very hard and complex task.

As seen previously in section 2.3.5, [Reh01a] reveals that a lot of applications (all of them with UI) violate basic *HCI* guidelines. So the scenario is not auspicious and, naturally, the work to be done should continue harder, more tests and evaluations processes will be needed.

[Mac96] describes the main problems present in these processes, which explain why current research focus this area. We can also see that formal methods appear as a way to get some systematic tests and, because of their rigor, to get interesting measures. The main problems are:

1. Lack of systematic approaches for testing
2. Validity, consistency and reliability of these approaches
3. Too many evaluation dimensions
4. Variety in specification
5. Evaluation costs
6. Adaptability to various environments
7. Extensibility to advances in technology
8. Adaptability to multiple devices

Along this work we make contributions to the body of work on the first problem presented above, applying *VDM TestCoverage* and *Invariants* proof features to analyze and certify *UI* formal specifications.

2.9 Conclusions

The aim of this chapter was to present the main and most recent technological concerns and contributions for user interfaces design.

This chapter offered arguments to analyze methodologies, technologies and frameworks suitable to UI design, and also described the main UI properties that must be implemented under HCI models.

It should be clear now that all main phases of the process: analysis and specification, design and evaluation, can be supported by different technologies and tools, where the programmer needs to work the interfaces and data under distinct and independent processes. From several existing UI programming methods, the markup paradigm appeared as an important alternative, allowing data description and representation.

Describing the VDM language, this chapter showed the applicability of *VDM-SL* as an important contribute against the unaccomplished application of formal methods on UI specification, following its rigor and capacity to model real problems.

In the next chapter we will see in more detail the importance and applicability of markup languages for user interface description.

Chapter 3

Markup Languages for User Interface description

3.1 Presentation

Over the past few years, there have been a number of industrial and academic initiatives to standardize many data types towards application interoperability. However, it seems that the same did not happen in the interface design area.

It is relieving to know that the *W3C - World Wide Web Consortium* [Con03b] and *OASYS - Organization for the Advancement of Structured Information Systems* [OAS03] have worked towards the great dissemination of XML and of some standards for application interoperability (like SOAP[1]) and several XML applications for multiple distinct areas[2]. Focusing on our main goal, there are already several XML applications aiming user interface specifications[3].

EAI[4] [Lin00]is a scientific group concerned with application integration which alerts for the importance of user interfaces in the overall integration process. However, due to the non existence of a standard to describe and manipulate interaction data, a great opportunity is being lost. It is well-known that user-interface "engineering" was deferred to second level in previous computing models [PE02b]. So, there is a great opportunity for better interface design support, both at operation and evaluation levels. In dealing with standard methods and technologies with some concerns for sharing and reusing, caution, rules and concepts are required. If we wish to share information (of any type) across different information systems it is important to have these assumptions [BCMS02] in mind and, in particular, a standard *ontology* to rely on.

In the following sections we are going to present some of this work, trying to show its importance and influence in state-of-the-art software engineering. Our work research has been concerned with *UIML*, one of these efforts, which we will describe later in some detail.

[1]SOAP - Simple Object Access Protocol

[2]*http://xml.coverpages.org/xmlApplications.html*

[3]http://xml.coverpages.org/userInterfaceXML.html

[4]EAI - Enterprise Application Integration

3.2 XIML - eXtensible Interface Markup Language

The *XIML - eXtensible Interface Markup Language* is concerned with defining a common representation and manipulation operators for interaction data [PE02b]. As argued by the XIML Organization [For], "…it can provide a standard mechanism for application and tools to interchange interaction data and to inter-operate with integrated user-interface engineering processes, from design, to operation, to evaluation…".

3.2.1 Scope

This markup language is concerned with the description of abstract, concrete and relational interface data items, as well as features to design, operate and organize such data in the context of user interface engineering.

3.2.2 Requirements

As a new proposed representation methodology, there are requirements in terms of notation, scope and underlying support technologies, which can be summarized as follows:

- a *central repository of data*, enabling a storage mechanism for interaction data;
- a *comprehensive lifecycle support*, enabling support functionality throughout the whole user interface development process;
- *abstract and concrete elements* - allowing for the representation of abstract and concrete (implementation level) interface data;
- *relational support* - for the various interface elements;
- *Underlying technology* - on the one hand, this must be supported by a technology adopted by recent computing models - XML is clearly one of it; on the other hand, it must coexist with existing methodologies and tools (for compatibility).

3.2.3 Structure

Figure 3.1 presents the main structure of the *XIML* language, characterized for an hierarchy of distinct object types.

In its most basic sense, XIML is an organized collection of interface *elements* that are categorized into one or more major interface *components* [PE02b]. As is usual in XML markup language types, there is not a limit for the number and type of elements. However, its specification predefines five basic interface components, namely *task, domain, user, dialog and presentation*.

Tasks (and subtasks) capture the business process and/or user tasks that the interface supports. An example can be "Enter Address", for instance.

The *domain* component is an organized collection of data objects and classes of objects that is structured into a hierarchy. An example could be "Address", for instance.

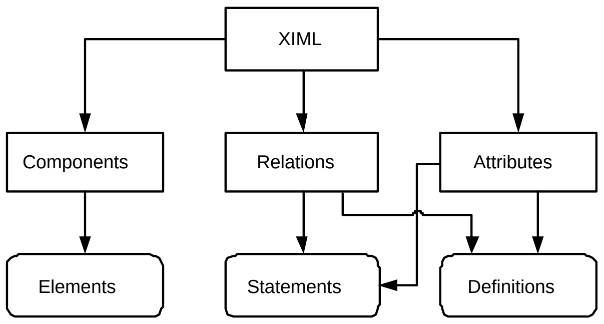


Figure 3.1: *Basic structure of XIML language (adapted from [PE02b])*

The *user* component defines a hierarchy a *tree* of users and associated characteristics.

The *presentation* component defines a hierarchy of interaction elements (concrete objects) that communicate with users in an interface. Examples of these are a *pushbutton*, a *slide* or even an ActiveX [PSM+03] control.

The *dialog* component defines a structured collection of elements that determine the possible interaction actions (“Click”, “OnMouseOver”, etc.) as also specifies the flow among those interaction actions that constitute the allowable navigation of the user interface.

To summarize, the first three of these components can be characterized as contextual and abstract, while the last two can be described as implementational and concrete. Components are then mapped into elements (*widgets* [Mic03a] are an example of such concrete representations). It is also possible to define other components, because of the extensibility of these languages.

A *relation* in XIML is a definition or a statement that links any two or more XIML elements either within one component or across components. For example,“Data type A *is displayed with* Presentation Element B or Presentation Element C” (relation in italic) is a link between a domain-component element and a presentation component element. XIML supports relation *definitions* that specify the canonical form of a relation, and relation *statements* that specify recent instances of relations. However, XIML does not specify the semantics of those relations [PE02b, PE02a].

Attributes are features or properties of elements to which values of several types can be assigned . These attributes follow the XML attribute definition rules, that is, they are pairs of name-values.

3.2.4 Importance

In order to evaluate the importance and usefulness of XIML, we have tried to find recent projects supported and eventually test results. However, both are difficult to find, because most are non academic projects.

Anyway, the promoters [For] of this language announce its applicability to several significant issues [PE02a] in computer science:

- Hand coded representation of interfaces
- Multi-platform interface development
- Intelligent interaction management
- Task modelling
- Reverse engineering

Taking these "may-be" advantages into account, our assessment of the XIML language, underscores its concern with logical separation of user interface and its implementation, as previous computing models announced [Mar95], and its intention to respond to multiple devices, enabling a rendering process of the XIML interface definition to different platforms (PDA, Web browser, etc.)

3.2.5 Remarks

We will see later on, as also announced by XIML promoters, that XIML is "similar" to UIML, the markup language analyzed later in section 3.4. Maybe the difference is the context of each one, XIML being more an industrial initiative against the more academic flavour of UIML. This fact may justify the "separation" between them.

Anyway, there is insufficient bibliography available and source code examples to explore! A little privacy would be better.

3.3 XUL - XML-based User-interface Language

We shall avoid re-inventing the wheel [Con02]

XUL, which must be pronounced as *zool*, stands for *XML-based User Interface Language*. This cross platform way for describing user interface [Rec03c], was first built by the *Mozilla* project[5] to be used in their Mozilla web-browser. Before this work was carried out, there was almost "nothing" available. Many W3C standards appeared as consequence of this work, such are the case of XML 1.0, HTML 4.0 [Rec04a]; Cascading Style Sheets (CSS) [Rec03a] 1 and 2; Document Object Model (DOM) Levels 1 and 2 [Rec03b]; and JavaScript 1.5 [Mic03f], including ECMA-262 Edition 3 (ECMAscript) [Hel03].

For all these contributions, this language was very expressive and relevant during a particular period. It was the first significant documented effort to represent and describe user interfaces focused mainly on Web applications.

3.3.1 Scope

Created by Mozilla, XUL is used mainly to construct the graphical user interface for the Mozilla browser client. Because it is XML-based, all XUL components could

[5]http://www.mozilla.org

be reused for any other browser, since there exists an appropriate parser. It is not a meta-language. It describes directly widgets or graphical objects.

For cross-platform applications it is crucial to have a set of technologies which hide operating system dependent logic from the application logic. Mozilla and XUL together allow for this [Pla03].

3.3.2 Requirements

All requirements are directly associated to the Mozilla browser platform. It must be possible to represent not only basic UI controls (such as Menus, Command buttons, TextBox) but also more comprehensive composite widgets, such as *Tree* widget, *ComboBox*, and *File Picker*, etc. All XML processors for XUL must support Unicode encoding [Che99].

The XUL Consortium [Rec03c], created XUL as being:

- *Standard Compliant*. Both existing internationalization solutions and new proposals shall be taken into consideration.
- *Simple*. The solution shall be easy to implement, and will integrate seamlessly with core development.
- *Portable*. The majority of Mozilla modules are platform-independent (available on Unix, Windows, Macintosh and others) .
- *Extensible*. Having in mind that a solution adopted now might not be valid in the future, they created it flexible enough to support future extension.
- *Separable*. Resources shall be placed into external files (as done with CSS files).
- *Consistent*. Same schema solutions for different modules.
- *Dynamic binding*. Some of the resources requiring translation may be dynamic, usually because they require string composition, e.g., “installing item 5 of 10”.
- *Validatable*. The result shall be easily validated by translators.
- *Invisible*. As much as possible, the standard tools that create English UI should emit files ready to XUL authoring process.
- *Efficient*. The implementation of the solution shall not cause any consequent application problems (performance, memory, etc.).

As a XML-based language, XUL has adhered to the W3C XML1.0 standard specification and follows the syntax rules defined there. In particular,

- all events and attributes must be written in lowercase;
- all strings must be double quoted, including those for attributes values;
- every XUL element must use close tags (either <tag>< /tag> or <tag/ > for empty elements), to be a valid document;

- all attributes must have a value.

A main concern in the design of this language has been its capacity to be *Scriptable* (changes without recompilation), and to support multiple platforms - *Cross-platform* and *Customizable* using *RDF*[6] [Moz03] technology.

3.3.3 Structure

A typical XUL file can contain standard XML elements, XUL specific elements, style information (CSS), HTML, and JavaScript functions. Having a *window* element as root all XUL interface must be defined inside it. Typically, a XUL interface is a combination of *menubars*, *toolbars* and *content areas*, like *window* [Con02]. So, there could be:

- XML declarations and elements
- UI elements and tags (*Menus*,*ComboBox*,*Tree widget*, etc.)
- HTML markups
- Style and layout (using CSS)
- Script functions (using Javascript)
- Chrome URLs (top level window which contains groups of UI elements of various types)

Being a user interface describing language, XUL supports directly objects description and representation. The bulleted list which follows describes the supported widget types [Moz03]:

- *windows*
- *box*
- *menus and menubars*
- *toolboxes and toolbar*
- *tab widget*
- *checkbox*
- *titled buttons*
- *scrollbar*
- *splitter*
- *progessmeter*

[6]*RDF - Resources Description Framework*, not explored in this document.

Let us see an example, adapted from [Moz03], which represents a particular *window* with a *XUL Grid* element (in form format):

Listing 3.1: *XUL User Interface example*

```
<?xml version="1.0"?>
<?xml-stylesheet href="chrome://global/skin" type="text/css"?>
<!DOCTYPE window>
<window id="by lufer" title="Mozilla XUL Example"
  xmlns:html="http://www.w3.org/1999/xhtml"
  xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul"
  style="background-color: #cc9933;"
  width="300" height="215" onload="centerWindowOnScreen( )">
<script type="application/x-javascript"
src="chrome://global/content/dialogOverlay.js" />
<vbox align="center">
  <label value=" " />
  <label style="font-weight: bold; font-size:12pt"
value="XUL example for UI"/>
  <label value=" " />
<image src=""/>
</vbox>
<grid>
  <columns><column flex="1"/><column flex="2"/></columns>
  <rows>
    <row align="center">
      <label value="Title"/>
      <textbox id="title-text" oninput="TextboxInput(this.id)"/>
    </row>
    <row align="center">
      <label value="Author"/>
      <textbox id="author-text" oninput=" TextboxInput(this.id)"/>
    </row>
    <row align="center">
      <label value="About"/>
      <textbox id="about-text" oninput=" TextboxInput(this.id)"/>
    </row>
  </rows>
</grid>
<vbox align="center">
  <label value=" " />
  <label style="font-weight: bold;" value="Simple XUL Window example
for Describing User Interfaces" />
  <label value=" " />
  <image src="xfly.gif" />
  <button label="test" oncommand="alert('Hello World');" />
</vbox>
</window>
```

Figure 3.2 depicts the Mozilla output for this XUL sample:

3.3.4 Importance

As stressed in our previous analysis of XIML, the expressiveness of a particular language must reflect into their use or application. In case of XUL this is clearly demonstrated by looking at the consequences of its creation. Several current XML "assumptions" arise from such work, mainly the recent XML standard, release 1.0.

From a technological perspective, XUL is:

- a XML language with an abstract notation distant from usual languages like Java, HTML, etc;

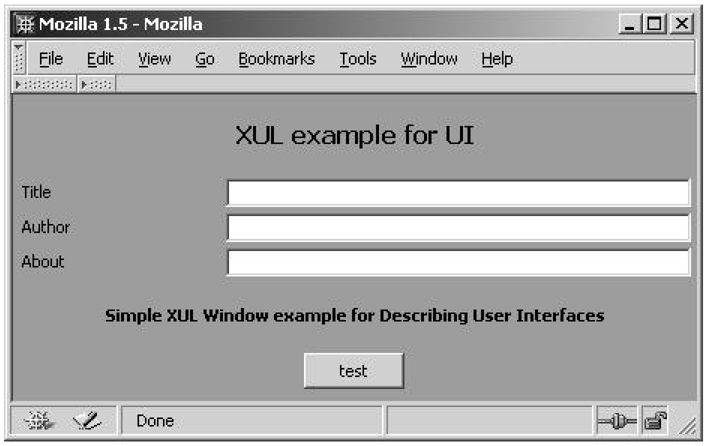

Mozilla 1.5 - Mozilla
File Edit View Go Bookmarks Tools Window Help
XUL example for UI
Title
Author
About
Simple XUL Window example for Describing User Interfaces
test
Done

Figure 3.2: *XUL window example*

- with clearly separations of interface contents from its structure;
- with clearly separations of business logic from the presentation logic, as many UI models advise [Mar95].

This turns this language into a significant reference, which guarantees its portability across platforms. As in other languages, in XUL the designer can concentrate work only on interface layout and ergonomic questions, leaving other responsibilities to other programmers.

3.3.5 Remarks

XUL looks at time of writing like an abandoned initiative, apparent from the lack of recent events or published results. However, one can see that several other languages share XUL as their foundation (UIML is one of them). The rigid relation to Mozzila could be the cause for this almost "hang on" moment. On the one hand, many people use different browsers and on the other hand, there is not sufficient information about XUL portability and support.

3.4 UIML - User Interface Markup Language

"One application, multiple User Interfaces" [Vög03]

UIML, abbreviation of *User Interface Markup Language*, has been proposed by *Harmonia* (*http://www.harmonia.com*) since 1998, as a XML-language to describe user interface elements and respective behavior.

3.4.1 Scope

Following their principle "one application, multiple interfaces", the main reason for this investment resides in the fact that it is important to use the required information in any output device (Phone, PDA, etc.) [AA01]. In short, developers describe the UI regardless of its concrete future platform. The existing rendering mechanism specific for the selected platform does the rest, generating the intended interface, be it HTML for Web browsers, or WML for WAP [For03] devices, or VoiceXml [Rec04b] for multimedia devices, or CHTML [Rec98a] for devices like SmartDevices or PDA, etc.

3.4.2 Requirements

Following Harmonia [Har98], the main reasons for the *UIML* initiative, are as follows:

- *UIML* is a XML-based Meta-Language. As in XML, there is no specific *tag* name directly associated to a concrete visual component (ListBox, PushButton, etc.). It allows the possibility to constitute a specific vocabulary for describing its concreteness. In this way, *UIML* can be adapted to almost any platform.
- *UIML* can be transformed ("transcoded") to any imperative language like C++, Java, Tcl/Tk, as well to any known Markup language as XHTML, WML, etc.
- *UIML* enables a multiple description of interface and respective behavior without the need to appeal to auxiliary technologies like Javascript, Embedded languages, etc.

In other words, what we can say about *UIML* is that:

- there are no specific tags like <pushbutton>, <listbox>, etc.
- there exists a particular set of restrict tags but only for describing UI structure, always far from concrete representation.
- a specific vocabulary is necessary to make it "talk", ie, to allow for final code generation - *Rendering process* - to the intended platform.
- one does not need to change *UIML* to handle a new device. It will only be necessary to get its particular vocabulary.

3.4.3 Structure

Any *UIML* document structure is restricted to its DTD (listed in appendix C of this document) or XSchema . Let us analyze an example of *UIML* in order to better understand its structure.

Listing 3.2: UIML *Hello World example (adapted from [AH02])*

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC "-//Harmonia//DTD UIML 3.0 Draft//EN"
"http://uiml.org/dtds/UIML3_0a.dtd">
<uiml>
        <interface>
                <structure>
                        <part id="TopHello">
                                <part id="hello" class="helloC"/>
                        </part>
                </structure>
                <style>
                        <property part-name="TopHello" name="rendering">Container
                        </property>
                        <property part-name="TopHello" name="content">Hello
                        </property>
                        <property part-class="helloC" name="rendering">String
                        </property>
                        <property part-name="hello" name="content">Hello World!
                        </property>
                </style>
        </interface>
        <peers> ... </peers>
</uiml>
```

Should we intend to represent the target platform, the corresponding <peers> element must be defined. For instance, if one intends to generate an interface to a WAP device (e.g. mobile phone), the *peers* element will be that of Listing 3.3 and the respective WML rendering result (using *Harmonia* WML render) should be that represented on Listing 3.4.

Listing 3.3: *WML <peers> example*

```
<peers>
        <presentation name="WML">
            <component name="Container" maps-to="wml:card">
                <attribute name="content" maps-to="wml:card.title"/>
            </component>
            <component name="String"  maps-to="wml:p">
                <attribute name="content" maps-to="PCDATA"/>
            </component>
        </presentation>
</peers>
```

Listing 3.4: *WML rendering result*

```
<?xml version="1.0"?>
<!DOCTYPE wml   PUBLIC "-//WAPFORUM//DTD WML 1.0//EN"
                "http://www.wapforum.org/DTD/wml.xml">
<wml>
        <card title="Hello">
            <p>Hello World!</p>
        </card>
</wml>
```

As we can see, an *UIML* document is a typical XML document, with a normal “prolog” and a XML tree started by root element called *uiml*. This and all other rules for document structuring are presented on its associated DTD (appendix C). Let us describe the main *UIML* elements a little more:

- in *prolog*, there must be a reference to the *UIML* DTD;
- As <uiml> (the root element) child elements, there are:
  - an (optional) <header> element (there is none in this example) which records metadata about the document;
  - an (optional) <interface> element in which the interface parts, their structure, content, style, and behavior must be described;

- an (optional) <peers> element which must map from each property and event name used to a presentation toolkit and to the application logic:
- an (optional) <template> element which allows for reusing *UIML* fragments:

Considering this, a typical *UIML* document can have the structure of Listing 3.5.

Listing 3.5: *Typical* UIML *document*

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE uiml PUBLIC
        "-//Harmonia//DTD UIML 2.0 Draft//EN" "UIML2_0e.dtd">
<uiml>
        <interface>
            <structure>...</structure>
            <style>...</style>
            <content>...</content>
            <behavior>..</behavior>
        </interface>
        <peers>
            <logic>...</logic>
            <presentation>...</presentation>
        </peers>
        <template>...</template>
</uiml>
```

Because *UIML* will be used on most experiments of this thesis, it is important to have more details about each element of an *UIML* specification. In Chapter 4 it is possible to make a detailed analysis of all *UIML* specification [AH02] elements. Figures 3.3, 3.4 and 3.5 depict the hierarchy of main UIML elements (release 3.0): *uiml*, *peers* and *interface* (the appendix D of this document depicts the complete *UIML* hierarchy).

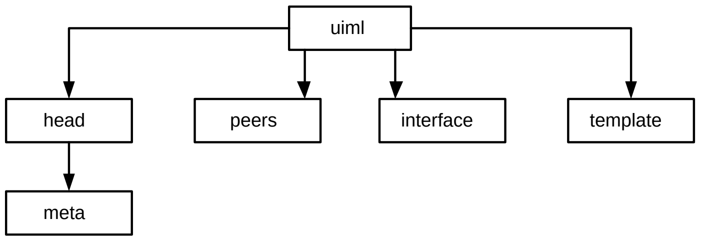


Figure 3.3: *The* uiml *element*

3.4.4 Importance

UIML appears to be an easy-to-use markup language, independent from any concrete representation or device platform, helping those programmers who find it hard to adopt a particular programming paradigm (like OO, logic programming or others), or less technical information to support a particular device, to go further. Our understanding of the main advantages of *UIML* is as follows:

- it allows developers or programmers to implement UI for almost any device without the need to completely understood the respective API;

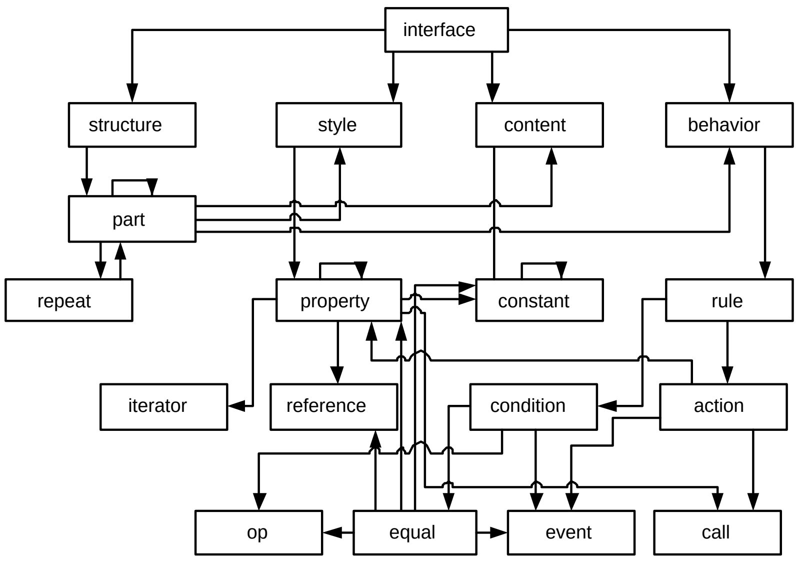


Figure 3.4: *The* peers *element*

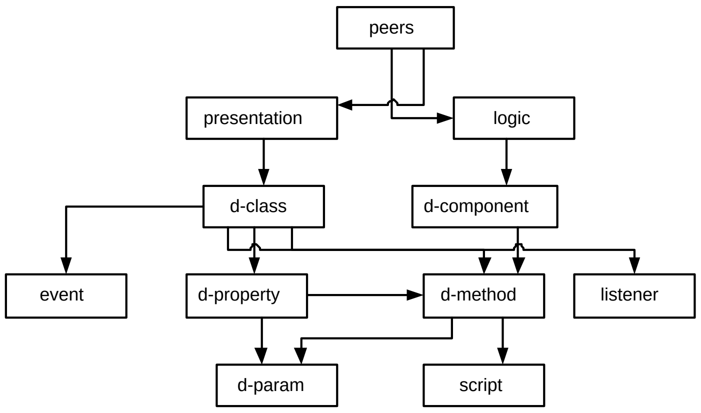


Figure 3.5: *The* interface *element*

- it provides a clear separation of programming logic and end user UI requirements, working for the specialization and maximization of working team capacity.
- it enables a UI description without knowing a particular programming language (UIML is almost a natural text language).
- the portability and internationalization are ensured thanks to XML.
- it provides important security possibilities, because there is only one document to deal with.
- it allows for rapid prototyping of UI.
- it provides support for future technologies.

At the moment, there are eleven vocabularies available (e.g, Java AWT, WML, HTML, etc.) [Org98]. However, research are under way on a *Generic Vocabulary* which should allow the conversion of generic terms to a particular toolkit. The main features of this new vocabulary are based on (a) the possibility to describe any UI for a family of devices and (b) being generic enough to be used without having expertise in all multiple platforms and toolkits within the family [APQA02].

There is, however, an important handicap: at time of writing, *UIML* is not yet a recognized standard.

3.4.5 Remarks

In section 2.3.4 we anticipated that *UIML* is an outcome of the *Meta-Interface Model*. Figure 2.8 on page 21 shows the proximity of *UIML* architecture and that model.

It has become a significant markup language to describe user interfaces, with a substantial expressiveness on UI development. It has advanced greatly once *Harmonia* and *UIML.org* proposed a *Technical Committee* to *OASIS* [OAS03], towards a standardization process.

There is a clear separation of content, structure and style, and the ability to render to any specific UI language, allowing for the *canonical representation*[7] of any UI suitable for mapping to existing languages [Hel03].

UIML could combine with other existent initiatives inheriting their most interesting features and developing new nonexistent ones. In what follows we present a possible comparison of *UIML* with other significant markup languages. Later, in Chapter 4, we will see the application of *UIML* on user interface specification and description.

3.5 Relationship between *UIML* and other UI Markup languages

From the above we can conclude that there are a few technologies enabling user interface description. In this context, we must highlight *Web applications* and *Markup languages*.

Markup languages can be justified by (a) being familiar for the Web community; (b) reducing the amount of necessary expertise; (c) giving direct support to Web-based applications;(d) Leveraging XML and (e) reducing the number of distinct technologies to assimilate [PA99].

Web applications are relevant because (a) they run on HTML browsers available for a variety of platforms; (b) they are accessible over the Internet and (3) they are simple and easy to build.

Figure 3.6 shows where *UIML* can be placed with respect to portability and abstraction capabilities.

Let us now take a look at the most significant XML-based markup technologies,

[7] A *canonical representation* can be seen as a metaphor-independent enumeration of the parts, behaviors, content and style of user interface [AH02].

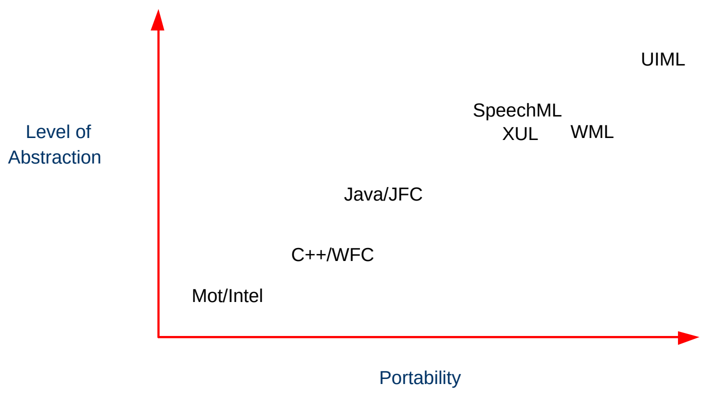


Figure 3.6: *Portability of* UIML *(adapted from [PA99])*

dealing with user interfacing, and see how *UIML* compares with them [Hel03].

3.5.0.1 UIML and XUL

XUL and *UIML* have several key similarities and it is difficult to say which of them is the source of the other [Hel03]:

Similarities:

- Both are XML-based;
- Both provide desktop platform portability by abstracting the user interface definition away from concrete languages such as Java, C++, or HTML.
- Both provide ease of localization, customization, and personalization by separating content from structure.
- Both separate the business logic from the presentation logic.

Differences:

- The programmer must combine XUL, JavaScript, XBL, CSS, and DTD of string bundles, while *UIML* packages everything into a language.
- *UIML* is a meta-language.
- *UIML* seeks to break the definition of a user interface into its six fundamental components: structure, content, style, behavior, presentation, and logic.
- Vocabulary is not a part of the *UIML* specification and can be customized for whatever domain and/or widget set is appropriate to the target application.

3.5.0.2 UIML and AUIML

AUIML - Abstract User Interface Markup Language[8] was developed by IBM, with the main goal to allow UI designers to develop the interface independently of any concrete

[8]http://xml.coverpages.org/userInterfaceXML.html

device specific realizations.

As *AUIML* has a specific vocabulary using tags like <group>, <choice>, <caption>, and <string> to represent data [Luy01], a new vocabulary can be defined for *UIML* that maps directly to AUIML's tag set. This will allow *UIML* to easily represent any interface that can be defined in AUIML [Hel03].

At the time of writing AUIML looks a standby initiative.

3.5.0.3 UIML and XIML

As seen in section 3.2, XIML and *UIML* seem to be both overlapping and complementary at the same time. The overall goal of both languages is to serve as a universal, canonical language for specifying user interfaces on computing devices. However, XIML's emphasis on the early design level, generic components of a user interface and UIML's ability as a meta-language to apply to late-design and implementation leaves room for both languages to complement each other [Hel03].

However, XIML has an important "joker" over UIML. It is prepared to deal with interaction context data to support knowledge-based system functions and has a clear separation of interface description from its rendered platform. Is this or not an advantage?

3.5.0.4 UIML and XForms

In section 2.3.2 (pag. 18) we could see the relevance of *XForms* in *Web Programming*. On the other hand, we could analyze the actual expressiveness of *UIML* as another expressive and significant language which operates over user interfaces. It seems, in a first analysis, that they are very similar technologies. Thus, it is very important to try and compare them.

Basically, *UIML* provides flexibility in form and function that can be used to represent any UI. So, *UIML* can be used to define interfaces that represent, not only the XForms sections of the UI, but also the *XHTML* [W3C03] in which the XForms tags are embedded [Rec03f].

Without trying to select a "winner", there are some situations where these languages work differently:

- XForms allows great advances on web forms, including remote process;
- *UIML* would be capable of defining a canonical representation of any UI.
- *UIML* can be used to serve as an intermediary so that transformers designed by one researcher from a language or model to *UIML* can be used in conjunction with transformers designed by another researcher from *UIML* to another language or model [Vög03]. XForms is not powerful enough to allow that.
- *UIML* provides strict separation of the user interface elements (we could find it in section 2.3.4 (on page 20) when we presented the model *MIM*) . This does not happen in XForms.

- *UIML* describes connection methods between the UI and whatever the UI intends to interact with. In particular, UIML should describe the wiring of the UI to business logic that uses a procedure call model or a Web services server that uses SOAP; or a web server using HTTP GET and PUTs; or a publish/subscribe protocol using events to push content to the UI; and so on [Vög03]. In contrast, XForms builds on the HTTP GET/PUT model of the Web.

- Because of its extensibility, *UIML* allows any UI metaphor [Mar95]. XForms is restricted to the web-forms model of interaction.

- *UIML* should represent the real world, allowing for descriptions of UIs expressed using almost free vocabulary abstractions rather than the selected future support (language, visual components, etc.). XForms is conditioned by its restricted notation.

- *UIML* should support behaviors, so that much of the JavaScript in traditional web pages can be described in a device-independent form. In contrast, XForms uses XEvents [Rec03e].

3.5.0.5 Remarks

UIML has been identified with some other languages and even several work groups like W3C, HCI, *WAI - Web Accessibility Initiative*, *DIWG - W3C Device Independence Working Group*, etc. For a deeper perspective of these relationships, see [Hel03].

We can also confirm that *UIML* does not walk alone on this path, ignoring all other initiatives. For instance, in the current *UIML* version (3.0) it is even better to use XForms with *UIML* than to work directly with HTML.

Being *UIML* an XML schema targeted to serve as an universal interchange format to represent any UI - regardless of UI metaphor, language, operating system and device [Pha00], we decided to explore it more deeply. If this work focused only web applications the decision could be different.

XUL was also explored because of its easy and clear representation of visual components, allowing us to ignore more complex details and to analyze direct mapping of interaction objects. However if we need more abstraction, XUL is not enough. So the choice of *UIML* followed.

Chapter 4

UIML Formal Specification

"The design objective of the *UIML* is to provide a canonical representation of any UI suitable for mapping to existing languages."[1]

The specification of a user interface system should agree with an *IDL - Interface Description Language*[2], able to describe each visual component of a Visual Component Library (VCL), its behavior (actions), channels, interaction and in and out relationships.

The *UIML* should not be considered a pure VCL because it "works" away from any concrete representation. However it can be understood as so in the sense that it describes visual components that can be represented in a concrete graphical environment.

The purpose of this chapter is to formally define, in *VDM-SL* notation, (recall section 2.4.3), the specification of the graphic elements of the *UIML* visual component library, including system properties and operationality.

For our intention to evaluate the components behavior and operationality (ease of use, ergonomics, etc.), it is important to animate the specification and the application using prototyping models. The VDMTOOLS [IFA00c] application will support this simulation process.

4.1 VDM-SL Specification

4.1.1 Terminologies

This section presents the main definitions of *UIML* 3.0 specification, as described in [AH02].

Application: Wherever we speak of building an UI, the UI itself along with the underlying logic that implements the functionality visible through the interface is called the application.

[1] UIML3.0 Language Specification. Harmonia, Inc.
[2] Acronym IDL arises in systems integration terminology

Canonical Representation: An UI metaphor-independent enumeration of the parts, behaviors, content, and style of a user interface.

End-user: The person that uses the application.

Application Logic: Code that is part of the application but not part of the UI. In a three-tier system architecture model (section 2.6.2 on page 39), the application logic is the middle layer that mediates communication between the database and presentation layers.

Device: A device is a physical object with which an end-user interacts using an UI, such as a PC or a Smart Device.

UI Toolkit: A toolkit is the markup language or software library upon which the UI of an application runs. *Harmonia* [Har98] uses the word "toolkit" to mean both markup languages that are capable of representing UI (e.g., Wireless Markup Language - *WML* [For03], HTML [Rec04a], and *VoiceXML* [Rec04b]) as well as APIs for imperative programming languages (e.g., *Java AWT*, *Java Swing* [Mic03e], *Microsoft Foundation Classes* [Mis99]).

Platform: A platform is a combination of a device, an operating system (OS) and an UI toolkit. An example of a platform is a PC running Windows on which applications use the Java Swing [Mic03e] toolkit. Another example is a cellular phone running a manufacturer-specific OS and a WML [For03] render.

Rendering: Is the process of converting a *UIML* document into a form that can be displayed (e.g., through sight or sound) to an end-user, and with which an end-user can interact. Rendering can be accomplished in two ways:

- By compiling *UIML* into another language (e.g., WML, Java), which allows for displaying and interacting with the UI described in UIML. Compilation might be accomplished by XSL [Rec01a, Rec99b], or by a program written in a traditional programming language (like Perl [Per03], for instance).
- By interpreting UIML, meaning that a program reads *UIML* and makes calls to an API that displays the UI and allows interaction. Interpretation is the same process that a Web browser uses when presented with an HTML document.

Rendering engine: Software that performs the actual process or rendering a *UIML* document. It could be a StyleSheet (for instance, written in XSL [Rec01a]), a script (for instance, written in Perl [Per03, Dev03]) or a simple compiled program (for instance, written in $C\sharp$ [SJ03]).

UI **Widget:** *UIML* describes how to combine UI widgets. The term widget is traditionally used in conjunction with a graphical UI. However *UIML* uses it in a more general sense, to mean presentation of elements in the context of any UI paradigm. For example, a widget might be a component in the *Microsoft Foundation Classes* or *Java Swing* toolkits, or a card or a text field in a WML document.

Method: *UIML* vocabulary uses the term method in generically referring to any code entity (that uses a language other than UIML) that a rendering engine can invoke, and which may optionally return a value. Examples include functions, procedures, and methods in an object-oriented language, database queries, and directory accesses.

4.1.2 Presentation

UIML can be viewed as a meta or extensible language, analogous to XML. *UIML* does not contain specific tags to a particular *UI* toolkit (e.g., <WINDOW> or <MENU>). Instead, it uses a set of generic tags (e.g., <part>, <property>).

As mentioned earlier on in section 3.4, page 60, *UIML* captures the elements that are common to any UI: the *UI* parts, events that occur for which parts, presentation style, content, and interconnection to the application logic. *UIML* authors specify instance and class names of their own choice for interface parts, events, and methods.

The *UIML* document specifies a mapping from those names to names that are vocabulary specific to a particular target platform. For example, if the target is *Java AWT*, the vocabulary might consist of the "java.awt" and "java.awt.event" class names, such as "JFrame", "JMenu" and "JButton". If the target is *WML*, the vocabulary might be tag names such as "card", "select" and "input". The vocabulary of target platforms is not a part of UIML. That vocabulary only appears in *UIML* as the value of attributes in UIML. Thus *UIML* only needs to be standardized once, and different constituencies of end users can define vocabularies that are suitable for various toolkits independently of UIML.

In addition to creating vocabularies for particular toolkits (e.g., Java AWT), a vocabulary for generic classes of toolkits (e.g., mapping to any graphical UI) could be devised.

4.2 *UIML* Overview

In the sequel, we are going now to describe the main parts of a typical *UIML* document, their importance and interoperability.

4.2.1 The Structure of an *UIML* Document

In *UIML* version 3.0, an *UI* is a set of interface elements with which the end-user interacts. Each interface element is called a part; just as an automobile or a computer is composed of a variety of parts, so is an UI. The interface parts may be organized differently for different categories of end-users and different families of devices. Each interface part has content (e.g., text, sounds, images) used to communicate information to the end-user. Some interface parts can receive input from the end-user. This is usually achieved through the use of interface artifacts like a scrollable selection list, pressable button, etc. Since the artifacts vary from device to device, the actual mapping (rendering) between an interface part and the associated artifact (widget) is done using either a <presentation> element or a special <property> element in the <style> element.

My definition of an user interface in *UIML* should provide answers for the following five questions:

- What parts comprise the UI?
- What presentation style for each part (rendering, font size, color, etc.)?
- What contents for each part (text, sounds, image, etc.)?
- What is the behavior of each part?
- How to connect to the outside world? (business logic, data sources, *UI* toolkit)

As we could see in Chapter 2.3.4 (page 20), *UIML* is based on the MIM model [Pha00].

4.2.2 *UIML* document

A typical *UIML* 3.0 document is composed of two main parts:

1. A *prolog* identifying the XML language version, encoding, and the location of the *UIML* 3.0 document type definition (DTD). The next excerpt illustrates this *prolog*:

   ```
   <?xml version="1.0"?> <!DOCTYPE UIML PUBLIC "-//Harmonia//DTD UIML
   3.0 Draft//EN" http://uiml.org/dtds/UIML3_0a.dtd">
   ```

2. The root element of the document, which is the *uiml* tag:

   ```
   <uiml xmlns='http://uiml.org/dtds/UIML3_0a.dtd'> ... </uiml>
   ```

   The <uiml> element contains four child elements:

   (a) An optional header element giving some more metadata about the document:

   ```
       <head> ... </head>
   ```

   The <head> element will be discussed in Section 4.3.3.2.

   (b) An optional element that allow for reusing *UIML* fragments:

   ```
   <template> ... </template>
   ```

   This element will be discussed in section 4.3.6.2.

   (c) An optional *UI* description, which describes the parts comprising the *UI*, and their structure, content, style, and behavior:

   ```
   <interface> ... </interface>
   ```

This element will be described in section 4.3.4.3.

(d) An optional element that describes the mapping of classes and names used in the *UIML* document to a UI toolkit and to the application logic:

`<peers> ... </peers>`

The discussion of the <peers> element will take place in section 4.3.5.1.

Listing 4.1 presents a *UIML* structured document. Listing 4.2 provides an example of a <peers> specification preparing for the *WML* [For03] code generation. Having this, the *WML* output code would be as presented in Listing 4.3.

Listing 4.1: UIML *"Hello World" example*

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC "-//Harmonia//DTD UIML 3.0 Draft//EN"
"http://uiml.org/dtds/UIML3_0a.dtd">
<uiml>
        <interface>
                <structure>
                        <part id="TopHello">
                                <part id="hello" class="helloC"/>
                        </part>
                </structure>
                <style>
                        <property part-name="TopHello" name="rendering">Container
                        </property>
                        <property part-name="TopHello" name="content">Hello
                        </property>
                        <property part-class="helloC" name="rendering">String
                        </property>
                        <property part-name="hello" name="content">Hello World!
                        </property>
                </style>
        </interface>
        <peers> ... </peers>
</uiml>
```

Listing 4.2: UIML *"Hello World" example* - WML *<peers> description*

```
<peers>
        <presentation name="WML">
                <component name="Container" maps-to="wml:card">
                        <attribute name="content" maps-to="wml:car.title"/>
                </component>
                <component name="String" maps-to="wml:p">
                        <attribute name="content" maps-to="PCDATA"/>
                </component>
        </presentation>
</peers>
```

Listing 4.3: UIML *"Hello World" example - WML output"*

```
<?xml version="1.0"?>
<!DOCTYPE wml    PUBLIC "-//WAPFORUM//DTD WML 1.0//EN"
                 "http://www.wapforum.org/DTD/wml.xml">
<wml>
        <card title="Hello">
            <p>Hello World!</p>
        </card>
</wml>
```

Our next example is a case study dealt with in detail in this work: a Stack *UIML* specification. This classic example [Oli02] is well known as having several different *UIML* elements. We will provide simulated *UIML* code excerpts where required, always refering source code lines.

Conceptually, the stack problem is a simple one. Its interface is mainly supported by methods $Push$, Pop, Top and $Empty$, where:

- *Push* - adds a new element;
- *Pop* - removes the top element;
- *Top* - shows the first element;
- *Empty* - tests if the stack is empty.

In Appendix H, we present the complete *VDM-SL* specification, as presented in [Oli02]. Figure 4.1 shows the *UIML* Java rendered for our stack *UIML* specification, which is presented in Listing 4.4. This is an excerpt of the complete *UIML* specification of the Stack interface available from Appendix G.1 in this document.

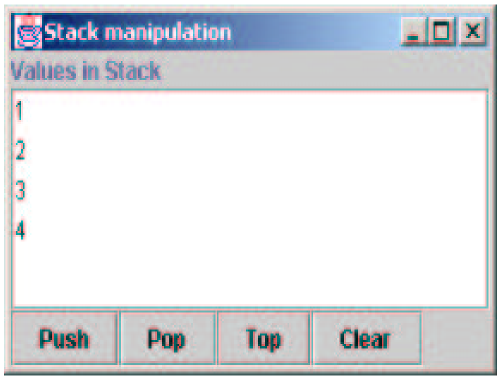


Figure 4.1: *Stack Java/*UIML *interface*

Listing 4.4: UIML *Stack Interfaces specification*

```
<interface>
  <structure>
    <part id="Top" class="JFrame">
      <style>
              <property name="layout_hgap">10</property>
              <property name="layout_vgap">25</property>
      </style>
      <part id="Label" class="JLabel"/>
      <part id="ScrollPane"   class="JScrollPane">
        <part id="List"       class="JList"/>
      </part>
      <part id="ButtonPanel" class="JPanel">
        <part id="AddButton"    class="JButton"/>
        <part id="RemoveButton" class="JButton"/>
        <part id="TopButton" class="JButton"/>
        <part id="Clear" class="JButton"/>
      </part>
     </part>
  </structure>
```

```
      <style>
        <part id="Top" class="JFrame">
            ...
        </part>
      </style>

      <behavior>
        <rule>
            ...
        </rule>
  </interface>

  <peers>
    <logic>
            ...
    </logic>
    <presentation base="Java_1.3_Harmonia_1.0"/>
  </peers>
</uiml>
```

In summary, the skeleton of a *UIML* document can be approached as presented in Listing 4.5:

Listing 4.5: *Skeleton of a* UIML *document*

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC
"-//Harmonia//DTD UIML 3.0 Draft//EN" "http://uiml.org/dtds/UIML3_0a.dtd">
<uiml xmlns='http://uiml.org/dtds/UIML3_0a.dtd'>
        <head> ... </head>
        <template> ... </template>
        <interface> ... </interface>
        <peers> ... </peers>
</uiml>
```

4.2.3 *UIML* Namespaces

UIML is designed to work with existing standards. These include other markup languages that specify platform-dependent formatting (i.e., HTML for text, WML for Wap, etc.). XML Namespaces remove the problem of recognition and collisions between elements and attributes of two or more markup vocabularies in the same file. All <uiml> elements and attributes are inside the *uiml* namespace, identified by the URI $http://uiml.org/dtds/UIML3\_0a.dtd$. Note that this URI has not been activated yet. The next two examples show this behavior. The excerpt which follows combines *UIML* and HTML vocabularies:

```
<uiml:UIML xmlns:uiml='http://uiml.org/dtds/UIML3_0a.dtd'>
<uiml:interface>
    <uiml:structure>
        <uiml:part uiml:name="A"/>
    </uiml:structure>
    <uiml:style>
        <uiml:property uiml:name="content" uiml:part-name="A">
        <html:em xmlns:html='http://www.w3.org/TR/REC-html40'
        >Emphasis</html:em>
        </uiml:property>
    </uiml:style>
</uiml:interface>
</uiml:uiml>
```

The above code can be simplified by making *UIML* the default namespace, as follows:

```
<UIML xmlns='http://uiml.org/dtds/UIML3_0a.dtd'> <interface>
    <structure>
        <part name="A"/>
    </structure>
    <style>
        <property name="content" part-name="A">
        <html:em xmlns:html='http://www.w3.org/TR/REC-html40'
        >Emphasis</html:em>
        </property>
    </style>
</interface>
</uiml>
```

To learn more about XML Namespaces please, refer to [Con00].

4.2.4 *UIML* Elements

Table 4.1 provides an overview of all elements in *UIML* and an index to where they are discussed in the remainder of this chapter. The *UIML* 3.0 DTD is presented in Appendix C.

Figures 4.2, 4.3 and 4.4, depict the hierarchy of main *UIML* elements: <uiml>, <interface> and <peers>. However, the complete *UIML* 3.0 hierarchy elements is depicted on Appendix D of this document.

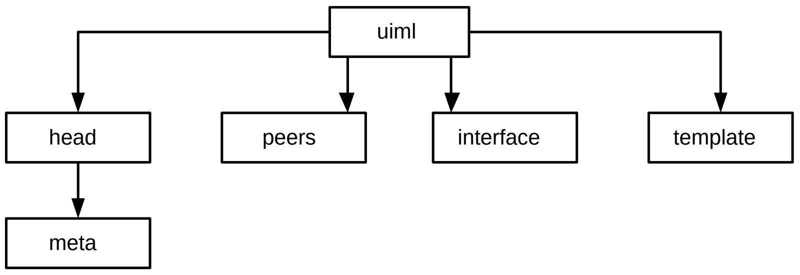


Figure 4.2: *<uiml> hierarchy*

4.3 *UIML* Formalization

4.3.1 Considerations

In this section we describe informally all *UIML* elements and attributes and then propose a corresponding formal specification (in *VDM-SL* notation).

Element	Purpose	Page
<action>	Perform an action if the condition of a rule is true	98
<behavior>	Specify rules for runtime behavior	93
<by-default>	Set of actions to be executed when *op* conditional is undefined	105
<call>	Call a function or method external to *UIML* document	100
<condition>	Specify a condition for a rule	96
<constant>	Define a constant value	92
<content>	Specify a set of constant values	90
<d-class>	Maps class names that can be used for parts and events to an *UI* toolkit	111
<d-component>	Maps a name used in a *call* element to application logic external to *UIML* document	110
<d-method>	Maps a method to a callable method or function in the API of the application logic	112
<d-param>	Defines a single formal parameter to a *d-method*	114
<d-property>	Maps a property name, for parts or events, to methods in an *UI* toolkit that get and set the properties value	112
<equal>	Compares the value of an event with another value	96
<event>	Specify a runtime *UI* event	97
<head>	A container for metadata information	80
<interface>	A container for the interface description	83
<iterator>	A tag controlling the number of times a virtual tree contained in a *repeat* element is replicated.	101
<logic>	A container for computation components	109
<meta>	Define a piece of metadata as a name/value pair	81
<op>	Define a conditional expression or operation	98
<param>	Actual parameter used in a *call* element	105
<part>	Specify a single interface part	86
<peers>	Mapping from property and event names to *UI* toolkit	107
<presentation>	A container for presentation components	108
<property>	Specify an interface property	89
<reference>	Reference a constant	93
<repeat>	Groups parts which are repeated one or more times in a user interface	100
<restructure>	Modify the current virtual tree of parts	102
<rule>	A condition/action pair	95
<script>	A container for executable script code	115
<structure>	Specify an interface physical structure	84
<style>	Specify a set of style properties for the interface	88
<template>	A container for reusing *UIML* elements	116
<uiml>	Top-level element in each *UIML* document	79
<when-true>	Set of actions to execute when *op* condition is true	104
<when-false>	Set of actions to execute when *op* condition is false	104

Table 4.1: *UIML elements*

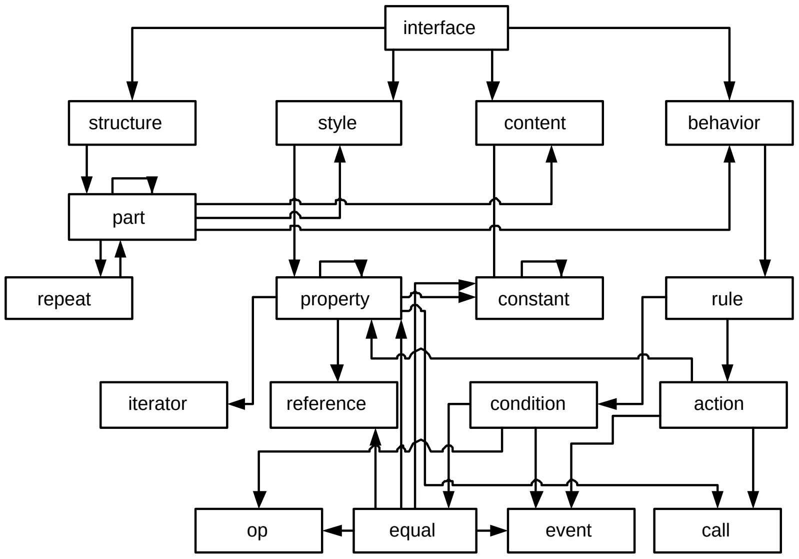


Figure 4.3: *<interface> hierarchy*

Whenever a new element is introduced in the remainder of the document, we first give the appropriate DTD fragment. Then a succinct description precedes the corresponding *VDM-SL* specification. Some examples related to our case study will be also presented. In the *VDM-SL* specification [Hop01], for types and functions identifiers, we adopt the well known "camel" (or Pascal)[3] notation convention, i.e., the *Name-Name* format. If an identifier name has more than one tag, every tag (except the first) starts with a capital letter. Selectors of compound types all start with lowercase letters.

4.3.2 *VDM-SL* Types and common *UIML* attributes

To get the mathematical accuracy of the intended formalization, this *VDM-SL* specification is supported by finite *Sequences* (ié *seq of* in *VDM*), thus respecting the order of *UIML* elements instantiation. In the next step of the process, an abstraction process will transform every element to be supported by other - more abstract - model.

```
module UIMLSpec
    exports all
definitions
```

```
    String = char⁺;
```

First of all, some common attributes common to several *UIML* elements:

[3] $http://www.cob.sjsu.edu/johnson_f/notation.htm$

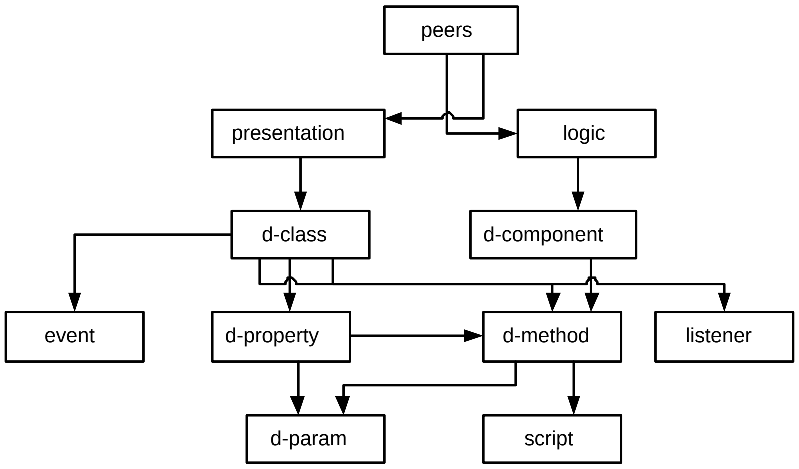


Figure 4.4: *<peers> hierarchy*

$ID :: \ String;$
$SourcesModes = \text{UNION} \mid \text{CASCADE} \mid \text{REPLACE} \mid \text{APPEND};$
$ExportOptions = \text{HIDDEN} \mid \text{OPTIONAL} \mid \text{REQUIRED};$
$SourceElements = Behavior \mid D\text{-}class \mid D\text{-}component \mid Constant \mid Interface \mid Logic \mid Part \mid Peers \mid Presentation \mid Property \mid Rule \mid Script \mid Structure \mid Style \mid Content \mid Restructure;$
$Member = Peers \mid Interface \mid Template;$
$WhereOptions = \text{FIRST} \mid \text{LAST} \mid \text{BEFORE} \mid \text{AFTER};$

Other required types are:

$InterfaceElements = Structure \mid Style \mid Content \mid Behavior;$
$TypesOptions :: type : (\text{INPUT} \mid \text{OUTPUT} \mid \text{INOUT} \mid \text{NONE});$

4.3.3 *UIML* top elements

4.3.3.1 The <uiml> element

DTD

```
<!ELEMENT uiml  (head?, (peers | interface| template)*)>
```

Description

The *UIML* element is the root element in a *UIML* document. It has no attributes.

```
<uiml>...</uiml>
```

Usually, one *uiml* element equates to one file, in much the same way that there is one HTML element per file when developing HTML-based applications. However, other arrangements are possible. For example, the *uiml* element might be retrieved from a database or the elements contained within the *uiml* element might be stored in multiple files.

When multiple markup vocabularies are used within the same *UIML* file, then the *UIML* Namespace must be specified as follow:

```
<uiml xmlns='http://uiml.org/dtds/UIML3_0a.dtd'>...</uiml>
```

The *VDM-SL* specification of this is as follows:

$$\begin{aligned}
&Uiml :: head : [Head] \\
&\qquad members : Member^* \\
&\mathsf{inv}\ uiml \triangleq uniqueIDs\,(uiml) \land \\
&\quad validProperties\,(uiml);
\end{aligned}$$

Predicates *uniqueIDs* and *validProperties* are defined in pages 118 and 142 respectively of this document.

4.3.3.2 The <head> element

DTD

```
<!ELEMENT head (meta)*>
```

Description

UIML The <head> element contains metadata about the current *UIML* document. Elements in the <head> element are not considered part of the interface, and have no effect on the rendering or operation of the UI. The *UIML* element

```
<head>...</head>
```

is used by *UIML* authoring tools to store information about the document (e.g., author, date, version, etc) and other proprietary information.

VDM-SL Specification

$$Head :: meta : Meta^*;$$

Along with *head*, *peers* (described in Section 4.3.5.1, page 107), *interface* (described in Section 4.3.4.3, page 83) and *template* (described in Section 4.3.6.2, page 116) elements, compose the main top elements of all *UIML* DTD.

4.3.3.3 The <meta> element

DTD

```
<!ELEMENT meta EMPTY>

<!ATTLIST meta
        name    NMTOKEN #REQUIRED
        content CDATA   #REQUIRED
>
```

Description

The meta element has the same semantics as the meta element in HTML. It describes a single piece of metadata about the current *UIML* document. This may include author information, date of creation, etc.The name attribute specifies the meta-information and the content attribute its content.

Example

```
  <head>
    <meta name="lufer" content="UIML Simple Stack"/>
    <meta name="Date" content="July, 2003"/>
    <meta name="Description" content= ‘‘This is an example
    of how to use the UIML to specify real problems."/>
  </head>
```

VDM-SL Specification

$$\begin{aligned} Meta :: \; & name : String \\ & content : String \end{aligned}$$

4.3.4 Interface description

This section describes the elements that go inside the *interface* element, their attributes, and their syntax. Examples from our case study are provided.

4.3.4.1 Overview

The *interface* element contains a sequence of four elements: *structure*, *style*, *content*, and *behavior*:

```
<interface>
    <structure> </structure>
    <style>     </style>
    <content>   </content>
    <behavior>  </behavior>
</interface>
```

- The *structure* element enumerates a set of interface parts and their organization for various platforms.
- The *style* element defines the values of various properties associated with interface parts (analogous to style sheets for HTML).
- The *content* element gives the words, sounds, and images associated with interface parts to facilitate internationalization or customization of UIs to various user groups.
- The *behavior* element defines which UI events should be acted on and what should be done.

4.3.4.2 Attributes common to multiple elements

Before explaining each of the elements presented in Table 4.1 (page 77), we first describe some attributes that are used in several elements.

The *id* and *class* Attributes: The <part>, <event>, and <call> elements in *UIML* may have an *id* and a *class* attribute.

The id attribute assigns a unique identifier to an element, i.e., no two elements can have the same id within the same *UIML* document (see Section 4.4).

The *class* attribute assigns a class name to an element. Any number of elements may be assigned the same class name.

The use of the attribute *class* is based on the CSS [3] concept of a class: it specifies an object *type*, while the element id uniquely identifies an instance of that type. A style associated with all instances of a class is associated with all elements that specify the same value for their class attribute; a style associated with a specific instance of a class is associated with any elements that specify the same value for their *id* attribute.

The *source* and *how* Attributes: Certain *uiml* elements (*behavior, d_component, d_class, d_method, constant, content, interface, logic, part, peers, presentation, property, rule, script, structure, and style*) may contain a *source* attribute. Like HTML, the *source* attribute specifies a link from the *UIML* document to a Web resource named by a URI. However, the reason for using a link in *UIML* differs from HTML.

A *source* attribute can refer to two things:

- *A URI to a resource that does not contain* UIML *code*. In this case, the file can be textual (e.g. HTML) or binary (e.g., JPEG). This case is analogous to the IMG tag in HTML; for instance:

  ```
  <constant id="Logo"
      source="http://uiml.org/UIMLLogo.jpg"/>
  ```

- *A URI to a resource that does contain* UIML *code*. The *UIML* code is inserted into the element that contains the *source* (templates rules). Inserting code has several uses as will be explained in Section 4.3.6:

  - Splitting an *UI* definition into several *UIML* documents
  - Creating a library of reusable *UI* components
  - Achieving the cascading behavior of CSS style sheets.

  The URI may either be an element in the same document where the *source* appears, or in a different document:

  - *URI names the same document*. The two elements must either have the same tag or the URI must name a template element.

    ```
    <style name="Simple"> ... </style>
    <style name="Complex" source="#Simple" how="cascade">
    ...
    </style>
    ```

  - *URI names another document*. Again, the two elements must either have the same tag or the URI must name a template element.

    ```
    <part name="Dialog"
    source=
    "http://uiml.org/templates/Dialog.uiml#SimpleDialog"
    how="replace"
    />
    ```

    A *how* attribute of *cascade* achieves behavior similar to cascading in CSS, while *replace* allows a *UIML* document to be split into multiple files.

The *export* Attribute: The *export* attribute is used in the context of templates. See Section 4.3.6 for details.

4.3.4.3 The <interface> Element

DTD

```
<!ELEMENT interface (structure|style|content|behavior)*>
<!ATTLIST interface
    id      NMTOKEN                         #IMPLIED
    source  CDATA                           #IMPLIED
    how     (union | cascade | replace)     "replace"
    export  (hidden | optional | required)  "optional">
```

Description

All *uiml* elements that describe the interface are contained in the *interface* element. The *interface* element describes an UI, *not* the interaction of the UI and the application logic. The *UIML* can employ various interface technologies (e.g., voice, graphics, and 3D).

An *interface* is composed of four elements: *structure* (see Section 4.3.4.4), *style* (see Section 4.3.4.5), *content* (see Section 4.3.4.6), and *behavior* (see Section 4.3.4.7).

VDM-SL Specification

$$\begin{aligned}
&\mathit{Interface} :: \mathit{intele} : \mathit{InterfaceElements}^{*}\\
&\qquad\qquad \mathit{id} : [\mathit{ID}]\\
&\qquad\qquad \mathit{source} : \mathit{String}\\
&\qquad\qquad \mathit{how} : [\mathit{SourcesModes}]\\
&\qquad\qquad \mathit{export} : [\mathit{ExportOptions}]\\
&\mathsf{inv}\ i \triangleq \mathit{uniqueIDs}\,(i);
\end{aligned}$$

4.3.4.4 The <structure> Element

Just as a bridge over a river is a structure that consists of many parts (e.g., steel beam, bolts), an UI consists of a structure (its organization) and many parts (e.g., widgets). [Har02].

DTD

```
<!ELEMENT structure (part*)>
<!ATTLIST structure
    id      NMTOKEN                        #IMPLIED
    source  CDATA                          #IMPLIED
    how     (union | cascade | replace)    "replace"
    export  (hidden | optional | required)  "optional">
```

Description

An application program can have an UI with one or more organizations associated with it. By "organization" we mean the set of UI widgets that are present in the interface, and the relationship of those widgets to each other when the interface is rendered. The relationship might be spatial (e.g., in a graphical UI) or temporal (e.g., in a voice interface).

The *structure* element defines the initial organization of the interface represented by the *UIML* document. This organization can be envisioned as a virtual tree of parts with each part associated content, behavior, etc. attached to it.

All interface description must include at least one structure description.

There may be more than one structure element, each representing a different organization of the interface (thus in the PC and voice interface example above, there are two *structure* elements). Each structure element is given a unique name.

If a *UIML* document contains more than one *structure* element, then a *UIML* render must select by *id* exactly one *structure* element and ignore all other *structure* elements. The *structure* element whose *id* matches the supplied *id* is then used, and all other *structure* elements are ignored. If the supplied *id* does not match the *id* attribute of any structure, or if no id is supplied, then the last <structure> element appearing in the *UIML* document must be used.

For example,

```
<structure id="ComplexUI">
    <part class="c2" id="n3">
        <part class="c1" id="n2"/>
    </part>
    </structure

<structure id="SimpleUI">
    <part class="c1" id="n1"/>
</structure>

<structure id="default">
    <part class="c1" id="n1"/>
    <part class="c2" id="n2"/>
</structure>
```

VDM-SL Specification

$$\begin{aligned} Structure :: \; & parts : Part^* \\ & id : [ID] \\ & source : String \\ & how : [SourcesModes] \\ & export : [ExportOptions] \\ \text{inv } s \triangleq \; & uniqueIDs\,(s); \end{aligned}$$

Example

In our Stack example, we can clearly see the structure element. Several part elements are present, each one with appropriate properties id and $class$.

```
<structure>
  <part id="Top" class="JFrame">
    <style>
        <property name="layout_hgap">10</property>
        <property name="layout_vgap">25</property>
    </style>
    <part id="Label" class="JLabel"/>
    <part id="ScrollPane" class="JScrollPane">
        <part id="List" class="JList"/>
    </part>
    <part id="ButtonPanel" class="JPanel">
```

```
            <part id="AddButton" class="JButton"/>
            <part id="RemoveButton" class="JButton"/>
            <part id="TopButton" class="JButton"/>
            <part id="Clear" class="JButton"/>
        </part>
       </part>
    </structure>
```

Dynamic Structure

The question remains as to how this initial virtual tree can be modified over the lifetime of the interface. Several types of dynamics exist in user interfaces. The three types that can be represented in *UIML* are described below:

- Content is dynamically supplied when the *UI* is rendered. This is handled by the *reference* element in Section 4.3.4.6.
- The virtual tree of *UI* parts is modified during the lifetime of an UI. See the *restructure* element in Section 4.3.4.7.
- The *UI* contains a sub-tree of parts that is repeated 1 or more times, where the number of times is determined at render time. This is the purpose of the *repeat* element (see Section 4.3.4.7).

The <part> Element

DTD

```
<!ELEMENT part (style?, content?, behavior?, part*, repeat*)>
<!ATTLIST part
    id          NMTOKEN                         #IMPLIED
    source      CDATA                           #IMPLIED
    how         (union | cascade | replace)     "replace"
    export      (hidden | optional | required)  "optional"
    class       NMTOKEN                         #IMPLIED
    where       (first|last|before|after)       "last"
    where-part  NMTOKEN                         #IMPLIED>
```

Description

Each *part* element corresponds either to one *UI* widget or to nothing (null). It is sometimes useful to associate a part with nothing; for example, a part might be needed for a large screen *UI*, but be omitted from a small device screen. In the former case, the part corresponds to an *UI* widget, and in the later case the part corresponds to nothing.

Parts may be nested to represent an hierarchical relationship of parts. Let *a* and *b* denote two part elements. If part *b* is nested inside part *a*, and both *a* and *b* correspond to *UI* widgets (i.e., neither *a* nor *b* correspond to null), then the *UI* widget *b* must be "contained in" widget *a*, where "contained in" is defined in terms of the *UI* toolkit.

For example, the *Java Swing* [Mic03e] toolkit has a notion of containers and components. Containers contain other containers or components, forming a hierarchy. Or, in a voice-based language, the oral equivalent of menus can be nested, again forming a hierarchy.

Each part must be associated with a single class. However, if multiple *structure* elements exist, then a part can be associated with a different class in each structure (see the example in 4.3.4.4). When the interface is rendered, only one structure is used (as discussed in "Description" under 4.3.4.4); thus, a part is always associated with a unique class.

UIML allows the style, content, and behavior information associated with a particular part to be specified within the part itself. Usually, this information is specified in the corresponding *style*, *content*, and *behavior* elements.

VDM-SL Specification

$$
\begin{array}{ll}
\mathit{Part} :: & \mathit{style} : [\mathit{Style}] \\
& \mathit{content} : [\mathit{Content}] \\
& \mathit{behavior} : [\mathit{Behavior}] \\
& \mathit{parts} : \mathit{Part}^{*} \\
& \mathit{repeats} : \mathit{Repeat}^{*} \\
& \mathit{id} : [\mathit{ID}] \\
& \mathit{source} : \mathit{String} \\
& \mathit{how} : [\mathit{SourcesModes}] \\
& \mathit{export} : [\mathit{ExportOptions}] \\
& \mathit{class} : \mathit{String} \\
& \mathit{where} : [\mathit{WhereOptions}] \\
& \mathit{where\text{-}part} : \mathit{String} \\
\mathsf{inv}\ p \triangleq \mathit{uniqueIDs}\,(p);
\end{array}
$$

Example

Below we can see that there is a *style* member of *part* $id =$ "Top". Also we can see that the *part* $id =$ "ButtonPanel" has several parts inside it.

```
<part id="Top" class="JFrame">
  <style>
      <property name="layout_hgap">10</property>
      <property name="layout_vgap">25</property>
  </style>
  <part id="Label" class="JLabel"/>
  <part id="ScrollPane" class="JScrollPane">
      <part id="List" class="JList"/>
  </part>
  <part id="ButtonPanel" class="JPanel">
      <part id="AddButton"    class="JButton"/>
      <part id="RemoveButton" class="JButton"/>
      <part id="TopButton"    class="JButton"/>
      <part id="Clear"        class="JButton"/>
  </part>
 </part>
```

4.3.4.5 The <style> Element

DTD

```
<!ELEMENT style (property*)>
<!ATTLIST style
    id      NMTOKEN                         #IMPLIED
    source  CDATA                           #IMPLIED
    how     (union | cascade | replace)     "replace"
    export  (hidden | optional | required)  "optional">
```

Description

The *style* element contains a list of properties and values that are used to render the interface. Such as in *CSS* [Rec03a] and *XSL* [Rec01a] specifications, *UIML* properties specify attributes of how the interface will be rendered on various devices, such as fonts, colors, layout, and so on.

There must be at least one *style* element, and there may be more than one. There is normally one *style* element for each toolkit to which the *UIML* document will be mapped. For a given toolkit, there may be multiple *style* elements serving a variety of purposes: to generate different interface presentations for accessibility, to support a family of similar but not identical devices (e.g., phones that differ in the number of characters that their displays support), to support different target audiences (e.g., children versus adults), and so on.

Style sheets may also use the mechanism for cascading (deeply described in [AH02]). However, unlike *CSS* and *XSL*, the style sheet is used to achieve device independence.

VDM-SL Specification

$$
\begin{array}{l}
\mathit{Style} :: \mathit{property} : \mathit{Property}^{*} \\
\qquad\quad \mathit{id} : [\mathit{ID}] \\
\qquad\quad \mathit{source} : \mathit{String} \\
\qquad\quad \mathit{how} : [\mathit{SourcesModes}] \\
\qquad\quad \mathit{export} : [\mathit{ExportOptions}] \\
\textsf{inv}\ s \triangleq \mathit{uniqueIDs}\,(s);
\end{array}
$$

Example

In this *UIML* fragment of our Stack specification, we can see more than one property as members of *style* element. For instance, in this example the single part named "Top" has size 380×230.

```
<style>
      <property part-name="Top" name="size">380,230</property>
      <property part-name="Top" name="location">100,100</property>
      <property part-name="Top" name="layout">javax.swing.BoxLayoutY</property>
      <property part-name="Top" name="title">Stack manipulation</property>
      ....
      <property part-name="List" name="content">
        <constant model="list">
          <constant id="1"  value="1"/>
          <constant id="2"  value="2"/>
          <constant id="3"  value="3"/>
          <constant id="4"  value="4"/>
        </constant>
      </property>
</style>
```

In the next *UIML* fragment, we can see more than one property as members of *style* element. In this example with property *name* instantiated with Java properties $layout\_hgap$ and $layout\_vgap$ of Java Frame class.

```
        <style>
            <property name="layout_hgap">10</property>
            <property name="layout_vgap">25</property>
        </style>
```

The <property> Element

DTD

```
<!ELEMENT property (#PCDATA|constant|property|reference|call|iterator)*>
<!ATTLIST property
    name        NMTOKEN                         #IMPLIED
    source      CDATA                           #IMPLIED
    how         (union | cascade | replace)     "replace"
    export      (hidden | optional | required)  "optional"
    part-name   NMTOKEN                         #IMPLIED
    part-class  NMTOKEN                         #IMPLIED
    event-name  NMTOKEN                         #IMPLIED
    event-class NMTOKEN                         #IMPLIED
>
```

Description

A property associates a name and value pair with a *part*, *event* (see Section 4.3.4.7), or *call* (see Section 4.3.4.7). For example, an UI part named “button” might be associated with a property name “color” and value ”blue”. The *property* element provides the syntax to make the association between the name *color* and value *blue* with the part *button*.

Property names are not defined by the *UIML* specification. This is a powerful concept, because it permits *UIML* to be extensible: one can define whatever property names are appropriate for a particular device. For example, a “color” might be a useful property name for a device with a screen, while “loudness” might be appropriate for a voice-based device.

Instead property names are defined by the *peers* element (see Section 4.3.5.1). Normally whoever creates a *UIML* document does not define the property names. Instead, someone else defines a set of properties. For example for the $Java\ AWT\ UI$

toolkit, the *peers* element simply specifies an *URI* that defines those property names. The compiler or interpreter that renders *UIML* should also access this *URI* to map property names in the *UIML* document to the desired UI toolkit.

Thus to use *UIML* one needs both a copy of this specification and a document defining the property names used in a particular *peers* element.

VDM-SL Specification

$$\begin{aligned}
\mathit{Property} :: \; & \mathit{property} : (\mathit{String} \mid \mathit{Constant} \mid \mathit{Property} \mid \mathit{Reference} \mid \mathit{Call} \mid \mathit{Iterator})^* \\
& \mathit{name} : [\mathit{String}] \\
& \mathit{source} : [\mathit{String}] \\
& \mathit{how} : [\mathit{SourcesModes}] \\
& \mathit{export} : [\mathit{ExportOptions}] \\
& \mathit{p\text{-}name} : [\mathit{String}] \\
& \mathit{p\text{-}class} : [\mathit{String}] \\
& \mathit{e\text{-}name} : [\mathit{String}] \\
& \mathit{e\text{-}class} : [\mathit{String}] \\
\mathsf{inv}\ p \; \triangleq \; & \mathit{uniqueIDs}\,(p);
\end{aligned}$$

Example

The excerpt below of our case study shows the sequence of *Constant* elements members of *Property*.

```
<property part-name="List" name="content">
   <constant model="list">
     <constant id="1"  value="1"/>
     <constant id="2"  value="2"/>
     <constant id="3"  value="3"/>
     <constant id="4"  value="4"/>
   </constant>
</property>
```

In the example below of *UIML* property, the *name* attribute is instantiated with $layout\_vgap$ and value 25.

```
<property name="layout_vgap">25</property>
```

4.3.4.6 The <content> Element

DTD

```
<!ELEMENT content (constant*)>
<!ATTLIST content
    id          NMTOKEN                         #IMPLIED
    source      CDATA                           #IMPLIED
    how         (union | cascade | replace)     "replace"
    export      (hidden | optional | required)  "optional">
```

Description

A *part* in an UI can be associated with various contents such as words, characters, sounds or images. *UIML* permits separation of the content from the structure in an UI. Separation is useful when different contents should be displayed under different circumstances. For example, an UI might display the content in English or Portuguese. Or an UI might use different words for an expert versus a novice user, or different icons for a color-blind user. *UIML* can express this.

Normally one would set the content associated with an UI part through the *property* element, as in the following example:

```
<structure>
    ....
    <part id="AddButton" class="JButton"/>
    ....
</structure>
<style>
    ....
    <property part-name="AddButton" name="text">
    Push
    </property>
    ....
</style>
```

In the *UIML* Stack fragment above, the *JButton* text is hard-wired to the string “Push”. Suppose we wanted to internationalize the interface. In this case *UIML* allows the value of a property to be what a programmer would think of as a variable reference using the *Reference* element:

```
<style>
    <property part-name="AddButton" name="text">
        <reference constant-name="AddButtonText"/>
    </property>
</style>
```

The *reference* element refers to a constant-name, which is defined in the *content* element in a *UIML* document. The important fact is that there may be multiple *content* elements in a *UIML* document, each with a different name. When the interface is rendered, one of the *content* elements is specified, and the *content* elements inside are then used to satisfy the *referenced* element.

VDM-SL Specification

$$
\begin{array}{l}
Content :: constant : Constant^{*} \\
\qquad\qquad id : ID \\
\qquad\qquad source : String \\
\qquad\qquad how : [SourcesModes] \\
\qquad\qquad export : [ExportOptions] \\
\mathsf{inv}\ c \triangleq uniqueIDs\,(c);
\end{array}
$$

The <constant> Element

DTD

```
<!ELEMENT constant (constant)*>
<!ATTLIST constant
    id          NMTOKEN                          #IMPLIED
    source      CDATA                            #IMPLIED
    how         (union | cascade | replace)      "replace"
    export      (hidden | optional | required)  "optional"
    model       Model                            #IMPLIED
    value       Value                            #IMPLIED>
```

Description

Constant elements contain the actual text strings, sounds, and images associated with UI parts from the *part* element. Each *constant* element is identified by an *id* attribute and is referred to by the *reference* element.

The following example shows how to create constant elements that point to a string. Similarly, you can create constants that point to images, video clips, binary files, and other objects.

```
<constant model="list">
      <constant id="1"  value="1"/>
      <constant id="2"  value="2"/>
      <constant id="3"  value="3"/>
      <constant id="4"  value="4"/>
</constant>
```

The *constant* element can also be used to represent literal strings used inside the *condition* element (see Section 4.3.4.7). For example:

```
<condition>
    <equal>
        <event part-name="inYear" class="filled"
               name="content"/>
        <constant>2000</constant>
    </equal>
</condition>
```

VDM-SL Specification

$$\begin{aligned} Constant :: \; & constant : (Constant)^* \\ & id : [ID] \\ & source : String \\ & how : [SourcesModes] \\ & export : [ExportOptions] \\ & model : String \\ & value : String \end{aligned}$$

$\text{inv } c \triangleq uniqueIDs\,(c);$

The <reference> Element

DTD

```
<!ELEMENT reference EMPTY>
<!ATTLIST reference
    constant-name   NMTOKEN     #IMPLIED
    url-name        NMTOKEN     #IMPLIED>
```

Description

The *reference* element refers to the value of the *constant* element specified by the *constant-name* attribute.

There are several uses for references:

- The same text string might be used in two or more places in a *UIML* document. In this case a *constant* element can be defined containing the string and anywhere the string is required (e.g., as values of a property) the *reference* element can be used. Thus, if we can modify the text in the constant element, the change propagates to all the places in the *UIML* document that is referred to.
- Often an interface part is initialized to contain several text strings, and when an event later occurs for the part, an equal element tests to see which text string the end user selected in triggering the event (for example, lists and choices in *Java SWING* contain multiple text items). In this case, a *constant* element can be defined in the content section, and then the part's values can be initialized in the style section using a property element containing a *reference* element as its value. In the behavior element, the rule element handling events for the part can test whether the item selected corresponded to the *constant* element by using a *reference* element. An example of this appears in Section 4.2.2.

The semantics of a *reference* element is to replace the element with the *constant* element whose *id* attribute matches the *constant-name* attribute of the *reference* element. Or, if the *url-name* attribute is specified, to replace the *constant* element contained within the document located by the URI given as the value of the url-name attribute. If no such element exists, then the *UIML* document cannot be rendered.

VDM-SL Specification

$$\begin{aligned} Reference :: \ & constant\text{-}name : String \\ & url\text{-}name : String \end{aligned}$$

4.3.4.7 The <behavior> Element

DTD

```
<!ELEMENT behavior (rule*)>
<!ATTLIST behavior
    id          NMTOKEN                         #IMPLIED
    source      CDATA                           #IMPLIED
    how         (union | cascade | replace)     "replace"
    export      (hidden | optional | required)  "optional">
```

Description

The *behavior* element describes what happens when an end-user interacts with a user interface. For example, the *behavior* element might describe what happens when an end-user presses a button. The *behavior* element also describes when and how the user interface invokes methods (recall from Section 4.1.1 that a *method* refers to functions, procedures, database queries, and so on.)

The *behavior* element contains a sequence of rules. Each rule contains a condition and a list of actions. Whenever a condition holds, the associated action is performed. If conditions for more than one rule hold simultaneously, there is an algorithm to determine the order of execution, as follows.

- *UIML* allows two types of conditions. The first is true when an event occurs (e.g., a button is pressed). The second is true when an event occurs and the value of some data associated with the event is equal to a certain value (e.g., a list selection is made and the selected item is “1”, the first condition element in the stack example in Section 4.2.2.
- Actions can be *internal* to the *UIML* document – specifying a change in a property value – or *external* – invoking a method in a script, program, or object.

A unique aspect of *UIML* is that events are also described in a device-independent fashion, by giving each event a name and identifying the class to which it belongs. As we discussed for parts, the *UI* implementor uses instance and class names of his/her choice for events, and those names are mapped to an event in the underlying platform in the *style* element. For example, the end user might use the class “selection” and the *style* element for a graphical *UI* maps “selection” to a “mouse click” event.

In *UIML* you can specify the following behavior:

- Assign a value to a part property.
- Call an external function or method.
- Fire an event.

VDM-SL Specification

$$\begin{aligned}
&\mathit{Behavior} :: \mathit{rules} : \mathit{Rule}^{*}\\
&\qquad\qquad \mathit{id} : [\mathit{ID}]\\
&\qquad\qquad \mathit{source} : \mathit{String}\\
&\qquad\qquad \mathit{how} : [\mathit{SourcesModes}]\\
&\qquad\qquad \mathit{export} : [\mathit{ExportOptions}]\\
&\textsf{inv}\ b \triangleq \mathit{uniqueIDs}\,(b);
\end{aligned}$$

Example

Next example shows the *behavior* of button *AddButton*. The action performed will be *addElement* once it is pressed.

```
<behavior>
      <rule>
        <condition>
          <event part-name="AddButton" class="actionPerformed"/>
        </condition>
        <action>
          <call name="stack.addElement"/>
        </action>
      </rule>
      ...
</behavior>
```

The <rule> Element

DTD

```
<!ELEMENT rule (condition,action)?>
<!ATTLIST rule
    id          NMTOKEN                         #IMPLIED
    source      CDATA                           #IMPLIED
    how         (union | cascade | replace)     "replace"
    export      (hidden | optional | required)  "optional">
```

Description

The *rule* element defines a binding between a *condition* element and an *action* element. Whenever the *condition* element within the rule is satisfied, then any elements inside the *action* element are executed sequentially (i.e., property assignment, external function or method call, or event firing).

VDM-SL Specification

$$\begin{aligned}
&\mathit{Rule} :: \mathit{condition} : \mathit{Condition}\\
&\qquad\quad \mathit{action} : \mathit{Action}\\
&\qquad\quad \mathit{id} : [\mathit{ID}]\\
&\qquad\quad \mathit{source} : \mathit{String}\\
&\qquad\quad \mathit{how} : [\mathit{SourcesModes}]\\
&\qquad\quad \mathit{export} : [\mathit{ExportOptions}]\\
&\textsf{inv}\ r \triangleq \mathit{uniqueIDs}\,(r);
\end{aligned}$$

Example

Continuing the previous example, there is a *rule* for each event which we need to support. As written before, the *action addElement*, inside Java module *stack.java*, will be executed when the button $AddButton$ is pressed.

```
<rule>
  <condition>
    <event part-name="AddButton" class="actionPerformed"/>
  </condition>
  <action>
    <call name="stack.addElement"/>
  </action>
</rule>
```

The <condition> Element

DTD

```
<!ELEMENT condition (equal|event|op)>
```

Description

The *condition* element contains as a child either an *event* element or a Boolean expression. The *action* element associated with this *condition* by the parent *rule* element is executed whenever either the event named in the *event* element fires or the Boolean expression in the *equal* or *op* evaluates to true.

VDM-SL Specification

$$Condition :: type : (Equal \mid Event \mid Op);$$

Example

This example shows the *condition* to execute some *action*. Everything happens during *event contentsChanged* supported on List *part*. Afterwards the *action* to take is described. In this example, the text of part Label changes to “An element was Pushed.”.

```
<condition>
      <event part-name="List" class="contentsChanged"/>
</condition>
<action>
      <property part-name="Label" name="text">
      An element was Pushed.
      </property>
</action>
```

The <equal> Element

DTD

```
<!ELEMENT equal (event,(constant|property|reference|op))>
```

Description

The *equal* element is a Boolean expression with value true or false. Every *equal* element must have exactly two children. Each child must be a *constant*, *property*, *reference*, or *op* element. The semantics of *equal* are as follows. Whenever the two children named in the *equal* element resolve to the same value then the *equal* element has value true. Otherwise the *equal* element has value false.

VDM-SL Specification

$$\begin{aligned} &\mathit{Equal} :: \mathit{event} : \mathit{Event} \\ &\qquad\quad \mathit{other} : \mathit{Constant} \mid \mathit{Property} \mid \mathit{Reference} \mid \mathit{Op} \\ &\textsf{inv}\ e \triangleq \mathit{uniqueIDs}\,(e); \end{aligned}$$

The <event> Element

DTD

```
<!ELEMENT event EMPTY>
<!ATTLIST event
    name        NMTOKEN     #IMPLIED
    part-name   NMTOKEN     #IMPLIED
    part-class  NMTOKEN     #IMPLIED
    class       NMTOKEN     #IMPLIED>
```

Description

The *event* element is used in two contexts:

- As the child of a *condition* element. The parent condition is satisfied whenever the event occurs.
- As the child of an action element. The event is fired.

VDM-SL Specification

$$\begin{aligned} &\mathit{Event} :: \mathit{name} : \mathit{String} \\ &\qquad\quad \mathit{p\text{-}name} : \mathit{String} \\ &\qquad\quad \mathit{p\text{-}class} : \mathit{String} \\ &\qquad\quad \mathit{class} : \mathit{String} \end{aligned}$$

Example

This example shows the declaration of *event contentsChanged*. This element is part of a *condition* element (as described before). This *event* occurs when any changes happen on List items.

```
    <rule>
        <condition>
          <event part-name="List" class="contentsChanged"/>
        </condition>
        <action>
          <property part-name="Label" name="text">
          Contents changed.
          </property>
        </action>
     </rule>
```

The <op> Element

DTD

```
    <!ELEMENT op (constant|property|reference|call|op|event)*>
    <!ATTLIST op
        name    CDATA       #REQUIRED>
```

Description

The *op* element allows multiple complex logic conditions to be expressed in $UIML$. In the previous examples simple conditions were used to control whether or not elements under *action* have been executed. The simplicity of the previous examples allows for only one condition to hold true, usually this was reserved to check if a particular *event* has occurred. Even with the functionality introduced with the *equal* tag, an author can only evaluate two different conditions; furthermore, the *equal* tag provides a limited logical condition, testing only if two values are equal. However, with the *op* element, basic logical conditions (less than, greater than, equal, not equal, and, or) may be expressed along with the ability to structure complex condition statements involving multiple values.

The name attribute of *op* describes the conditional applied to the expression that you wish to create. The value of the name attribute can either be symbols representing operators or the written name of the operator itself.

The introduction of *op* element brings upon instances where certain actions may want to be defined as a result from the evaluation of the *op* element. Three new elements *when-true*, *when-false*, *by-default* have been introduced to define a set of actions when a conditional is found to be true, false, or undefined. The elements are found as children of *action*.

VDM-SL Specification

$$
\begin{aligned}
&Op :: type : (Constant \mid Property \mid Reference \mid Call \mid Op \mid Event)^{*} \\
&\qquad name : String \\
&\text{inv } op \triangleq uniqueIDs\,(op);
\end{aligned}
$$

The <action> Element

DTD

```
<!ELEMENT action (((property|call|restructure)*, event?)|(when-true?,
when-false?,by-default?))>
```

Description

Action elements contain one or more elements that are executed in the order they appear in the *UIML* document. Each element can be either a *property* element to set a property of an element, a *call* element, which invokes code (e.g., a function or method), a *restructure* element to restructure an interface (see <restructure>, page 102), an *event* element to fire another event, or a *when-true*, *when-false*, *by-default* element to determine a set course of actions depending on the value of the conditional expressed in the *condition* element (see <bydefault>, page 105, <whentrue>, page 104 and <whenfalse>, page 104). The *event* element, if present, must be the last element inside the action. As result of this, you can only fire one event within the *action* element.

VDM-SL Specification

$$\begin{aligned}
ActionType1 &:: type : (Property \mid Call \mid Restructure)^{*} \\
&\quad\; event : [Event] \\
ActionType2 &:: whentrue : [When\text{-}true] \\
&\quad\; whenfalse : [When\text{-}false] \\
&\quad\; bydefault : [By\text{-}default] \\
Action &= ActionType1 \mid ActionType2 \\
\textsf{inv}\ a &\triangleq uniqueIDs\,(a);
\end{aligned}$$

Example

The two following rule examples show the *action* performed under a particular *condition*. The former shows the internal execution of some operation (change text *Label*). The second one shows the activation by a *call* element of an external method or function (*addElement* in class stack) (see <call>, page 100).

```
<rule>
    <condition>
      <event part-name="List" class="contentsChanged"/>
    </condition>
    <action>
      <property part-name="Label" name="text">
      Contents changed.
      </property>
    </action>
</rule>

<rule>
    <condition>
```

```
          <event part-name="AddButton" class="actionPerformed"/>
        </condition>
        <action>
          <call name="stack.addElement"/>
        </action>
    </rule>
```

The <call> Element

DTD

```
<!ELEMENT call (param*)>
<!ATTLIST call
    name        NMTOKEN     #IMPLIED>
```

Description

The *call* element is an abstraction of any type of code invocation (that uses a language other than $UIML$). The code is referred to in this specification as a method, which in Section 4.1.1 is defined to include functions, procedures, and methods in an object-oriented language, database queries, and directory accesses.

The $UIML$ philosophy on specifying function calls is to allow the *UIML* author to freely choose a set of names for widgets, events, and functions referred to in the <interface> section. Each of these names is then mapped in the *peers* section to implementing entities (e.g., Swing user-interface components, methods in memory or remote object instances, entry points in remote procedures, functions in user scripts, etc.).

VDM-SL Specification

$$\begin{aligned} Call :: \; & params : Param^* \\ & name : String \end{aligned}$$

Example

We can see in the example bellow the calling of an external method *addElement*, of stack Java class. If the method or function demands values as arguments, the *param* element allows for this (see 4.3.4.7).

```
        <action>
          <call name="stack.addElement"/>
        </action>
```

The <repeat> Element

DTD

```
    <!ELEMENT repeat (iterator,part*)>
```

Description

A *repeat* element must enclose one *iterator* element and a set of one or more *part* elements. The *part* elements denoted as children of the *repeat* element are repeated with their children a number of times as determined by the *iterator* element. The *repeat* element parent *part* element will not be repeated.

A *repeat* element has the following legal children, ordering of the children does not matter:

- Each *repeat* element must have one and only one *iterator* child. The *iterator* element denotes how many times the specified interface components will be repeated. If more than one *iterator* child is defined than the implementation must produce a warning and use the last *iterator* defined in textual order.
- Each *repeat* must have one or more *part* elements as children. These *part* elements represent the components to be repeated. If the components are named (i.e. have a defined *name* attribute), then each repetition of the *part* will have "_#" appended onto the part name, where # is the integer representation of this iteration.

Nested repeats are allowed, meaning that a *repeat* can be a child of another *part* element descendant of *repeat* (not just the first level children). This allows for the dynamic construction of more complicated interface elements such as tables and static depth trees.

VDM-SL Specification

$$\begin{aligned} Repeat :: \; & iterator : Iterator \\ & parts : Part^{*} \end{aligned}$$

The <iterator> Element

DTD

```
<!ELEMENT iterator (#PCDATA|constant|property|call)>
<!ATTLIST iterator
    id      NMTOKEN     #REQUIRED>
```

Description

The *iterator* element defines the number of times the interface components should be repeated. *Iterator* elements can have only one child, which can be of four forms:

- A text string
- A *call* element
- A *property* element

- A *constant* element

The form of the child is irrelevant as long as it resolves to an integer value N. This integer N is then used as the maximum number of iterations that the repeat will perform, counting from 1 to N. Note that the step value for an *iterator* element is currently always 1.

The *iterator* element can be used in *property* and *param* elements to provide an integer value representing the iteration number that is currently processing. In this way, the *iterator* element behaves very similarly to the *property* element.

It is important to note that an *iterator* is defined within the scope of the *repeat* it is a child of. Thus, no other *iterator* elements may have the same *id* if they are defined within a descendent of the current *repeat*. This also implies that an *iterator* whose *repeat* is an ancestor of another *iterator* can be accessed within the scope of the descendent *iterator*.

VDM-SL Specification

$$
\begin{array}{l}
\mathit{IteratorOptions} = \mathit{String} \mid \mathit{Constant} \mid \mathit{Property} \mid \mathit{Call};\\
\mathit{Iterator} :: \mathit{iterator} : \mathit{IteratorOptions}\\
\qquad\qquad\quad \mathit{id} : \mathit{ID}\\
\mathsf{inv}\ \mathit{iter} \triangleq \mathit{uniqueIDs}\,(\mathit{iter});
\end{array}
$$

Example

```
<uiml>

    <part class=JDialog>
        <repeat>
            <iterator id=i>10</iterator>
            <part class = JCheckBox>
                <style>
                    <property name=text>
                        <iterator id=i/>
                    </property>
                </style>
            </part>
        </repeat>
    </part>

</uiml>
```

The example above illustrates the two uses of the *iterator* element and would result in the appearance of a *JDialog* containing ten *JCheckBoxes*. The *JCheckBoxes* would be numbered 1 to 10.

The <restructure> Element

DTD

```
<!ELEMENT restructure (template)?>
<!ATTLIST restructure
    at-part     NMTOKEN                          #IMPLIED
    how         (union|cascade|replace|delete)  replace
    where       (first|last|before|after)       last
    where-part  NMTOKEN                          #IMPLIED
    source      CDATA                            #IMPLIED>
```

Description The *restructure* element provides a way for the *UI* to change as a result

of some condition being met. Most conditions include (but are not limited to) user interactions. For example, using the Java AWT/Swing vocabulary for *UIML*, an *UI* containing a window with a button and a panel is described as follows:

```
<structure>
    <part class="JFrame" name="F">
        <part class="JButton" name="B"/>
        <part class="JPanel" name="A"/>
    </part>
</structure>
```

Suppose that, when the initial *UI* is displayed, we want only the button to appear. When the user clicks the button, the panel appears. We would use the *restructure* element to define the necessary changes within the *UI* to remove the button and display the panel.

The semantics of *UIML* are changed to include the concept of a virtual *UI* tree. During the lifetime of an *UI*, the parts comprising the *UI* may change. (All parts that exist but are invisible to an end user are still part of the tree.) The parts present in the *UI* have a hierarchical relationship, therefore forming a tree. At any moment during the *UI* lifetime, one could enumerate the tree of parts that currently exist, and this is the virtual *UI* tree. Each node in this tree corresponds to a *part* element in the *UI* generated by *UIML*. We call the tree "virtual" because it may or may not be physically represented as a data structure on a computer, depending on how a rendering engine is implemented.

UIML adds nodes using the *restructure* tag. (The *restructure* tag is so-named because it modifies the *structure* section's representation in the virtual *UI* tree.) The *restructure* tag can only appear inside an *action* element in *UIML*.

The *restructure* element may not contain a body if one of the following holds:

- The source attribute is present
- how="delete" is present

Otherwise, the *restructure* element must contain a body, and that body must contain exactly one *template* element, which must contain exactly one *part* element that matches the part specified in the at-part attribute.

VDM-SL Specification

$$\begin{aligned}
&\mathit{Restructure} :: \mathit{template} : [\mathit{Template}]\\
&\qquad \mathit{at\text{-}part} : \mathit{String}\\
&\qquad \mathit{how} : \text{UNION} \mid \text{CASCADE} \mid \text{REPLACE} \mid \text{DELETE}\\
&\qquad \mathit{where} : [\mathit{WhereOptions}]\\
&\qquad \mathit{where\text{-}part} : \mathit{String}\\
&\qquad \mathit{source} : \mathit{String}\\
&\mathsf{inv}\ r \triangleq \mathit{uniqueIDs}\,(r);
\end{aligned}$$

The <when-true> Element

DTD

```
<!ELEMENT when-true ((property|call)*,restructure?,op?,equal?,event?)>
```

Description

The *when-true* element defines a set of actions to be executed when a conditional expression defined by the element *op* evaluates to true.

VDM-SL Specification

$$\begin{aligned}
&\mathit{When\text{-}true} :: \mathit{type} : (\mathit{Property} \mid \mathit{Call})^*\\
&\qquad \mathit{restructure} : [\mathit{Restructure}]\\
&\qquad \mathit{op} : [\mathit{Op}]\\
&\qquad \mathit{equal} : [\mathit{Equal}]\\
&\qquad \mathit{event} : [\mathit{Event}]
\end{aligned}$$

The <when-false> Element

DTD

```
<!ELEMENT when-false ((property|call)*,restructure?,op?,equal?,event?)>
```

Description

The *when-false* element defines a set of actions to be executed when a conditional expression defined by the element *op* evaluates to false.

VDM-SL Specification

$$\begin{aligned}
&\mathit{When\text{-}false} :: \mathit{type} : (\mathit{Property} \mid \mathit{Call})^*\\
&\qquad \mathit{restructure} : [\mathit{Restructure}]\\
&\qquad \mathit{op} : [\mathit{Op}]\\
&\qquad \mathit{equal} : [\mathit{Equal}]\\
&\qquad \mathit{event} : [\mathit{Event}]
\end{aligned}$$

The <by-default> Element

DTD

```
<!ELEMENT by-default ((property|call)*,restructure?,op?,equal?,event?)>
```

Description

The *by-default* element defines a set of actions to be executed when the evaluation of a conditional expression defined by the element *op* results to be undefined.

VDM-SL Specification

$$\begin{aligned} \textit{By-default} :: \ & \textit{type} : (\textit{Property} \mid \textit{Call})^{*} \\ & \textit{restructure} : [\textit{Restructure}] \\ & \textit{op} : [\textit{Op}] \\ & \textit{equal} : [\textit{Equal}] \\ & \textit{event} : [\textit{Event}] \end{aligned}$$

The <param> Element

DTD

```
<!ELEMENT param
(#PCDATA|property|reference|call|op|event|constant|iterator)>
<!ATTLIST param
    name        NMTOKEN             #IMPLIED>
```

Description

This element describes a single parameter of the call described by the parent *call* element. Note that all parameters are character strings. See <dparam>, pag 114, for details on the conversion of the arguments to the types required by the formal parameters of the method being called.

If the number of *param* elements equals the number of formal parameters in the method being called, then the following hold:

- The name attribute is optional, and is ignored by the rendering engine if present.
- The order of *param* elements within the *call* element must match the order of the formal parameters in the method being called.

Otherwise, there must be fewer *param* elements than formal parameters in the method being called, and the following holds:

- The name attribute is required on all param elements.

- The name attribute must be used by the rendering engine to match each *param* element to a formal parameter in the method being called.

A *param* element must have exactly one child.

VDM-SL Specification

$$\begin{aligned} &ParamType = String \mid Property \mid Reference \mid Call \mid Op \mid Event \mid \\ &\qquad\qquad\quad Constant \mid Iterator; \\ &Param :: type : ParamType \\ &\qquad\quad\; name : String \end{aligned}$$

Example

Our stack example is a simple one and does not illustrates parameter utilization. The value added by the *Push* method is automatically generated. Suppose that *Push* method has a value argument *ele* (this value must be inserted from the interface), which must be added to the top of stack. This assumption can be specified by using the following *UIML* code. We denote the presence of a *param* element in the *call* element named *ele*. This value comes from *part TextField* of the interface.

```
    <rule>
        <condition>
          <event part-name="AddButton" class="actionPerformed"/>
        </condition>
        <action>
          <call name="stack.addElement">
            <param name="ele">
              <property part-name="TextField" name="text"/>
            </param>
          </call>
          <property part-name="TextField" name="text"></property>
        </action>
      </rule>
```

4.3.5 Peer Components

These are *UIML* property values to specific tags or objects in the target platform:

```
<peers>
    <presentation name="Java-AWT">
        <component name="MenuItem" maps-to="java.awt.MenuItem">
            ...
    </presentation>
    ...
<peers>
```

In summary, *UIML* uses three levels of names for interface parts and events. The first is chosen by the *UIML* author. The second name is in the *style* element and maps the mnemonic to an abstract widget name (e.g., MenuItem). The second level allows a mapping from one abstract set of names (e.g., BigWindow) to multiple platforms (e.g., *MFC* or *Java Swing*) without modifying the rest of the interface description. Finally, the third name in the *peers* element is part of a toolkit-specific vocabulary and maps the abstract widget name to a name of a widget from the target platform (e.g., java.awt.TextField).

4.3.5.1 The <peers> Element

DTD

```
<!ELEMENT peers (presentation|logic)*>
<!ATTLIST peers
    id          NMTOKEN                          #IMPLIED
    source      CDATA                            #IMPLIED
    how         (union | cascade | replace)      "replace"
    export      (hidden | optional | required)  "optional">
```

Description

To allow extensibility, *UIML* includes a *peers* element that defines mappings from class *UIML*:

- The *presentation* element contains mappings of part and event classes, property names, and event names to an *UI* toolkit. This mapping defines a vocabulary to be used with a *UIML* document, such as a vocabulary of classes and names for *VoiceXML*. Normally, a *UIML* author does not write a *presentation* element, but instead uses names in a *UIML* document that have been defined in the list of vocabularies at *http://uiml.org/toolkits* (in element <presentation>, page 108, we discuss vocabularies).

- The *logic* element maps names and classes used in *call* elements to application logic external to the *UIML* document. In large-scale software development, the *logic* element is defined once to represent the API of the application logic (typically as a *template* element), and then included in each *UI* for the project.

The following excerpt of *UIML* code tries to show this.

```
<peers>
    <presentation
    name="Java" source="http://uiml.org/toolkits/Java20Swing.ui"/>
    <presentation
    name="wml" source="http://uiml.org/toolkits/wml.ui"/>
        ...
        <logic
        name="Java" source="http://uiml.org/apps/CalendarApp.logic"/>
        <logic name="Scripts"
        source="http://uiml.org/apps/scripts/CalendarApp.logic"/>
</peers>
```

VDM-SL Specification

$$\begin{array}{l}
Peers :: prelog : (Presentation \mid Logic)^{*} \\
\qquad\qquad id : [ID] \\
\qquad\qquad source : String \\
\qquad\qquad how : [SourcesModes] \\
\qquad\qquad export : [ExportOptions] \\
\textsf{inv}\ p \triangleq uniqueIDs\,(p);
\end{array}$$

Example

In our example we need a *peers* component to map all evoked external methods: *removeElement, addElement, Top and Clear*. As we are working with Java Swing components, and using an Harmonia Java *UIML* render release, the presentation refers $Java\_1.3\_Harmonia\_1.0$.

```
    <peers>
        <logic>
            <component id="stack" maps-to="stack">
                <method id="removeElement" maps-to="removeTopElement"/>
                <method id="addElement" maps-to="Push"/>
                <method id="Top" maps-to="Top"/>
                <method id="Clear" maps-to="Clear"/>
            </component>
        </logic>
        <presentation base="Java_1.3_Harmonia_1.0"/>
    </peers>
```

4.3.5.2 The <presentation> Element

DTD

```
<!ELEMENT presentation (d-class*)>
<!ATTLIST presentation
    id          NMTOKEN                         #IMPLIED
    source      CDATA                           #IMPLIED
    how         (union | cascade | replace)     "replace"
    export      (hidden | optional | required)  "optional"
    base        CDATA                           #REQUIRED>
```

Description

Every *UIML* document uses a *vocabulary*. The vocabulary defines the legal class names that can be used for parts and events in a *UIML* document, as well as the legal property names. The formal definition of a vocabulary is done through a *presentation* element containing $d\text{-}class$ elements (element <d-class>, page 111). Each $d\text{-}class$ element defines a legal class name.

At present, the list of standard vocabularies is posted on *http://uiml.org/toolkits*, in the form of a set of *presentation* templates that may be included into *UIML* documents.

VDM-SL Specification

$$
\begin{aligned}
\mathit{Presentation} :: \ & \mathit{dclass} : \mathit{D\text{-}class}^* \\
& \mathit{id} : [\mathit{ID}] \\
& \mathit{source} : \mathit{String} \\
& \mathit{how} : [\mathit{SourcesModes}] \\
& \mathit{export} : [\mathit{ExportOptions}] \\
& \mathit{base} : \mathit{String} \\
\mathsf{inv}\ \mathit{pres} \triangleq & \ \mathsf{len}\,(\mathit{pres.base}) > 0 \land \mathit{uniqueIDs}\,(\mathit{pres});
\end{aligned}
$$

Example

As remarked in the previous example, we are working with an Harmonia Java *UIML* render release. So, the *presentation* element refers $Java\_1.3\_Harmonia\_1.0$.

```
<peers>
  ....
    <presentation base="Java_1.3_Harmonia_1.0"/>
</peers>
```

4.3.5.3 The <logic> Element

DTD

```
<!ELEMENT logic (d-component*)>
<!ATTLIST logic
    id          NMTOKEN                          #IMPLIED
    source      CDATA                            #IMPLIED
    how         (union | cascade | replace)      "replace"
    export      (hidden | optional | required)  "optional">
```

Description

The *logic* element describes how the UI interacts with the underlying logic that implements the functionality apparent in the interface. The underlying logic might be implemented by middleware in a three tier application, or it might be implemented by scripts in some scripting language, or it might be implemented by a set of objects whose methods are invoked as the end user interacts with the UI, or by some combination of these, or in other ways.

Thus, the *logic* element acts as the glue between an UI described in *UIML* and other code. It describes the calling conventions for methods in application logic that the UI invokes. Examples of such functions include objects in languages such as *C++* or *Java*, *CORBA* objects, programs, legacy systems, server-side scripts, databases, and scripts defined in various scripting languages.

VDM-SL Specification

$$\begin{aligned} Logic &:: dcomponent : D\text{-}component^* \\ &\quad\; id : [ID] \\ &\quad\; source : String \\ &\quad\; how : [SourcesModes] \\ &\quad\; export : [ExportOptions] \\ \mathsf{inv}\ l &\triangleq uniqueIDs\,(l); \end{aligned}$$

Example

The logic element in our Stacks case study describes the underlying Java functions which will be used. This Java functions are implemented in a separate module (stack.class).

```
<logic>
    <d-component id="stack" maps-to="stack">
        <d-method id="removeElement"
            maps-to="removeTopElement"/>
        <d-method id="addElement" maps-to="Push"/>
        <d-method id="Top" maps-to="Top"/>
        <d-method id="Clear" maps-to="Clear"/>
    </d-component>
</logic>
```

4.3.5.4 Subelements of *presentation* and *logic*

The <d-component> Element

DTD

```
<!ELEMENT d-component (d-method)*>
<!ATTLIST d-component
    id          NMTOKEN                        #REQUIRED
    source      CDATA                          #IMPLIED
    how         (union | cascade | replace)    "replace"
    export      (hidden | optional | required)  "optional"
    maps-to     CDATA                          #IMPLIED
    location    CDATA                          #IMPLIED>
```

Description

The $d\text{-}component$ (a child of *logic* only) acts as a container for application methods (e.g., a class in an object oriented language). A $d\text{-}component$ contains $d\text{-}methods$.

The $maps\text{-}to$ attribute specifies the platform-specific type of component or container that is being bound. The location attribute gives additional information (e.g., an URI) that is used by the rendering engine to locate the widget, event, or application class at runtime.

VDM-SL Specification

$$\begin{aligned} \mathit{D\text{-}component} :: \; & \mathit{dmethod} : \mathit{D\text{-}method}^* \\ & \mathit{id} : \mathit{ID} \\ & \mathit{source} : \mathit{String} \\ & \mathit{how} : [\mathit{SourcesModes}] \\ & \mathit{export} : [\mathit{ExportOptions}] \\ & \mathit{maps\text{-}to} : \mathit{String} \\ & \mathit{location} : \mathit{String} \end{aligned}$$

Example

Below, the local methods *removeTopElement*, *Push*, *Top* and *Clear* are mapped to Java methods *removeElement*, *addElement*, *Top* and *Clear*, respectively.

```
<d-component id="stack" maps-to="stack">
    <d-method id="removeElement" maps-to="removeTopElement"/>
    <d-method id="addElement" maps-to="Push"/>
    <d-method id="Top" maps-to="Top"/>
    <d-method id="Clear" maps-to="Clear"/>
</d-component>
```

The <d-class> Element

DTD

```
<!ELEMENT d-class (d-method*, d-property*, event*, listener*)>
<!ATTLIST d-class
    id          NMTOKEN                         #REQUIRED
    source      CDATA                           #IMPLIED
    how         (union | cascade | replace)     "replace"
    export      (hidden | optional | required)  "optional"
    maps-to     CDATA                           #REQUIRED
    maps-type   (attribute | tag | class)       #REQUIRED
    used-in-tag (event | listener | part)       #REQUIRED>
```

Description

The $\mathit{d\text{-}class}$ (a child of *presentation* only) element binds a name used in the rendering property of a part or an *event* element elsewhere in the interface to a component that is part of the presentation toolkit.

The maps-to attribute specifies the platform-specific type of the component or container that is being bound.

VDM-SL Specification

$$
\begin{aligned}
&\mathit{mapsTypes} = \text{ATTRIBUTE} \mid \text{TAG} \mid \text{CLASS};\\
&\mathit{used\text{-}in\text{-}tagTypes} = \text{EVENT} \mid \text{PART} \mid \text{LISTENER};\\
&\mathit{D\text{-}class} :: \mathit{dmethod} : \mathit{D\text{-}method}^*\\
&\qquad\qquad \mathit{dproperty} : \mathit{D\text{-}property}^*\\
&\qquad\qquad \mathit{event} : \mathit{Event}^*\\
&\qquad\qquad \mathit{listener} : \mathit{Listener}^*\\
&\qquad\qquad \mathit{id} : \mathit{ID}\\
&\qquad\qquad \mathit{source} : \mathit{String}\\
&\qquad\qquad \mathit{how} : [\mathit{SourcesModes}]\\
&\qquad\qquad \mathit{export} : [\mathit{ExportOptions}]\\
&\qquad\qquad \mathit{maps\text{-}to} : \mathit{String}\\
&\qquad\qquad \mathit{maps\text{-}type} : \mathit{mapsTypes}\\
&\qquad\qquad \mathit{used\text{-}in\text{-}tag} : \mathit{used\text{-}in\text{-}tagTypes}
\end{aligned}
$$

The <d-property> Element

DTD

```
<!ELEMENT d-property (d-method*,d-param*)>
<!ATTLIST d-property
    id          NMTOKEN                                   #REQUIRED
    maps-type (attribute|getMethod|setMethod|method) #REQUIRED
    maps-to     CDATA                                     #REQUIRED
    return-type CDATA                                     #IMPLIED>
```

Description

The *d-property* element specifies the mapping between the name appearing in a *property* element and the associated methods that assign or retrieve a value for the property.

VDM-SL Specification

$$
\begin{aligned}
&\mathit{D\text{-}property} :: \mathit{dmethod} : \mathit{D\text{-}method}^*\\
&\qquad\qquad \mathit{dparam} : \mathit{D\text{-}param}^*\\
&\qquad\qquad \mathit{id} : \mathit{ID}\\
&\qquad\qquad \mathit{maps\text{-}type} : (\text{ATTRIBUTE} \mid \text{GETMETHOD} \mid \text{SETMETHOD} \mid \text{METHOD})\\
&\qquad\qquad \mathit{maps\text{-}to} : \mathit{String}\\
&\qquad\qquad \mathit{return\text{-}type} : \mathit{String}\\
&\mathsf{inv}\ \mathit{dpro} \triangleq \mathit{length}\,(\mathit{dpro.id}) > 0 \wedge \mathit{length}\,(\mathit{dpro.maps\text{-}to}) > 0;
\end{aligned}
$$

The <d-method> Element

DTD

```
<!ELEMENT d-method (d-param*, script?)>
```

```
<!ATTLIST d-method
    id          NMTOKEN                          #REQUIRED
    source      CDATA                            #IMPLIED
    how         (union | cascade | replace)      "replace"
    export      (hidden | optional | required)  "optional"
    maps-to     CDATA                            #REQUIRED
    return-type CDATA                            #IMPLIED>
```

Description

The *d-method* element describes a method in the external application logic or presentation toolkit in terms of its optional formal parameters and optional return value.

The *maps-to* attribute specifies the name that is being bound. The value of *maps-to* points to the name of an actual method that can be executed. The method can represent a toolkit method (if it is inside a *presentation* element), an application method (if it is inside a logic element), or scripting code (with scripting nested inside the *d-method* element).

The method element supports three different execution models:

1. The method represents a remote (outside the render) executable code. This code executes outside the sandbox of the render and is treated like a black box. The render will package all the parameters, send them to the server executing the code (which can be on the same machine or across the network), and wait for a reply. Here is an example:

```
<d-component name="Math" maps-to="myClass.Math.CommonRoutines">
    <d-method name="findMean" maps-to="calcMean">
        <d-param name="a"/>
        <d-param name="b"/>
    </d-method>
</d-component>
```

2. The method represents a local script. This script is embedded inside the method and is executed within the sandbox of the render. If the *maps-to* attribute for the component is missing, this means that all the code is local. Here is an example:

```
<d-component name="Math">
    <d-method name="findMean" maps-to="calcMean">
        <d-param name="a"/>
        <d-param name="b"/>
        <script type="text/javascript">
            <![CDATA[
                calcMean(int a, int b) {
                return (a+b)/2;
                }
            ]]>
        </script>
    </d-method>
</d-component>
```

3. The method represents a combination of the above. This is useful if you want to do some error checking locally before calling a remote method or manipulate the result after it is returned. The semantics of how to do this are under revision [Har02].

VDM-SL Specification

$$
\begin{aligned}
\mathit{D\text{-}method} :: \; & \mathit{dparam} : \mathit{D\text{-}param}^{*} \\
& \mathit{script} : [\mathit{Script}] \\
& \mathit{id} : \mathit{ID} \\
& \mathit{source} : \mathit{String} \\
& \mathit{how} : [\mathit{SourcesModes}] \\
& \mathit{export} : [\mathit{ExportOptions}] \\
& \mathit{maps\text{-}to} : \mathit{String} \\
& \mathit{return\text{-}type} : \mathit{String}
\end{aligned}
$$

The <d-param> Element

DTD

```
<!ELEMENT d-param (#PCDATA)>
<!ATTLIST d-param
    id      NMTOKEN         #IMPLIED
    type    CDATA           #IMPLIED>
```

Description

Describes a single formal parameter of the function described by the parent method element. Note that all parameters are character strings. It is up to some intermediary to convert parameters from character strings to other data types (e.g., integer or Boolean) if required.

The order of *param* elements within the method element is significant. This order must correspond to the order in which the parameters were originally declared in the external application. For example, if we have

```
<d-param>23</d-param>
```

which is mapped to the parameter of function $f(double)$ in this Java class

```
public class Demo {
    static void f(double);
}
```

then string “23” is converted by some intermediary to type double in Java.

VDM-SL Specification

$$\begin{aligned} \mathit{D\text{-}param} :: \; & \mathit{data} : \mathit{String} \\ & \mathit{id} : [\mathit{ID}] \\ & \mathit{type} : \mathit{String} \end{aligned}$$

4.3.5.5 The <script> Element

DTD

```
<!ELEMENT script (#PCDATA)>
<!ATTLIST script
        id          NMTOKEN                        #REQUIRED
        source      CDATA                          #IMPLIED
        how         (union | cascade | replace)    "replace"
        export      (hidden | optional | required)  "optional"
        type        NMTOKEN                        #IMPLIED>
```

Description

The *script* element contains a program written in the scripting language identified by the *type* attribute (this is similar to the script element in HTML 4.0).

VDM-SL Specification

$$\begin{aligned} \mathit{Script} :: \; & \mathit{data} : \mathit{String} \\ & \mathit{id} : \mathit{ID} \\ & \mathit{source} : \mathit{String} \\ & \mathit{how} : [\mathit{SourcesModes}] \\ & \mathit{export} : [\mathit{ExportOptions}] \\ & \mathit{type} : \mathit{String} \\ \mathsf{inv}\; \mathit{scri} \triangleq \; & \mathit{uniqueIDs}\,(\mathit{scri}); \end{aligned}$$

4.3.6 Templates - Reusable Interface Components

UIML templates enables interface implementors to design parts or to make their entire UI reusable as a component in another UI. For example, many UI for electronic commerce applications include a credit-card entry form. If such a form is described in *UIML* as a template, then it can be reused multiple times either within the same UI or across another UI. This reduces the amount of *UIML* code needed to develop an UI and also ensures a consistent presentation across enterprise-wide UI. End users tend to make fewer mistakes and are more efficient when presented with familiar UI ([AH02] describes all template rules).

4.3.6.1 The <listener> Element

DTD

```
<!ELEMENT listener EMPTY>
<!ATTLIST listener
        class        NMTOKEN             #IMPLIED
        attacher      CDATA              #IMPLIED>
```

Description

listener records that a name defined with *d-class used-in-tag*=“*listener*” should be attached as a listener to the $d\text{-}class$, which contains this *listener* element.

VDM-SL Specification

$$\begin{aligned} Listener :: \; & class : String \\ & attacher : String \end{aligned}$$

4.3.6.2 The <template> Element

DTD

```
<!ENTITY % SourceElements  "
    (behavior|d-class|d-component|constant|content|interface|
    logic|part|peers|presentation|property|rule|script|
    structure|style|restructure)">

<!ELEMENT template %SourceElements;>
<!ATTLIST template
    id    NMTOKEN     #IMPLIED>
```

Description

The template element permits several handy shortcuts when writing UIML. It allows

- one fragment of *UIML* to be inserted in multiple places in a *UIML* document,
- one *UIML* document to include a *UIML* fragment from another document, and
- style and other elements to be cascaded, in a manner analogous to the CSS specification.

Templates work as follows. Most elements (see *SourceElements* list) can contain the source attribute; call such an element E. The *source* attribute names a *template* element (either within the same document or in another document). The *template* named must contain an element of the same type as the element *E* (i.e., have the same tag name). The *source* attribute causes the body of the element inside the *template* to be combined with the body of *E*.

The section “Rules for Templates” of [Har02], describes the rules to control how this combining is done.

VDM-SL Specification

$$\begin{aligned} & \mathit{Template} :: \mathit{src\text{-}ele} : \mathit{SourceElements} \\ & \qquad\qquad\quad \mathit{id} : \mathit{ID} \\ & \mathsf{inv}\ t \triangleq \mathit{uniqueIDs}\,(t); \end{aligned}$$

4.4 Invariants

4.4.1 Overview

Invariants are conditions on datatypes which must be preserved before or after execution of a particular function or event involving such datatypes. Some conditions are naturally preserved by *VDM-SL* notation and definitions of its basic types, like *String*.

We also must know that the use of a DTD to certify the XML structure represents an important way to ensure the consistency of the *UIML* document, although there are some other situations which require explicit control. In the following we describe the main invariants involved in the UIML specification:

1. No two elements can have the same *id* within the same *UIML* document
   The predicate *uniqueID* checks this;

2. In a *reference* element there must be one *constant-name* attribute
   Ensured by *VDM-SL* operators;

3. The *equal* element must have exactly two children
   Ensured by *VDM-SL* operators;

4. An *attribute* element must have a *name* attribute
   Ensured by *VDM-SL* operators;

5. The *call* element has one mandatory attribute, name
   Ensured by *VDM-SL* operators;

6. There must be at least one *style* element
   Ensured by *VDM-SL* operators;

7. A *param* element must have exactly one child
   Ensured by *VDM-SL* operators;

8. The *condition* element must have only one child
   Ensured by *VDM-SL* operators.

9. The *property* attribute *part-name* must refer an existing *part ID* attribute;

10. The *event* attribute *part-name* must refer an existing *part ID* attribute;

11. The *constant ID* must be declared of *const-name* on *reference* element;

12. *Part* class name must be defined in *d-class.ID*;

13. *property* attribute *name* must be defined in *d-property* attribute *part-name*;

14. The *event* attribute *name* must be defined in *d-class* attribute *ID*;

15. the *d-method type* attribute must be defined in *d-class ID* attribute.

In the sequel, we will describe the predicates which preserve these invariants.

4.4.2 Auxiliary Data Types

$$CountID = ID \xrightarrow{m} \mathbb{N};$$

$$\begin{aligned}
&UIMLMembers :: P : Peers^* \\
&\qquad\qquad\qquad I : Interface^* \\
&\qquad\qquad\qquad T : Template^* \\
&PropertyType = String \mid Constant \mid Property \mid Reference \mid Call \mid \\
&\qquad\qquad Iterator \mid Op \mid Event; \\
&UIMLElements = Uiml \mid Peers \mid \\
&\qquad Behavior \mid D\text{-}class \mid D\text{-}component \mid Constant \mid Interface \mid Logic \mid \\
&\qquad Part \mid Presentation \mid Property \mid Rule \mid Script \mid Structure \mid Style \mid \\
&\qquad Content \mid Restructure \mid Equal \mid Op \mid Iterator \mid Template \mid \\
&\qquad ActionType1 \mid ActionType2
\end{aligned}$$

4.4.3 The uniqueness of ID’s

4.4.3.1 Function $uniqueIDs$

Specification:

```
$uniqueIDs : UIMLElements \rightarrow \mathbb{B}$
$uniqueIDs\,(t) \triangleq$
  cases $t$ :
    mk-$Uiml$ (-, -) $\rightarrow$
        rng $allIDs\,(t) = \{1\}$,
    mk-$Peers$ (-, -, -, -, -) $\rightarrow$
        rng $peersIDs\,([t], \{\mapsto\}) = \{\}$,
    mk-$Presentation$ (-, -, -, -, -, -) $\rightarrow$
        rng $preslogIDs\,([t], \{\mapsto\}) = \{1\}$,
    mk-$Logic$ (-, -, -, -, -) $\rightarrow$
        rng $preslogIDs\,([t], \{\mapsto\}) = \{1\}$,
    mk-$Part$ (-, -, -, -, -, -, -, -, -, -, -, -, -) $\rightarrow$
        rng $partsIDs\,([t], \{\mapsto\}) = \{1\}$,
    mk-$Equal$ (-, -) $\rightarrow$
        rng $equalIDs\,(t, \{\mapsto\}) = \{1\}$,
    mk-$Op$ (-, -) $\rightarrow$
        rng $opIDs\,([t], \{\mapsto\}) = \{1\}$,
    mk-$Constant$ (-, -, -, -, -, -, -, -) $\rightarrow$
        rng $constantsIDs\,([t], \{\mapsto\}) = \{1\}$,
    mk-$Property$ (-, -, -, -, -, -, -, -, -, -) $\rightarrow$
        rng $propertiesIDs\,([t], \{\mapsto\}) = \{1\}$,
    mk-$Restructure$ (-, -, -, -, -, -, -) $\rightarrow$
        rng $restructureIDs\,(t, \{\mapsto\}) = \{1\}$,
    mk-$Iterator$ (-, -) $\rightarrow$
        rng $iteratorIDs\,(t, \{\mapsto\}) = \{1\}$,
    mk-$Style$ (-, -, -, -, -) $\rightarrow$
        rng $styleIDs\,(t, \{\mapsto\}) = \{1\}$,
    mk-$Content$ (-, -, -, -, -) $\rightarrow$
        rng $contentIDs\,(t, \{\mapsto\}) = \{1\}$,
    mk-$Behavior$ (-, -, -, -, -) $\rightarrow$
        rng $behaviorIDs\,(t, \{\mapsto\}) = \{1\}$,
    mk-$Interface$ (-, -, -, -, -) $\rightarrow$
        rng $interfaceIDs\,([t], \{\mapsto\}) = \{1\}$,
    mk-$Template$ (-, -) $\rightarrow$
        rng $templatesIDs\,([t], \{\mapsto\}) = \{1\}$,
    mk-$Rule$ (-, -, -, -, -, -) $\rightarrow$
        rng $rulesIDs\,([t], \{\mapsto\}) = \{1\}$,
    others $\rightarrow$ true
  end;
```

Description:

Tests the uniqueness of identity attributes in a UIML *document.*

Calls:

$allIDs, peersIDs, preslogIDs, partsIDs, equalIDs, opIDs, constantsIDs,$ $propertiesIDs,$ $restructureIDs, iteratorIDs, styleIDs, contentIDs, behaviorIDs, interfaceIDs,$

$templatesIDs$,$rulesIDs$

4.4.3.2 Function $allIDs$

Specification:

```
$allIDs : Uiml \rightarrow CountID$
$allIDs\,(u) \triangleq$
   let $memb =$
            $split\,(u.members,$ mk-$UIMLMembers\,([], [], []))$,
      $pid = peersIDs\,(memb.P, \{\mapsto\})$,
      $iid = interfacesIDs\,(memb.I, pid)$ in
   $templatesIDs\,(memb.T, iid)$;
```

Description:

Counts the occurrences of each id *in an* UIML *document.*

Calls:

$split$,$peersIDs$,$interfaceIDs$,$templatesIDs$

4.4.3.3 Function $peersIDs$

Specification:

```
$peersIDs : Peers^* \times CountID \rightarrow CountID$
$peersIDs\,(sp, ci) \triangleq$
   if $sp = []$
   then $ci$
   else let $pe =$ hd $(sp)$ in
         let $cp = addMunion\,(pe.id, ci)$,
             $cprelog = preslogIDs\,(pe.prelog, cp)$ in
         $peersIDs\,($tl $(sp), cprelog)$;
```

Description:

Counts the occurrences of each id *in a* peers *element.*

Calls:

$addMunion$,$preslogIDs$

4.4.3.4 Function $preslogIDs$

Specification:

```
$preslogIDs : (Presentation \mid Logic)^{*} \times CountID \to CountID$
$preslogIDs\,(spl, ci) \triangleq$
  if $spl = []$
  then $ci$
  else let $pl =$ hd $(spl)$,
           $cpi = addMunion\,(pl.id, ci)$ in
      cases $pl$ :
        mk-$Presentation$ (-, -, -, -, -, -) $\to$
            let $cdc = dclassesIDs\,(pl.dclass, cpi)$ in
            $preslogIDs$ (tl $(spl)$, $cdc$),
        mk-$Logic$ (-, -, -, -, -) $\to$
            let $cdc = dcomponentsIDs\,(pl.dcomponent, cpi)$ in
            $preslogIDs$ (tl $(spl)$, $cdc$)
      end;
```

Description:

Counts the occurrences of each id *in a* presentation *and* logic *elements.*

Calls:

$addMunion$,$dclassesIDs$,$dcomponentsIDs$

4.4.3.5 Function $dcomponentsIDs$

Specification:

```
$dcomponentsIDs : D\text{-}component^{*} \times CountID \to CountID$
$dcomponentsIDs\,(sdc, ci) \triangleq$
  if $sdc = []$
  then $ci$
  else let $dc = $ hd $(sdc)$ in
      let $cdc = dcomponentIDs\,(dc, ci)$ in
      $dcomponentsIDs$ (tl $(sdc)$, $cdc$);
```

Description:

Counts the occurrences of each id *in a* dcomponent *sequence.*

Calls:

$dcomponentIDs$

4.4.3.6 Function $dcomponentIDs$

Specification:

```
$dcomponentIDs : D\text{-}component \times CountID \rightarrow CountID$
$dcomponentIDs\,(dc, ci) \triangleq$
  let $cdc = addMunion\,(dc.id, ci)$ in
  $dmethodsIDs\,(dc.dmethod, cdc)$;
```

Description:

Counts the occurrences of each id *in a* dcomponent *element.*

Calls:

$addMunion$,$dmethodsIDs$

4.4.3.7 Function $dclassesIDs$

Specification:

```
$dclassesIDs : D\text{-}class^{*} \times CountID \rightarrow CountID$
$dclassesIDs\,(sdc, ci) \triangleq$
  if $sdc = []$
  then $ci$
  else let $dc = \text{hd}\,(sdc)$ in
       let $cdc = dclassIDs\,(dc, ci)$ in
       $dclassesIDs\,(\text{tl}\,(sdc), cdc)$;
```

Description:

Counts the occurrences of each id *in a* dclass *sequence.*

Calls:

$dclassIDs$

4.4.3.8 Function $dclassIDs$

Specification:

```
$dclassIDs : D\text{-}class \times CountID \rightarrow CountID$
$dclassIDs\,(dc, ci) \triangleq$
    let $cdc = addMunion\,(dc.id, ci)$,
        $cdm = dmethodsIDs\,(dc.dmethod, cdc)$ in
    $dpropertiesIDs\,(dc.dproperty, cdm)$;
```

Description:

Counts the occurrences of each id *in a* dclass *element.*

Calls:

$addMunion$,$dmethodsIDs$,$dpropertiesIDs$

4.4.3.9 **Function** $dmethodsIDs$

Specification:

```
$dmethodsIDs : D\text{-}method^{*} \times CountID \rightarrow CountID$
$dmethodsIDs\,(sdm, ci) \triangleq$
    if $sdm = []$
    then $ci$
    else let $dm =$ hd $(sdm)$ in
            let $cdm = dmethodIDs\,(dm, ci)$ in
            $dmethodsIDs\,($tl $(sdm), cdm)$;
```

Description:

Counts the occurrences of each id *in a* dmethod *sequence.*

Calls:

$dmethodIDs$

4.4.3.10 **Function** $dmethodIDs$

Specification:

```
$dmethodIDs : D\text{-}method \times CountID \rightarrow CountID$
$dmethodIDs\,(dm, ci) \triangleq$
    let $cdm = addMunion\,(dm.id, ci)$ in
    $dparamsIDs\,(dm.dparam, cdm)$;
```

Description:

Counts the occurrences of each id *in a* dmethod *element.*

Calls:

$addMunion$,$dparamsIDs$

4.4.3.11 Function $dparamsIDs$

Specification:

```
$dparamsIDs : D\text{-}param^* \times CountID \rightarrow CountID$
$dparamsIDs\,(sdp, ci) \triangleq$
  if $sdp = []$
  then $ci$
  else let $dp = \mathsf{hd}\,(sdp)$ in
        let $cdp = addMunion\,(dp.id, ci)$ in
        $dparamsIDs\,(\mathsf{tl}\,(sdp), cdp)$;
```

Description:

Counts the occurrences of each id *in a* dparams *sequence.*

Calls:

$addMunion$

4.4.3.12 Function $dpropertiesIDs$

Specification:

```
$dpropertiesIDs : D\text{-}property^* \times CountID \rightarrow CountID$
$dpropertiesIDs\,(sdp, ci) \triangleq$
  if $sdp = []$
  then $ci$
  else let $dp = \mathsf{hd}\,(sdp)$ in
        let $cdp = addMunion\,(dp.id, ci)$,
            $cdm = dmethodsIDs\,(dp.dmethod, cdp)$,
            $cdpar = dparamsIDs\,(dp.dparam, cdm)$ in
        $dpropertiesIDs\,(\mathsf{tl}\,(sdp), cdpar)$;
```

Description:

Counts the occurrences of each id *in a* dproperty *sequence.*

Calls:

$addMunion, dmethodsIDs, dparamsIDs$

4.4.3.13 Function $usedintagIDs$

Specification:

```
$usedintagIDs : (Event \mid Part) \times CountID \rightarrow CountID$
$usedintagIDs\ (ep, ci) \triangleq$
  cases $ep$ :
    mk-$Part$ (-,-,-,-,-,-,-,-,-,-,-,-,-) $\rightarrow partIDs\ (ep, ci)$,
    others $\rightarrow ci$
  end;
```

Description:

Counts the occurrences of each id *in a* used-in-tag *attribute.*

Calls:

$partIDs$

4.4.3.14 Function $partsIDs$

Specification:

```
$partsIDs : Part^* \times CountID \rightarrow CountID$
$partsIDs\ (sp, ci) \triangleq$
  if $sp = []$
  then $ci$
  else let $p =$ hd $(sp)$ in
      let $cp = partIDs\ (p, ci)$ in
      $partsIDs$ (tl $(sp), cp)$;
```

Description:

Counts the occurrences of each id *in a* part *sequence.*

Calls:

$partIDs$

4.4.3.15 **Function** $partIDs$

Specification:

```
$partIDs : Part \times CountID \rightarrow CountID$
$partIDs\,(p, ci) \triangleq$
    let $cs = addMunion\,(p.id, ci)$,
        $cc = contentIDs\,(p.content, cs)$,
        $cb = behaviorIDs\,(p.behavior, cc)$ in
    $partsIDs\,(p.parts, cb)$;
```

Description:

Counts the occurrences of each id *in a* part *element.*

Calls:

$addMunion$,$contentIDs$,$behaviorIDs$,$partsIDs$

4.4.3.16 **Function** $contentIDs$

Specification:

```
$contentIDs : Content \times CountID \rightarrow CountID$
$contentIDs\,(c, ci) \triangleq$
    let $cc = addMunion\,(c.id, ci)$ in
    $constantsIDs\,(c.constant, cc)$;
```

Description:

Counts the occurrences of each id *in a* content *element.*

Calls:

$addMunion$,$constantsIDs$

4.4.3.17 Function $constantsIDs$

Specification:

```
$constantsIDs : Constant^* \times CountID \to CountID$
$constantsIDs\,(sc, ci) \triangleq$
  if $sc = []$
  then $ci$
  else let $c =$ hd $(sc)$ in
      let $cc = addMunion\,(c.id, ci),$
        $cc1 = constantsIDs\,(c.constant, cc)$ in
      $constantsIDs$ (tl $(sc), cc1);$
```

Description:

Counts the occurrences of each id *in a* constant *sequence.*

Calls:

$constantsIDs$,$addMunion$

4.4.3.18 Function $behaviorIDs$

Specification:

```
$behaviorIDs : Behavior \times CountID \to CountID$
$behaviorIDs\,(b, ci) \triangleq$
  let $cb = addMunion\,(b.id, ci)$ in
  $rulesIDs\,(b.rules, cb);$
```

Description:

Counts the occurrences of each id *in a* behavior *element.*

Calls:

$addMunion$,$rulesIDs$

4.4.3.19 Function $rulesIDs$

Specification:

```
$rulesIDs : Rule^* \times CountID \rightarrow CountID$
$rulesIDs\,(sr, ci) \triangleq$
  if $sr = []$
  then $ci$
  else let $r =$ hd $(sr)$ in
      let $cr = addMunion\,(r.id, ci),$
        $cc = conditionIDs\,(r.condition, cr)$ in
      $rulesIDs$ (tl $(sr), cc);$
```

Description:

Counts the occurrences of each id *in a* constant *sequence.*

Calls:

$addMunion$,$conditionIDs$

4.4.3.20 **Function** $conditionIDs$

Specification:

```
$conditionIDs : [Condition] \times CountID \rightarrow CountID$
$conditionIDs\,(c, ci) \triangleq$
  if $c =$ nil
  then $ci$
  else cases $c.type$ :
        mk-$Equal$ (-, -) $\rightarrow equalIDs\,(c.type, ci),$
        mk-$Op$ (-, -) $\rightarrow opIDs\,(c.type.type, ci)$
      end;
```

Description:

Counts the occurrences of each id *in a* condition *element.*

Calls:

$equalIDs$,$opIDs$

4.4.3.21 **Function** $actionIDs$

Specification:

```
$actionIDs : [Action] \times CountID \to CountID$
$actionIDs\,(a, ci) \triangleq$
  if $a$ = nil
  then $ci$
  else cases $a$ :
        mk-$ActionType1\,(-,-) \to pcrsIDs\,(a.type, ci)$,
        mk-$ActionType2\,(-,-,-) \to$
            let $cwt = whentrueIDs\,(a.whentrue, ci)$,
                $cwf = whenfalseIDs\,(a.whenfalse, cwt)$ in
            $bydefaultIDs\,(a.bydefault, cwf)$
      end;
```

Description:

Counts the occurrences of each id *on* action *element.*

Calls:

$pcrsIDs, whentrueIDs, whenfalseIDs, bydefaultIDs$

4.4.3.22 **Function** $equalIDs$

Specification:

```
$equalIDs : Equal \times CountID \to CountID$
$equalIDs\,(e, ci) \triangleq$
  $cprIDs\,(e.other, ci)$;
```

Description:

Counts the occurrences of each id *on* equal *element.*

Calls:

$cprIDs$

4.4.3.23 **Function** $cprIDs$

Specification:

```
$cprIDs : (Constant \mid Property \mid Reference) \times CountID \to CountID$
$cprIDs\ (cpr, ci) \triangleq$
  cases $cpr$ :
    mk-$Constant$ (-, -, -, -, -, -, -) $\to constantsIDs\ ([cpr], ci)$,
    mk-$Property$ (-, -, -, -, -, -, -, -, -) $\to propertiesIDs\ ([cpr], ci)$
  end;
```

Description:

Counts the occurrences of each id *on* (Constant — Property — Reference) *elements.*

Calls:

$constantsIDs$,$propertiesIDs$

4.4.3.24 Function $pcrsIDs$

Specification:

```
$pcrsIDs : (Property \mid Call \mid Restructure)^* \times CountID \to CountID$
$pcrsIDs\ (pcr, ci) \triangleq$
  if $pcr = []$
  then $ci$
  else let $h =$ hd $(pcr)$ in
        cases $h$ :
          mk-$Property$ (-, -, -, -, -, -, -, -, -) $\to$
                let $cp = propertiesIDs\ ([h], ci)$ in
                $pcrsIDs$ (tl $(pcr), cp)$,
          mk-$Call$ (-, -) $\to$
                let $cp = paramsIDs\ (h.params, ci)$ in
                $pcrsIDs$ (tl $(pcr), cp)$,
          mk-$Restructure$ (-, -, -, -, -, -) $\to$
                let $ct = templatesIDs\ ([h.template], ci)$ in
                $pcrsIDs$ (tl $(pcr), ct)$
        end;
```

Description:

Counts the occurrences of each id *on* (Property — Call — Restructure) *elements.*

Calls:

$propertiesIDs$,$paramsIDs$,$templatesIDs$

4.4.3.25 Function $propertiesIDs$

Specification:

```
$propertiesIDs : Property^* \times CountID \to CountID$
$propertiesIDs\,(sp, ci) \triangleq$
  if $sp = []$
  then $ci$
  else let $p = \mathsf{hd}\,(sp)$ in
          let $cp = propertytypesIDs\,(p.property, ci)$ in
          $propertiesIDs\,(\mathsf{tl}\,(sp), cp)$;
```

Description:

Counts the occurrences of each id *on* property *sequence.*

Calls:

$propertytypeIDs$

4.4.3.26 Function $propertytypesIDs$

Specification:

```
$propertytypesIDs : PropertyType^* \times CountID \to CountID$
$propertytypesIDs\,(sp, ci) \triangleq$
  if $sp = []$
  then $ci$
  else let $p = \mathsf{hd}\,(sp)$ in
          cases $p$ :
            mk-$Constant\,(\text{-},\text{-},\text{-},\text{-},\text{-},\text{-},\text{-}) \to$
                let $cp = addMunion\,(p.id, ci)$,
                    $cc = constantsIDs\,([p], cp)$ in
                $propertytypesIDs\,(\mathsf{tl}\,(sp), cc)$,
            mk-$Property\,(\text{-},\text{-},\text{-},\text{-},\text{-},\text{-},\text{-},\text{-},\text{-}) \to$
                let $cp = addMunion\,(p.id, ci)$,
                    $cps = propertiesIDs\,([p], cp)$ in
                $propertytypesIDs\,(\mathsf{tl}\,(sp), cps)$,
            mk-$Call\,(\text{-},\text{-}) \to$
                let $cc = paramsIDs\,(p.params, ci)$ in
                $propertytypesIDs\,(\mathsf{tl}\,(sp), cc)$,
            mk-$Iterator\,(\text{-},\text{-}) \to$
                let $cp = addMunion\,(p.id, ci)$,
                    $cit = iteratorIDs\,(p.iterator, cp)$ in
                $propertytypesIDs\,(\mathsf{tl}\,(sp), cit)$
          end;
```

Description:

Counts the occurrences of each id *in a* PropertyType *element.*

Calls:

$addMunion$,$constantsIDs$,$propertiesIDs$,$paramsIDs$, $iteratorIDs$

4.4.3.27 Function $iteratorIDs$

Specification:

```
$iteratorIDs : (Constant \mid Property \mid Call) \times CountID \rightarrow CountID$
$iteratorIDs\,(i, ci) \triangleq$
  cases $i$ :
    mk-$Constant$ (-,-,-,-,-,-,-) $\rightarrow$
        let $cp = addMunion\,(i.id, ci)$ in
        $constantsIDs\,([i], cp)$,
    mk-$Property$ (-,-,-,-,-,-,-,-,-) $\rightarrow$
        let $cp = addMunion\,(i.id, ci)$ in
        $propertiesIDs\,([i], cp)$,
    mk-$Call$ (-,-) $\rightarrow$
        $paramsIDs\,(i.params, ci)$
  end;
```

Description:

Counts the occurrences of each id *in an* Iterator *element.*

Calls:

$addMunion$,$constantsIDs$,$propertiesIDs$,$paramsIDs$

4.4.3.28 Function $paramsIDs$

Specification:

```
$paramsIDs : Param^{*} \times CountID \rightarrow CountID$
$paramsIDs\,(sp, ci) \triangleq$
  if $sp = []$
  then $ci$
  else let $p =$ hd $(sp)$,
           $cp = paramIDs\,(p.type, ci)$ in
       $paramsIDs$ (tl $(sp)$, $cp$);
```

Description:

Counts the occurrences of each id *in a* param *sequence.*

Calls:

$paramIDs$

4.4.3.29 Function $paramIDs$

Specification:

```
$paramIDs : ParamType \times CountID \to CountID$
$paramIDs\,(p, ci) \triangleq$
  cases $p$ :
    mk-$Constant\,(\text{-},\text{-},\text{-},\text{-},\text{-},\text{-},\text{-}) \to constantsIDs\,([p], ci)$,
    mk-$Property\,(\text{-},\text{-},\text{-},\text{-},\text{-},\text{-},\text{-},\text{-},\text{-}) \to propertiesIDs\,([p], ci)$,
    mk-$Call\,(\text{-},\text{-}) \to paramsIDs\,(p.params, ci)$,
    mk-$Iterator\,(\text{-},\text{-}) \to$
        let $cp = addMunion\,(p.id, ci)$ in
        $iteratorIDs\,(p.iterator, cp)$
  end;
```

Description:

Counts the occurrences of each id *in a* param *element.*

Calls:

$addMunion$,$constantsIDs$,$propertiesIDs$,$paramsIDs$,$iteratorIDs$

4.4.3.30 Function $opIDs$

Specification:

```
$opIDs : PropertyType^* \times CountID \rightarrow CountID$
$opIDs\,(sp, ci) \triangleq$
  if $sp = []$
  then $ci$
  else let $p =$ hd $(sp)$ in
      cases $p$ :
        mk-$Constant$ (-, -, -, -, -, -, -) $\rightarrow$
            let $cp = addMunion\,(p.id, ci)$,
                $cc = constantsIDs\,([p], cp)$ in
            $opIDs$ (tl $(sp), cc)$,
        mk-$Property$ (-, -, -, -, -, -, -, -, -) $\rightarrow$
            let $cp = addMunion\,(p.id, ci)$,
                $cps = propertiesIDs\,([p], cp)$ in
            $opIDs$ (tl $(sp), cps)$,
        mk-$Call$ (-, -) $\rightarrow$
            let $cc = paramsIDs\,(p.params, ci)$ in
            $opIDs$ (tl $(sp), cc)$,
        mk-$Op$ (-, -) $\rightarrow$
            let $co = opIDs\,(p.type, ci)$ in
            $opIDs$ (tl $(sp), co)$,
        mk-$Iterator$ (-, -) $\rightarrow$
            let $cp = addMunion\,(p.id, ci)$,
                $cit = iteratorIDs\,(p.iterator, cp)$ in
            $opIDs$ (tl $(sp), cit)$
      end;
```

Description:

Counts the occurrences of each id *in an* op *element.*

Calls:

$addMunion$,$constantsIDs$,$propertiesIDs$,$iteratorIDs$,$paramsIDs$

4.4.3.31 Function $whentrueIDs$

Specification:

```
$whentrueIDs : [When\text{-}true] \times CountID \rightarrow CountID$
$whentrueIDs\,(w, ci) \triangleq$
  if $w =$ nil
  then $ci$
  else let $cpc = pcIDs\,(w.type, ci)$,
          $cr = restructureIDs\,(w.restructure, cpc)$,
          $ce = equalIDs\,(w.equal, cr)$ in
      $opIDs\,(w.op.type, ce)$;
```

Description:

Counts the occurrences of each id *in a* when-true *element.*

Calls:

$restructureIDs$,$equalIDs$,$opIDs$,$pcIDs$

4.4.3.32 **Function** $pcIDs$

Specification:

```
$pcIDs : (Property \mid Call)^{*} \times CountID \rightarrow CountID$
$pcIDs\,(pc, ci) \triangleq$
  if $pc = []$
  then $ci$
  else let $p =$ hd $(pc)$ in
      cases $p$ :
        mk-$Property$ (-, -, -, -, -, -, -, -, -) $\rightarrow$
            let $cp = propertiesIDs\,([p], ci)$ in
            $pcIDs$ (tl $(pc), cp)$,
        mk-$Call$ (-, -) $\rightarrow$
            let $cc = paramsIDs\,(p.params, ci)$ in
            $pcIDs$ (tl $(pc), cc)$
      end;
```

Description:

Counts the occurrences of each id *in* sequence of property or call *elements.*

Calls:

$propertiesIDs$,$paramsIDs$

4.4.3.33 **Function** $whenfalseIDs$

Specification:

```
$whenfalseIDs : [When\text{-}false] \times CountID \rightarrow CountID$
$whenfalseIDs\,(w, ci) \triangleq$
  if $w =$ nil
  then $ci$
  else let $cpc = pcIDs\,(w.type, ci),$
           $cr = restructureIDs\,(w.restructure, cpc),$
           $ce = equalIDs\,(w.equal, cr)$ in
         $opIDs\,(w.op.type, ce);$
```

Description:

Counts the occurrences of each id *on* when-false *element.*

Calls:

$restructureIDs, equalIDs, opIDs, pcIDs$

4.4.3.34 Function $bydefaultIDs$

Specification:

```
$bydefaultIDs : [By\text{-}default] \times CountID \rightarrow CountID$
$bydefaultIDs\,(w, ci) \triangleq$
  if $w =$ nil
  then $ci$
  else let $cpc = pcIDs\,(w.type, ci),$
           $cr = restructureIDs\,(w.restructure, cpc),$
           $ce = equalIDs\,(w.equal, cr)$ in
         $opIDs\,(w.op.type, ce);$
```

Description:

Counts the occurrences of each id *on* by-default *element.*

Calls:

$restructureIDs, equalIDs, opIDs, pcIDs$

4.4.3.35 Function $restructureIDs$

Specification:

```
$restructureIDs : Restructure \times CountID \rightarrow CountID$
$restructureIDs\,(r, ci) \triangleq$
    $templateIDs\,(r.template, ci);$
```

Description:

Counts the occurrences of each id *on* restrucure *element.*

Calls:

$templateIDs$

4.4.3.36 **Function** $interfacesIDs$

Specification:

```
$interfacesIDs : Interface^{*} \times CountID \rightarrow CountID$
$interfacesIDs\,(si, ci) \triangleq$
    if $si = []$
    then $ci$
    else let $p =$ hd $(si)$,
                 $cid = interfaceIDs\,(p.intele, ci)$ in
             $interfacesIDs$ (tl $(si), cid);$
```

Description:

Counts the occurrences of each id *in an* Interface *sequence.*

Calls:

$interfaceIDs$

4.4.3.37 **Function** $interfaceIDs$

Specification:

```
$interfaceIDs$ : $InterfaceElements^* \times CountID \to CountID$
$interfaceIDs\,(si, ci) \triangleq$
  if $si = []$
  then $ci$
  else let $p =$ hd $(si)$ in
      cases $p$ :
        mk-$Structure$ (-, -, -, -, -) $\to$
            let $cp = addMunion\,(p.id, ci)$,
                $cc = structureIDs\,(p, cp)$ in
            $interfaceIDs$ (tl $(si), cc)$,
        mk-$Style$ (-, -, -, -, -) $\to$
            let $cp = addMunion\,(p.id, ci)$,
                $cps = styleIDs\,(p, cp)$ in
            $interfaceIDs$ (tl $(si), cps)$,
        mk-$Content$ (-, -, -, -, -) $\to$
            let $cp = addMunion\,(p.id, ci)$,
                $cc = contentIDs\,(p, cp)$ in
            $interfaceIDs$ (tl $(si), cc)$,
        mk-$Behavior$ (-, -, -, -, -) $\to$
            let $cb = addMunion\,(p.id, ci)$,
                $co = behaviorIDs\,(p, cb)$ in
            $interfaceIDs$ (tl $(si), co)$
      end;
```

Description:

Counts the occurrences of each id *in an* Interface Type *sequence.*

Calls:

$addMunion$, $structureIDs$, $styleIDs$, $contentIDs$, $behaviorIDs$

4.4.3.38 Function $templatesIDs$

Specification:

```
$templatesIDs$ : $Template^* \times CountID \to CountID$
$templatesIDs\,(s, ci) \triangleq$
  if $s = []$
  then $ci$
  else let $t =$ hd $(s)$,
          $cid = templateIDs\,(t, ci)$ in
      $templatesIDs$ (tl $(s), cid)$;
```

Description:

Counts the occurrences of each id *in a sequence of* template *element.*

Calls:

$templateIDs$

4.4.3.39 Function $templateIDs$

Specification:

```
$templateIDs : Template \times CountID \to CountID$
$templateIDs\,(t, ci) \triangleq$
  let $aux = addMunion\,(t.id, ci)$ in
  cases $t.src\text{-}ele$ :
    mk-$Behavior$ (-,-,-,-,-) $\to$
        $behaviorIDs\,(t.src\text{-}ele, aux)$,
    mk-$Structure$ (-,-,-,-,-) $\to$
        $structureIDs\,(t.src\text{-}ele, aux)$,
    mk-$Style$ (-,-,-,-,-) $\to$
        $styleIDs\,(t.src\text{-}ele, aux)$,
    mk-$Content$ (-,-,-,-,-) $\to$
        $contentIDs\,(t.src\text{-}ele, aux)$,
    mk-$Constant$ (-,-,-,-,-,-,-) $\to$
        $constantsIDs\,([t.src\text{-}ele], aux)$,
    mk-$Property$ (-,-,-,-,-,-,-,-,-) $\to$
        $propertiesIDs\,([t.src\text{-}ele], aux)$,
    mk-$Peers$ (-,-,-,-,-) $\to$
        $peersIDs\,([t.src\text{-}ele], aux)$,
    mk-$Presentation$ (-,-,-,-,-,-) $\to$
        $preslogIDs\,([t.src\text{-}ele], aux)$,
    mk-$Logic$ (-,-,-,-,-) $\to$
        $preslogIDs\,([t.src\text{-}ele], aux)$,
    mk-$Part$ (-,-,-,-,-,-,-,-,-,-,-,-,-) $\to$
        $partsIDs\,([t.src\text{-}ele], aux)$,
    mk-$Restructure$ (-,-,-,-,-,-) $\to$
        $restructureIDs\,(t.src\text{-}ele, aux)$,
    mk-$Interface$ (-,-,-,-,-) $\to$
        $interfaceIDs\,([t.src\text{-}ele], aux)$,
    mk-$Rule$ (-,-,-,-,-,-) $\to$
        $rulesIDs\,([t.src\text{-}ele], aux)$,
    mk-$Script$ (-,-,-,-,-,-) $\to$
        $scriptIDs\,(t.src\text{-}ele, aux)$,
    mk-$D\text{-}class$ (-,-,-,-,-,-,-,-,-,-,-,-) $\to$
        $dclassIDs\,(t.src\text{-}ele, aux)$,
    mk-$D\text{-}component$ (-,-,-,-,-,-,-) $\to$
        $dcomponentIDs\,(t.src\text{-}ele, aux)$
  end;
```

Description:

Counts the occurrences of each id *on* template *element.*

Calls:

$behaviorIDs$,$structureIDs$,$styleIDs$,$contentIDs$,$constantsIDs$, $propertiesIDs$,$peersIDs$, $preslogIDs$,$partsIDs$,$restructureIDs$,$interfaceIDs$, $rulesIDs$,$scriptIDs$,$dclassIDs$, $dcomponentIDs$

4.4.3.40 Function $structureIDs$

Specification:

```
$structureIDs : Structure \times CountID \rightarrow CountID$
$structureIDs\,(s, c) \triangleq$
    let $aux = addMunion\,(s.id, c)$ in
    $partsIDs\,(s.parts, aux)$;
```

Description:

Counts the occurrences of each id *in a* structure *element.*

Calls:

$addMunion$,$partsIDs$

4.4.3.41 Function $scriptIDs$

Specification:

```
$scriptIDs : Script \times CountID \rightarrow CountID$
$scriptIDs\,(s, c) \triangleq$
    $addMunion\,(s.id, c)$;
```

Description:

Counts the occurrences of each id *in a* script *element.*

Calls:

$addMunion$

4.4.3.42 Function $styleIDs$

Specification:

```
$styleIDs : Style \times CountID \rightarrow CountID$
$styleIDs\,(s, c) \triangleq$
    let $aux = addMunion\,(s.id, c)$ in
    $propertiesIDs\,(s.property, aux)$;
```

Description:

Counts the occurrences of each id *in a* style *element.*

Calls:

$addMunion$,$propertiesIDs$

4.4.4 Attribute *part-name* must refer an existing *part ID* attribute

Invariant 9 **-** *The* property *attribute* part-name *must be defined in* ID *attribute of* part *elements*

The same reasoning must be applied to invariants 10,11,12,13, 14 and 15.

We must remember that several elements do not have part-name attributes. So they can be ignored.

4.4.4.1 Function $validProperties$

Specification:

```
$validProperties : Uiml \to \mathbb{B}$
$validProperties\,(u) \triangleq$
    let $memb = split\,(u.members, \text{mk-}UIMLMembers\,([], [], []))$,
        $sPN = interfacesPN\,(memb.I, \{\})$,
        $sPId = interfacesIDs\,(memb.I, \{\mapsto\})$ in
    $\forall\, x \in sPN \cdot$
        $\text{mk-}ID\,(x) \in \text{dom}\,(sPId)$;
```

Description:

Verify if all properties of an $UIML$ document are correct.

Calls:

$split$,$interfacesPN$,$interfacesIDs$

4.4.4.2 Function $interfacesPN$

Specification:

```
$interfacesPN$ : $Interface^* \times String$-set $\rightarrow$ $String$-set
$interfacesPN\ (si, ss) \triangleq$
  if $si = []$
  then $ss$
  else let $p =$ hd $(si)$,
           $cid = interfacePN\ (p.intele, ss)$ in
       $interfacesPN$ (tl $(si), cid)$;
```

Description:

Verify if all part-name *properties of all* interfaces *are correct.*

Calls:

$interfacePN$

4.4.4.3 Function $interfacePN$

Specification:

```
$interfacePN$ : $InterfaceElements^* \times String$-set $\rightarrow$ $String$-set
$interfacePN\ (si, ss) \triangleq$
  if $si = []$
  then $ss$
  else let $i =$ hd $(si)$ in
       cases $i$ :
         mk-$Structure$ (-, -, -, -, -) $\rightarrow$
             let $sPN = partsPN\ (i.parts, ss)$ in
             $interfacePN$ (tl $(si), sPN)$,
         mk-$Style$ (-, -, -, -, -) $\rightarrow$
             let $sPN = stylePN\ (i.property, ss)$ in
             $interfacePN$ (tl $(si), sPN)$,
         others $\rightarrow$ $interfacePN$ (tl $(si), ss)$
       end;
```

Description:

Verify if all part-name *of* interface *elements are correct.*

Calls:

$partsPN$,$stylePN$

4.4.4.4 Function $partsPN$

Specification:

```
$partsPN : Part^{*} \times String$-set $\rightarrow String$-set
$partsPN\ (sp, ss) \triangleq$
  if $sp = []$
  then $ss$
  else let $p =$ hd $(sp)$ in
       let $sPN = stylePN\ (p.style.property, ss)$,
           $pPN = partsPN\ (p.parts, sPN)$ in
       $partsPN$ (tl $(sp), pPN)$;
```

Description:

Verify if all part-name *of all* part *elements are correct.*

Calls:

$stylePN$

4.4.4.5 Function $stylePN$

Specification:

```
$stylePN : Property^{*} \times String$-set $\rightarrow String$-set
$stylePN\ (s, ss) \triangleq$
  if $s = []$
  then $ss$
  else let $p =$ hd $(s)$ in
       let $sPN = propertiesPN\ (p.property, ss)$ in
       $stylePN$ (tl $(s), sPN)$;
```

Description:

Verify if all part-name *of all* property *elements of* style *elements are correct.*

Calls:

$propertiesPN$

4.4.4.6 Function $propertiesPN$

Specification:

```
$propertiesPN : Property^* \times String$-set $\rightarrow String$-set
$propertiesPN\ (sp, ss) \triangleq$
  if $sp = []$
  then $ss$
  else let $p =$ hd $(sp)$,
             $cp = propertytypesPN\ (p.property, \{p.p\text{-}name\} \cup ss)$ in
        $propertiesPN$ (tl $(sp), cp)$;
```

Description:

Verify if all part-name *of each* property *sequence are correct.*

Calls:

$propertytypesPN$

4.4.4.7 Function $propertytypesPN$

Specification:

```
$propertytypesPN : PropertyType^* \times String$-set $\rightarrow String$-set
$propertytypesPN\ (sp, ss) \triangleq$
  if $sp = []$
  then $ss$
  else let $p =$ hd $(sp)$ in
        cases $p$ :
          mk-$Property$ (-, -, -, -, -, -, -, -, -) $\rightarrow$
              let $cps = propertytypesPN\ ([p], ss)$ in
              $propertytypesPN$ (tl $(sp), cps)$,
          mk-$Call$ (-, -) $\rightarrow$
              let $cc = paramsPN\ (p.params, ss)$ in
              $propertytypesPN$ (tl $(sp), cc)$
        end;
```

Description:

Verify if all part-name *of each* property type *sequence are correct.*

Calls:

$paramsPN$

4.4.4.8 Function $paramsPN$

Specification:

```
$paramsPN : Param^* \times String$-set $\rightarrow String$-set
$paramsPN\ (sp, ss) \triangleq$
  if $sp = []$
  then $ss$
  else let $p =$ hd $(sp)$,
           $pn = paramPN\ (p.type, ss)$ in
       $paramsPN$ (tl $(sp), pn)$;
```

Description:

Verify if all part-name *of each* param *element are correct.*

Calls:

$paramPN$

4.4.4.9 Function $paramPN$

Specification:

```
$paramPN : ParamType \times String$-set $\rightarrow String$-set
$paramPN\ (p, ss) \triangleq$
  cases $p$ :
    mk-$Property$ (-, -, -, -, -, -, -, -, -) $\rightarrow$
        $propertytypesPN\ ([p], ss)$,
    mk-$Call$ (-, -) $\rightarrow paramsPN\ (p.params, ss)$,
    mk-$Iterator$ (-, -) $\rightarrow iteratorPN\ (p, ss)$
  end;
```

Description:

Verify if all part-name *of each* param *element are correct.*

Calls:

$propertytypesPN$,$paramsPN$,$iteratorPN$

4.4.4.10 Function $iteratorPN$

Specification:

```
$iteratorPN : IteratorOptions \times String$-set $\rightarrow String$-set
$iteratorPN\ (i, ss) \triangleq$
   cases $i$ :
      mk-$Property$ (-, -, -, -, -, -, -, -, -) $\rightarrow$
            $propertytypesPN\ ([i], ss)$,
      mk-$Call$ (-, -) $\rightarrow paramsPN\ (i.params, ss)$
   end;
```

Description:

Verify if all part-name *of each* param *element are correct.*

Calls:

$propertytypesPN$,$paramsPN$

4.4.5 Auxiliary Functions

4.4.5.1 Function $split$

Specification:

```
$split : Member^* \times UIMLMembers \rightarrow UIMLMembers$
$split\ (m, spit) \triangleq$
   if $m = []$
   then $spit$
   else let $x =$ hd $(m)$ in
         cases $x$ :
            mk-$Peers$ (-, -, -, -, -) $\rightarrow$
                  $split$ (tl $(m)$,
                        mk-$UIMLMembers\ (spit.P \curvearrowright [x], spit.I, spit.T)$),
            mk-$Interface$ (-, -, -, -, -) $\rightarrow$
                  $split$ (tl $(m)$,
                        mk-$UIMLMembers\ (spit.P, spit.I \curvearrowright [x], spit.T)$),
            mk-$Template$ (-, -) $\rightarrow$
                  $split$ (tl $(m)$,
                        mk-$UIMLMembers\ (spit.P, spit.I, spit.T \curvearrowright [x])$)
         end;
```

Description:

Splits the three different members of uiml *element in different bags.*

Calls:

Standard VDM-SL only

4.4.5.2 Function $addMunion$

Specification:

```
$addMunion : [ID] \times CountID \to CountID$
$addMunion\,(id, sid) \triangleq$
  if $id =$ nil
  then $sid$
  else if $sid = \{\mapsto\}$
      then $id \mapsto 1 \rightharpoonup$
      else if $id \in$ dom $sid$
          then $id \mapsto sid\,(id) + 1 \rightharpoonup \biguplus(\{id\} \lhd sid)$
          else $id \mapsto 1 \rightharpoonup \biguplus sid;$
```

Description:

The data type CountID maps IDs to its number of occurrences. The function adds a new ID to an existing CountID.

Calls:

Standard VDM-SL only

4.4.5.3 Function $length$

Specification:

```
$length : (String \mid ID) \to \mathbb{N}$
$length\,(x) \triangleq$
  cases $x$ :
    mk-$ID\,(s) \to$ len $(s)$,
    others $\to$ len $(x)$
  end
```

Description:

Returns string length of Id, Source, Class, Model *and* Value *attributes.*

Calls:

Standard VDM-SL only

---

end *UIMLSpec*

4.5 Remarks

Once formalized, the *UIML VDM-SL* specification represents one of the main goals of this work. During this process, several direct interactions with *Harmonia* (Dr. Marc. Adams[4] and Dr. James Helms[5] were necessary, because of some inconsistencies in the *UIML* language definitions.

Due to the extension of *UIML* language, some particularities were not considered on this work. Mainly rules associated to *template* and *property* elements.

Now we are going to test the resultant *VDM-SL* specification, using a table object, that is the focus of the next chapter.

[4]mabrams@harmonia.com
[5]jhelms@harmonia.com

Chapter 5

Case study: Table IO

5.1 Overview

With the advent of personal computers (vulg. *laptop*) everybody should be nowadays familiar with spread-sheets and the tabular presentation of data on a computer screen. However, data reality is not always 2-dimensional. This entails the need for data-analytical processing, often disguised in the *GUI* for improved user friendliness.

In this chapter, we address the *UIML* specification and animation of such a basic GUI component - the *tabular* graphical interaction component.

The process will start by defining an abstract *VDM-SL* model for a *table* and by creating new functional methods (operators, functions and transformers) under the *UIML* specification of Chapter 4, mainly methods to support the most common features of OLAP technology (section 2.7.2).

5.2 Fundamentals of Table IO Formalization

Our case study — a *Table* as an *interaction IO* object — is very common in many *GUI*s, for direct structure data display, for immediate calculus framework like spread-sheets [JNZM93], Data Mining, statistical analysis, etc.

5.2.1 Considerations

When we talk about usual table visual components, we are referring to a bi-dimensional structure, a rectangle of cells, with a specific height and width. Nevertheless, we will see that more dimensions can arise.

In the literature, reference [Nig01] regards as a $dimension$ to the set of columns or rows, having a structure graphically represented by a *cube* (as we have seen in the section 2.7.2). Each cube face represents a particular set of three *dimensions*. This makes it easy to understand why this kind of structure is called a *Multidimensional Structure*.

Figure 5.1 depicts an example of this kind of cube. Looking at any of its faces, we see a table, considered as a structure with two or even three dimensions (in case where cells values are considered a dimension). Using the standard terminology, a table can be seen as a *Cross Tab View* or *Data Matrix*.

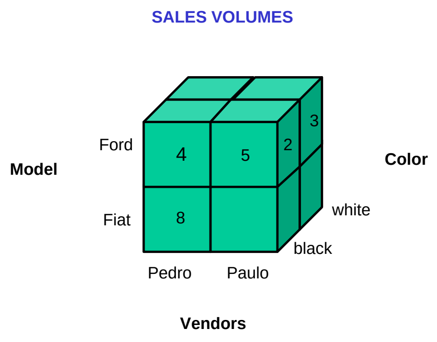


Figure 5.1: *Sales volumes HyperCube*

Informally, table contents are spread into units called *cells* disposed in *rows* and *columns*. Each cell is referred to by a pair of values (r, c), where r and c are the row and column numbers, respectively. There is a diversity of operations covering cells, rows and columns, as well as a large possibility of contents: literal values, equations, etc.

It is possible to enumerate several common basic attributes, properties and operators, which support table behavior, and organize them in three main classes: operators, functions and transformers [GLS96]. Section 5.3 summarizes these concepts. Because of its extension, we will only explore some of them in this work.

Using a mathematical notation, a tentative tabular model can be represented by a pair:

$$T = \langle N, V \rangle$$

where

N: represents a finite set of names (relations and attributes)

V: represents the set of possible values

Graphically, a table can be described as a set of singular *cells*, each uniquely determined by a pair (r, c) of the *Cartesian product* $\mathbb{N} \times \mathbb{N}$, corresponding to row and $column$ identifiers, respectively. Each $cell$ should host either a value $v \in V$ or *no value* (Table 5.1). To support *no values* we will use a special symbol $\perp$ (read $novalue$). In this way, to represent all possible displayable symbols, we will use $S = N \cup V$. where $V = \mathbb{N} \cup \{\perp\}$.

Let V_A be the the set of natural values for $column\ A$. To also include the $\perp$ value, we define:

$$V_A^{\perp} = V_A \cup \{\perp\}$$

In case of more than one column, say B, expression

SUN Microsystems Stock		
	Price	
Month	low	high
Jan 2001	25.438	34.875
Dec 2000	26.938	45.875
Nov 2000	$\perp$	56.532

Table 5.1: *Example of table data display*

$$V_A^{\perp} \times V_B^{\perp}$$

denotes the set of all values associated to all possible combinations of columns A and B, where

$$A \times B \quad = \quad \{(a, b) \mid a \in A \wedge b \in B\} \tag{5.1}$$

It may seem a little strange to justify this new value $\perp$. We will see this in a more detailed way when exploring transformer operators.

We do not intend to create, prove or improve a tabular algebra, as does [GLS96]. However, we will try to show how several different perspectives of data can be displayed, by just using a combination of known operators.

Distinct possibilities to manipulate and eventually transform a table to hold new operators/results, or even map to a different type of IO, is the main topic of the following section.

5.2.2 Table model

Resorting to $SETs$ mathematical notation [Hal60], the following definition for 2-dimensional tables

$$T \quad \cong \quad (\mathbb{N} \times \mathbb{N}) \to S \tag{5.2}$$

makes sense: for each Row and Column pair $(r, c) \in \mathbb{N} \times \mathbb{N}$, there is an associated value $v \in S$ where, as seen above, $\perp \in S$. Alternatively, T can be made into a finite partial map via isomorphism

$$(B + 1)^A \quad \cong \quad A \rightharpoonup B \tag{5.3}$$

(read the exponential as a functional space, $+$ as disjoint union and $1 = \{\perp\}$):

$$T \quad \cong \quad (\mathbb{N} \times \mathbb{N}) \rightharpoonup V \tag{5.4}$$

The fact the finite mappings are special cases of finite binary relations[1],

$$A \rightharpoonup B \quad \trianglelefteq \quad 2^{A\times B}_{fdp} \tag{5.5}$$

enables us to refine (5.2) into

$$T \quad \trianglelefteq \quad 2^{(\mathbb{N}\times\mathbb{N}\times V)} \tag{5.6}$$

which works as convenient (albeit redundant) definition of Table IO: *a set of triples Row, Column and Value*, i.e, a set of cells.

Moreover, from different instances of a particular table, we can see that the number of $columns$ stands invariable, contrary to the number of rows which can be more or less, depending of data display.

Let us focus on the original finite mapping definition (5.2). We know that the number of columns or rows in a table could be characterized as *dynamic*. Should we need operators to give the total number of elements in a table $t \in T$, we can compute $card(dom(t))$, where $card(s)$ denotes the cardinality of a finite set s and $dom(t)$ denotes the domain of definition of a partial finite map t. Alternatively, from facts

$$(A \times B) \rightharpoonup C \quad \trianglelefteq \quad A \rightharpoonup (B \rightharpoonup C) \tag{5.7}$$
$$2^A \quad \trianglelefteq_{elems} \quad A^* \tag{5.8}$$

(where $elems(l)$ returns the set of elements of sequence s) and equation (5.5), we can apply the following transformations to (5.4),

$$\begin{aligned} T &\trianglelefteq \mathbb{N} \rightharpoonup (\mathbb{N} \rightharpoonup V) \\ &\trianglelefteq \mathbb{N} \rightharpoonup (\mathbb{N} \times V)^* \\ &\trianglelefteq (\mathbb{N} \times (\mathbb{N} \times V)^*)^* \end{aligned} \tag{5.9}$$

Regarding the last expression above, $(\mathbb{N} \times V)^*$ can be seen as the list of columns and $(\mathbb{N} \times (\mathbb{N} \times V)^*)^*$ as the list of table rows.

Tables (5.2, 5.3, 5.4, 5.5), depict some table structures for different cases of data representation.

Month	Low	High
Jan 2001	25.438	34.875
Dec 2000	26.938	45.875
Nov 2000	$\perp$	56.532

Figure 5.2: $Sales : Month \rightharpoonup Low \times High$

Month	High
Jan 2001	25.438
Dec 2000	26.938
Nov 2000	$\perp$

Figure 5.3: $Sales : Month \rightharpoonup High$

In the sequel we will focus on formalizing the behaviour associated with our table model using the *VDM-SL* specification language, as earlier on in this document. As we will see, the domain of our table model can be single (represented by $String$) or composed elements (in our *VDM-SL* represented by Exp) using SET constructors,

[1]For details on the $\trianglelefteq$-ordering on data models, see e.g. [Oli92]

Month	Color	High
Jan 2001	Red	14.275
Dec 2000	Blue	12.600
Nov 2000	Gray	$\perp$

Figure 5.4: $Sales : Month \times Color \rightharpoonup High$

North	South	Total
Paul		140
	Mary	720
Sophy		210

Figure 5.5: $Sales : North \oplus South \rightharpoonup Total$

Cartesian Products $(A \times B)$, *Unions* $(A + B)$, *Finite Functions* $(A \rightharpoonup B)$, *Sets* (*set of Exp*), *Sequences* (*seq of Exp*)[2], etc.

Let us now relate this abstract model of a table with the *UIML VDM-SL* model presented earlier on.

In almost all common GUI applications, table components are used "inside" another component, usually named the *container object*, like $frames$, $forms$, etc, so a table must be considered as part of the defined GUI.

After analyzing several *UIML* fragments which describe table objects (Appendix G.2 presents an example), a conclusion can be clearly reached: in *UIML* notation, a simple table can be defined by expression (5.10), derived from the *UIML part* element:

$$Table \cong Style \times Part^* \times (ID + 1) \tag{5.10}$$

where $Style$, being optional, behaves as the table's graphical information, $Part^*$ lists the table rows and ID is an optional, unique table identifier. We will substitute $Part$ by Par_r, $Style$ by Sty_t and ignore table identifiers [3]. So, we rewrite:

$$Table \cong Sty_t \times Par_r^* \tag{5.11}$$

From the *UIML VDM-SL* specification, a *part* element (page 86) is defined, using abbreviated names, as:

$$Par \cong Sty \times Con \times Beh \times Par^* \times Rep^* \times ID \times S_{att} \tag{5.12}$$

Considering that we are working with simple tables, we will ignore Content (*Con*), Behavior (*Beh*), Repeat (*Rep*) and remainder attributes (S_{att}). So we define *UIML* $part$ element Par_r which abbreviates (5.12) to

$$Par_r \cong Sty_r \times Par_c^* \times ID \tag{5.13}$$

and describes table rows. In this expression, Par_c^* models the $Column$ set and Sty_r supports row graphic information.

Par_c — also derived from $part$ element — is defined as follows:

$$Par_c \cong Sty_c \times Par^* \times ID$$

In the same line of thought, Par^* will represent all $cells$.

[2]*seq of Exp* is pretty printed as Exp^* in *VDM-SL*

[3]Subscript *r, t, c* stand for: *t* - table, *r* - row and *c* - column.

Summarizing this process, the three equations which model our *UIML* table are:

$$\begin{aligned} Table &\cong Sty_t \times Par_r^* && (5.14)\\ Par_r &\cong Sty_r \times Par_c^* \times ID && (5.15)\\ Par_c &\cong Sty_c \times Par^* \times ID && (5.16) \end{aligned}$$

Looking now to the *style* element and recalling the corresponding *UIML* definition (page 88), it is defined as:

$$Sty \cong Pro^* \times (ID + 1) \times S_{att}$$

Again we can ignore optional attributes and abbreviate this definition into

$$Sty \cong Pro^*$$

where Pro models the *UIML* $property$ element (page 89). Finally, Pro can restrict itself to only its attribute *name* and respective value. Considering this, Pro is defined as:

$$Pro \cong Name \times String$$

The following *UIML* code fragment represents an example of this model:

```
<part id="col_x" class="Th">
    <style>
        <property name="content">SPANISH</property>
    </style>
</part>
```

To conclude our reasoning, it is important to consider the following transformation rules:

1. Suppose datatype is a sequence of tuples defined as follows:

$$NT = (A \times \cdots \times ID \times \cdots \times B)^*$$

   If the order in the sequences can be ignored, NT can be rewritten as

$$NT = \mathcal{P}(A \times \cdots \times ID \times \cdots \times B)$$

2. If there is a factor in Cartesian product $A \times \cdots \times B$ which can determine any other, such as *ID* in equation,

$$\begin{aligned} NT &= (A \times \cdots \times ID \times \cdots \times B)^* && (5.17)\\ inv\ l &== uniqueIDs(l) \end{aligned}$$

   it is possible to further refine NT as follows,

$$NT \trianglelefteq (\mathbb{N} \rightharpoonup ID) \times (ID \rightharpoonup (A \times \cdots \times B)) \qquad (5.18)$$

the first map capturing the sequence order and the functional dependency being recorded by the second map. In case where the order can be ignored, the first map can be deleted and NT can be refined as,

$$NT = ID \rightarrow (A \times \cdots \times B) \tag{5.19}$$

3. In case the expressions considered above are of type

$$\begin{aligned} NT &= (A \times ID \times B + \cdots + C \times ID \times D)^* \\ inv\ l &== uniqueIDs(l) \end{aligned}$$

and considering the distributive property of Cartesian product

$$A \times (B + C) \cong (A \times B) + (A \times C) \tag{5.20}$$

NT can be rewritten into

$$\begin{aligned} NT &= (ID \times (A \times B + \cdots + C \times D))^* \\ inv\ l &== uniqueIDs(l) \end{aligned}$$

whereby, via equation (5.17), NT can be converted into

$$NT = ID \rightarrow (A \times B + \cdots + C \times D) \tag{5.21}$$

Note that Par_r defined by equation (5.13) and Par_r^* are instances of equation (5.17). From equation (5.19), Par_r^* can be refined into

$$Par_r^* \cong ID \rightarrow Sty_r \times Par_c$$

The same reasoning can be applied to Par_c^*. By renaming Par_r^* by *Rows* and Par_c^* by *Cols*, our table model can finally be specified as follows:

$$\begin{aligned} Table &\cong Sty_t \times Rows && (5.22) \\ Rows &\cong ID \hookrightarrow Sty_r \times Cols && (5.23) \\ Cols &\cong ID \hookrightarrow Sty_c \times Part && (5.24) \\ Sty &\cong Pro^* && (5.25) \\ Pro &\cong Name \times Value && (5.26) \end{aligned}$$

To get some practical insight on this model and its representation in *UIML*, we will proceed to its animation in *VDM-SL*.

5.3 Table *VDM-SL* specification

This section presents the specification of the most common OLAP methods (functions, operators or transformers), grouped as general, multidimensional or auxiliary methods. To better understand their behavior, these methods can be considered as **attributes** if they support column and row characteristics (ex. *column name*, *column width*, etc.); **operators** if they work with table structure (*addRow*, *delRow*, etc.); **functions** if they support data calculus (*sum*, *avg*, *sort*, etc.) or even **transformers** if they manipulate the original table structure (*hideCol*, *drill-down*, *rotation*, etc.).

Considering operators which can change the original table structure, like adding and removing rows or columns, their most common impact results in a new different table.

As an implementation detail, we use a "mark" in *Style* the element of *T*, *Cols* and *Rows* elements of our *VDM-SL* specification. Every time one intends to identify a particular row or column, this mark can be used. We shall see that several operators will use it.

The tables which follow summarize all implemented methods, followed by their detailed specification analysis.

General table methods

These methods support common table features such as column, row and table properties as well as table structure manipulation.

Method	Description	Page
$mkTable$	creates an empty table	160
$rows$	returns the number of rows	161
$hideRow$	hides a particular row	161
$showRows$	shows all hiding rows	162
$delRow$	removes a table row	163
$addRow$	adds a new table row	163
$getRows$	returns the table rows	164
$rowValues$	returns all row values	164
$colValues$	returns all column values	165
$project$	returns a partial table	165
$rowcolValues$	returns a set of row/column values	166
$addCols$	adds a set of columns	167
$addCol$	adds a new empty column	168
$hideCol$	hides a column	169
$showCol$	renders a particular column visible	170
$getCols$	returns a column set	170
$getColsIds$	returns all column ID	171
$setCell$	sets a new cell value	171
$getCellValue$	returns a cell value	172
$getCell$	returns a cell value and style information	172

Multidimensional methods

These methods support multidimensional analysis operations, a set of OLAP important features.

Operator	Description	Page
$rotate$	executes a table rotation, changing rows by columns	174
$average$	calculates the average of a particular column (‘slice and dice’)	175
mda	calculates a column multidimensional analysis	176
$collect$	allows correlation between columns	177
$rollUp$	associates an upper hierarchy value	179
$rollColsHier$	allows the preservation of previous hierarchy value	180
$parent$	returns the previous value of a hierarchy	181
$children$	returns all direct hierarchy children	182
$family$	returns all hierarchy members	182
$childrenOf$	returns all children of a particular element	183
$drillDown$	executes a drill-down over a particular column	184
$drillColsHier$	gets details for a particular column value	184
$consolidate$	applies a total function to a Set of values (high order VDM function)	185
$summarize$	applies a binary function to a Set of values (high order VDM function)	185

Auxiliary methods

These methods are called in others methods.

Method	Description	Page
$markRow$	marks a particular table row	189
$unmarkRows$	unmarks all marked table rows	189
$addColsRows$	adds a new empty column	168
sum	binary function which calculates the sum of a set numbers	186
avg	binary function which calculates the average of a set numbers	187
max	determines the maximum of two values	187
min	determines the minimum of two values	188
$t2troll$	converts between table formats	190
$applyMon$	applies the function to all elements	190
$ff2set$	gets one set element	191
$mda2rows$	converts a mda result to rows	191
$mda2table$	converts a mda result to table	192
$setApplyElems$	applies a binary function to each set element	193
$setApply$	applies a function to a set	193
$map2set$	converts a mapping function to set	194
$map2map$	remove duplicate mapping elements	194
$set2seq$	converts a set to string (sequence)	195
$visibleRows$	counts only unmarked rows	195
$outHtml$	exports the table data to HTML	196
$outUiml$	exports the table data to *UIML*, using HTML vocabulary	196
$outUimlJ$	exports the table data to *UIML*, using JAVA vocabulary	197

As suggested in [MO85], we have defined a kind of $archetype$ with $signatures$ and definitions of main table *OLAP* operators. Perhaps this work could contribute towards a *table specification algebra* [GLS96].

Let us now analyze the *VDM-SL* specification itself.

module $UIMLSpecTab$

  imports

    from IO all ,

    from $UIMLSpec$ all

  exports all

definitions

$String = UIMLSpec`String;$
$Nat = \mathbb{N};$
$ID = String;$
$Rid = String;$
$Tid = String;$
$Cid = String;$
$Hier = Value \xrightarrow{m} Hier;$
$T = Style \times Rows;$
$Rows = Rid \xrightarrow{m} (Style \times Cols);$
$Cols = Cid \xrightarrow{m} (Style \times Value);$
$Style = Pro\text{-set};$
$Pro = String \times Value;$
$Value = [String \mid Nat \mid \mathbb{B} \mid \text{char} \mid \mathbb{Z} \mid \mathbb{R}]$

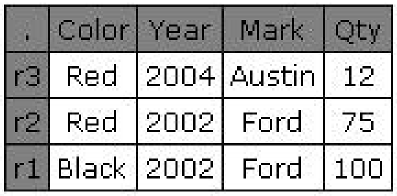
We shall now explore more deeply all defined operators. All definitions will be applied to our “working case” depicted in next table *VDM* specification, which represents a 3×4 table sales information of vehicles for a particular color, mark and year. This particular example[4] will be explored again in section 5.3.6 (page 206).

[4]This example could be created using functions *mkTable*, *addRows* and *addCol*. Because of its extension, we decided to define it directly as a *VDM-SL* data value.

```
values
    $t0 = $ mk- $(\{\}, \{\mapsto\});$
    $t = $ mk- $(\{\},$

              $"r3" \mapsto$ mk- $(\{\},$
                  $"Mark" \mapsto$ mk- $(\{$mk- $("content", "Austin")\}, "Austin"),$
                  $"Color" \mapsto$ mk- $(\{$mk- $("content", "Red")\}, "Red"),$
                  $"Qty" \mapsto$ mk- $(\{$mk- $("content", "12")\}, 12),$
                  $"Year" \mapsto$ mk- $(\{$mk- $("content", "2004")\}, 2004) \rightharpoonup),$
              $"r2" \mapsto$ mk- $(\{\},$
                  $"Mark" \mapsto$ mk- $(\{$mk- $("content", "Ford")\}, "Ford"),$
                  $"Color" \mapsto$ mk- $(\{$mk- $("content", "Red")\}, "Red"),$
                  $"Qty" \mapsto$ mk- $(\{$mk- $("content", "75")\}, 75),$
                  $"Year" \mapsto$ mk- $(\{$mk- $("content", "2002")\}, 2002) \rightharpoonup),$
              $"r1" \mapsto$ mk- $(\{\},$
                  $"Mark" \mapsto$ mk- $(\{$mk- $("content", "Ford")\}, "Ford"),$
                  $"Color" \mapsto$ mk- $(\{$mk- $("content", "Black")\}, "Black"),$
                  $"Qty" \mapsto$ mk- $(\{$mk- $("content", "100")\}, 100),$
                  $"Year" \mapsto$ mk- $(\{$mk- $("content", "2002")\}, 2002) \rightharpoonup) \rightharpoonup)$
```

This can be represented graphically as:



.	Color	Year	Mark	Qty
r3	Red	2004	Austin	12
r2	Red	2002	Ford	75
r1	Black	2002	Ford	100

Figure 5.6: *Sales table information*

5.3.1 General table methods

5.3.1.1 Function $mkTable$

Specification:

```
    $mkTable : Style \times Rows \rightarrow T$
    $mkTable\,(s, r) \triangleq$
        mk- $(s, r);$
```

Description:

> ***mkTable(s, rows)*** *creates an empty table. It prepares the workspace with the table header, for instance.*

Calls:

Standard VDM-SL only

For example, the creation of a default table, with no rows and no graphical information (background color, cell spacing, etc.), could be obtained by the following expression:

$$mkTable(\texttt{"dT"}, \{\}, \{| \text{-} >\});$$

Our table *t* could start its construction using:

$$mkTable(, \texttt{"r1"} \mid \text{-} > mk_(, \texttt{"Mark"} \mid \text{-} > mk_(, \texttt{"Ford"})));$$

(recall that $mk\_$ is an internal *VDM-SL* operator to construct records types [Hop01]).

We shall now focus on row operators, where processes like *adding*, *hiding* and *getting row values* are modeled.

5.3.1.2 Function $rows$

Specification:

$$\begin{array}{l} rows : T \to Nat \\ rows\,(t) \triangleq \\ \quad visibleRows\,(t.\#2); \end{array}$$

Description:

rows(t) *returns the number of rows. It only counts those rows which are not "marked".*

Calls:

$visibleRows$ (page 195)

Only rows not marked are "counted". Should we have no premise of "rows marked", the count would be directly given by $card(t.\#2)$ in our previous *VDM-SL* specification.

In our example, expression[5]:

$$rows(t1)$$

will return 3;

[5]In *VDM-Tools* this can be observed with command *print rows(t1).*

5.3.1.3 Function $hideRow$

Specification:

```
$hideRow : T \times Rid \rightarrow T$
$hideRow\,(t, rid) \triangleq$
   mk- $(t.\#1, markRow\,(t.\#2, rid, "h"))$
pre $rid \in$ dom $(t.\#2)$
   ;
```

Description:

hideRow(t,rid) *hides a particular Row in table* t.

Calls:

$markRow$ (page 189)

As previously mentioned, this operator marks the *style* of a specific row (*Rid*) with an h (as one can see in the third row of above *VDM-SL* code). In this way, row does not “appear” in subsequent operations, like $rows$, which counts table rows. It uses the auxiliary function $markRow$ (page 189).

5.3.1.4 Function $showRows$

Specification:

```
$showRows : T \rightarrow T$
$showRows\,(t) \triangleq$
   mk- $(t.\#1, unmarkRows\,(t.\#2));$
```

Description:

showRows(t) *shows all hidden rows.*

Calls:

$unmarkRows(page 189)$

This is the “converse” of *hideRow*. It performs a kind of “rollback”, rendering all rows “visible” again. So, one would expect the following property to hold:

$$showRows(hideRow(t, rid)) = t \qquad (5.27)$$

Note, however, that given sets A and B, in general,

$$(A \cup B)\text{-}B = A\text{-}B \neq A \tag{5.28}$$
$$(A\text{-}B) \cup B = A \cup B \neq A \tag{5.29}$$

Therefore, the equality in equation (5.28) hols only in case $A \cap B = \emptyset$. Concerning (5.29), equality holds in case $B \subseteq A$.

In consequence, property (5.27) fails when t already has some hidden rows. Equations (5.28) and (5.29) show this in relating set-difference (-) with *hideRow* and set-union ($\cup$) to *showRows*.

5.3.1.5 Function $delRow$

Specification:

$$\begin{aligned}
&delRow : T \times Rid \to T\\
&delRow\ (t, rid) \triangleq\\
&\quad \text{mk-}\ (t.\#1, \{rid\} \lhd t.\#2);
\end{aligned}$$

Description:

delRow(t, rid) *removes table rows.*

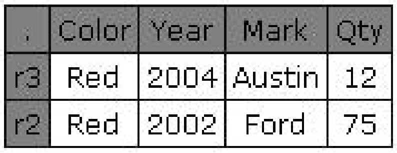
Calls:

Standard VDM-SL only

In our example, expression

$$delRow(t, \text{"r1"})$$

will remove all the information concerning $Black\ Ford$, and consequently, expression $rows(t)$ will now return *2* (see figure 5.7).



:	Color	Year	Mark	Qty
r3	Red	2004	Austin	12
r2	Red	2002	Ford	75

Figure 5.7: *delRow operation result*

5.3.1.6 Function $addRow$

Specification:

$$\begin{array}{l} addRow : T \times Rid \times Style \to T \\ addRow\,(t, rid, sty) \triangleq \\ \quad \mathsf{mk\text{-}}\,(t.\#1, t.\#2 \uplus^{m} rid \mapsto \mathsf{mk\text{-}}\,(sty, \{\mapsto\})\ \rightharpoonup); \end{array}$$

Description:

***addRow(t,rid,s)** adds a new table row width graphical information.*

Calls:

Standard VDM-SL only

So, the second row of our case study could have been created by the expression:

$$addRow(t, \texttt{"r2"}, \{\})$$

considering no style (graphical) information (third argument is $\{\}$).

The next operator $rowValues$ gets all specific row values present in the table. Formally, the specification is similar to *colValues*, thinking now in terms of horizontal values.

5.3.1.7 Function $rowValues$

Specification:

$$\begin{array}{l} rowValues : T \times Rid \to Value\text{-}\mathsf{set} \\ rowValues\,(t, rid) \triangleq \\ \quad \{t.\#2\,(rid).\#2\,(ci).\#2 \mid ci \in \mathsf{dom}\,(t.\#2\,(rid).\#2)\} \\ \mathsf{pre}\ \ rid \in \mathsf{dom}\,(t.\#2) \\ \quad ; \end{array}$$

Description:

***rowValues(t,rid)** returns all values in rid.*

Calls:

Standard VDM-SL only

In our example, the expression:

$$rowValues(t, \texttt{"r1"})$$

will return the set $\{\texttt{"100"}, \texttt{"2002"}, \texttt{"Ford"}, \texttt{"Black"}\}$.

5.3.1.8 Function $getRows$

Specification:

$$
\begin{array}{l}
getRows : T \rightarrow Rows \\
getRows\,(t) \triangleq \\
\quad t.\#2;
\end{array}
$$

Description:

***getRows(t)** gets table rows. It works as an auxiliary function.*

Calls:

Standard VDM-SL only

We will see an example of *getRows* application under *unmarkRows* auxiliary function description (on page 189).

The same reasoning is applicable to columns. Hiding columns, selecting column data, adding columns, etc., are also necessary operators. Let us now analyze some of them.

Focusing on the $Projection$ operator of *Relational Algebra*, which gives all values for a specific table column, we have created the $colValues$ operator.

5.3.1.9 Function $colValues$

Specification:

$$
\begin{array}{l}
colValues : T \times Cid \rightarrow Value\text{-set} \\
colValues\,(t, cid) \triangleq \\
\quad \{t.\#2\,(ri).\#2\,(cid).\#2 \mid ri \in \mathsf{dom}\,(t.\#2)\};
\end{array}
$$

Description:

***collValues(t,cid)** returns all values in column* cid

Calls:

Standard VDM-SL only

In our example, expression:

$$colValues(t, \text{"Mark"})$$

will return the set $\{Ford, Austin\}$. Remember that there are two values $Ford$. But, as we are working with set expressions, this redundancy is eliminated.

Another instance of the traditional $projection$ operator could be supported for the following method $project$.

5.3.1.10 Function $project$

Specification:

```
$project : T \times Cid$-set $\rightarrow T$
$project\,(t, scid) \triangleq$
  if $scid = \{\}$
  then $t$
  else mk- $(t.\#1, map2map[Rid, Style, Cols]$
                 (
                  $getRows$ (mk- $(t.\#1, \{rid \mapsto$ mk- $(t.\#2\,(rid).\#1,$
                                   $\{cid \mapsto getCell\,(t, rid, cid) \mid cid \in$
$scid\}) \mid$
                                $rid \in$ dom $(t.\#2)\})),$
                  $\{\mapsto\}));$
```

Description:

***project(t,sc)** returns a partial table*

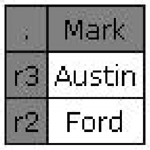
Calls:

$map2map$ (page 194), $getRows$ (page 164), $getCell$ (page 172)

In our example, expression:

$$project(t, \{\text{"Mark"}\})$$

will return a single column table with the *Mark* column information, as depicted in Figure 5.8.



.	Mark
r3	Austin
r2	Ford

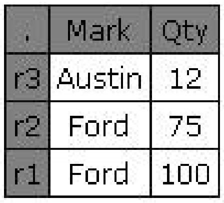
Figure 5.8: *Single column project operation*

Otherwise, the following expression will return all information of *Mark* and *Qty*, in a two column table (Figure 5.9):

$$project(t, \{\text{"Mark"}, \text{"Qty"}\})$$

Unlike $colValues$, which returns a set of values, this $project$ operator returns a new table (sub-table) formed by the selected columns.

We must remember now the operator $rowValues$ (specified in page 164), which does the same as this one, but working with horizontal values.



.	Mark	Qty
r3	Austin	12
r2	Ford	75
r1	Ford	100

Figure 5.9: *Multiple column project operation*

5.3.1.11 Function $rowcolValues$

Specification:

$$
\begin{aligned}
&rowcolValues : Rows \times Cols \times Cid \to Value\text{-set}\\
&rowcolValues\,(r, c, cid) \triangleq\\
&\quad \{r\,(x).\#2\,(cid).\#2 \mid x \in \mathsf{dom}\,(r) \cdot (\mathsf{dom}\,(c) \lhd r\,(x).\#2) = c\};
\end{aligned}
$$

Description:

rowcolValues(rows, cols, cid) *returns a particular set of column values for a particular set of rows*

Calls:

Standard VDM-SL only

Let us now consider the operators $addCols$, $addCol$ and $addNCol$, which allow for column insertion. The last one allows for the insertion of a new column in a particular position (N). The corresponding signatures are:

$$addCols : Table * Cols \to Table$$

$$addCol : Table * Cid * Style \to Table$$

$$addNCol : Table * Col * N \to Table$$

Let us explore $addCol$ and $addCols$. The same modelling decision applies to $addNcol$.

5.3.1.12 Function $addCols$

Specification:

$$
\begin{aligned}
&addCols : T \times Cols \to T\\
&addCols\,(t, sc) \triangleq\\
&\quad \mathsf{mk\text{-}}\,(t.\#1, addColsRows\,(t.\#2, sc));
\end{aligned}
$$

Description:

> ***addCols(t, cols)** adds a set of columns information (value and graphical information) to table*

Calls:

> $addColsRows$ (page 168)

In our example, the "Mark" column could have been created using the following expression (considering no row cell values):

$$addCols(t, \{\text{"Mark"} \mid \text{-} > \{\}\})$$

5.3.1.13 Function $addCol$

Specification:

```
$addCol : T \times Cid \times Style \to T$
$addCol\,(t, cid, sty) \triangleq$
  $\mathsf{mk\text{-}}\,(t.\#1,$
      $t.\#2 \dagger \{rid \mapsto \mathsf{mk\text{-}}\,(t.\#2\,(rid).\#1,$
            $t.\#2\,(rid).\#2 \uplus cid \mapsto \mathsf{mk\text{-}}\,(sty, \text{""}) \rightharpoonup) \mid$
          $rid \in \mathsf{dom}\,(t.\#2)\});$
```

Description:

> ***addCol(t, cid, s)** adds a new empty column to table*

Calls:

> Standard VDM-SL only

In our example, the "Color" column could have been created using expression:

$$addCol(t, \text{"Color"}, \{\})$$

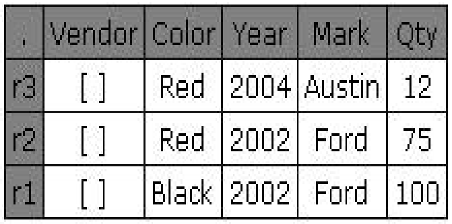
From the following expression,

$$addCol(t, \text{"Color"}, \{\}) \cong addCols(t, \{\text{"Color"} \mid \text{-} > \{\}\})$$

we can see that $addCol$ is an instance of $addCols$.
Considering this, the creation of our table (without cell values) could have been made by the following expression:

$$addCol(addCol(addCol(mkTable(\text{"Sales"}, \{\}, \{\mid \text{-} >\}), \text{"Mark"}), \text{"Color"}), \text{"Qty"})$$

Figure 5.10 depicts the result of applying $addCol(t, \text{"Vendor"})$ to our initial table.



.	Vendor	Color	Year	Mark	Qty
r3	[ ]	Red	2004	Austin	12
r2	[ ]	Red	2002	Ford	75
r1	[ ]	Black	2002	Ford	100

Figure 5.10: *New column applying* $addCol$

5.3.1.14 Function $addColsRows$

Specification:

$$
\begin{aligned}
&addColsRows : Rows \times Cols \rightarrow Rows\\
&addColsRows\,(rs, sc) \triangleq\\
&\quad \{x \mapsto \mathsf{mk\text{-}}\,(rs\,(x).\#1, (rs\,(x).\#2) \mathbin{\underset{m}{\cup}} sc) \mid x \in \mathsf{dom}\,(rs)\};
\end{aligned}
$$

Description:

***addColsRows(rows,cols)** is an auxiliary function used by* $addCols$.

Calls:

Standard VDM-SL only

As we have seen in the specification of row operators, it should be possible to hide some table information. The method $hideCol$ is specified for this purpose. In these situations, the original table state must not be changed.

5.3.1.15 Function $hideCol$

Specification:

$$
\begin{aligned}
&hideCol : T \times Cid \rightarrow T\\
&hideCol\,(t, cid) \triangleq\\
&\quad \mathsf{mk\text{-}}\,(t.\#1,\\
&\qquad\quad t.\#2 \dagger \{rid \mapsto \mathsf{mk\text{-}}\,(t.\#2\,(rid).\#1, t.\#2\,(rid).\#2 \dagger\\
&\qquad\qquad\quad cid \mapsto \mathsf{mk\text{-}}\\
&\qquad\qquad\qquad (\\
&\qquad\qquad\qquad\ t.\#2\,(rid).\#2\,(cid).\#1 \cup\\
&\qquad\qquad\qquad\ \{\mathsf{mk\text{-}}\,(\text{"}h\text{"}, \text{"}0\text{"})\},\\
&\qquad\qquad\qquad\ t.\#2\,(rid).\#2\,(cid).\#2) \rightharpoonup) \mid\\
&\qquad\qquad rid \in \mathsf{dom}\,(t.\#2)\});
\end{aligned}
$$

Description:

> ***hideCol(t,cid)*** *hides a particular column on table t. As in* <u>hideRow</u> *function, a "mark" h is activated on* style *attribute.*

Calls:

> Standard VDM-SL only

For instance, expression:

$$hideCol(t, \texttt{"Mark"})$$

will hide vehicle mark information in our running example. The table will keep only two columns visible.

An operator which does the reverse of $hideCol$ can be inferred from it. Let us call it $showCol$. Our reasoning concerning *showRows* and $hideRow$ (page 161) should be considered here too. So, property

$$showCol(hideCol(t, c_i), c_i) = t$$

is not be always valid.

5.3.1.16 Function $showCol$

Specification:

```
$showCol : T \times Cid \to T$
$showCol\,(t, cid) \triangleq$
  mk- $(t.\#1,$
      $t.\#2 \dagger \{rid \mapsto$ mk- $(t.\#2\,(rid).\#1, t.\#2\,(rid).\#2 \dagger$
            $cid \mapsto$ mk-
                $($
                 $t.\#2\,(rid).\#2\,(cid).\#1 \setminus$
                 $\{$mk- $(\texttt{"}h\texttt{"}, \texttt{"}0\texttt{"})\},$
                 $t.\#2\,(rid).\#2\,(cid).\#2) \rightharpoonup) \mid$
          $rid \in$ dom $(t.\#2)\});$
```

Description:

> ***showCol(t, cid)*** *renders visible the column Cid. It is the converse function of $hideCol$.*

Calls:

> Standard VDM-SL only

5.3.1.17 Function $getCols$

Specification:

$$\begin{array}{l}
getCols : Rows \times Cid\text{-set} \to Cols\text{-set} \\
getCols\,(r, scid) \triangleq \\
\quad \text{let } cols = \lambda\, ri : Rid \cdot (scid \lhd r\,(ri).\#2) \text{ in} \\
\quad \{cols\,(ri) \mid ri \in \text{dom } r\};
\end{array}$$

Description:

***getCols(rows, cols)** returns a particular column set.*

Calls:

Standard VDM-SL only

5.3.1.18 Function $getColsIds$

Specification:

$$\begin{array}{l}
getColsIds : T \to Cid\text{-set} \\
getColsIds\,(t) \triangleq \\
\quad \bigcup\{\text{dom } (t.\#2\,(rid).\#2) \mid rid \in \text{dom } (t.\#2)\};
\end{array}$$

Description:

***getColsIds(t)** returns all column ID.*

Calls:

Standard VDM-SL only

If we intend to set or get values to/from a specific cell, the next functions will be useful.

5.3.1.19 Function $setCell$

Specification:

$$\begin{array}{l}
setCell : T \times Rid \times Cid \times Value \to T \\
setCell\,(t, rid, cid, v) \triangleq \\
\quad \text{mk-}\,(t.\#1, t.\#2 \dagger \\
\qquad\quad rid \mapsto \text{mk-}\,(t.\#2\,(rid).\#1, \\
\qquad\qquad\quad t.\#2\,(rid).\#2 \dagger \\
\qquad\qquad\quad cid \mapsto \text{mk-}\,(t.\#2\,(rid).\#2\,(cid).\#1, v) \rightharpoonup) \rightharpoonup);
\end{array}$$

Description:

setCell(t, rid, cid, v) *set a new value v to a particular cell.*

Calls:

Standard VDM-SL only

In our example, the values for column *Mark* could have been by the following expressions:

$$\begin{aligned} t1 &= setCell(t, \text{"r1"}, \text{"Mark"}, \text{"Ford"}) \\ t2 &= setCell(t1, \text{"r2"}, \text{"Mark"}, \text{"Ford"}) \\ t3 &= setCell(t2, \text{"r3"}, \text{"Mark"}, \text{"Austin"}) \end{aligned}$$

In the same way, to get the respective values, one can use the function $getCellValue$, defined as follows.

5.3.1.20 Function $getCellValue$

Specification:

```
$getCellValue : T \times Rid \times Cid \to Value$
$getCellValue\,(t, rid, cid) \triangleq$
  if $t = $ mk- $(\{\}, \{\mapsto\})$
  then nil
  else $t.\#2\,(rid).\#2\,(cid).\#2;$
```

Description:

getCellValue(t, rid, cid) *returns the value of a particular cell.*

Calls:

Standard VDM-SL only

Considering this, the following expression should be correct:

$$getCellValue(setCell(t, \text{"r2"}, \text{"Qty"}, 123), \text{"r2"}, \text{"Qty"}) = 123$$

5.3.1.21 Function $getCell$

Specification:

$$
\begin{aligned}
&getCell : T \times Rid \times Cid \to (Style \times Value)\\
&getCell\,(t, rid, cid) \triangleq\\
&\quad \textsf{mk-}\,(t.\#2\,(rid).\#2\,(cid).\#1, t.\#2\,(rid).\#2\,(cid).\#2);
\end{aligned}
$$

Description:

***getCell(t, rid, cid)** returns all cell information.*

Calls:

Standard VDM-SL only

5.3.2 Towards Multidimensional Analysis

5.3.2.1 Overview

As we have seen in section 2.7.1, a dimension is a perspective or view of a specific dataset. A different view of the same data is an alternative dimension. A system that supports simultaneous, alternative views of datasets is said to be multidimensional. Dimensions are typical categories such as time, accounts, regions, markets, budgets, and so on. Each dimension contains additional categories that could have various relationships one to another.

Considering this, many aspects motivate current advances on database technologies or more clearly, on Data Center technologies. Questions related to storage capacity are well-known (replication, clustering, etc.), but in what it concerns to demand, we may say that it is still evolving. Issues like forecasting, comparative analysis, "What-if" analysis, etc., are very important and demand support.

Spreadsheet manipulation, arranging and storing related data, summarizing data, partitioning repositories, results from multidimensional databases and respective multidimensional analysis, like *OLAP* [Ma98]. This will be the context that we will explore in the formal specification which follows. Our table example should be used again to animate these operators.

VDM-SL has a particular feature which supports this kind of operation. We are referring to $high\ order\ VDM$ [IFA00c, Hop01] functions[6] which offer polymorphic properties, ie, the same definition allows operations over different data types. Informally, summarizing data, for instance, results from applying some calculus over a dataset, like sum, $maximum$, etc. It should be possible to work with any data type. The symbol @ in the next expressions, represent type parameters.

Supposing that we need to analyze the data display from different points of view, i.e, organizing data under different criteria, such as:

[6]Functions can receive others functions as arguments

- Calculate the total of values for a specific expression (column)
- Rotate the whole table, i.e, change columns by rows
- See more or less details for a specific expression
- Consolidate data
- Apply some calculus to all elements on a specific expression

We are going through *OLAP* [Ma98, SCJS01] rules and procedures, dealing with *Multidimensional Databases* [Nig01] and *Data Mining* [MK97] questions.

As mentioned in section 2.7.2, operations like *Rotation*, $Roll\text{-}Up/Drill\text{-}Down$, $Slicing$, $Consolidate$, etc., seem to be useful and worthwhile table features. Next we will try to illustrate some of them.

5.3.2.2 Function $rotate$

Specification:

```
$rotate : T \to T$
$rotate\,(t) \triangleq$
  let $ri \in$ dom $(t.\#2)$ in
  let $cols = t.\#2\,(ri).\#2$ in
  mk- $(t.\#1,$
      $\{ci \mapsto$ mk- $(t.\#2\,(ri).\#1,$
           $\{r \mapsto t.\#2\,(r).\#2\,(ci) \mid r \in$ dom $(t.\#2)\}) \mid$
         $ci \in$ dom $(cols)\});$
```

Description:

***rotate(t)** performs a table rotation, changing rows by columns.*

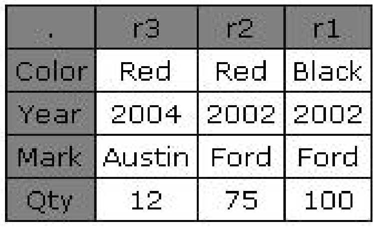
Calls:

Standard VDM-SL only

As mentioned in section 2.7.1, *rotation* is a most useful *OLAP* feature. On a multidimensional $2 \times 2 \times 2$ representation, as illustrated by the *hypercube* of Figure 5.1 (page 151), there are 8 $cells$ corresponding to the 8 records in an equivalent relational representation. A desired analysis may require any combination of dimensions to be reported. So one should be able to “rotate” the cube data view to allow the visualization of all faces.

This method should be idempotent:

$$rotate(rotate(t)) = t$$



.	r3	r2	r1
Color	Red	Red	Black
Year	2004	2002	2002
Mark	Austin	Ford	Ford
Qty	12	75	100

Figure 5.11: *Result of rotation*

Figure 5.11 shows the application of this method in our *Sales* example.

Even if this behavior looks simple, there are in fact several situations to explore. Because this is not in the scope of this thesis, all other situations will not be worked out here. Note that a $rotation$ is sometimes referred to as $dataslicing$ because each rotation yields a different $slice$ or two dimensional table of data [Nig01].

Now suppose that we want to simplify some data, calculate or group some values, etc. This is the purpose of *OLAP* $Roll\text{-}Up/Drill\text{-}Down$ and $Consolidate$ operators. In our work, the $consolidate$ operator applies some unary operators to all elements in a dataset. We are talking about usual operations like sum; $count$, $average$, etc.

5.3.2.3 Functions *Slice and Dice*

Quite often, the *dataset* must be scoped down to a subset grouping, formed by smaller tables, rows or column subsets. It is the area of $Ranging$ operations, in $OLAP$ terminology, many times called also by "$slice$ and $dice$" calculations. In our experience, we can assume that these operations are supported by functions $colValues$ (page 165), $rowValues$ (page 164) and even $partition$ (another name for the $project$ method (page 165).

5.3.2.4 Function $averag$

Specification:

$$\begin{array}{l} averag : T \times Cid \to \mathbb{Z} \\ averag\,(t, cid) \triangleq \\ \quad avg\,(\{t.\#2\,(ri).\#2\,(cid).\#2 \mid ri \in \mathsf{dom}\,(t.\#2)\}) \\ \mathsf{pre}\;\; cid \in getColsIds\,(t) \end{array}$$

Description:

***average(t,cid)** calculates the average of a particular column.*

Calls:

avg (page 187), $getColsIds$ (page 171)

Considering our table t, expression

$$averag(t, \text{"Qty"})$$

will return 62.

Let us consider now the following auxiliary VDM type to be used in next functions.

$$ColRelatedSimple = ((Cid \times Value)\text{-set}) \xrightarrow{m} (Cid \xrightarrow{m} Value)$$

5.3.2.5 Function mda

Specification:

```
$mda[@A] : T \times Cid\text{-set} \times Cid \times (@A \to @A) \times Value \to ColRelatedSimple$
$mda\,(t, scid, cid, f, v) \triangleq$
  let $rows = getRows\,(t)$ in
  $applyMon[@A]\,(collect\,(rows, getCols\,(rows, scid), \{\mapsto\}, cid), f, v)$
```

Description:

mda(cids, cid, func, null_value) *calculates a multidimensional analysis over a particular column, using a particular function.*

Calls:

$applyMon$ (page 190), $collect$ (page 177), $getRows$ (page 164), $getCols$ (page 170)

As mentioned earlier on, multidimensional analysis is a feature which allows for data analysis over dimensions and calculations over datasets. These calculations should be supported by usual mathematical functions, e.g. *addition*.

Considering this, we will see that the *mda* method implements this feature for a particular set of binary functions.

This method uses *high order VDM properties* (identified by character @), which allow for polymorphic properties, ie, support several data types (integer, booleans, etc.) as arguments.

Our next example shows its application to integer (*int*) data types, using as argument the sum binary operation,

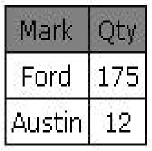
$$mda[int](t, \{\text{"Qty"}, \text{"Mark"}, \text{"Color"}\}, \text{"Qty"}, sum, 0)$$

which should perform a sum over the Qty column, getting the respective Qty sum over $Mark$ and $Color$ vehicle.

If we intend to get Qty for each $Mark$ vehicle, ignoring its color, we can use the following expression:

$$mda[int](t, \{\text{"Qty"}, \text{"Mark"}\}, \text{"Qty"}, sum, 0)$$

Figure 5.12 depicts this example, where value *175* results from adding *100* to *75* (both *Ford* vehicle quantities).



Mark	Qty
Ford	175
Austin	12

Figure 5.12: *Result of applying* mda

Sometimes it is important to identify relations between column or rows. For instance, to analyze the sales quantity of vehicles for a particular color, get the most sold vehicle mark; etc. These kind of relations can be achieved by the method which follows.

$$ColRelated = ((Cid \times Value)\text{-set}) \xrightarrow{m} (Cid \xrightarrow{m} Value\text{-set})$$

5.3.2.6 Function $collect$

Specification:

```
$collect : Rows \times Cols\text{-set} \times ColRelated \times Cid \rightarrow ColRelated$
$collect\,(sr, scol1, m, cid) \triangleq$
  if $scol1 = \{\}$
  then $m$
  else let $x \in scol1$ in
      let $y = \{cid\} \lhd x$,
         $z = \{cid\} \ntriangleleft x$,
         $sc = f\!f2set[Cid, Style, Value]\,(y)$ in
      if $sc \in \mathrm{dom}\,(m)$
      then let $v = getOne[Value\text{-set}]\,(\mathrm{rng}\,(m\,(sc)))$ in
          if $z = \{\mapsto\}$
          then $collect\,(sr, scol1 \setminus \{x\}, m\,\dagger$
                    $sc \mapsto cid \mapsto rowcolValues\,(sr, y, cid)$ ⇀⇀,
                    $cid)$
          else let $setv = getOne[Cid \times Value]\,(f\!f2set[Cid, Style, Value]\,(z))$,
                 $col = setv.\#1$,
                 $v1 = setv.\#2$ in
               $collect\,(sr, scol1 \setminus \{x\}, m \dagger sc \mapsto col \mapsto v \cup \{v1\}$ ⇀⇀, $cid)$
      else if $z = \{\mapsto\}$
          then $collect\,(sr, scol1 \setminus \{x\}, m \uplus$
                    $sc \mapsto cid \mapsto rowcolValues\,(sr, y, cid)$ ⇀⇀,
                    $cid)$
          else let $col = getOne[Cid \times Value]\,(f\!f2set[Cid, Style, Value]\,(z)).\#1$,
                 $v = getOne[Cid \times Value]\,(f\!f2set[Cid, Style, Value]\,(z)).\#2$ in
               $collect\,(sr, scol1 \setminus \{x\}, m \uplus f\!f2set[Cid, Style, Value]\,(y) \mapsto$
$col \mapsto \{v\}$ ⇀⇀, $cid)$
```

Description:

collect(rows,cols,colr,cid) *allows for the correlation among columns*

Calls:

$f\!f2set$ (page 191), $getOne$ (page 191), $colValues$ (page 165)

Suppose one intends to know all different sale quantities for each vehicle's color. This can be obtained by evaluating:

$collect(getRows(t), getCols(getRows(t), \{$"Qty", "Color"$\}), \{|$ - $>\}, $"Qty"$)$

and the result should be:

$Red \rightarrow \{12, 75\}$

$Black \rightarrow \{100\}$

Supposing now one intends to know detailed information about all vehicle sale by mark. A possible expression could be:

$$collect(getRows(t), getCols(getRows(t), \{\text{"Qty"}, \text{"Mark"}\}), \{|\text{ - }>\}, \text{"Qty"})$$

and the result should be:

$Ford \rightarrow \{75, 100\}$
$Austin \rightarrow \{12\}$
Consider now the next VDM types to be used in following functions.

$TRoll = Style \times RowsRoll;$
$RowsRoll = Rid \xrightarrow{m} (Style \times ColRoll);$
$ColRoll = Cid \xrightarrow{m} (Style \times Value \times Value^*)$

5.3.2.7 Function *rollUp*

Specification:

```
$rollUp : TRoll \times Hier \times Cid \rightarrow TRoll$
$rollUp\ (t, h, cid) \triangleq$
  mk- $(t.\#1,$
      $\{ri \mapsto$ mk- $(t.\#2\ (ri).\#1,$
            $rollColsHier\ (t.\#2\ (ri).\#2, h, cid))\ |$
         $ri \in$ dom $(t.\#2)\})$
```

Description:

rollUp(t,h,c) *associates an upper hierarchy value*

Calls:

rollColsHier (page 180)

As referred to in section 2.7.1, *roll-up* operations allow for simplification of data representation.

To model a *roll-up* process, there must be a hierarchical representation of data which have to be related. For instance,

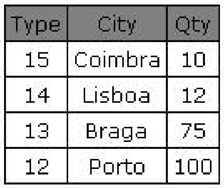
```
values
  $h = \text{"}Region\text{"} \mapsto$
      $\text{"}North\text{"} \mapsto \text{"}Porto\text{"} \mapsto \{\mapsto\}, \text{"}Braga\text{"} \mapsto \{\mapsto\} \rightharpoondown,$
      $\text{"}Middle\text{"} \mapsto \text{"}Coimbra\text{"} \mapsto \{\mapsto\} \rightharpoondown,$
      $\text{"}South\text{"} \mapsto \text{"}Lisboa\text{"} \mapsto \{\mapsto\} \rightharpoondown\rightharpoondown\rightharpoondown;$
```

describes such relation between *country*, *region* and *city*. Interpreting them, for instance, we can say that *Porto* and *Braga* cities are in *North*. In this case, *Region* is the hierarchy top.

Considering this, let us analyze a $roll\text{-}up$ application over table

$$\begin{array}{l}
th = \textsf{mk-}(\{\},\\
\qquad\qquad "r4" \mapsto \textsf{mk-}(\{\},\\
\qquad\qquad\qquad "City" \mapsto \textsf{mk-}(\{\}, "Porto"),\\
\qquad\qquad\qquad "Qty" \mapsto \textsf{mk-}(\{\}, 100),\\
\qquad\qquad\qquad "Type" \mapsto \textsf{mk-}(\{\}, 12) \rightharpoonup),\\
\qquad\qquad "r3" \mapsto \textsf{mk-}(\{\},\\
\qquad\qquad\qquad "City" \mapsto \textsf{mk-}(\{\}, "Braga"),\\
\qquad\qquad\qquad "Qty" \mapsto \textsf{mk-}(\{\}, 75),\\
\qquad\qquad\qquad "Type" \mapsto \textsf{mk-}(\{\}, 13) \rightharpoonup),\\
\qquad\qquad "r2" \mapsto \textsf{mk-}(\{\},\\
\qquad\qquad\qquad "City" \mapsto \textsf{mk-}(\{\}, "Lisboa"),\\
\qquad\qquad\qquad "Qty" \mapsto \textsf{mk-}(\{\}, 12),\\
\qquad\qquad\qquad "Type" \mapsto \textsf{mk-}(\{\}, 14) \rightharpoonup),\\
\qquad\qquad "r1" \mapsto \textsf{mk-}(\{\},\\
\qquad\qquad\qquad "City" \mapsto \textsf{mk-}(\{\}, "Coimbra"),\\
\qquad\qquad\qquad "Qty" \mapsto \textsf{mk-}(\{\}, 10),\\
\qquad\qquad\qquad "Type" \mapsto \textsf{mk-}(\{\}, 15) \rightharpoonup) \rightharpoonup)
\end{array}$$

which can be graphically depicted by Figure 5.13,



Type	City	Qty
15	Coimbra	10
14	Lisboa	12
13	Braga	75
12	Porto	100

Figure 5.13: *Sales per region*

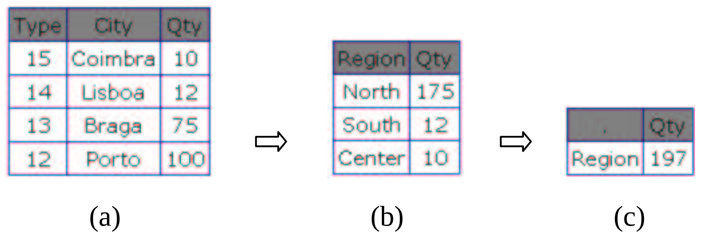
Representing quantities (*Qty*) by region, expression

$$rollUp(t2troll(th), h, \text{"City"})$$

can be successively applied until the top of hierarchy is reached (auxiliary function t2troll is defined on page 190). Each *rollUp* operation applies an aggregating function (*sum* in this case). Figure 5.14 shows the outcome of successive *rollUp* applications.

5.3.2.8 Function $rollColsHier$

Specification:



(a)

Type	City	Qty
15	Coimbra	10
14	Lisboa	12
13	Braga	75
12	Porto	100

(b)

Region	Qty
North	175
South	12
Center	10

(c)

.	Qty
Region	197

Figure 5.14: *rollUp process: these figures depict two rollUp iterations:* ***(a)*** *initial dataset;* ***(b)*** *first rollUp over Qty;* ***(c)*** *second rollUp.*

```
$rollColsHier : ColRoll \times Hier \times Cid \rightarrow ColRoll$
$rollColsHier\,(sc, h, cid) \triangleq$
  if $(sc = \{\mapsto\} \vee h = \{\mapsto\})$
  then $sc$
  else $\{ci \mapsto \text{mk-}\,(sc\,(ci).\#1, sc\,(ci).\#2, [\,]) \mid ci \in \text{dom}\,(sc) \setminus \{cid\}\} \uplus_m$
      $cid \quad \mapsto \quad \text{mk-}\,(sc\,(cid).\#1, parent\,(h, sc\,(cid).\#2), [sc\,(cid).\#2] \curvearrowright$
$sc\,(cid).\#3) \rightharpoonup$
```

Description:

rollColsHier(cols, h, c) *allows for the preservation of previous hierarchy value*

Calls:

parent (page 181)

In this case, the following expression

$$rollColsHier(t2troll(th)(\text{"Sales"}).\#2(\text{"r1"}).\#2, h, \text{"Qty"})$$

returns

```
{
     "Qty" |-> mk_({},nil,[100]),
     "City" |-> mk_({},"Porto",[]),
     "Type" |-> mk_({},12,[])
}
```

```
$LLGraph = Value \xrightarrow{m} Value$
$Hier2LLGraph : Hier \rightarrow LLGraph$
$Hier2LLGraph\,(t) \triangleq$
  merge $\{$let $tt = t\,(x)$ in
    $\{y \mapsto x \mid y \in \text{dom}\ tt\} \uplus_m Hier2LLGraph\,(tt) \mid$
      $x \in \text{dom}\ t\};$
```

5.3.2.9 Function $parent$

Specification:

```
$parent : Hier \times Cid \rightarrow Value$
$parent\,(h, v) \triangleq$
  if $(h = \{\mapsto\} \vee \neg\, v \in \mathrm{dom}\,(Hier2LLGraph\,(h)))$
  then nil
  else let $x = Hier2LLGraph\,(h)$ in
      $x\,(v)$;
```

Description:

parent(h,cid) *returns the previous value of a hierarchy of values.*

Calls:

Standard VDM-SL only

In our example, expression

$$parent(h, \texttt{"Porto"})$$

will return “North”.

5.3.2.10 Function $children$

Specification:

```
$children : Hier \rightarrow Value\text{-set}$
$children\,(h) \triangleq$
  $\bigcup(\{\mathrm{dom}\,(x) \mid x \in \mathrm{rng}\,(h)\})$;
```

Description:

children(h) *returns all direct children of hierarchy h.*

Calls:

Standard VDM-SL only

Example: expression

$$children(h)$$

returns

```
{ "South","North","Center" }
```

5.3.2.11 Function $family$

Specification:

$$\begin{array}{l} family : Hier \rightarrow Value\text{-set} \\ family\,(h) \triangleq \\ \quad \text{dom}\,(h) \cup \bigcup(\{family\,(hi) \mid hi \in \text{rng}\,(h)\}); \end{array}$$

Description:

***family(h)** return all members of hierarchy h.*

Calls:

Standard VDM-SL only

Again in our running example, expression

$$family(h)$$

returns

```
{"South","Braga","North","Porto","Middle","Lisboa","Region","Coimbra"}
```

5.3.2.12 Function $childrenOf$

Specification:

$$\begin{array}{l} childrenOf : Hier \times Value \rightarrow Value\text{-set} \\ childrenOf\,(h, v) \triangleq \\ \quad \text{if } v \in \text{dom}\,(h) \\ \quad \text{then dom}\,(h\,(v)) \\ \quad \text{else } \bigcup(\{childrenOf\,(hi, v) \mid hi \in \text{rng}\,(h)\}); \end{array}$$

Description:

***childrenOf(hier,value)** returns all children of a particular hierarchy element*

Calls:

Standard VDM-SL only

The expression

$$childrenOf(h, \text{"North"})$$

returns

```
{"Braga","Porto" }
```

It can be also important to be able to get details of a particular result. Thus the following $drill\text{-}down$ operator.

5.3.2.13 Function $drillDown$

Specification:

```
$drillDown : TRoll \times Cid \to TRoll$
$drillDown\,(t, cid) \triangleq$
  mk- $(t.\#1,$
        $\{ri \mapsto$ mk- $(t.\#2\,(ri).\#1,$
                $drillColsHier\,(t.\#2\,(ri).\#2, cid))\ |$
            $ri \in$ dom $(t.\#2)\});$
```

Description:

drillDown(t,c) *performs a drill-down over column specified*

Calls:

drillColsHier (page 184)

A $drill\text{-}down$ operation like,

$$drillDown(rollUp(t2troll(th), h, \text{"City"}), \text{"City"})$$

will restore the initial table.

Rolling-up and drill-down are seen as OLAP complementary operations, so they must respect the property

$$drillDown(rollUp(t, h, v), v) = t$$

5.3.2.14 Function $drillColsHier$

Specification:

```
$drillColsHier : ColRoll \times Cid \to ColRoll$
$drillColsHier\,(sc, cid) \triangleq$
  if $sc = \{\mapsto\}$
  then $\{\mapsto\}$
  else $\{ci \mapsto$ mk- $(sc\,(ci).\#1, sc\,(ci).\#2, [sc\,(ci).\#2])\ |\ ci \in$ dom $(sc) \setminus \{cid\}\} \dagger$
        $cid \mapsto$ mk- $(sc\,(cid).\#1,$ hd $(sc\,(cid).\#3),$ tl $(sc\,(cid).\#3)) \rightharpoonup;$
```

Description:

drillColsHier(cols,column) *gets details for a particular column value*

Calls:

Standard VDM-SL only

This operation does the reverse of $rollColsHier$ (on page 180). It restores the original hierarchy value.

5.3.2.15 Function $consolidate$

Specification:

```
$consolidate[@A, @B] : T \times Cid \times (@A \to @B) \times @B \to @B$
$consolidate\,(t, cid, f, n) \triangleq$
  if $t =$ mk- $(\{\}, \{\mapsto\})$
  then $n$
  else let $cols = \{t.\#2\,(ri).\#2\,(cid).\#2 \mid ri \in$ dom $(t.\#2)\}$ in
      $setApply[@B, \mathbb{Z}]\,(f, cols, 0)$;
```

Description:

consolidate(t, cid, f, nullValue) *applies a total function (e.g.* sum*) to set* s.

Calls:

$setApply$ (page 193)

This method uses *high order VDM function* feature, which allows polymorphic properties, ie, support several types as arguments. By applying $consolidate$ to our example, we can calculate the average of all Qty values using expression

$$consolidate[int, int](t, \text{"Qty"}, avg, 0)$$

yielding 62.3. To work with binary operations (maximum -*max* or minimum -*min*), we create the $summarize$ method.

5.3.2.16 Function $summarize$

Specification:

```
$summarize[@A, @B] : T \times String \times (@A \times @A \rightarrow @B) \times @B \rightarrow @B$
$summarize\,(t, cid, f, n) \triangleq$
    if $t =$ mk- $(\{\}, \{\mapsto\})$
    then $0$
    else let $colvalues = \{t.\#2\,(ri).\#2\,(cid).\#2 \mid ri \in$ dom $(t.\#2)\}$ in
        $setApplyElems[@B]\,(f, colvalues, n)$;
```

Description:

summarize(t, cid, f, nullValue) *applies a binary function (e.g.* max*) to all set* s *elements.*

Calls:

$setApplyElems$ (page 193)

Suppose that we need to "get the largest quantity sold". Interpreting this as evaluating the maximum Qty in our table, with the following expression

$$summarize[int, int](t, \texttt{"Qty"}, max, 0)$$

we yield the intended sales quantity.

The data volume in a repository is often very high. Because people need to decide quickly and efficiently, it will be more efficient to work only with a part of such data. It is the context of $Partitioning$ databases, where a single large table is split into smaller ones, thereby improving response time for queries and other processes [Nig01]. The initial table information structure should result from the union of all those parts.

This operation is supported by function $partition$ and it is implemented using $project$ operator (page 165).

As mentioned earlier on, many other operators could be explored. We can enumerate some of the most usual [Nig01]:

- Pivoting rows and columns
- Ranging a data subset
- Suppressing missing values (like $null$, $zero$, etc)
- Ranking and comparing
- Sorting and filtering

Because of the extension of such a specification, we did not work them out.

5.3.2.17 Aggregating functions

These functions are created just to be used in combination with *consolidate* or *summarize* functions, described before. We are talking about traditional *aggregate operations* on relational database algebra, such as *sum, avg, min, max,* etc.

5.3.2.18 Function sum

Specification:

```
$sum : \mathbb{Z}\text{-set} \to \mathbb{Z}$
$sum\,(s) \triangleq$
    $setApplyElems[\mathbb{Z}]\,(\lambda\, x : \mathbb{Z}, y : \mathbb{Z} \cdot x + y, s, 0)$
pre  card $(s) > 0$
    ;
```

Description:

sum(set) *is a binary function which calculates the sum of a set of numbers*

Calls:

Standard VDM-SL only

5.3.2.19 Function avg

Specification:

```
$avg : \mathbb{Z}\text{-set} \to \mathbb{Z}$
$avg\,(s) \triangleq$
    let $tot = setApplyElems[\mathbb{Z}]\,(\lambda\, x : \mathbb{Z}, y : \mathbb{Z} \cdot x + y, s, 0)$ in
    $tot/$card $(s)$
pre  card $(s) > 0$
    ;
```

Description:

avg(set) *is a binary function which calculates the average of a non empty set of numbers*

Calls:

Standard VDM-SL only

5.3.2.20 Function max

Specification:

```
$max : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$
$max\ (x, y) \ \underline{\triangle}$
    if $x > y$
    then $x$
    else $y$;
```

Description:

max(int,int) *determines the maximum of two values*

Calls:

Standard VDM-SL only

5.3.2.21 Function min

Specification:

```
$min : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$
$min\ (x, y) \ \underline{\triangle}$
    if $x < y$
    then $x$
    else $y$;
```

Description:

min(int,int) *determines the minimum of two values*

Calls:

Standard VDM-SL only

5.3.3 Auxiliary functions

5.3.3.1 Function $toStr$

Specification:

```
$toStr : Value \to String$
$toStr\ (v) \ \underline{\triangle}$
    $[v]$;
```

Description:

Convert a Value to String.

Calls:

Standard VDM-SL only

5.3.3.2 Function $markRow$

Specification:

$$
\begin{array}{l}
markRow : Rows \times Rid \times String \rightarrow Rows \\
markRow\,(r, rid, s) \triangleq \\
\quad r \dagger rid \mapsto \textsf{mk-}\,(r\,(rid).\#1 \cup \{\textsf{mk-}\,(s, \texttt{"}0\texttt{"})\}, r\,(rid).\#2) \rightharpoonup \\
\textsf{pre}\;\; rid \in \textsf{dom}\,(r) \\
\quad ;
\end{array}
$$

Description:

markRow(Rows, rowId, mark) *marks row* rid *on a table. It can be used for hiding rows, for example.*

Calls:

Standard VDM-SL only

5.3.3.3 Function $unmarkRows$

Specification:

$$
\begin{array}{l}
unmarkRows : Rows \rightarrow Rows \\
unmarkRows\,(r) \triangleq \\
\quad \{x \mapsto \textsf{mk-}\,(r\,(x).\#1 \setminus \{\textsf{mk-}\,(\texttt{"}h\texttt{"}, \texttt{"}0\texttt{"})\}, r\,(x).\#2) \mid x \in \textsf{dom}\,(r)\};
\end{array}
$$

Description:

unmarkRows(r) *unmarks all table rows. It can be used to show/hide rows, for example. It works like an auxiliary function.*

Calls:

Standard VDM-SL only

Having this, the next expression must be also valid:

$$unmarkRows(markRow(getRows(t), \texttt{"Color"}, \texttt{"h"})) = t$$

5.3.3.4 Function $t2troll$

Specification:

```
$t2troll : T \to TRoll$
$t2troll\,(t) \triangleq$
   if $t = $ mk- $(\{\}, \{\mapsto\})$
   then mk- $(\{\}, \{\mapsto\})$
   else mk- $(t.\#1, \{ri \mapsto$ mk- $(t.\#2\,(ri).\#1,$
                          $\{ci \mapsto$ mk-
                               (
                                $t.\#2\,(ri).\#2\,(ci).\#1,$
                                $t.\#2\,(ri).\#2\,(ci).\#2, []) \mid$
                            $ci \in$ dom $(t.\#2\,(ri).\#2)\}) \mid$
                      $ri \in$ dom $(t.\#2)\});$
```

Description:

t2troll(table) *converts between table formats*

Calls:

Standard VDM-SL only

5.3.3.5 Function $applyMon$

Specification:

```
$applyMon[@A] : ColRelated \times (@A \to @A) \times Value \to ColRelatedSimple$
$applyMon\,(m, f, n) \triangleq$
   if $m = \{\mapsto\}$
   then $\{\mapsto\}$
   else $\{x \mapsto \{ci \mapsto setApply[@A, \mathbb{Z}]\,(f, m\,(x)\,(ci), n) \mid$
              $ci \in$ dom $(m\,(x))\} \mid$
            $x \in$ dom $m\};$
```

Description:

applyMon(Cols, f, null_value) *applies the function f to all $Cols$ elements*

Calls:

$setApply$ (page 193)

5.3.3.6 Function $\mathit{ff2set}$

Specification:

```
$ff2set[@A, @B, @C] : @A \xrightarrow{m} (@B \times @C) \to (@A \times @C)$-set
$ff2set\,(m) \triangleq$
   if $m = \{\mapsto\}$
   then $\{\}$
   else $\{$mk- $(x, m\,(x).\#2) \mid x \in$ dom $(m)\};$
```

Description:

ff2set(f) *converts to set a mapping function.*

Calls:

Standard VDM-SL only

5.3.3.7 Function $getOne$

Specification:

```
$getOne[@A] : (@A)$-set $\to @A$
$getOne\,(s) \triangleq$
   let $x \in s$ in
   $x$
pre  $s \neq \{\}$
   ;
```

Description:

getOne(s) *gets one of the element set.*

Calls:

Standard VDM-SL only

5.3.3.8 Function $mda2rows$

Specification:

```
    $mda2rows : ColRelatedSimple \times Value \to Rows$
    $mda2rows\,(m, v) \triangleq$
      if $m = \{\mapsto\}$
      then $\{\mapsto\}$
      else let $x \in$ dom $(m)$ in
          let $f \in$ dom $(m\,(x))$ in
          $v \mapsto$ mk- $(\{\}, \{y.\#1 \mapsto$ mk- $(\{\}, y.\#2) \mid y \in x\} \uplus_m f \mapsto$
mk- $(\{\}, m\,(x)\,(f)) \rightharpoonup) \rightharpoonup \uplus_m$
          $mda2rows\,(\{x\} \lhd m, v + 1);$
```

Description:

mda2rows(cols) *converts a* mda *result to* $Rows$.

Calls:

Standard VDM-SL only

5.3.3.9 Function $mda2table$

Specification:

```
    $mda2table : Rows \to T$
    $mda2table\,(r) \triangleq$
      mk- $(\{\}, r);$
```

Description:

mda2table(rows) *converts a* mda *result to* $Table$.

Calls:

Standard VDM-SL only

5.3.3.10 Function $mda2html$

Specification:

```
$mda2html : ColRelatedSimple \times String \to \mathbb{B}$
$mda2html\,(m, f) \triangleq$
  if $m = \{\mapsto\}$
  then true
  else $outHtml\,(mda2table\,(mda2rows\,(m, 0)),$
                $f, 0);$
```

Description:

mda2html(cols) *writes to* $HTML$ *format a* mda *result.*

Calls:

mda2rows (page 191), $mda2table$ (page 192), $outHtml$ (page 196)

5.3.3.11 Function $setApplyElems$

Specification:

```
$setApplyElems[@A] : (@A \times @A \to @A) \times @A\text{-set} \times @A \to @A$
$setApplyElems\,(f, s, n) \triangleq$
  if $s = \{\}$
  then $n$
  else let $x \in s$ in
      $f\,(x, setApplyElems[@A]\,(f, s \setminus \{x\}, n));$
```

Description:

setApplyElems(function, set, nullValue) *applies the binary function* f *to each element of Set* s*. It works as an auxiliary function.*

Calls:

Standard VDM-SL only

5.3.3.12 Function $setApply$

Specification:

```
$setApply[@A, @B] : (@A\text{-set} \to @B) \times @A\text{-set} \times @B \to @B$
$setApply\,(f, s, n) \triangleq$
  if $s = \{\}$
  then $n$
  else $f\,(s);$
```

Description:

***setApply(function, set, nullValue)** applies function f to set s.*

Calls:

Standard VDM-SL only

5.3.3.13 Function $map2set$

Specification:

```
$map2set[@A, @B] : @A \xrightarrow{m} @B \to (@A \times @B)\text{-set}$
$map2set\,(m) \triangleq$
    if $m = \{\mapsto\}$
    then $\{\}$
    else let $i \in \text{dom}\,(m)$ in
        $\{\text{mk-}\,(i, m\,(i))\} \cup map2set[@A, @B]\,(\{i\} \lhd m)$;
```

Description:

***map2set(map)** converts a mapping function to sets.*

Calls:

Standard VDM-SL only

5.3.3.14 Function $map2map$

Specification:

```
$map2map[@A, @B, @C] : (@A \xrightarrow{m} (@B \times @C)) \times (@A \xrightarrow{m} (@B \times @C))$
$\to @A \xrightarrow{m} (@B \times @C)$
$map2map\,(m1, m2) \triangleq$
    if $m1 = \{\mapsto\}$
    then $m2$
    else let $x \in \text{dom}\,(m1)$ in
        if $(m1\,(x) \in \text{rng}\,(m2))$
        then $map2map[@A, @B, @C]\,(\{x\} \lhd m1, m2)$
        else $map2map[@A, @B, @C]\,(\{x\} \lhd m1, m2 \uplus x \mapsto m1\,(x) \rightharpoonup)$;
```

Description:

map2map(map1,map2) *removes duplicate elements in mapping* $map1$.

Calls:

Standard VDM-SL only

5.3.3.15 Function $set2seq$

Specification:

```
$set2seq : Value\text{-set} \rightarrow Value^*$
$set2seq\,(s) \triangleq$
  if $s = \{\}$
  then $[]$
  else let $x \in s$ in
      $[x] \curvearrowright set2seq\,(s \setminus \{x\});$
```

Description:

set2seq(set) *converts a Set of values to String.*

Calls:

Standard VDM-SL only

5.3.3.16 Function $visibleRows$

Specification:

```
$visibleRows : Rows \rightarrow Nat$
$visibleRows\,(r) \triangleq$
  card $\{x \mid x \in \text{dom}\,(r) \cdot \neg\,\text{mk-}\,("h", "0") \in (r\,(x).\#1)\};$
```

Description:

visibleRows(rows) *is an auxiliary function to count only unmarked rows.*

Calls:

Standard VDM-SL only

5.3.4 Pretty print functions

The functions in this subsection allow for outputting the result of all previous methods. It is possible to have *UIML*, HTML or Java output.

5.3.4.1 Operation $outHtml$

Specification:

```
$outHtml : T \times String \times Value \rightarrow [\mathbb{B}]$
$outHtml\,(t, f, v) \triangleq$
  $IO`fecho\,(f, "\backslash n", \text{START}) \wedge$
  let $x \in$ dom $(t.\#2)$ in
  ((if $v = 1$
    then $IO`fecho\,(f, " < TRbgcolor = 'gray' > \backslash n < TD > . < /TD > ", \text{APPEND})$
    else $IO`fecho\,(f, " < TRbgcolor = 'gray' > ", \text{APPEND})) \wedge$
  $\forall\, ci \in$ dom $(t.\#2\,(x).\#2) \cdot$
      (if (mk- $("h", "0") \notin (t.\#2\,(x).\#2\,(ci).\#1))$
       then $(IO`fecho\,(f, " < TD > " \curvearrowright ci \curvearrowright " < /TD > \backslash n", \text{APPEND}))$
       else true)) $\wedge$
  $IO`fecho\,(f, " < /TR > \backslash n", \text{APPEND}) \wedge$
  $\forall\, ri \in$ dom $(t.\#2) \cdot$
     (if (mk- $("h", "0") \notin (t.\#2\,(ri).\#1))$
      then ((if $v = 1$
             then $IO`fecho\,(f, " < TR >< TDbgcolor = 'gray' > " \curvearrowright ri \curvearrowright " < /TD > \backslash n", \text{APPEND})$
             else $IO`fecho\,(f, " < TR > \backslash n", \text{APPEND})) \wedge$
           $\forall\, ci \in$ dom $(t.\#2\,(ri).\#2) \cdot$
              (if (mk- $("h", "0") \notin (t.\#2\,(ri).\#2\,(ci).\#1))$
               then $(IO`fecho\,(f, " < TD > ", \text{APPEND}) \wedge$
                    $IO`fecho\,(f, t.\#2\,(ri).\#2\,(ci).\#2, \text{APPEND}) \wedge$
                    $IO`fecho\,(f, " < /TD > \backslash n", \text{APPEND}))$
               else true))
      else true) $\wedge$
     $IO`fecho\,(f, " < /TR > \backslash n", \text{APPEND});$
```

Description:

outHtml(t, fileName) *exports table data to HTML.*

Calls:

Standard VDM-SL only

Note that all previous HTML figures of our example which depict method application results, were generated with *outHtml*.

5.3.4.2 Operation $outHtmlValue$

Specification:

$$\begin{array}{l} outHtmlValue : Value \times String \rightarrow [\mathbb{B}] \\ outHtmlValue\,(v, f) \triangleq \\ \quad IO`fecho\,(f, v, \textsc{start}); \end{array}$$

Description:

outHtmlValue(Value, fileName) *writes single result to file.*

Calls:

Standard VDM-SL only

5.3.4.3 Operation $outUiml$

Specification:

```
    $outUiml : T \times String \to [\mathbb{B}]$
    $outUiml\,(t, f) \triangleq$
      $IO`fecho\,(f, "<?xmlversion = '1.0'? > \backslash n", \textsc{start}) \wedge$
      $fWrite\,(f, "<!DOCTYPEuimlPUBLIC'\text{-}//UIT//DTDUIML" \curvearrowright$
              $"2.0Draft//EN'\backslash n") \wedge$
      $fWrite\,(f, "'UIML2_0g.dtd' > \backslash n") \wedge$
      $fWrite\,(f, " < uiml > \backslash n") \wedge$
      $fWrite\,(f, " < interface > \backslash n") \wedge$
      $fWrite\,(f, " < structure > \backslash n") \wedge$
      $fWrite\,(f, " < partid = 'top'class = 'Html' > \backslash n") \wedge$
      $fWrite\,(f, " < partid = 'body'class = 'Body' > \backslash n") \wedge$
      $fWrite\,(f, " < partclass = 'Table' > \backslash n") \wedge$
      $fWrite\,(f, " < style > \backslash n") \wedge$
      $fWrite\,(f, " < /style > \backslash n") \wedge$
      let $x \in$ dom $(t.\#2)$ in
      $(fWrite\,(f, " < partid = 'Theader'class = 'Tr' > \backslash n") \wedge$
       $\forall\, ci \in$ dom $(t.\#2\,(x).\#2) \cdot$
            $(fWrite\,(f, " < partclass = 'Th' > \backslash n") \wedge$
             $fWrite\,(f, " < style > \backslash n") \wedge$
             $fWrite\,(f, " < propertyname = 'content' > " \curvearrowright ci \curvearrowright " <$
 $/property > \backslash n") \wedge$
             $fWrite\,(f, " < /style > \backslash n") \wedge$
             $fWrite\,(f, " < /part > \backslash n"))) \wedge$
      $fWrite\,(f, " < /part > \backslash n") \wedge$
      $\forall\, ri \in$ dom $(t.\#2) \cdot$
          $(fWrite\,(f, " < partid = '" \curvearrowright ri \curvearrowright "'class = 'Tr' > \backslash n") \wedge$
           $\forall\, ci \in$ dom $(t.\#2\,(ri).\#2) \cdot$
                (if (mk-$("h", "0") \notin (t.\#2\,(ri).\#2\,(ci).\#1)$)
                 then $(fWrite\,(f, " < partclass = 'Td' > \backslash n") \wedge$
                       $fWrite\,(f, " < style > \backslash n") \wedge$
                       $fWrite\,(f, " < propertyname = 'content' > ") \wedge$
                       $fWrite\,(f, t.\#2\,(ri).\#2\,(ci).\#2) \wedge$
                       $fWrite\,(f, " < /property > \backslash n") \wedge$
                       $fWrite\,(f, " < /style > \backslash n") \wedge$
                       $fWrite\,(f, " < /part > \backslash n"))$
                 else true$)) \wedge$
          $fWrite\,(f, " < /part > \backslash n") \wedge$
          $fWrite\,(f, " < /part > \backslash n") \wedge$
          $fWrite\,(f, " < /part > \backslash n") \wedge$
          $fWrite\,(f, " < /part > \backslash n") \wedge$
          $fWrite\,(f, " < /structure > \backslash n") \wedge$
          $fWrite\,(f, " < /interface > \backslash n") \wedge$
          $fWrite\,(f, " < peers > \backslash n") \wedge$
          $fWrite\,(f, " < presentationhow = 'replace'source =$
 $'HTML_3.2_Harmonia_1.0.uiml\#vocab'" \curvearrowright$
                  $"base = 'HTML_3.2_Harmonia_1.0'/ > \backslash n") \wedge$
          $fWrite\,(f, " < /peers > \backslash n") \wedge$
          $fWrite\,(f, " < /uiml > ");$
```

Description:

outUiml(Table, fileName) *exports the table data to* UIML.

Calls:

Standard VDM-SL only

5.3.4.4 Operation $outUimlJ$

Specification:

```
$outUimlJ : T \times String \times Value \to [\mathbb{B}]$
$outUimlJ\,(t, f, a) \triangleq$
  $IO`fecho\,(f, \texttt{"} <?xmlversion = \text{'}1.0\text{'}? > \backslash n\texttt{"}, \textsc{start}) \wedge$
  $fWrite\,(f, \texttt{"} <!DOCTYPEuimlPUBLIC\text{'}\text{-}//UIT//DTDUIML\texttt{"} \curvearrowright$
          $\texttt{"}2.0Draft//EN\text{'}\backslash n\texttt{"}) \wedge$
  $fWrite\,(f, \texttt{"}\text{'}UIML2_0g.dtd\text{'} > \backslash n\texttt{"}) \wedge$
  $fWrite\,(f, \texttt{"} < uiml > \backslash n\texttt{"}) \wedge$
  $fWrite\,(f, \texttt{"} < interface > \backslash n\texttt{"}) \wedge$
  $fWrite\,(f, \texttt{"} < structure > \backslash n\texttt{"}) \wedge$
  (if $a = 1$
   then $fWrite\,(f, \texttt{"}\backslash t < partid = \text{'}f\text{'}class = \text{'}JApplet\text{'} > \backslash n\texttt{"})$
   else $fWrite\,(f, \texttt{"}\backslash t < partid = \text{'}f\text{'}class = \text{'}JFrame\text{'} > \backslash n\texttt{"})) \wedge$
  $fWrite\,(f, \texttt{"}\backslash t\backslash t < partid = \text{'}s1\text{'}class = \text{'}JScrollPane\text{'} > \backslash n\texttt{"}) \wedge$
  $fWrite\,(f, \texttt{"}\backslash t\backslash t\backslash t < partclass = \text{'}JTable\text{'} > \backslash n\texttt{"}) \wedge$
  $fWrite\,(f, \texttt{"}\backslash t\backslash t\backslash t < /part > \backslash n\texttt{"}) \wedge$
  $fWrite\,(f, \texttt{"}\backslash t < /part > \backslash n\texttt{"}) \wedge$
  $fWrite\,(f, \texttt{"} < style > \backslash n\texttt{"}) \wedge$
  $fWrite\,(f, \texttt{"}\backslash t < propertypart\text{-}name = \text{'}f\text{'}name = \text{'}size\text{'} > 200, 150 <$
$/property > \backslash n\texttt{"}) \wedge$
  $fWrite\,(f, \texttt{"}\backslash t < propertypart\text{-}name = \text{'}t1\text{'}name =$
$\text{'}preferredScrollableViewportSize\text{'} > 300, 150 < /property > \backslash n\texttt{"}) \wedge$
  $fWrite\,(f, \texttt{"}\backslash t < propertypart\text{-}name = \text{'}t1\text{'}name = \text{'}columnNames\text{'} >$
$\backslash n\texttt{"}) \wedge$
  $fWrite\,(f, \texttt{"}\backslash t < constantmodel = \text{'}list\text{'} > \backslash n\texttt{"}) \wedge$
  let $x \in$ dom $(t.\#2)$ in
  $(\forall\, ci \in$ dom $(t.\#2\,(x).\#2) \cdot$
      (if (mk- $(\texttt{"}h\texttt{"}, \texttt{"}0\texttt{"}) \notin (t.\#2\,(x).\#2\,(ci).\#1))$
       then $(fWrite\,(f, \texttt{"}\backslash t\backslash t < constantvalue = \text{'}\texttt{"} \curvearrowright ci \curvearrowright \texttt{"}\text{'}/ > \backslash n\texttt{"}))$
       else true$)) \wedge$
  $fWrite\,(f, \texttt{"}\backslash t < /constant > \backslash n\texttt{"}) \wedge$
  $fWrite\,(f, \texttt{"}\backslash t < /property > \backslash n\backslash n\texttt{"}) \wedge$
  $fWrite\,(f, \texttt{"}\backslash t < propertypart\text{-}name = \text{'}t1\text{'}name = \text{'}content\text{'} > \backslash n\texttt{"}) \wedge$
  $fWrite\,(f, \texttt{"}\backslash t < constantmodel = \text{'}table.rowMajor\text{'} > \backslash n\texttt{"}) \wedge$
  $\forall\, ri \in$ dom $(t.\#2) \cdot$
      (if (mk- $(\texttt{"}h\texttt{"}, \texttt{"}0\texttt{"}) \notin (t.\#2\,(ri).\#1))$
       then $(fWrite\,(f, \texttt{"}\backslash t < constant > \backslash n\texttt{"}) \wedge$
            $(\forall\, ci \in$ dom $(t.\#2\,(ri).\#2) \cdot$
                (if (mk- $(\texttt{"}h\texttt{"}, \texttt{"}0\texttt{"}) \notin (t.\#2\,(ri).\#2\,(ci).\#1))$
                 then $(fWrite\,(f, \texttt{"}\backslash t < constantvalue = \text{'}\texttt{"}) \wedge$
                      $fWrite\,(f, t.\#2\,(ri).\#2\,(ci).\#2) \wedge$
                      $fWrite\,(f, \texttt{"}\text{'}/ > \backslash n\texttt{"}))$
                 else true$)) \wedge$
            $fWrite\,(f, \texttt{"}\backslash t < /constant > \backslash n\texttt{"}))$
       else true $\wedge\, fWrite\,(f, \texttt{"}\backslash t < /constant > \backslash n\texttt{"})) \wedge$
      $fWrite\,(f, \texttt{"}\backslash t < /constant > \backslash n\texttt{"}) \wedge$
      $fWrite\,(f, \texttt{"} < /property > \backslash n\texttt{"}) \wedge$
      $fWrite\,(f, \texttt{"} < /style > \backslash n\texttt{"}) \wedge$
      $fWrite\,(f, \texttt{"} < /interface > \backslash n\texttt{"}) \wedge fWrite\,(f, \texttt{"} < peers > \backslash n\texttt{"}) \wedge$
      $fWrite\,(f, \texttt{"} < presentationbase = \text{'}Java_1.3_Harmonia_1.0\text{'}/ >$
$\backslash n\texttt{"}) \wedge$
      $fWrite\,(f, \texttt{"} < /peers > \backslash n\texttt{"}) \wedge$
      $fWrite\,(f, \texttt{"} < /uiml > \texttt{"})$
```

Description:

outUimlJ(Table, fileName) *exports table T to Java* UIML *vocabulary.*

Calls:

Standard VDM-SL only

5.3.4.5 Operation $fWrite$

Specification:

$$\begin{array}{l} fWrite : String \times String \xrightarrow{o} [\mathbb{B}] \\ fWrite\,(fn, ui) \triangleq \\ \quad IO`fecho(fn, ui, \textsc{append}) \end{array}$$

Description:

fwrite(s,fn) *writes the string* s *in file* fn.

Calls:

$IO`fecho$

5.3.5 AST Conversion

The functions which follow allow for the type conversion between *UIMLSpecTab* and *UIMLSpec*, as well as relative expressions.

To support this conversion, we need the following sub-conversion functions:

$$UIMLSpecTab`Table \rightarrow UIMLSpec`Uiml$$

$$UIMLSpecTab`Row \rightarrow UIMLSpec`Part$$

$$UIMLSpecTab`Col \rightarrow UIMLSpec`Part$$

$$UIMLSpecTab`Style \rightarrow UIMLSpec`Property$$

Function *TU2U* allows some animation to the process, preparing the *UIML* for HTML output.

5.3.5.1 Function $TU2U$

Specification:

```
$TU2U : T \to UIMLSpec`Uiml$
$TU2U\,(t) \triangleq$
  if $t =$ mk- $(\{\}, \{\mapsto\})$
  then mk-$UIMLSpec`Uiml$ (nil , $[]$)
  else mk-$UIMLSpec`Uiml$ (nil ,
                              $[toInter\,([$
                                    $toStru\,([$
                                          $toPart$
                                              $($
                                               $toSty\,([toPro\,("1", "border")]), [$
                                               $toPart\,(toSty\,(TS2P\,(t.\#1)), TR2P\,(t.\#2), "$
                                               $"Table")])])]);$
```

Description:

Exports a UIMLSpecTab *Table element to* UIMLSpec *uiml element*

Calls:

$toInter$, $toStru$, $toPart$, $toPro$, $toPart$, $TS2P$, $TR2P$

5.3.5.2 Function $TS2P$

Specification:

```
$TS2P : Style \to UIMLSpec`Property^*$
$TS2P\,(p) \triangleq$
  if $p = \{\}$
  then $[]$
  else let $x \in p$ in
       $[toPro\,(x.\#2, x.\#1)] \curvearrowright$
       $TS2P\,(p \setminus \{x\});$
```

Description:

Auxiliary function to convert UIMLSpecTab‘Style into UIMLSpec‘Property elements.

Calls:

Standard VDM-SL only

5.3.5.3 Function $TR2P$

Specification:

```
$TR2P : Rows \rightarrow UIMLSpec`Part^*$
$TR2P\,(r) \triangleq$
  if $r = \{\mapsto\}$
  then $[]$
  else let $x \in$ dom $(r)$ in
       $[toPart\,($nil , $[$
                 $toPart\,(toSty\,(TS2P\,(r\,(x).\#1)),$
                          $TC2P\,(r\,(x).\#2),$ "$Td$"$)],$
                "$Tr$"$)] \curvearrowright$
       $TR2P\,(\{x\} \triangleleft r);$
```

Description:

Auxiliary function to convert UIMLSpecTab‘Rows into sequences of UIML-Spec‘Part elements.

Calls:

$TS2P$, $TC2P$, $toPart$,$toSty$

5.3.5.4 Function $TC2P$

Specification:

```
$TC2P : Cols \rightarrow UIMLSpec`Part^*$
$TC2P\,(c) \triangleq$
  if $c = \{\mapsto\}$
  then $[]$
  else let $x \in$ dom $(c)$ in
       $[toPart\,(toSty\,(TS2P\,(c\,(x).\#1)), [],$ "$Td$"$)] \curvearrowright$
       $TC2P\,(\{x\} \triangleleft c);$
```

Description:

Auxiliary function to convert UIMLSpecTab‘Cols into sequences of UIML-Spec‘Part elements.

Calls:

$toPart$, $toSty$, $TS2P$

5.3.5.5 Function $toPro$

Specification:

$$\begin{array}{l} toPro : String \times String \rightarrow UIMLSpec`Property \\ toPro\,(v, c) \ \triangleq \\ \quad \textsf{mk-}UIMLSpec`Property\,([v], c, \texttt{""}, \textsf{nil}\,, \textsf{nil}\,, \texttt{""}, \texttt{""}, \texttt{""}, \texttt{""}); \end{array}$$

Description:

Auxiliary function to construct an UIMLSpec‘Property element.

Calls:

Standard VDM-SL only

5.3.5.6 Function $toSty$

Specification:

$$\begin{array}{l} toSty : UIMLSpec`Property^* \rightarrow UIMLSpec`Style \\ toSty\,(p) \ \triangleq \\ \quad \textsf{mk-}UIMLSpec`Style\,(p, \textsf{nil}\,, \texttt{""}, \textsf{nil}\,, \textsf{nil}\,); \end{array}$$

Description:

Auxiliary function to construct an UIMLSpec‘Style element.

Calls:

Standard VDM-SL only

5.3.5.7 Function $toStru$

Specification:

$$\begin{array}{l} toStru : UIMLSpec`Part^* \rightarrow UIMLSpec`Structure \\ toStru\,(sp) \ \triangleq \\ \quad \textsf{mk-}UIMLSpec`Structure\,(sp, \textsf{nil}\,, \texttt{""}, \textsf{nil}\,, \textsf{nil}\,); \end{array}$$

Description:

Auxiliary function to construct an UIMLSpec‘Structure element.

Calls:

Standard VDM-SL only

5.3.5.8 Function $toPart$

Specification:

$toPart : [UIMLSpec`Style] \times UIMLSpec`Part^* \times String \to UIMLSpec`Part$
$toPart\ (sty, sp, cl) \triangleq$
$\quad \mathsf{mk}\text{-}UIMLSpec`Part\ (sty, \mathsf{nil}, \mathsf{nil}, sp, [\,], \mathsf{nil}, "", \mathsf{nil}, \mathsf{nil}, cl, \mathsf{nil}, "");$

Description:

Auxiliary function to construct an UIMLSpec‘Part element.

Calls:

Standard VDM-SL only

5.3.5.9 Function $toInter$

Specification:

$toInter : UIMLSpec`Structure^* \to UIMLSpec`Interface$
$toInter\ (s) \triangleq$
$\quad \mathsf{mk}\text{-}UIMLSpec`Interface\ (s, \mathsf{nil}, "", \mathsf{nil}, \mathsf{nil})$
end *UIMLSpecTab*

Description:

Auxiliary function to construct an UIMLSpec‘Interface element.

Calls:

Standard VDM-SL only

Considering this, to get an *UIMLSpec* specification of our *t* example table, we use expression

$$UIMLSpecTab`TU2U(UIMLSpecTab`t)$$

For it to be rendered with *Harmonia* rendering engine, we evaluate expression

$$VDM2UIML`toFileHTML(UIMLSpecTab`TU2U(UIMLSpecTab`t), \text{"t.uiml"})$$

where *toFileHTML* is a method defined on *VDM2UIML* "tool", presented in Appendix E.2.

5.3.6 UIML visualization

This section describes our work on *UIML* visualization which was performed in two main phases: first, generate *UIML* from *VDM-SL*; second, generate HTML from *UIML*. These are phases *a)* and *b)* respectively in Figure 5.15.

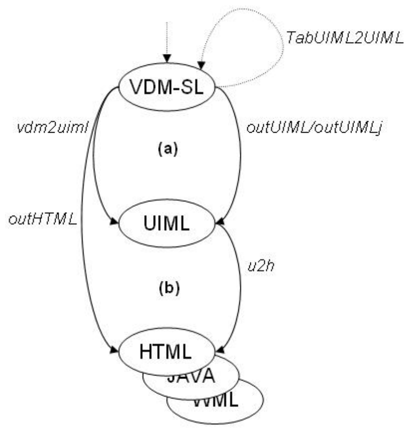


Figure 5.15: UIML *Visualization process*

One of the main announced features of *UIML* [Pha00] is its possibility to be rendered to a different platform under a very simple process. Considering this, we are going to make all referred transformations over tables using also native *UIML* rendering mechanisms.

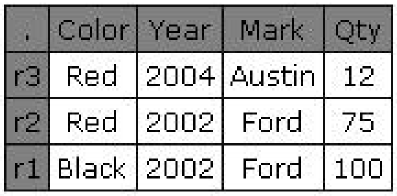
As we can see in our *table* specification, there is an auxiliary operator which exports to *HTML* the resulted specification calculus: this is the *outHtml* (page 196) operator.

Let us consider again our information table t of page 160:

Listing 5.1: *VDM-SL Sales table*

```
t = mk_({},
    {
    "r1"  |->mk_({},{
                    "Mark"|->mk_({},"Ford"),
                    "Color"|->mk_({},"Black"),
                    "Qty"|->mk_({},"100"),
                    "Year"|->mk_({},"2002")
                    }),
    "r2" |->mk_({},{
                    "Mark"|->mk_({},"Ford"),
                    "Color"|->mk_({},"Red"),
                    "Qty"|->mk_({},"75"),
                    "Year"|->mk_({},"2002")
                    }),
    "r3" |->mk_({},{
                    "Mark"|->mk_({},"Austin"),
                    "Color"|->mk_({},"Red"),
                    "Qty"|->mk_({},"12"),
                    "Year"|->mk_({},"2004")
                    })
    }
    );
```

and its HTML representation in Figure 5.16 after applied the HTML pretty print *outHtml*:



.	Color	Year	Mark	Qty
r3	Red	2004	Austin	12
r2	Red	2002	Ford	75
r1	Black	2002	Ford	100

Figure 5.16: *Result of applying* outHtml *operator*

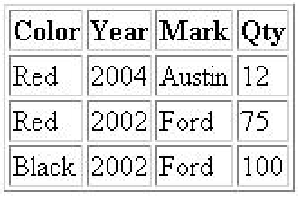
The same can be done with the *UIML* pretty print $outUiml$ (page 197), which generates *UIML* as depicted on Listing 5.2.

Listing 5.2: *UIML table code generated by* outUiml

```
<?xml version='1.0'?>
<!DOCTYPE uiml PUBLIC '-//UIT//DTD UIML 2.0 Draft//EN'
'UIML2_0g.dtd'>
<uiml>
<interface id='Sales'>
<structure>
<part id='top' class='Html'>
<part id='body' class='Body'>
<part id='Sales' class='Table'>
        <style>
        </style>
        <part id='Theader' class='Tr'>
                <part class='Th'>
                        <style>
                        <property name='content'>Color</property>
                        </style>
                </part>
                <part class='Th'>
                        <style>
                        <property name='content'>Mark</property>
                        </style>
                </part>
                <part class='Th'>
                        <style>
                        <property name='content'>Qty</property>
                        </style>
                </part>
        </part>
        <part id='r3' class='Tr'>
                <part class='Td'>
                        <style>
                        <property name='content'>Red</property>
                        </style>
                </part>
                <part class='Td'>
                        <style>
                        <property name='content'>Austin</property>
                        </style>
                </part>
                <part class='Td'>
                        <style>
                        <property name='content'>12</property>
                        </style>
                </part>
        </part>
        <part id='r2' class='Tr'>
                <part class='Td'>
                        <style>
                        <property name='content'>White</property>
                        </style>
                </part>
                <part class='Td'>
                        <style>
                        <property name='content'>Ford</property>
                        </style>
                </part>
                <part class='Td'>
                        <style>
                        <property name='content'>75</property>
                        </style>
                </part>
        </part>
        <part id='r1' class='Tr'>
                <part class='Td'>
                        <style>
                        <property name='content'>Black</property>
                        </style>
                </part>
                <part class='Td'>
                        <style>
                        <property name='content'>Ford</property>
                        </style>
                </part>
                <part class='Td'>
                        <style>
                        <property name='content'>100</property>
                        </style>
                </part>
        </part>
</part>
</part>
</part>
</structure>
</interface>
<peers>
<presentation how='replace' source='HTML_3.2_Harmonia_1.0.uiml#vocab'
        base='HTML_3.2_Harmonia_1.0'/>
</peers>
</uiml>
```

The outcome of rendering this *UIML* fragment with *Harmonia* rendering engine *u2h* [Har98] is the *HTML* table depicted in Figure 5.17.

This sequence of processes shows one of the possibilities to render HTML from



Color	Year	Mark	Qty
Red	2004	Austin	12
Red	2002	Ford	75
Black	2002	Ford	100

Figure 5.17: *Result of applying* u2h *render to* $table.uiml$

UIML of Figure 5.15, here illustrated for our table model. It is important to render *UIML* using our main *UIML* specification. This is the main topic of the chapter which follows.

5.4 Summary

This Chapter constitutes an important contribute of the outcome of this work. It describes a practical case of graphical objects formalization, using a table as the case study. It describes how *VDM-SL* can be used to formally specify the state and methods of this kind of objects.

Having considered the basic table operations (such as row/column manipulation) and the basic *OLAP* operators (such as Filter or roll/drill), it was possible to experiment the application of such objects on multidimensional analysis processes, responding to different criteria of data visualization.

All the results were mapped to *UIML* and then rendered to HTML, focusing their immediate visualization via normal web browser. The great capabilities of *VDM-SL* (as is the case of High-Order functions) on one hand, and the capability of reuse or combine objects, on the other, were determinant for the reached outcomes. It was possible to see how a new method can be defined using existent ones. The rigor of used specification, ensured this.

Although this case of study can not be enough to support a general conclusion, we are sure that it subscribes the possibility and importance to work towards a Visual Component Library.

Next chapter will slightly present the main developed tools as well as the created animation prototype.

Chapter 6

Prototype and Supporting Tools

Recall from previous chapters that we have two main *VDM-SL* specifications, one, *UIMLSpec*, specifies the original *UIML* language (Chapter 4) and the other corresponding to the abstract model of a particular visual object, a table graphical object model (Chapter 5).

In this chapter we describe how to animate such specifications using the VDM tools. We will develop a prototype to animate the *OLAP* features and test the resultant *VDM-SL* specifications.

6.1 Prototype

In order to experiment the animation of VDM methods, mainly *OLAP* functions, we have decided to create a HTML prototype whereby we can visualize all table transformations. This application uses our VDM specifications, *UIMLSpec* and *UIMLSpecTab*. From CGI HTML forms behavior, the *VDM-SL* methods are called, then *UIML* is generated and rendered again to HTML. Figure 6.1 depicts the architecture of our prototype and Figure 6.2 depicts its front-end.

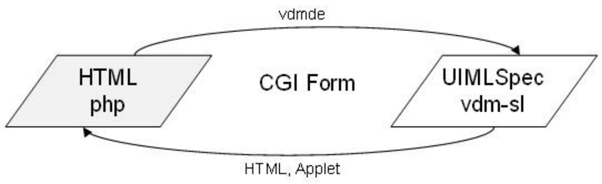
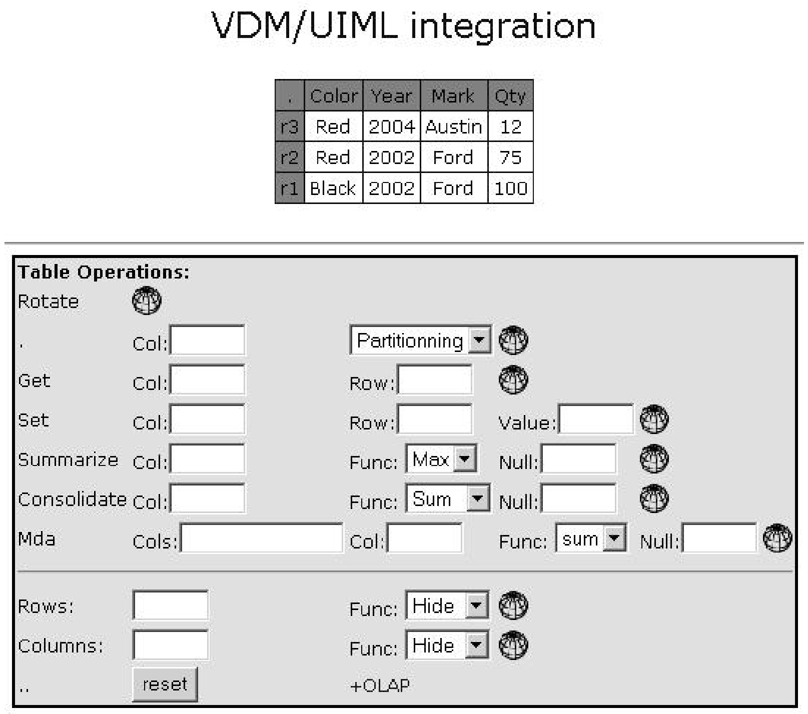


Figure 6.1: *Prototype architecture*

As we can see from Figure 6.2, the methods prototyped are:

- Rotate
- Partitioning and Projection
- Get and Set column
- Summarize



VDM/UIML integration

.	Color	Year	Mark	Qty
r3	Red	2004	Austin	12
r2	Red	2002	Ford	75
r1	Black	2002	Ford	100

Table Operations:

Rotate							
.	Col:	Partitionning					
Get	Col:	Row:					
Set	Col:	Row:	Value:				
Summarize	Col:	Func: Max	Null:				
Consolidate	Col:	Func: Sum	Null:				
Mda	Cols:	Col:	Func: sum	Null:			
Rows:		Func: Hide					
Columns:		Func: Hide					
..	reset	+OLAP					

Figure 6.2: *VDM/UIML integration prototype*

- Consolidate
- Multidimensional analysis
- Hide, Show, Add, Delete columns and rows
- Roll-Up and Drill-Down

The results can be seen in Java Applets too. Appendix F presents this prototype with more detail.

6.2 Supporting tools

Recall the schema of Figure 1.6, in Chapter 1. Each of the four depicted phases *transcoding*, *abstraction*, *validation* and *rendering*, is supported by auxiliary tools which facilitate its application. In the sequel we shall explore their main features.

6.2.1 Phase 1 - Transcoding *UIML* to VDM-SL

The main goal of this phase (seen as the first one) is to obtain a formal representation (in VDM-SL) of *UIML* source code. In formal terms, to obtain an *AST*[1] in *VDM-SL*. So, *transcoding* (or transforming) an *UIML* document to a new *VDM-SL* one, demands

[1]AST - Abstract Syntax Tree

a kind of parsing grammar which directs the code generation for each *UIML* (XML) tag. Since *UIML* is a XML markup language, it is possible to create a processing style sheet (with XSLT technology) to support this transformation.

As previously described in sections 2.4.3 (page 29) and 4.3.2 (page 78), the *VDM-SL* formal specification must model each *UIML* element.

Considering again the <uiml> DTD element (section 4.3.3.1, page 79)

$$<!ELEMENT\ \mathit{UIML}\ (head?, (peers \mid interface \mid template)^{*})\ >$$

and respective *VDM-SL* model (page 79),

$$UIML :: head\ :\ [Head]\ member : Members^{*}$$

since the *Member* is type defined in *VDM-SL* as:

$$Member = Peers \mid Interface \mid Template$$

the *XSL* template which allows it transcoding process is defined as follows:

Listing 6.1: *XSL template to "transcode" <uiml> element*

```
<xsl:template match="uiml">VDM2UIML`uiml2str(mk_UIMLSpec`Uiml(
    <xsl:if test="count(head)=0">nil,</xsl:if>
    <xsl:if test="count(head) &gt; 0">
        <xsl:apply-templates select="head"/>,
    </xsl:if>
    <xsl:if test="count(interface | peers | template) = 0">[],</xsl:if>
    <xsl:if test="count(interface | peers | template) &gt; 0">
    [<xsl:for-each select="interface | peers | template">
            <xsl:apply-templates select="."/>
            <xsl:if test="position()!=last()">,</xsl:if>
        </xsl:for-each>]
    </xsl:if>))
</xsl:template>
```

To help in understanding this process, Listing 6.2 shows the resulting *VDM-SL* code of our previous *UIML* Hello example (page 73).

Listing 6.2: *Excerpt of* UIML *Hello example*

```

uiml = mk_UIMLSpec`Uiml(nil, mk_UIMLSpec`Interface({
     mk_UIMLSpec`Structure({ mk_UIMLSpec`Part( {
   mk_UIMLSpec`Part( {nill},mk_UIMLSpec`Name("hello"),
      nill,
          <REPLACE>,mk_UIMLSpec`Class("helloC"))
},mk_UIMLSpec`Name("TopHello"),
      nill,
          <REPLACE>, nill)
},nill,

      ....
      )
```

Appendix E.1 depicts the source code of this style sheet (*uiml2vdm.xsl*) responsible for all this transcoding process. The structure of this type of documents is clear and easily understood. Readers unfamiliar with this technology should consult some XSL technical reference, for instance [Rec01a, Rec99b].

6.2.2 Pretty Print

Developed directly in VDM-SL, this tool - *vdm2uiml.vdm* - works as a script[2] and executes a parsing process over *VDM-SL* specifications. It generates a correspondent *UIML* syntax for each specified element. Listing 6.3 is an extract of this module, and shows the *uiml2vdm*, a VDM function which generates the <uiml> element.

Listing 6.3: *Extract of* vdm2uiml VDM-SL *script*

```
uiml2str : UIMLSpec`Uiml -> UIMLSpec`String
uiml2str(ui) == PI ^
                "<uiml>"^
                head2str(ui.head)^
                members2str(ui.members)^
                end_doc;
}
```

Recalling our initial *Hello* example, in this case using *hello.vdm* which has resulted from transcoding *UIML* to VDM, we can see that the application of *vdm2uiml* shall generates the *UIML* document of Listing 6.4:

Listing 6.4: *UIML generated from vdm2uiml VDM "script"*

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<uiml>
        <interface>
                <structure>
                        <part class=' ' where='last' >
                                <part class='helloC' where='last' ></part>
                        </part>
                </structure>
                <style></style>
        </interface>
        <peers>
                <presentation base=''> </presentation>
        </peers>
</uiml>
```

Appendix E.2 contains the complete source code of this VDM pretty print module. To better understand how this module really works, the main syntax characteristics of *VDM-SL* must be present (described in section 2.4.3).

6.2.3 Phase 2 - Verifier

The verifier is a tool which was developed to support the *validation* phase (depicted on Figure 1.6, on Chapter 1) of our process.

This validation process aims at generating *UIML* from *VDM-SL* code, corresponding to the transcoding reverse process. It allows us to test the consistency of the generated VDM-SL code along the transcoding process, verifying the "similarity" between both *UIML* code.

Because of the extension of this process, it was simplified appealing to some *VDM-SL* methods, mainly *TabUIML2UIML* method (*UIMLSpecTab VDM-SL* specification), which converts between specifications, and *toFile* (VDM2UIML *VDM-SL* module). Next section 6.2.5 will focus also on this topic.

[2]VDM2UIML *VDM-SL* module

6.2.4 Phase 3 - Abstraction

This phase of our process has not been completely developed. All reasoning implicit in it is sketched in Chapter 7. The process starts from the table *VDM-SL* abstraction towards the creation of a visual components library.

This work should show that it is possible to perform transformations by calculation. Then it is necessary to find different candidate implementation objects. This is a natural property of adaptable interfaces.

In our case study, the idea stays as "it is possible to get a new *UIML* interface description" which represents the same. Chapter 7 will address this idea and its main guidelines.

6.2.5 Phase 4 - Rendering *UIML*

Markup languages are by their nature declarative, which means that a runtime mechanism (interpreter, script or other) must be present, which decides what and how to render them for a particular supporting platform [BHW02].

As a markup language, *UIML* uses the <style> element to specify the binding of the markup description to the semantics, to control the rendering process. Recently, some scripting features like assignment and comparison operations can be directly supported in the markup language. This happens already with XSLT without compromising its portability.

Considering this, to get an application UI, is necessary to render the *UIML* to the desired platform. We use the actual *UIML* rendering engines, from Harmonia[3], which supports several platforms (Desktop PC, handheld PC, Cell phone, PDA and others) with several languages (JAVA, HTML, WML, VoiceXml, etc.), rendering widgets and events specified in *UIML* <presentation> elements.

If we choose $HTML$ as the destination platform and use our operators to generate $HTML$ directly (ex. *outHtml* method), for instance, we risk to deface the whole process. Considering the existing $UIML$ renders [AA01] which allow HTML, Java, WML, etc., code generation, we can also generate $UIML$ from our specification and then experiment to apply these renders too.

Let us consider a table example t. If we use the following expression:

$$outUiml(t, \texttt{"}\text{table.uiml}\texttt{"})$$

we will get the file $table.uiml$ with $UIML$ code for our table (as referred in section 5.3.6).

In another way, table t can be converted to our *UIML VDM-SL* specification which, using our *vdm2uiml* module, can be converted to *UIML* and rendered to the target platform.

As mentioned already, in order to convert our table specification (*UIMLSpecTab*) to our base specification (*UIMLSpec*), we created function *TabUIML2UIML*. Some other functions are available to convert also Rows (and Columns) and Style types, to Parts

[3]http://www.harmonia.com

and Property *UIML* elements respectively, as referred to in Section 5.3.5. Considering this, the expression

$$UIMLSpecTab`TabUIML2UIML(UIMLSpecTab`t)$$

will result in an *UIML* element corresponding to our table t.

Once in the *UIML* code, the following shell commands will respond for the rest of the process:

1. $u2h\ table.uiml\ table.html$

   Generates $HTML$ code for browser platforms. The render $u2h$ works over $HTML$ vocabulary.

2. $u2ji\ table.uiml\ table.java$

   Generates $Java$ code associated to $UIML$ descriptions, ready to be used on Java platforms. The $u2ji$ render works over $Java$ vocabulary.

3. $u2w\ table.uiml\ table.wml$

   Generates WML code for WAP devices. The render $u2w$ works over WML vocabulary.

For those who dislike using shell commands, Harmonia has a JAVA utility - *LiquidUI* UIML *Browser* (Figure 6.3), which allows for rendering to several platforms.

Figure 6.3: *Harmonia LiquidUI* UIML *browser*

Our example can only be observed on browser platforms, because (for the sake of simplicity) we only worked with $HTML$ vocabulary. A result of applying $u2h$ (shell command 1) to our case study can be observed in Figure 5.17 (on page 209).

There is also the possibility to generate Harmonia "General vocabulary" [Pha00], which allows the conversion to several platforms independently of base vocabularies. Because of some of its limitations we avoided using it.

Chapter 7

Conclusions and Future Work

7.1 Overview

It is a fact of all times that humans react and decide upon data perceived by their eyes. Thus the “must see to believe” legend...

Although not such a religious matter for software designers, this entails a need to worry about the way users see and use the application software they create, which has grown enormously in complexity in recent years: as technology grows up, so do user’s requirements. Human Computer Interaction (HCI) has thus become an important branch of Computer Science.

This dissertation focus on the design of graphical user interfaces which support state-of-the-art software applications. For the sake of rigor, it addresses the application of formal methods to recent UI markup language support technologies. The application of formal modelling techniques to user interface development imposes scientific rigor along the development process as a whole: specification, validation and transformation.

To better achieve its research aims, this work has been organized in the following steps:

- Review of the state of the art;
- Identification of recent UI models, available specifications, designs and programming tools;
- Analysis of available UI markup languages;
- Study of the application of formal methods to specify and develop UI layers;

Two main questions have triggered the research:

1. How does one take advantage of user interface graphical objects and associated features?
2. How does one adapt available user interfacing tools to new platforms? By developing “from scratch” or by reusing from a Visual Component Library?

To answer these questions, it was found important to identify the most recent advances in user interface properties, model analysis and specification, designs, programming and evaluation.

Web-technologies were found to be central to this research area. As a matter of fact, the advent of the web information processing model and newly associated business paradigms has led to increased interest in the separation of UI from its supporting platform and the multi-level organization of programming methods, with specialized working teams lead by a system architect.

In this context, one can also appreciate the strength of markup language applications in UI development. XML, although recent, is spreading and becoming by and large accepted as a standard way to ensure data portability among systems. So, why not use it in UI development?

Considering the three main initiatives in this context — UIML, XIML and XUL — at time of writing, UIML was the technology selected to address in detail in this work due to some important shortcomings of the other two.

UIML (the acronym of *User Interface Markup Language*) has been proposed by *Harmonia Inc.* as a XML-language to describe user interface elements and respective behavior. Under the "one application, multiple interface" principle, it looked perfect to support our goals. It is an easy-to-use markup language (thus inheriting all XML features) supported by a well structured Document Type Definition (DTD). Moreover, it is an OASIS standardization proposal and it allows for UI definition and implementation regardless the intended final device.

However, as it happens with other technologies, the loose (informal) semantics of the UIML markup language can lead it to a hard option for UI programmers. UI is a large and complex technology area, once it deals with humans and human needs. Although a well structured and almost natural textual language, its extensibility with <part> and <peers> elements, its dynamic behavior with <structure> elements, its multimodal support with <interface> elements, its capacity to integrate custom vocabularies with <presentation> elements and reusable interface components with <template> elements, UIML has become a complex and large syntax to assimilate. Large UIML source code modules are required to implement even simple UI designs.

Semantically, UIML authors have decided to define several conflict resolution policies (such as those concerning <property> elements) and to assume several behavior rules. Most of these are left to final rendering mechanisms or even external applications. This is where rigor of the UIML design process is compromised.

Our awareness of this problem is the main motivation for the idea of using formal methods: the need of rigor in the specification and validation processes.

For this purpose we resorted to VDM-SL, an ISO standard and general purpose formal specification language, which is among the best supported at tool level, to specify UIML. Chapter 4 describes this specification work.

Thanks to *VDM-SL* datatype invariant support, we could complement the specification with some invariants, corresponding to main UIML semantic properties, such as *id uniqueness* (XML does not control this restriction sufficiently well) and the fact that attribute *part-name* must refer to an existing *part id* attribute. This was implemented with VDM-SL auxiliary functions. Thanks to the possibility of using the IFAD VDM-

Tools, under the academic licence granted to the University of Minho, it was possible to animate the resultant specifications as well as to perform tests on the validity of the invariants defined.

Because of the comprehensiveness of UIML, the VDM-SL specification results naturally in a quite large and hard to manipulate document[1].

Our main experiments have to do with the specification of a table graphical object — a very common user interface component in almost any kind of software interface. It was important to analyze, on the one hand, its UIML abstract VDM-SL representation and, on the other hand, its behavior in supporting OLAP features. The outcome of tablular, OLAP-like operations such as projection, column/row insertion, rotation, etc. can be exported to HTML, UIML canonical representation and Java Applets.

The whole conversion from VDM-SL to UIML/HTML/JAVA, and vice-versa, is supported by a XML Stylesheet *(uiml2vdm)* and VDM-SL predicates (as are the case of *outHtml*, *outUiml* and *outUimlJ*). The stylesheet *uiml2vdm* does not support all UIML syntactic definitions.

Our case study behaved as a simulation or prototype platform and proved the applicability of VDM to animate prototypes and the role of rigor in the achieved UIML formal specification, even using only a few UIML elements. However simple, this case study was enough to provide evidence of the potential and substantial complexity of the UIML specification.

7.2 Discussion and Future Research

Our research on user interface development has revealed an interesting and very complex area. Despite the existence of several research projects focused on this, there are still questions which remain unanswered.

One of these questions is as basic as follows: are markup languages, even developed with up-to-date technology, the best way to support this process? Further than a portable and platform independent description technique, XML subscribes to an enormous vocabulary set, which demands parser and validation mechanisms. As a "swirl", this can be characterized as a "déjà vu", but now with different actors?

Our UIML VDM-SL specification leads us to believe that there is room for some conceptual simplification. This conclusion is sustained by observing the achieved abstract specification, where several replicated elements and repeated patterns can be identified.

Considering this, an interesting path for future work is that of UIML's *refactoring* process towards a simpler specification.

Another interesting topic for future developments is that of creating a formal *Visual Component Library* (VCL). Our table specification is but the beginning of one such repository of useful graphical interaction objects (IO objects). Over such a repository one should be able to build new interfaces just by combining/reusing available components. Once made available, a VCL would contribute to answering another relevant question: "What is the best IO object able to represent this particular piece of

[1]Most of this is due to the absence of inheritance mechanisms in VDM-SL.

data?", Ideally, a "matching" process should become available for searching the library and retrieving all adequate components.

As a matter of fact, our research has included some work along these two vectors — UIML abstract syntax refactoring and visualization. However, the outcome cannot be regarded as finished work. Despite their incompleteness, perhaps our results can still be useful to anyone wishing to pursue them. This explains why we have decided to include them as sections 7.2.1 and 7.2.2 which follow.

Readers uninterested in the details which follow should jump straight to our final remarks in section 7.3. Anyway, our motivation for formal methods in this work can be perceived in section 7.2.1 better than anywhere else in this dissertation — this is where we actually reason about the specification and deviate from current specification practice using informal notation, where such reasoning is not mathematically sound.

7.2.1 UIML formal specification refactoring

Our UIML formal specification (in VDM-SL) allows us to go further and apply transformations rules supported by SETS [Oli98, Hal60, Hal01] theory.

Considering all *UIML* elements specified in Chapter 4, it is possible to define a system of mathematical equations which represent them. Towards this target, let us assume the following notation considerations:

1. UIML elements will be abbreviated as presented in Table 7.1;
2. *String* type elements will be denoted by symbol S;
3. The VDM type $X = A \mid B$ will be written in $SETS$ as $X \equiv A + B$.
4. Every VDM record structure will be abstracted by $SETS$ tuples. So record $X :: a : A \ b : B$ will be transformed into tuple $A \times B$.

Considering this, a first mathematical equational system representing all UIML elements is defined as follows:

Abbrev=UIML	Abbrev=UIML
$U = Uiml$	$Log = Logic$
$H = Head$	$Per = Peers$
$Met = Meta$	$Tem = Template$
$S = String$	$Par = Part$
$Int = Interface$	$Pre = Presentation$
$Str = Structure$	$Rul = Rule$
$Pro = Property$	$Scr = Script$
$Ref = Reference$	$Res = Restructure$
$Ca = Call$	$Sty = Style$
$Ev = Event$	$Con = Content$
$Cot = Constant$	$Rep = Repeat$
$Ite = Iterator$	$Cod = Condition$
$Beh = Behavior$	$Act = Action$
$Dcl = D\text{-}class$	$Pa = Param$
$Dcm = D\text{-}component$	$WT = When\text{-}true$
$Dpr = D\text{-}property$	$WF = When\text{-}false$
$Dpa = D\text{-}param$	$BD = By\text{-}default$
$Dmt = D\text{-}method$	$Lis = Listener$

Table 7.1: *Abbreviations for* UIML *element names*

$$U \cong H \times (Int + Per + Tem)^* \tag{7.1}$$
$$H \cong Met^* \tag{7.2}$$
$$Met \cong S \times S \tag{7.3}$$
$$Int \cong (Str + Sty + Con + Beh) \times (ID + 1) \times S_{3attr} \tag{7.4}$$
$$Str \cong Par^* \times (ID + 1) \times S_{3attr} \tag{7.5}$$
$$Par \cong Sty \times Con \times Beh \times Par^* \times Rep^* \times (ID + 1) \times S_{6attr} \tag{7.6}$$
$$Sty \cong Pro^* \times (ID + 1) \times S_{3attr} \tag{7.7}$$
$$Pro \cong (S + Cot + Pro + Ref + Ca + Ite)^* \times S_{8attr} \tag{7.8}$$
$$Con \cong Cot^* \times (ID + 1) \times S_{3attr} \tag{7.9}$$
$$Cot \cong Cot^* \times (ID + 1) \times S_{5attr} \tag{7.10}$$
$$Ref \cong S \times S \tag{7.11}$$
$$Beh \cong Rul^* \times (ID + 1) \times S_{3attr} \tag{7.12}$$
$$Rul \cong Cod \times Act \times (ID + 1) \times S_{3attr} \tag{7.13}$$
$$Cod \cong Eq + Ev + Op \tag{7.14}$$
$$Eq \cong Ev \times (Cot + Pro + Ref) \tag{7.15}$$

$$\begin{aligned}
Ev &\cong S \times S \times S \times S && (7.16)\\
Op &\cong (Cot + Pro + Ref + Ca + Op + Ev)^* \times S && (7.17)\\
Act &\cong (Pro + Ca + Res)^* + (WT \times WF \times BD) && (7.18)\\
Ca &\cong Pa^* \times S && (7.19)\\
Rep &\cong Ite \times Par^* && (7.20)\\
Ite &\cong (S + Cot + Pro + Ca) \times (ID + 1) && (7.21)\\
Res &\cong Tem \times S_{4attr} && (7.22)\\
WT &\cong (Pro + Ca)^* \times Res \times Op \times Eq \times Ev && (7.23)\\
WF &\cong (Pro + Ca)^* \times Res \times Op \times Eq \times Ev && (7.24)\\
BD &\cong (Pro + Ca)^* \times Res \times Op \times Eq \times Ev && (7.25)\\
Pa &\cong (S + Pro + Ref + Ca + Op + Ev + Cot + Ite) \times S && (7.26)\\
Per &\cong (Pre + Log)^* \times (ID + 1) \times S_{3attr} && (7.27)\\
Pre &\cong Dcl^* \times (ID + 1) \times S_{4attr} && (7.28)\\
Log &\cong Dcm^* \times (ID + 1) \times S_{3attr} && (7.29)\\
Dcm &\cong Dmt^* \times ID \times S_{5attr} && (7.30)\\
Dcl &\cong Dmt^* \times Dpr^* \times Ev^* \times Lis^* \times ID \times S_{6attr} && (7.31)\\
Dpr &\cong Dmt^* \times Dpa^* \times ID \times S_{3attr} && (7.32)\\
Dmt &\cong Dpa^* \times Scr \times ID \times S_{5attr} && (7.33)\\
Dpa &\cong S \times (ID + 1) \times S && (7.34)\\
Scr &\cong S \times (ID + 1) \times S_{4attr} && (7.35)\\
Lis &\cong S \times S && (7.36)\\
Tem &\cong (Beh + Dcl + Dcm + Cot + Int + Log + Par + Per + && (7.37)\\
&\quad Pre + Pro + Ru + Scr + Str + Sty + Con + Res) \times S
\end{aligned}$$

From this mathematical system which represents all UIML element dependencies, it is possible to identify some particular occurrences, such as *replicated definitions*, *expression patterns* and *mapping candidates*.

Because text (string) is the type of the majority of attributes, we have decided to simplify some of the equations, by grouping attributes into S_{attr} (S_{3attr},S_{4attr},S_{5attr},S_{6attr}, S_{8attr}). Attribute *ID* is considered in isolation because of its role in the overall referential integrity. Wherever ID is optional, we use $(ID + 1)$.

Let us now recall some important notions concerning such a transformation process.

7.2.1.1 Sets and Sequences

$Sets$[2] in $VDM\text{-}SL$ notation are represented by type constructor *set of*, for instance, *set of A (A-set in pretty printed VDM-SL)* must be read as *set of A elements*. In $SETS$ notation, one writes 2^A. So,

[2]Do not confuse *Sets* with *SETS*: the first one refers to a set of elements, while the second one refers to a theory.

$$set\ of\ A \cong 2^A$$

Sequences in *VDM-SL* notation are represented by type constructor *seq of*, having *seq of A* interpreted as sequence of A elements *(A-seq in pretty printed VDM-SL)*. In *SETS* notation, sequences are represented by *A**. So,

$$seq\ of\ A \cong A^*$$

$Sets$ and $Sequences$ are related by refinement,

$$2^A \leq_{elems} A^* \tag{7.38}$$

demanding the acceptance of the following two conditions:

- $s \in A^*$ is an ordered sequence
- $s \in 2^A$ has no repeated elements

7.2.1.2 Replicated elements

Let S denote any datatype (e.g. string). This cartesian product $S \times S \times \cdots \times S$ can be abbreviated by exponential S^n:

$$S^n \cong \overbrace{S \times S \times S \times \cdots \times S}^{\text{n}} \tag{7.39}$$

where $n \in \mathbb{N}$. Considering this and working with mathematical variables substitution, many expressions should have their *arity* reduced.

7.2.1.3 Mapping transformations

Our analysis of $UIML$ shows that some of the elements result from composition of different type elements. As happens in a database normalization, there exist elements which have some particular element in their composition which can determine the others. This means that this element is unique and can identify a particular instance of that object. This happens in those sequences of $UIML$ elements which have a required $(ID + 1)$.

To understand all these properties, please recall the mapping model transformation properties described in previous section 5.2.2.

7.2.1.4 $SETS$ patterns

In the previous system of equations 7.1 to 7.37, one can identify some expressions which are common to several equations. For example, there are several instances of the following expressions:

- B^n
- $B^* \times C^n$
- $B^* \times C \times D^n$

So, it could be really possible that one equation could be an instance of another one. This can be easily identified in equations (7.23), (7.24) and (7.25). All of them can be considered as modelling the same.

Let us analyze equations (7.9) and (7.10) for instance, having to do with *Content* and *Constant* elements, respectively:

$$\begin{aligned} Con &\cong Cot^* \times (ID + 1) \times S_{3attr} \\ Cot &\cong Cot^* \times (ID + 1) \times S_{5attr} \end{aligned}$$

Since $S_{3attr} \leq S_{5attr}$, i.e, all attributes of *Con* element are also attributes of *Cot* elements, we can denote that one equation can be an instance of the other. In this way both equations ((7.9) and (7.10)) can be substituted by a new one. For instance, a new element called $anyCont$ can be defined as:

$$anyCont \cong Cot^* \times (ID + 1) \times S_{5attr}$$

In practice, this can be viewed as a different $UIML$ element called $anyContent$, with attributes to distinguish between $Constant$ or $Content$. For example, in $UIML$ specification 3.0, <constant> elements are defined as:

```
        <constant id="Cat" value="Cat"/>
        <constant id="Dog" value="Dog"/>
        <constant id="Mouse" value="Mouse"/>
```

and <content> elements as:

```
        <content id="Portugues">
            <constant id="Cat">Gato</constant>
            <constant id="Dog">Cão</constant>
            <constant id="Mouse">Rato</constant>
        </content>
```

Both could now be represented by $anyContent$ elements:

```
    <anyContent type="const" id="Cat" value="Cat"/>
    <anyContent type="const" id="Dog" value="Dog"/>
    <anyContent type="const" id="Mouse" value="Mouse"/>
    <anyContent type="cont" source="Portugues" id="Cat"  value="Gato"/>
    <anyContent type="cont" source="Portugues" id="Dog" value="Cão"/>
    <anyContent type="cont
    ‘‘ source="Portugues" id="Mouse" value="Rato"/>
```

Continuing this reasoning, there could be some more equations which can “disappear”, being a particular instance of others.

7.2.1.5 Applying abstraction

Let us now try to perform some $mapping\ transformations$.

Being in mind what was explained in section 7.2.1.3, let us consider some elements as a study case for this propose.

7.2.1.5.1 <uiml> element :

Let us abstract the <uiml> element and its member elements. Following previous assumptions, we shall structure our reasoning in two steps:

1. Mathematical equations

$$\begin{array}{rcl} \mathit{Uiml} & \cong & \mathit{Head} \times (\mathit{Interface} + \mathit{Peers} + \mathit{Template})^* \\ \mathit{Head} & \cong & \mathit{Meta}^* \\ \mathit{Meta} & \cong & \mathit{String} \times \mathit{String} \end{array}$$

2. Abbreviating names (following Table 7.1)

$$\begin{array}{rcl} U & \cong & H \times (\mathit{Int} + \mathit{Per} + \mathit{Tem})^* \\ H & \cong & \mathit{Met}^* \\ \mathit{Met} & \cong & S \times S \end{array}$$

From the equation system of page 219, we are going now to find some relevant patterns in most equations.

7.2.1.5.2 <interface> element :

As referred in previous equation (7.4, page 220), the $interface$ element is defined by

$$\mathit{Int} \cong (\mathit{Str} + \mathit{Sty} + \mathit{Con} + \mathit{Beh})^* \times (\mathit{ID} + 1) \times S_{3attr} \tag{7.40}$$

In the $VDM\text{-}SL$ specification, Ie (*InterfaceElements*) is a type defined by

$$\mathit{InterfaceElements} = \mathit{Structure} \mid \mathit{Style} \mid \mathit{Content} \mid \mathit{Behavior}$$

That is,

$$\mathit{Ie} \equiv \mathit{Str} + \mathit{Sty} + \mathit{Con} + \mathit{Beh}$$

once names are abbreviated.

On the other hand, we see that Str, Sty, Con and Beh are elements with $(A \times \cdots \times ID \cdots \times B)$ structure. So, Ie, can be rewritten as:

$$\mathit{Ie} \equiv (\mathit{ID} \times \mathit{Str}_{inf} + \mathit{ID} \times \mathit{Sty}_{inf} + \mathit{ID} \times \mathit{Con}_{inf} + \mathit{ID} \times \mathit{Beh}_{inf})$$

where Beh_{inf} abstracts the remainder element information.

By applying properties (5.17) (page155) and (5.20), (5.21) (page 156), and ignoring Int attributes, equation (7.40) is rewritten into

$$\begin{aligned} Int &\cong ID \hookrightarrow Int_{inf} \\ Int_{inf} &\cong (Str_{inf} + Sty_{inf} + Con_{inf} + Beh_{inf}) \end{aligned}$$

These final equations, mean the same as the original VDM specification. $Interface$ can have several *Structure*, *Style*, *Content* or *Behavior* elements, all of them identified by an unique identifier *(ID+1)*.

The same reasoning should be applied to other $UIML$ elements, like $Structure$, $Content$, $Constant$, $Behavior$, $Rule$, $Iterator$, $Peers$, $Logic$, $D\text{-}component$, $D\text{-}class$, $D\text{-}property$, $D\text{-}method$, $D\text{-}param$ and $Script$.

7.2.1.5.3 A practical example :

Listing 7.1 shows an *UIML* fragment which describes a *Java AWT* Frame containing a Java AWT Label:

Listing 7.1: *UIML code to describe a Java AWT label*

```
<structure>

<part id="TopHello" class="Frame">

    <style>
        <property name="rendering">Frame</property>
        <property name="title">Example UI</property>
        <property name="resizable">true</property>
        <property name="layout">java.awt.FlowLayout</property>
        <property name="background">blue</property>
        <property name="foreground">white</property>
        <property name="size">500,100</property>
        <property name="location">100,100</property>
    </style>

    <part name="L" class="Label">
        <style>
            <property name="text">Sample label.</property>
        </style>
    </part>
</part>

<structure>
```

It is easy to find in this code instances of the equations (7.5), (7.6), (7.7) and (7.8).

The diagram of Figure 7.1 presents an informal interpretation of this $UIML$ excerpt.

Note that the first property has attribute *name="rendering"* and value text *"Frame"*. These *name* attribute and value text (string) are members of S^n in equation (7.8).

Considering the involved equations, let us analyze this piece of code in more detail:

$Equation\ (7.5)$:

$$Str \cong Par^* \tag{7.41a}$$

To simplify, the $structure$ element has no attributes;

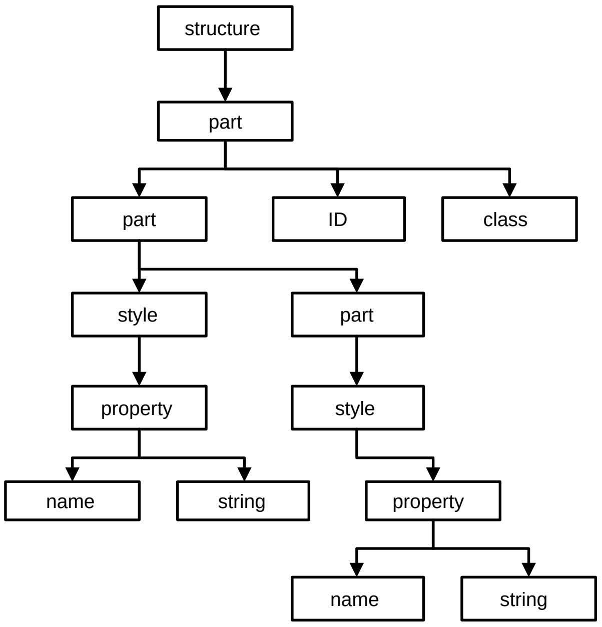


Figure 7.1: *Example of <uiml> hierarchy elements*

$Equation\ (7.6)$:

$$Par \cong Sty \times Par \times S \times S \tag{7.41b}$$

Here $S \times S$ represent $(ID + 1)$ and $class$ attributes;

$Equation\ (7.7)$:

$$Sty \cong Pro^* \tag{7.41c}$$

To simplify, the $style$ element has no attributes;

$Equation\ (7.8)$:

$$Pro \cong S \times S \tag{7.41d}$$

Here, the first S represents the *value* text (string), while the second one represents the $name$ attribute;

Grouping all, we have:

$$Str \cong Par^* \tag{7.42a}$$
$$Par \cong Sty \times Par \times S \times S \tag{7.42b}$$
$$Sty \cong Pro^* \tag{7.42c}$$
$$Pro \cong S \times S \tag{7.42d}$$

and, by replacing *Sty* and *Pro*, we have

$$Str' \cong Par' \tag{7.42e}$$
$$Par' \cong 2^{S \times S} \times Par' \tag{7.42f}$$

where $2^{N \times T}$ could be converted in $N \hookrightarrow T$.

Considering equations (7.42a) to (7.42d) we can anticipate a change in $UIML$ notation, with particular tags names, to describe the same. For instance,

```
<structure>
    <PART id="TopHello" class="Frame" title="Example UI"
    resizable="true" layout="java.awt.FlowLayout"
    background="blue" foreground="white"
    size="500,100" location="100,100">
        <PART id="L" class="Label" text="Sample label."/>
    </PART>
</structure>
```

So we can see the recursiveness of $part$ elements having a new part elements inside of it.

Comparing this code to the original one, it is obvious that it is more concise and perhaps, more intuitive. The translation between this and original $UIML$ can be mechanically implemented.

7.2.2 Tool support for language refactoring

The extension of UIML renders any attempt to do manual transformations as calculated above impracticable.

Resorting to some automated process will be useful. Starting from an VDM-SL AST, applying a set of transformations properties, towards a new, abstractly changed AST, is the focus of *VooDooM* [AS04] automated refinement tool. This shows the applications of Haskel[3] and Strafunski[4] functional strategies. Haskel and its powerful data type system, provides a natural way for defining abstract syntax. On top of it, the Strafunsky library implements functional strategies for generic traversals. From its definition, “generic functions that can traverse into terms of any type” it looks adequate to perform complex AST manipulation. Moreover, some Haskel libraries are available for markup languages, namely HaXML[5].

The reader is referred to [AS04] for results and conclusions, about the application of functional strategies to write specific transformation operators, without being concerned with all AST complexity.

So, perhaps a similar process can be applied to UIML refactoring.

[3]http://www.haskell.org/
[4]http://www.cs.vu.nl/Strafunski/
[5]http://www.cs.york.ac.uk/fp/HaXml/

7.2.3 IO Visualization

Our research has focused on formalizing $Interaction\ Objects$ behavior such as the $table$ object presented in Chapter 5. No less important is the possibility of integration or application of existent distinct UI objects, such as $ListBoxes$, $Frames$, etc. to get our data display more accurate.

This question was deeply explored in several research initiatives [JNZM93, GLS96], and from their results, we can deduce the existence of significant problems on UI generators, mainly on selecting components. The automatic selection of UI objects could not be the best choice.

Considering this part of our reasoning and experience, it is important to show how a composition or a particular use of an UI component library can have different results.

In practice, this is captured by the question "How can we represent graphically these data?" or "How can we select the best UI object, if there is one?".

This section presents some thoughts on an *Abstract visualization* process based on graphical attributes and eventual refinement on expressions intended for visualization. The whole experience starts from the basic elements of our abstraction process, in the context of which our $table$ object was considered. We refer to elements such as Exp in our $VDMType$ definitions (on page 159) which could represent expressions of type $A \times B$, $A \hookrightarrow B$ and constants ($Numbers$ and $Strings$). Here, a table and all its components (rows, columns and cells) will be graphically represented around a hypothetical rectangle.

7.2.3.1 Abstract visualization

Along this representative process we will work with a simple table object, having specific graphical information ignored, exploring the possibility of composing new representation forms after some reasoning over initial expressions. In practice, this means that a certain component could be replaced by another one, which could represent the same.

Let us assume a $rectangle$ as being the main visible shape of our visualization process, and let $visualizing$ be our operator, which can "represents" several different types of expressions. The homomorphic behavior of this operator is described as follows:

$$visualizing(Exp_i\ \theta\ Exp_j) = visualizing(Exp_i)\ \phi\ visualizing(Exp_j)$$

where ϕ represents the visual semantics of θ. The interpretation is that visualizing a composite objects must be equivalent to composing the visualization of each of its parts.

Starting with the basic elements of our model, Figure 7.2, describes the visualization of each possible expression.

Considering this, we will analyze how different terms (expressions) can be visualized. Just as a particular instance of concrete element representations, we can assume that:

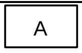
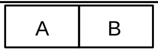
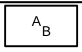
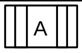
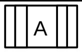
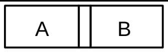
Exp	*visualizing(Exp)*	*GUI*
A		label; textBox
AxB		label; textBox
A+B		radioButton
A*		listBox; comboBox
A+		listBox; comboBox
A->B		table

Figure 7.2: *Abstract and Concrete graphical visualization of basic elements*

- Simple (numerical or textual constant) expressions, can be represented by $label$ objects;
- As A^* in $SETS$ notation is a sequence of A elements, the known $listBox$ object can display it;
- The same can be said about A^+;
- According to this reasoning, $A + B$ can be represented by a $radioButton$ object, evaluating to A or B data values;
- $A \times B$ can be represented by two $label$ objects;
- $A \hookrightarrow B$, as seen before, can be represented by a $table$ object.

What has influenced this kind of mapping from abstract to concrete graphical representations? Let us take A^* as an example.

In a sequence, the order of its elements is relevant. Position 1 must be occupied by element a_1, position 2 element by a_2, and so on, until all elements are positioned. This reminds us of an $array$ (Figure 7.3).

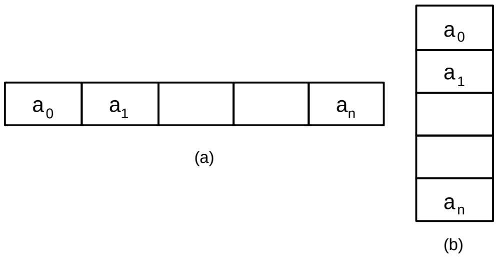


Figure 7.3: A^* *seen as an* $array$

As happens with $array$, it is possible to have operators to manipulate it, such as, giving a value for a particular array position (in programming language notation

$v = array[i]$); getting all values stored in it ($elems(array)$); getting the next free (i); etc.

Looking at Figure 7.3, we can see that (a) and (b) represent the same, being (b) a rotation or a vertical perspective of (a).

Now, how do we represent arbitrarily composite expressions (terms), such as e.g. $A \times B \times C$, $A \times B) \hookrightarrow C$, $A \hookrightarrow (B + C)$, etc.?

Let us analyze some examples.

7.2.3.1.1 Example 1 :

Suppose we want to have a GUI element or elements to represent the following expression:

$$E_1 \quad : \quad (A \times B) \hookrightarrow C$$

We can see that it is an instance of $A \hookrightarrow B$, substituting A by $(A \times B)$. So, according to our abstract graphical representation,

$$visualizing(E_1) \cong visualizing(Exp \hookrightarrow C)$$

giving a possible aspect showed on Figure 7.4. It is only necessary to know how to represent Exp, which is $A \times B$.

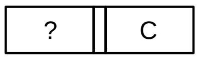


Figure 7.4: $(Exp \hookrightarrow C)$ *representation*

Looking again to our abstract graphical symbols (Figure 7.2), there is a direct representation of $A \times B$. So, our final representation is:

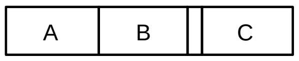


Figure 7.5: $(A \times B) \hookrightarrow C$

7.2.3.1.2 Example 2 :

Suppose we want to visualize values of type expression:

$$E_2 \quad : \quad A \hookrightarrow D \times (B \hookrightarrow C)$$

Once more, this is an instance of $A \hookrightarrow B$. The part which must be explored now is the mapping's range. So,

$$visualizing(E_2) \cong visualizing(A \hookrightarrow Exp)$$

As before, a possible representation is shown in Figure 7.6. Now it is necessary to know how to represent Exp, which is $D \times (B \hookrightarrow C))$.

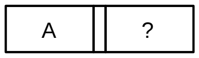


Figure 7.6: $(A \hookrightarrow Exp)$ *representation*

As $D \times (B \hookrightarrow C)$ is an instance of $(A \times B)$, having B substituted by $B \hookrightarrow C$ and instance of $A \hookrightarrow B$, the final representation will be

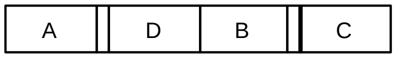


Figure 7.7: $A \hookrightarrow D \times (B \hookrightarrow C)$ *representation*

7.2.3.1.3 Example 3 :

Looking now at expression

$$E_3 \quad : \quad A \hookrightarrow D \times C^B$$

at first sight it looks quite strange, because we have no visualization for C^B. From $SETS$ properties, we know that

$$C^B \leq B \hookrightarrow C$$

In this way, E_3 can be considered as an instance of E_2 (example 2).

$$visualizing(E_3) \cong visualizing(E_2)$$

7.3 Final Remarks

So we reach our final conclusion: although $UIML$ is a recent domain-specific markup language, based on XML description, having associated DTD and $XMLschema$, it comprehends perhaps an excessive set of structures. Large code files will be necessary to specify what is intended.

So, trying to "say the same in fewer words" will be better. The $UIML$ creators work hard towards this. They are working on a more "simple and generic UIML" which is called *UIML Shorthand* [Har98].

Finally we would like to mention the opportunity and usefulness of this work to Harmonia on analyzing this kind of problems, as referred in this email message from Dr. Marc Adams:

> ...The inconsistencies in the draft spec will be very helpful to the OASIS UIML technical committee, so please make a list of them for the TC.
>
> I'll look forward to receiving a draft release — you are doing a very interesting topic.

Bibliography

[AA01] Mir Farooq Ali and Marc Adams. Simplifying Construction on Multi-Platform User Interfaces Using UIML, 2001.

[AAN03] S. Apostolos, T. Antonis, and S. Nick. *Digital Typography Using Latex*. Springer-Verlag, 2003.

[ABB+97] David Atkins, Thomas Ball, Michael Benedikt, Glenn Bruns, Kenneth Cox, Peter Mataga, and Kenneth Rehor. Experience with a Domain Specific Language for Form-based Services. In *ACM*, pages 37–50, 1997. http://citeseer.nj.nec.com/atkins97experience.html.

[AF00] Mehmet Altinel and Michael J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *The VLDB Journal*, pages 53–64, 2000. http://citeseer.nj.nec.com/altinel00efficient.html.

[AH00] Marc Abrams and James Helmes. User Interface Markup Language UIML Specification. UIML Version 2.0, 2000.

[AH02] Marc Abrams and James Helmes. User Interface Markup Language UIML Specification. UIML Version 3.0, 2002.

[And] Derek J. Andrews. An Overview of VDM–SL.

[APQA02] Mir Farooq Ali, Manuel Pérez-Quiñones, and Marc Adams. Building Multi-Platforms User Interfaces with UIML, 2002.

[AS04] Tiago M. L. Alves and Paulo F. A. Silva. Automated refining tool. Technical report, Universidade do Minho, 2004.

[BCMS02] Trevor Bench-Capon, Grant Malcolm, and Michael Shave. Semantics for Interoperability: relating ontologies and schemata, 2002. http://gunther.smeal.psu.edu/17827.html.

[BHW02] Judith Bishop, R Nigel Horspool, and Basil Worrall. Experience with integrating Java with new technologies: C#, XML and web services. *Computer Science Department*, 2002.

[Bja91] Stroustrup Bjarne. What is Object-Oriented Programming?, 1991. http://www.research.att.com/ bs/whatis.pdf.

[Bor03] Borland. Kylix, 2003. http://www.borland.com/kylix/.

[Bow96] Prof. Jonathan Bowen. *Formal Specification and Documentation using Z: A Case Study Approach*. Thomson Publishing, 1996.

[Bra98] Myers Brad. A brief history of Human Computer Interaction Technology, 1998. http://citeseer.nj.nec.com/myers98brief.html.

[BRS$^+$00] Klaus Bergner, Andreas Rausch, Marc Sihling, Alexander Vilbig, and Manfred Broy. A Formal Model for Componentware. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 189–210. Cambridge University Press, New York, NY, 2000. http://citeseer.nj.nec.com/bergner00formal.html.

[BRSV98] Klaus Bergner, Andreas Rausch, Marc Sihling, and Alexander Vilbig. An Integrated View On Componentware - Concepts, Description Techniques, and Development Process. In *IASTED 98, Proceedings of IASTED Conference on Software Engineering*, pages 77–82. IEEE, 1998. http://citeseer.nj.nec.com/bergner98integrated.html.

[BRSV99] Klaus Bergner, Andreas Rausch, Marc Sihling, and Alexander Vilbig. Componentware – Methodology and Process. In *CBSE '99 Proceedings of the International Workshop on Component-Based Software Engineering*. IEEE, 1999. http://citeseer.nj.nec.com/238431.html.

[Bus02] Warehouses For Business. Creating High Quality e-Data - Warehouses for Business, 2002. http://citeseer.ist.psu.edu/534668.html.

[BVE02] Laurent Bouillon, Jean Vanderdonckt, and Jacob Eisenstein. Model-Based Approaches to Reengineering Web Pages, 2002. http://citeseer.nj.nec.com/bouillon02modelbased.html.

[Byn98] Bill Bynum. *Latex Thesis Guide*, 1998.

[Che99] Tao Cheng. XUL - Creating Localizable XML GUI, 1999.

[Con00] XML Consortium. XML NameSpaces, 2000. http://www.w3.org/2000/xmlns/.

[Con02] XUL Consortium. XML-based User Interface Language, 2002. http://www.mozilla.org/projects/xul/.

[Con03a] The TCL/TK Consortium. Tool Command Language - TCL/TK, 2003. http://www.tcl.tk/.

[Con03b] W3C Consortium. World Wide Web Consortium, 2003. http://www.w3c.org.

[Cyp93] Allan Cypher. Watch What I Do: Programming by Demonstration, 1993.

[Dev03] Web Developers. Common Gateway Interface, 2003. http:www.wdvl.com/Authoring/CGI.

[dSP00] Paulo Pinheiro da Silva and Norman W. Paton. User Interface Modelling with UML. In H. Kangassalo, H. Jaakkola, and E. Kawaguchi, editors, *Proc. 10th European-Japanese Conference on Information Modelling and Knowledge Bases, Saariselkä (Finland), 2000*. IOS Press, Amsterdam, 2000. http://citeseer.nj.nec.com/dasilva00user.html.

[dSP03] Paulo Pinheiro da Silva and Norman W. Paton. Improving UML Support for User Interface Design: A Metric Assessment of UMLi, 2003. http://citeseer.nj.nec.com/pinheirodasilva03improving.html.

[eBC01] Intel eB̃usiness Center. N-tier Architecture Improves Scalability, Availability and Ease of Integration. *Infrastructure Best Practices*, 2001. http://citeseer.nj.nec.com/robert83using.html.

[Fer00] Carpani Fernando. Multidimensional Models: A State of Art., 2000. http://www.fing.edu.uy/inco/pedeciba/ bibliote/reptec/TR0012.pdf.

[FF93] Martin R. Frank and James D. Foley. Model-based User Interface Design by Example and by Interview. In *ACM Symposium on User Interface Software and Technology*, pages 129–137, 1993. http://citeseer.nj.nec.com/article/frank93modelbased.html.

[FME03] FME. Formal Methods Europe, 2003. http://www.fmeurope.org/.

[For] XIML Forum. eXtensible Interface Markup Language. http://www.ximl.org.

[For03] Wap Forum. Wireless Markup Language, 2003. http://www.wapforum.org/what/technical.htm.

[Fre03] Jeff Freund. Interface Scalability, 2003. http://www.cmswatch.com/.

[GC93] Gerald C. Gannod and Betty H. C. Cheng. A Two-Phase Approach to Reverse Engineering Using Formal Methods. In *Formal Methods in Programming and Their Applications*, pages 335–348, 1993. http://citeseer.nj.nec.com/gannod93twophase.html.

[GC99a] Gerald C. Gannod and Betty H. C. Cheng. A Formal Approach for Reverse Engineering: A Case Study. In *Working Conference on Reverse Engineering*, pages 100–111, 1999. http://citeseer.nj.nec.com/212304.html.

[GC99b] Gerald C. Gannod and Betty H.C. Cheng. A Formal Approach for Reverse Engineering: A Case Study, 1999.

[GCB$^+$97] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *J. Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[GeA02] Miguel Goulão and Fernando Brito e Abreu. From Objects to Components: a Quantitative Approach, 2002.

[Gee00] Haan Geert. *ETAG - A Formal Model of Competence Knowledge for User Interface Design*. PhD thesis, Vrije Universiteit, 2000.

[Ger01] Andreas Gerstinger. Improvement of Requirements and Test Cases in a Network Node for Air Traffic Control with the Vienna Development Method. Master's thesis, Institute for Software Technology, TU-Graz, Austria, March 2001. Supervisor: Peter Lucas and Bernhard Aichernig.

[GLS96] Marc Gyssens, Laks V. S. Lakshmanan, and Iyer N. Subramanian. Tables as a Paradigm for Querying and Restructuring. In *Symposium on Principles of Database Systems*, pages 93–103, 1996. http://citeseer.nj.nec.com/gyssens96tables.html.

[Gro03a] Apache Group. Embperl, 2003. http://perl.apache.org/embperl/.

[Gro03b] OMG Object Management Group. ARGOi, 2003. http://www.cs.man.ac.uk/img/umli/tutorial/short-tutorial01.html.

[Gro03c] OMG Object Management Group. UML - Unified Modelling Language, 2003. http://www.uml.org.

[Hal60] Paul Halmos. *Naive Set Theory*. Princeton, 1960.

[Hal90] Anthony Hall. Seven Myths of Formal Methods. *IEEE Softw.*, 7(5):11–19, 1990.

[Hal01] Paul Halmos. Wikipedia - The free Encyclopedia, 2001. http://en.wikipedia.org/wiki/Naive_set_theory.

[Har98] Inc. Harmonia. Harmonia inc., 1998. http://www.harmonia.com/.

[Har02] Inc. Harmonia. User Interface Markup Language UIML Specification. Technical report, Harmonia Inc., 2002.

[Hel03] James Helmes. The Relationship of the UIML 3.0 Spec. to other Standards/Working Groups, 2003.

[Her91] Jürgen Herczeg. A Design Environment for Graphical User Interfaces, 1991. http://citeseer.ist.psu.edu/342583.html.

[HM01] Carlos A. Hurtado and Alberto O. Mendelzon. Reasoning about Summarizability in Heterogeneous Multidimensional Schemas. In *Proceedings of the 8th International Conference on Database Theory*, pages 375–389. Springer-Verlag, 2001.

[Hop01] K. Hopper. VDM-SL - a tutorial, 2001.

[IAL03] IBM, Apple, and Lotus. OpenDoc Tehnology, 2003.

[IBM03] IBM. Enterprise PLI - Language Reference, 2003. http://publibfi.boulder.ibm.com/epubs/pdf/ibm3lr20.pdf.

[IFA00a] IFAD. VDMTools - VDM-SL Toolbox User Manual - V3.6. Technical report, IFAD, 2000. http://www.ifad.dk.

[IFA00b] IFAD. VDMTools - The IFAD VDM++ language - V6.6. Technical report, IFAD, 2000. http://www.ifad.dk.

[IFA00c] IFAD. VDMTools - The IFAD VDM-SL language - V3.6. Technical report, IFAD, 2000. http://www.ifad.dk.

[IFA03] IFAD. IFAD Company, 2003. http://www.ifad.com.

[ISO96] ISO. ISO/IEC 13817-1: Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language, 1996.

[JBK89] J. Grollmann J. Burgstaller and F. Kapsner. On the Software Structure of User Interface Management Systems, 1989.

[JFMdM92] Dennis J. M. J., James D. Foley, Kevin E. Mullet, and Charles A.van der Mast. Coupling application design and user interface design. Technical Report DUT-TWI-92-03, Georgia Institute of Tehcnology and Sun Microsystems, Delft, The Netherlands, 1992. http://citeseer.nj.nec.com/baar91coupling.html.

[JNZM93] Jeff A. Johnson, Bonnie A. Nardi, Craig L. Zarmer, and James R. Miller. ACE: building interactive graphical applications. *Communications of the ACM*, 36(4):40–55, 1993.

[Jon90] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1990. http://citeseer.nj.nec.com/jones95systematic.html.

[KP88] G. Krasner and S. Pope. A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 system. *Journal of Object Oriented Programming*, 1(3):26–49, 1988.

[KR02] George Coulouris Kasim Rehman, Frank Stajano. Interfacing with the Invisible Computer, 2002. http://www-lce.eng.cam.ac.uk/ kr241/Paper260702.pdf.

[Lam94] Leslie Lamport. *LaTeX: A Document Preparation System (2nd Edition)*. Addison-Wesley Professional, 1994.

[Lec96] Eric Lecolinet. XXL: A Dual Approach for Building User Interfaces. In *ACM Symposium on User Interface Software and Technology*, pages 99–108, 1996. http://citeseer.nj.nec.com/lecolinet96xxl.html.

[Lec99] Eric Lecolinet. XXL: A Visual+Textual Environment for Building Graphical User Interfaces, 1999. http://citeseer.nj.nec.com/lecolinet99xxl.html.

[LFJ95] Nigay L., Jambon F., and Coutaz J. Formal Specification of Multimodality. *CHI' 95 Workshop*, 1995.

[Lin00] David S. Linthic. *Enterprise Application Integration*. Addison Wesley, 2000.

[Luí03] Ferreira G. Luís. No passado...será XML, 2003.

[Luí04] Ferreira G. Luís. Formalizing Markup Languages for User Interface, 2004. Technical Report - 44 pages.

[Luy01] Kris Luyten. XML en User Interfaces, 2001. http://lumumba.luc.ac.be/kris/courses/ui/.

[Ma98] Yao Ma. Data Warehousing, OLAP, and Data Mining: An Integrated Strategy for Use at FAA, 1998. http://citeseer.nj.nec.com/ma98data.html.

[Mac96] Vijay Machiraju. A Survey on Research in Graphical Interfaces, August 1996.

[Mar95] F. M. Martins. *Métodos Formais na Concepção e Desenvolvimento de Sistemas Interactivos*. University of Minho, 1995. Ph. D. thesis (in Portuguese).

[MB86] Brad A. Myers and William Buxton. Creating Highly-Interative and Graphical User Interfaces by Demonstration, August 1986.

[McE04] Chris McEvoy. Usability Views, 2004. www.usabilityviews.com.

[MCM$^+$91] Brad A. Myers, Allen Cypher, David Maulsby, David C. Smith, and Ben Shneiderman. Demonstrational interfaces: Coming soon? In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 393–396. ACM Press, 1991.

[MHP00] Brad Myers, Scott E. Hudson, and Randy Pausch. Past, Present and Future of User Interface Software Tools, 2000. http://citeseer.nj.nec.com/231861.html.

[Mic96] Sun Microsystems. Javabeans v1.0, 1996. http://java.sun.com/beans.

[Mic03a] McLennan Michael. Incr Widgets, 2003. http://incrtcl.sourceforge.net/iwidgets/.

[Mic03b] Microsoft. Layered Application, 2003. http://msdn.microsoft.com/architecture/patterns/ArcLayeredApplication.

[Mic03c] Microsoft. Microsoft, 2003. http://www.microsoft.com/.

[Mic03d] Sun Microsystems. The Java Language, 2003. http://java.sun.com/.

[Mic03e] Sun Microsystems. The Java Swing API, 2003. http://java.sun.com/products/jfc/tsc/.

[Mic03f] Sun Microsystems. The Javascript Script Language, 2003. http://www.javascript.com.

[Mic04a] Microsoft. Community Resources for Architecture and Design, 2004. http://www.gotdotnet.com/team/architecture.

[Mic04b] Microsoft. Visual Studio .Net, 2004. http://msdn.microsoft.com/vstudio/.

[Mic04c] Sun Microsystems. J2EE - Java 2 Platform Enterprise Edition, 2004. http://java.sun.com/j2ee/.

[Mig97] Encarnação Miguel. Models of Human-Computer Interaction, 1997. www.gris.uni-tuebingen.de/gris/proj/guis/Papers/DISS.

[Mis99] Misosoft. Microsoft Foundations Classes, 1999. http://msdn.microsoft.com/library/.

[MJS$^+$00] Hausi A. Muller, Jens H. Jahnke, Dennis B. Smith, Margaret-Anne D. Storey, Scott R. Tilley, and Kenny Wong. Reverse engineering: a Roadmap. In *ICSE — Future of SE Track*, pages 47–60, 2000. http://citeseer.nj.nec.com/muller00reverse.html.

[MK97] R. Michalski and K. Kaufman. Data Mining and Knowledge Discovery: A Review of Issues and a Multistrategy Approach, 1997. http://citeseer.nj.nec.com/article/michalski97data.html.

[ml98] XML-DEV mailing list. Simple API for XML, 1998. http://www.w3c.org/TR/xslt.

[MO85] F. M. Martins and J. N. Oliveira. Graphics Programming with ‘Archetypes’ — A Preliminary Study. In *Proceedings of the EUROGRAPHICS’85 Conference*, pages 401–412, Nice, France, September 1985.

[Moz03] Mozdev.org. O’ REILLY’S creating applications with mozzila, 2003. http://books.mozdev.org/chapters/index.html.

[MT97] Nenad Medvidovic and Richard N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 60–76. Springer–Verlag, 1997. http://citeseer.nj.nec.com/medvidovic97framework.html.

[Mye95] Brad A. Myers. User Interface Software Tools, March 1995. http://citeseer.nj.nec.com/myers93user.html.

[Mye96] Brad A. Myers. UIMS, Toolkits, Interface Builders, May 1996. http://www.cs.cmu.edu/ bam.

[NB02] Walsh Norman and Stayton Bob. DocBook XSL Stylesheet Documentation, 2002.

[Nig01] Pendse Nigel. Multidimensional data structures, 2001. http://www.olapreport.com/MDStructures.htm.

[Nun01] Nunes D. Nuno. Object Modeling for User-Centered Development and User Interface Design: The Wisdom Approach, 2001. Ph. D. thesis.

[OAS03] OASIS. Organization for the Advancement of Structured Information Standards, 2003. http://www.oasis-open.org/.

[Oli92] J. N. Oliveira. A Reification Calculus for Model-Oriented Software Specification. *Formal Aspects of Computing, Vol.2, 1-23*, 1992.

[Oli98] J. Nuno Oliveira. Métodos Formais de Programação. Departamento de Informática, Universidade do Minho, 1998.

[Oli02] J. Nuno Oliveira. EVDM: a LATEX style extending article.sty+vdmsl-2e.sty. Technical report, Dep. Informática, Universidade do Minho, Portugal, 2002. http://www.di.uminho.pt/ jno.

[Oli03] J. Nuno Oliveira. An introduction to Data Refinement. Formal Methods II, 2003.

[oM03] University of Maryland. Guide to Usability for Software Engineers, 2003. http://www.otal.umd.edu/guse/.

[OMG02] OMG. Object Management Group, 2002. http://www.omg.org/.

[oRE03] WCRE Working Conference on Reverse Engineering. Reengineering Forum, 2003. http://reengineer.org/.

[org92] UIMS org. A metamodel for the runtime architecture of an interactive system: the UIMS tool developers workshop. *SIGCHI Bull.*, 24(1):32–37, 1992.

[Org98] UIM Org. Oasis UIML technical committee, 1998. http://www.oasis-open.org/committees/uiml/.

[Ovi99] Sharon Oviatt. Designing the User Interface for Multimodal Speech and Pen-based Gesture Applications, 1999.

[PA99] Constantinos Phanouriou and Marc Abrams. Using XML to Build User Interfaces, 1999. http://www.uiml.org/.

[Pal98] A.J. Palay. The Andrew Toolkit - An Overview. In *Winter Usenix Technical Conference*, 1998.

[Pan97] Markopoulos Panagiotis. *A compositional model for the formal specification of user interface software*. PhD thesis, Queen Mary and Westfield College - University of London., 1997.

[PE02a] Angel Puerta and Jacob Eisenstein. XIML: A Common Representation for Interaction Data, 2002.

[PE02b] Angel Puerta and Jacob Eisenstein. XIML: A Universal Language for User Interfaces, 2002. http://citeseer.nj.nec.com/587490.html.

[Per03] Perl.org. The Perl Language, 2003. http://www.perl.org/.

[Pfa85] G.E. Pfaff. User Interface Management Systems: Proceedings of the Seeheim Workshop, 1985.

[Pha00] Constantinos Phanouriou. UIML: A Device-Independent User Interface Markup Language, 2000.

[Pin00] Paulo Pinheiro da Silva. User Interface Declarative Models and Development Environments: A Survey. In Ph. Palanque and F. Paternò, editors, *Proceedings of DSV-IS2000*, volume 1946 of *LNCS*, pages 207–226, Limerick, Ireland, June 2000. Springer-Verlag. http://citeseer.nj.nec.com/article/dasilva00user.html.

[Pla03] XUL Planet. XML User Interface Language XUL, 2003. http://www.xulplanet.com.

[Pop01] Paul Pop. Design Principles of Human-Computer Interaction, 2001.

[PP00] Paulo Pinheiro da Silva and Norman W. Paton. UMLi: The Unified Modeling Language for Interactive Applications. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939, pages 117–132. Springer, 2000. http://citeseer.nj.nec.com/article/dasilva00umli.html.

[PSM$^+$03] Michael Palermo, Darshan Singh, Steve Mohr, Pieter Siegers, and Chris Knowles. *Professional ASP.NET 1.0 XML with C#.* Wrox Press, 2003.

[Pue93] Angel R. Puerta. The study of models of intelligent interfaces. In *Intelligent User Interfaces*, pages 71–78, 1993.

[Pyt03] Python.org. The Python Language, 2003. http://www.python.org/.

[Rec98a] W3C Recommendation. Compact HTML for Small Information Appliances, 1998. http://www.w3c.org/TR/1998/NOTE-compactHTML-19980209/.

[Rec98b] W3C Recommendation. DTD - Document Type Definition, 1998. http://www.w3.org/XML/1998/06/xmlspec-report-v21.htm.

[Rec99a] W3C Recommendation. XML Path Language (XPath) 1.0, 1999. http://www.w3.org/TR/xpath.

[Rec99b] W3C Recommendation. XSL Transformations XSLT, 1999. http://www.w3c.org/TR/xslt.

[Rec01a] W3C Recommendation. Extensible Stylesheet Language XSL Version 1.0, 2001. http://www.w3c.org/style/xsl/.

[Rec01b] W3C Recommendation. Standard Generalized Markup Language, 2001. http://www.w3c.org/Markup/sgml/.

[Rec01c] W3C Recommendation. XML Schema, 2001. http://www.w3c.org/XML/Schema.

[Rec02] W3C Recommendation. XML Pointer Language (XPointer), 2002. http://www.w3.org/TR/xptr/.

[Rec03a] W3C Recommendation. Cascading Style Sheets, level 2 CSS2 Specification, 2003. http://www.w3c.org/style/css/.

[Rec03b] W3C Recommendation. Document Object Model, 2003. http://www.w3c.org/DOM/.

[Rec03c] W3C Recommendation. Extensible Markup Language XML Version 1.0, 2003. http://www.w3.org/TR/REC-xml.

[Rec03d] W3C Recommendation. Web services, 2003. http://www.w3c.org/DesingIssues/WebServices.html/.

[Rec03e] W3C Recommendation. XEvents, 2003. http://www.w3.org/TR/xml-events/.

[Rec03f] W3C Recommendation. XForms 1.0, 2003. http://www.w3c.org/Markup/Forms/.

[Rec04a] W3C Recommendation. Hypertext Markup Language HTML, 2004. http://www.w3c.org/TR/html401/.

[Rec04b] W3C Recommendation. Voice eXtensible Markup Languages - Version 2.0, 2004. http://www.w3c.org/TR/2004/REC-voicexml20-20040316/.

[Reh01a] Kasim Rehman. 101 Ubiquitous Computing Applications, 2001. http://www-lce.eng.cam.ac.uk/ kr241/html/101-ubicomp.html.

[Reh01b] Kasim Rehman. A Graphical User Interface for the Real World, 2001.

[RSF97] Sébastien Romitti, Charles Santoni, and Philipe François. A design methodology and a prototyping tool to dedicate to adaptive interface generation. *UI4All*, 1997. http://ui4all.ics.forth.gr/ui4all97/.

[SC03] Nary Subramanian and Lawrence Chung. Adaptable User Interface Generation, 2003.

[Sch01] Schomaker. Software structure of UIMS, 2001. http://hwr.nici.kun.nl/ miami/taxonomy/node83.html.

[SCJS01] Chun S., Chung C., Lee J., and Lee S. Dynamic Update Cube for Range-Sum Queries. *The VLDB Journal*, pages 521–530, 2001. http://citeseer.nj.nec.com/chun01dynamic.html.

[Shn97] Ben Shneiderman. Direct Manipulation for Comprehensible, Predictable and Controllable User Interfaces. In *Intelligent User Interfaces*, pages 33–39, 1997. http://citeseer.nj.nec.com/shneiderman97direct.html.

[SIG] ACM SIGCHI. Curricula for human-computer interaction. http://sigchi.org/cdg/index.html.

[SJ03] Jonh Sharp and Jon Jagger. *Microsoft Visual C# .NET Step by Step*. Step by STep. Microsoft, 2003.

[Ste99] Zeil J. Steven. Formal Specification – Invariants. http://www.cs.odu.edu/z̃eil/cs451/Lectures/02reqts/specprepost/specprepost_ht.html, 1999.

[TC04] Pedersen Torben and Jensen S. Christian. Multidimensional Databases, 2004. http://www.cs.auc.dk/ tbp/Teaching/-DAT5E01/mddatabasesPJ.pdf.

[Tro03] Trolltech. Qt, 2003. http://www.trolltech.com/.

[UI97] UNU-IIST. Formal Software Specification Using RAISE, 1997. http://www.iist.unu.edu/home/Unuiist/newrh/II/2/1/1/page.html.

[Vad96] Engelson Vadim. *An Approach to Automatic Construction of Graphical User Interfaces for Applications in Scientific Computing*. PhD thesis, Department of Computing and Information Science - Linköoping University, 1996.

[vBMvR] Alan J.J. van Beek, Hans B.F. Mulder, and Victor E. van Reijswoud. Rapid Application Development in Dynamic Organisations with Business Modelling - A Practitioners Point of View. http://citeseer.nj.nec.com/361875.html.

[VBS01] Jean Vanderdonckt, Laurent Bouillon, and Nathalie Souchon. Flexible Reverse Engineering of Web Pages with VAQUISTA. In *Working Conference on Reverse Engineering*, pages 241–248, 2001. http://citeseer.nj.nec.com/vanderdonckt01flexible.html.

[Víc96] Vega V. Víctor. PAC - Presentation-Abstraction-Control, 1996. http://www.cs.lth.se/Education/Courses/96.Dokt.Patterns/L6.1.PAC.ps.

[Vög03] Gabriel Vögler. UIML - User Interface Markup Language, 2003.

[Vic01] Eijkhout Victor. Tex by Topic, a Technician's Reference, 2001.

[VS03] VBscript and Jscript Microsoft Windows Scripting. Windows Scripting, 2003. http://www.microsoft.com/scripting/.

[W3C96] W3C. Toward a Formalism for Communication on the Web, 1996. http://www.w3c.org/Collaboration/Knowledge#feb94.

[W3C03] W3C. Hypertext Markup Language, 2003. http://www.w3c.org/Markup/.

Appendix A

VDM-SL Notation

This appendix describes the main particularities of VDM-SL, namely data types and their constructors and operators [And]. This synopsis behaves as a *VDM-SL* language guide.

Type Definitions

An example of a simple data type definition is:

```
Amount = nat
```

This defines a data type with name “Amount” and states that the values which belong to this type are natural numbers (**nat** is one of the basic types described below).

Invariants

In *VDM-SL* is possible to attach invariants to a type definition.

```
Type name == type expression
inv pattern == logical expression
```

The *pattern* can be a single identifier representing a typical element of the type or a *mk_expression* if the type is a record.

Basic data types and type constructors

Basic types:

Type	Values
nat	Natural numbers
nat1	Natural numbers excl. 0
int	Integers
real	Real numbers
bool	Booleans
char	Characters
token	Tokens

Quote types are written as identifiers surrounded by angle brackets e.g. <Red>.

Type constructors:

Constructor	Description
set of _	Finite sets
seq of _	Finite sequences
map _ to _	Finite mappings
_ | _	Type union
[_]	Optional type
:: notation	Record types

Data type operators

Boolean type

Operator	Name	Type
not b	Negation	$bool \longrightarrow bool$
a and b	Conjunction	$bool * bool \longrightarrow bool$
a or b	Disjunction	$bool * bool \longrightarrow bool$
$a \Rightarrow b$	Implication	$bool * bool \longrightarrow bool$
$a \Leftrightarrow b$	Biimplication	$bool * bool \longrightarrow bool$

Numeric types

Operator	Name	Type
-x	Unary minus	$real \longrightarrow real$
abs x	Absolute value	$real \longrightarrow real$
x + y	Sum	real * $real \longrightarrow real$
x - y	Difference	$real * real \longrightarrow real$
x * y	Product	$real * real \longrightarrow real$
x / y	Division	$real * real \longrightarrow real$
x**y	Power	$real * real \longrightarrow real$
$x < y$	Less than	$real * real \longrightarrow real$
$x > y$	Greater than	$real * real \longrightarrow real$
$x \leq y$	Less or equal	$real * real \longrightarrow real$
$x \geq y$	Greater or equal	$real * real \longrightarrow real$

Character, Quote and Token types

Characters, quotes and token values can only be compared to each other by equality and inequality.

Set types

Set enumeration $\{e_1, e_2, \ldots, , e_n\}$ constructs a set of the enumerated elements. The empty set is represented as $\{\}$.

Set comprehension $\{e \mid bd_1, bd_2, \ldots, bd_m \ \& \ P\}$ constructs a set by evaluation the expression *e* on all the bindings for which the predicate *P* evaluates to true. The expression *e* uses the variables defined in the bindings.

Set range $\{e_1, \ldots, e_2\}$ where e_1 and e_2 are numeric expressions. Denotes the set of integers from e_1 to e_2 inclusive.

Operator	Name	Type
e in set s1	Membership	$set\ of\ A \rightarrow bool$
e not in set s1	Not membership	$set\ of\ A \rightarrow bool$
s1 union s2	Union	$set\ of\ A * set\ of\ A \rightarrow set\ of\ A$
s1 inter s2	Intersection	$set\ of\ A * set\ of\ A \rightarrow set\ of\ A$
s1 $\setminus$ s2	Difference	$set\ of\ A * set\ of\ A \rightarrow set\ of\ A$
s1 subset s2	Subset	$set\ of\ A * set\ of\ A \rightarrow bool$
card s1	Cardinality	$set\ of\ A \rightarrow nat$
dunion ss	Distributed union	$set\ of\ (set\ of\ A) \rightarrow set\ of\ A$
dinter ss	Distributed intersection	$set\ of\ (set\ of\ A) \rightarrow set\ of\ A$

Sequence types

Sequence enumeration $[e_1, e_2, ..., e_n]$ constructs a sequence of the enumerated elements. The empty sequence is $[]$.

Sequence comprehension : $[e \mid \ id\ in\ set\ S \ \&\ P]$ constructs a sequence by evaluating the expression e on all the bindings for which the predicate P evaluates to true. The expression e will use the identifier id. S is a set of numbers and id will be matched to the numbers in the normal order (the smallest number first).

Subsequence A *subsequence* of a sequence *l* is a sequence formed from consecutive elements of *l*; from n_1 up to and including n_2. It has the form: $l\ (n_l, \ldots, n_2)$ where n_1 and n_2 are positive integer expressions (less than the length of *l*).

Operator	Name	Type
hd l	Head	$seq\ of\ A \rightarrow A$
tl l	Tail	$seq\ of\ A \rightarrow seq\ of\ A$
len l	Length	$seq\ of\ A \rightarrow nat$
elems l	Elements	$seq\ of\ A \rightarrow set\ of\ A$
inds l	Indices	$seq\ of\ A \rightarrow seq\ of\ nat1$
l1 ˆ l2	Concatenation	$seq\ of\ A * seq\ of\ A \rightarrow seq\ of\ A$
conc l1	Distributed concatenation	$seq\ of\ (seq\ of\ A) \rightarrow seq\ of\ A$
l(i)	Sequence index	$seq\ of\ A\ *\ nat1 \rightarrow A$

Mapping types

Mapping enumeration $\{a_1\ \mid\text{-}>\ b_1, a_2\ \mid\text{-}>\ b_2, \ldots, a_n\ \mid\text{-}>\ b_n\}$ constructs a mapping of the enumerated maplets. The empty mapping will be written as {| - >}.

Mapping comprehension : Mapping comprehension: $\{ed \mid\text{-}> er \mid bd_1, \ldots, bd_n\ \&\ P\}$ constructs a mapping by evaluating the expressions ed and er on all the possible bindings for which the predicate P evaluates to true. $bd_1, \ldots, bd_n$ are bindings of free identifiers from the expressions ed and er to sets or types.

Operator	Name	Type
dom m	Domain	$map\ A\ to\ B \rightarrow set\ of\ A$
rngm	Range	$map\ A\ to\ B \rightarrow set\ of\ B$
m1 munion m2	Map union	$map\ A\ to\ B * map\ A\ to\ B \rightarrow map\ A\ to\ B$
ml ++ m2	Override	$map\ A\ to\ B * map\ A\ to\ B \rightarrow map\ A\ to\ B$
s <: m	Domain restrict to	$set\ of\ A\ * map\ A\ to\ B \rightarrow map\ A\ to\ B$
s <-: m	Domain restrict by	$set\ of\ A\ * map\ A\ to\ B \rightarrow map\ A\ to\ B$
m :> s	Range restrict to	$set\ of\ B\ * map\ A\ to\ B \rightarrow map\ A\ to\ B$
m :-> s	Range restrict by	$set\ of\ B\ * map\ A\ to\ B \rightarrow map\ A\ to\ B$
m(d)	Mapping apply	$map\ A\ to\ B * A \rightarrow B$

Record types

Record values are constructed using a record constructor written as $mk_R ecId(r_l, r_2, \ldots, r_n)$where the different rs are arbitrary values and RecId is the name of the record type. Record types are defined as:

```
Type :: component name: type
        component name: type
        ...
        component name: type
```

For example, for a type defined:

```
Date :: day     : Day
        month   : Month
        year    : Year
```

The record constructor for Date is $mk\_Date(\_,\_,\_)$. The field selectors are $\_.day$, $\_.month$ and $\_.year$

Union and Optional Types

Union types are written as:

```
MasterA = A | B | ...
```

An optional type is written as:

[T]

This denotes a union between the elements from the type *T* and the special value nil.

Expressions

A **let expression** has the form:

```
let p1 = e1,...,pn = en in e
```

where $p1,\ldots,pn$ are variables, $e1,\ldots,en$ are expressions and *e* is an expression involving $p1,\ldots,pn$.
An **if expression** has the form:

```
if e1 then e2 else e3
```

where *e1* is a Boolean expression, while *e2* and *e3* are expressions of any type.

Quantified Expressions have the form:

```
Universal:      forall bd1, bd2,...,bdn & e
Existential:    exists bd1, bd2,...,bdn & e
```

where each *bdi* is a binding (i.e. either a set binding of the form *pi* in set *s* or a type binding of the form *pi: type*), and e is a Boolean expression involving the bound variables.

Function Definition

An explicit function definition has the form:

```
f: A * B * ... * Z -> R
f(a,b,...,z) == expr
pre preexpr(a,b,...,z)
```

An implicit function definition has the form:

```
f(a:A,b:B,...,z:Z) res:R
pre preexpr(a,b,...,z)
post postexpr(a,b,...,z,res)
```

Appendix B

W3C XML

W3C XML - Extended Markup Language

If we accept that java offers code portability, so XML offers data portability.

Extended Markup Language (XML) is a W3C Recommendation [Rec03c] which deserves attention for almost every software programmer. The markup language most people are familiar with today is, of course, HTML, that we use to create standard Web pages. So the concept “markup” is not so recent. However, XML can be seen as the new way to represent information in a text-based document.

XML is extensible (once everyone can create its own markup set of tags) and it is a meta-language (language used to create other languages).

The large spread of XML in almost all areas of science (and not only in computer science), demanded the creation of many and different technical documentation. Almost everything about the essence of XML is already written.

In this work we are going to explore the main properties of XML, describing its main supporting technologies. We will start by analyzing the sample XML *BookStore* on Listing B.1.

Listing B.1: *XML document*

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?xml-stylesheet href="book.xsl" type="text/xsl"/>
3  <!--a xml simple example-->
4  <BookStore>
5          <Book id="Mc98">
6                  <Title>My Life and Times</Title>
7                  <Author>Paul McCartney</Author>
8                  <Date>1998</Date>
9                  <ISBN>1-56592-235-2</ISBN>
10                 <Publisher>McMillin Publishing</Publisher>
11         </Book>
12 </BookStore>
```

In this XML little example:

- *line 1* represents the *Prolog* of any XML document. On it must be specified XML version and character set to be used (UTF-8 is the default set);

- *line 2* is a XML Processing Instruction (in this case referring a stylesheet);
- *line 3* is a commentary (as in HTML, with <!–...–> syntax);
- *line 4* has the first element (<BookStore>). Root element is its usual name.
- *line 5* has a *Book* element, *child* of BookStore. It has also the *attribute id* with "Mc98" value. Note that every XML element must be "closed", i.e., there should be an end element. *Book* end element is on *line 11* (</Book>).
- *line 6-line 10* have the Book child elements (<Title>, <Author>, <Date>, <ISBN> and <Publisher>).

XML is not...

- **Another programming language like C++, Java, etc.** - XML is a meta-language. Syntax/rules defined by XML can be used to create other markup languages.
- **The new support only for Internet or Web applications -** XML is an important choice to transfer data over Internet, however, it is being used for a much wider variety of applications.
- **Something to replace or in competition with HTML -** When you first look at XML, it might look very similar to HTML. Both are markup languages and have hierarchical structures containing elements (start-tag and end-tag) and attributes. However, XML is there to deal with data while HTML is about presentation.
- **Some proprietary technology -** As referred before, XML is extensible and it is an open standard created by W3C [Con03b]. So everyone can support XML and provide tools or technologies to work with it.

Unlike HTML

- XML is all about data; it does not provide any display/presentation details.
- XML does not have a fixed set of tags
- XML is case-sensitive
- XML has strict rules
  - Each start tag should have an end tag
  - Attribute values must be in single or double quotes
  - Tags cannot overlap
  - There is only one root element
  - No element may have two attributes with the same name

Valid and well-formed XML documents

All valid documents are well-formed. The inverse is not true.

As it happens with other languages, there are syntax rules which determine the correctness of XML. If these rules are respected, the document is *well-formed*. To be a *valid* document, besides being well-formed it must be according to the rules specified DTD - Document Type Definition [Rec98b] or XML Schema [Rec01c] documents.

Figure B.1 depicts the essential syntax rules for any XML well-formed document.

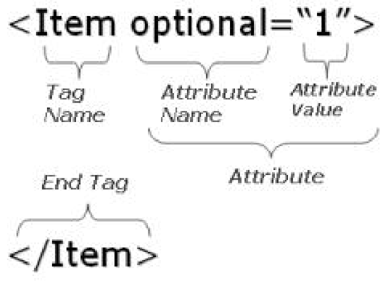


Figure B.1: *Essential XML rules*

A DTD has the responsibility of defining all possible element names (tag names), their occurrence and sequence, their attributes and respective types in a XML document. Unlike XML Schemas, a DTD does not offer semantic information.

Listing B.2 lists a possible DTD for our previous XML BookStore example.

Listing B.2: *A sample DTD*

```
<!ELEMENT BookStore (Book)+>
<!ELEMENT Book (Title, Author, Date, ISBN, Publisher)>
<!ELEMENT Title (#PCDATA)>
<!ELEMENT Author (#PCDATA)>
<!ELEMENT Date (#PCDATA)>
<!ELEMENT ISBN (#PCDATA)>
<!ELEMENT Publisher (#PCDATA)>
```

A XML Schema - often abbreviated by *XSchema* - provides a means to define the logical structure, content and semantics of XML documents [Con03b, Rec01c]. Unlike DTD, a XSchema does not offer grammatical information.

Listing B.3 shows a possible XSchema which validates our XML BookStore example.

Listing B.3: *A sample schema written in W3C XML Schema syntax*

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.books.org"
            xmlns="http://www.books.org"
            elementFormDefault="qualified">
    <xsd:element name="BookStore">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="Book" minOccurs="1" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="Book">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="Title" minOccurs="1" maxOccurs="1"/>
                <xsd:element ref="Author" minOccurs="1" maxOccurs="1"/>
                <xsd:element ref="Date" minOccurs="1" maxOccurs="1"/>
                <xsd:element ref="ISBN" minOccurs="1" maxOccurs="1"/>
                <xsd:element ref="Publisher" minOccurs="1" maxOccurs="1"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="Title" type="xsd:string"/>
    <xsd:element name="Author" type="xsd:string"/>
    <xsd:element name="Date" type="xsd:date"/>
    <xsd:element name="ISBN" type="xsd:string"/>
    <xsd:element name="Publisher" type="xsd:string"/>
</xsd:schema>
```

Deciding between DTD and XSchema depends on the problem type and dimension. If it is only necessary to ensure syntax, DTD should enough. Otherwise, if it is necessary to ensure data types (digits, characters, etc.) and their occurrences, the option must be XSchema. The programmer knowledge and experience can also influence the choice.

What is so important about XML?

There is a lot to say about XML which justifies the large appeal of XML. Next paragraphs describe the main XML properties which support this success [PSM+03]:

- **Everything is text -** All XML code is text which makes it highly portable. This is the reason XML is being heavily used for cross-platform data integration. If meaningful tag/attribute names are used to describe the data, the document becomes self-describing.
- **It is free and portable -** Many developers have started using XML in their application design/architecture because of the fact that XML is an open standard with excellent tools and vendor support; no one is the proprietary of XML. So it is free.
- **Just for Content -** XML appeared to describe or represent data (the content). There are other different available technologies to transform it into different presentation formats like html, pdf, etc. Next “Working with XML” topic will explore more clearly this process.
- **Easy to send -** Being easily to transform any XML document into any other format, it is easy to transfer information between different platforms.
- **Easy to parse and process -** Having a well defined structure and a textual property, any XML can be easily parsed and processed (like searching). There are a lot of parsers (in Java, C#, Perl, etc.) and other processing technologies available.

- **Easy to edit -** We do not need any special IDE to write or read XML. Because it is text, just *notepad* would do it.
- **Hierarchical Structure -** XML documents are hierarchical in nature with one top-level root element, and this way it is an excellent choice for modelling hierarchical data in an easy-to-read way.
- **Enables and supports other technologies -** XML is one of the core building blocks in the emerging Web services technology [Rec03d], for systems integration purpose. We can see also XML participating in multiple kinds of recent applications release, like office, graphic and multimedia applications.

If network communications bandwidth is limited, deciding to transfer XML documents could not be the best option. A XML document has a lot of redundant information which take a large amount of space. Considering this, binary code should be an important option!

Working with XML

The XML working process is clear and easy to assimilate. The source data is described in XML and then different technologies (several of them based also in XML) transform it in different formats. There must be present reader, parser, and interpreter applications. Figure B.2 depicts the main phases and associated technologies/tools on these processes.

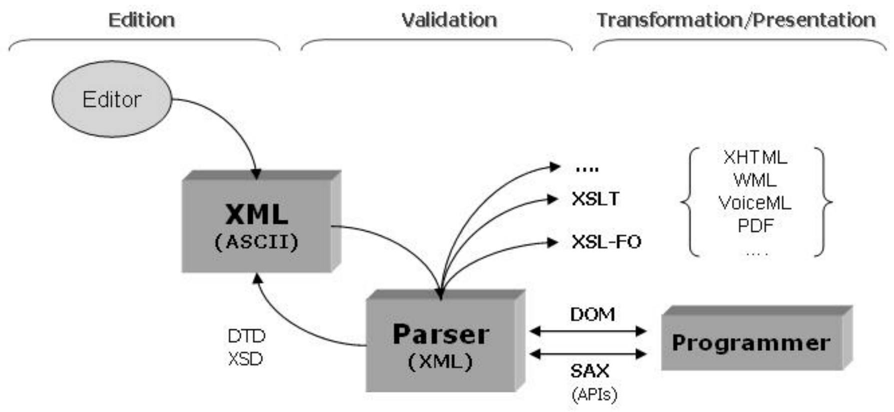


Figure B.2: *XML working process*

As we can see, XML working process is well structured and defined along three main phases: edition, validation, transformation and presentation.

Nevertheless an important question must be analyzed. As we have seen before, being extensible, XML allows everyone to define his own tags. Hence there is a large probability to use the same tag names to define different things, or the programmer himself when he repeats tag names when handling a large number of code lines, or even between different programs. This is a situation of *name conflicts*.

For instance, <*name*>*Luis Camoes*</*name*> and <*name*>*Os Lusiadas*</*name*> could be correct but could also represent different things. The former *name* could mean a *person name*; the later *name* could mean a *book name* or even its *title*.

There are two ways to resolve name conflicts: using attributes (like scope rules) and XML NameSpaces [Con00]. The former means the same we have said before: the attribute name scope is simply the element that it describes, ie, two attributes in the same element may not have the same name. The latter, Namespaces, represent the vocabulary used on tags set. If we intent to avoid the conflict, instead of simply using an element name like <name>, we use the following form: <prefix:name> The *name* part of the *prefix:name* construct is called a local name. The combination of prefix and name must be guaranteed to be unique. The used namespace is defined on XML document header using the *xmlns* tag. In our XML example could be:

```
<BookStore xmlns:bo="http=//www.mybook.pt/xml/book">
    <bo:Book>
        <bo:Title>My Life and Times</bo:Title>
        ...
    </bo:Book>
</BookStore>
```

Continuing to refer XML working process, we may say that, any text editor can be used in what concerns to its edition. Despite this there are more efficient tools, such as *XML Spy*[1] or *Butterfly XML*[2].

For validation and because of their complexity, DTD [Rec98b] or XSchema [Rec01c] must be created from wizard tools. In this way, they will be automatically created from XML source. Both previous noted applications are an example of these tools. Once created, DTD or XSchema will ensure the XML consistent format.

Process, interpret, search, "navigate" or even present XML documents demands a lot of specific technologies. We will present some of the most important:

- **Document Object Model - DOM** [Rec03b]: interface which describes methods to access, manipulate and manage the XML document. Using DOM one can build, navigate the structure, add, modify or delete elements and their content.
- **Simple API for XML Parsing - SAX** [ml98]: API to be implemented by event-based XML parsers, ie, as the parser moves through the document, events are reported via callbacks to an event handler.
- **XML Pointer Language - XPointer** [Rec02]: language to be used as the basis for a fragment identifier for any URI reference that locates a resource.
- **Extensible Stylesheet Language Family - XSL** [Rec01a] : family of recommendations for defining XML document transformation and presentation. Describes formatting and flow semantics for paginated presentation that can be expressed using an XML vocabulary of elements and attributes. Composed by XSLT, XSL-FO and XPath.

[1]http://www.altova.com
[2]http://www.butterflyxml.org/

- **XML Path Language - XPath** [Rec99a]: language for addressing parts of an XML document, designed to be used by both *XSLT* and *XPointer*.
- **XSL Transformations - XSLT** [Rec99b]: language for transforming XML documents into other XML documents.
- **XML Formatting Objects - XSLFO** [Rec01a]: XML vocabulary for specifying formatting semantics (like *PDF*, for instance).

In the following pages we are going to explore, in further detail, XSLT, which is one of the most important and used XML technology.

XSLT

XSL is both a transformation and a formatting language. The XSLT transformation part lets you scan through a document structure and rearrange its content any way you like. You can write out the content using a different set of XML tags, and generate text as needed. For example, you can scan through a document to locate all headings and then insert a generated table of contents at the beginning of the document, at the same time writing out the content marked up as HTML. XSL is also a rich formatting language, letting you apply typesetting controls to all components of your output. With a good formatting backend, it is capable of producing high quality printed pages[NB02].

An XSL stylesheet is written using XML syntax, and is itself a well-formed XML document. That makes the basic syntax familiar, and enables an XML processor to check for basic syntax errors. The stylesheet instructions use special element names, which typically begin with xsl: to distinguish them from any XML tags you want to appear in the output. The XSL namespace is identified at the top of the stylesheet file. As with other XML, any XSL elements that are not empty will require a closing tag. And some XSL elements have specific attributes that control their behavior. It helps to keep a good XSL reference book handy.

Here is an example of a simple XSL stylesheet applied to a simple XML file to generate HTML output.

Listing B.4: *Another XML example*

```
<?xml version="1.0"?>
<document>
        <title>Using a mouse</title>
        <para>
        It's easy to use a mouse. Just roll it around and click the buttons.
        </para>
</document>
```

Listing B.5: *XSL example*

```
<?xml version='1.0'?>
<xsl:stylesheet
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version='1.0'>
<xsl:output method="html"/>

<xsl:template match="document">
  <HTML><HEAD><TITLE>
    <xsl:value-of select="./title"/>
    </TITLE>
    </HEAD>
   <BODY>
      <xsl:apply-templates/>
    </BODY>
    </HTML>
</xsl:template>

<xsl:template match="title">
   <H1><xsl:apply-templates/></H1>
</xsl:template>

<xsl:template match="para">
   <P><xsl:apply-templates/></P>
</xsl:template>

</xsl:stylesheet>
```

Listing B.6: *HTML generated from XSL*

```
<HTML>
<HEAD>
<TITLE>Using a mouse</TITLE>
</HEAD>
<BODY>
        <H1>Using a mouse</H1>
        <P>It's easy to use a mouse. Just roll it around and click the buttons.</P>
</BODY>
</HTML>
```

XML Applications

Below is referred a list about the main and more recent XML applications[3].

- Data description - Perhaps the most common use of XML on transferring data between systems, over the Internet.
- Application Integration (EAI, ISI, legacy applications, etc.)
- CMS/LMS - Content Management
- Messaging and remote processing (SOAP and Web Services)
- File format
- Miscellaneous:
  - Configuration Files
  - Code documentation
  - RSS/RDF
  - Graphics (SVG)
  - Multimodal Applications (WML, VoiceML, etc.)
  - XForms and other data collection methods.

[3]URL *http://xml.coverpages.org/gen-apps.html* describe many others XML Applications in cross-domain and multi-disciplinary enterprises

Important XML links

- **XML Portals**
  - **http://xml.com/** - helps to learn how this new Internet technology can solve real-world problems in information management and electronic commerce.
  - **http://xmlhack.com/** - a news site for XML developers
- **XML Tutorials**
  - **http://xslt.com/resources_tutorials.htm** - A very good listing of XML and XSL(T) tutorials.
- **XML Resource Listings**
  - **http://www.xmlbooks.com/** - Charles F. Goldfarb's "All the XML Books in Print"
  - **http://www.dtd.com/** - Lists over 180 current XML-based language standards, pseudo-standards and developing standards in progress.
  - **http://www.xmlsoftware.com/** - XMLSOFTWARE: The XML Software Site
  - **http://www.dpawson.co.uk/xsl/xslfaq.html** - XSL Frequently Asked Questions (FAQ)
- **XML Tools and Software**
  - **http://4suite.org/** - Python
  - **http://www.altova.com/** - XML Spy - One of the best XML IDE
  - **http://www.butterflyxml.org/** Butterfly XML - One of the best Open source XML IDE.
- **XSLT**
  - **http://www.w3c.org/Style/XSL/** - The world wide web consortium on XSL.
  - **http://www.xslt.com/** - portal for things related xslt.
  - **http://xml.apache.org/** - Xalan, an open source, C++ and Java, implementation of xslt.

Appendix C

UIML DTD

Listing C.1: *UIML DTD 3.0*

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!--
    User Interface Markup Language (UIML)
    =====================================

    Developed by:

        Harmonia, Inc.

    Usage:

        <?xml version="1.0"?>
        <!DOCTYPE uiml PUBLIC "-//Harmonia//DTD UIML 3.0a Draft//EN"
        "http://uiml.org/dtds/UIML3_0a.dtd">
        NOTE: This URL has not yet been activated.

        <uiml>
          <head> ...      </head>
          <template> ...  </template>
          <peers> ...     </peers>
          <interface> ... </interface>
        </uiml>

    Description:

        This DTD corresponds to the UIML 3.0a specification.

    Change History:

        06 Feb 2002 - J Helms
                    - Initial Draft and added <repeat> and <iterator>
        02 July 2002 - K Rodriguez
                     - Correct syntax errors in param, repeat, and iterator


-->

<!-- =================== Content Models ====================== -->

<!--
    'uiml' is the root element of a UIML document.
-->

<!ELEMENT uiml (head?,(template|interface|peers)*) >
```

```
<!--
    The 'head' element is meant to contain metadata about the UIML
    document. You can specify metadata using the meta tag,
    this is similar to the head/meta from HTML.
-->

<!ELEMENT head (meta)*>
<!ELEMENT meta EMPTY>
<!ATTLIST meta
          name    NMTOKEN #REQUIRED
          content CDATA   #REQUIRED>

<!--
    The 'peers' element contains information that defines
    how a UIML interface component is mapped to the target platform's
    rendering technology and to the backend logic.
-->

<!ELEMENT peers (presentation|logic)*>
<!ATTLIST peers
          id     NMTOKEN                        #IMPLIED
          source CDATA                          #IMPLIED
          how    (union|cascade|replace)        "replace"
          export (hidden|optional|required)     "optional">

<!--
    The 'interface' element describes a user interface in terms of
    presentation widgets, component structure and behavior specifications.
-->

<!ELEMENT interface (structure|style|content|behavior)*>
<!ATTLIST interface
          id     NMTOKEN                        #IMPLIED
          source CDATA                          #IMPLIED
          how    (union|cascade|replace)        "replace"
          export (hidden|optional|required)     "optional">

<!--
    The 'template' element enables reuse of UIML elements.
    When an element appears inside a template element it can
    sourced by another element with the same tag.
-->

<!ELEMENT template (behavior|constant|content|d-class|d-component|interface
                   |logic|part|peers|presentation|property|restructure|rule
                   |script|structure|style)>
<!ATTLIST template
          id NMTOKEN #IMPLIED>

<!-- Peer related elements -->

<!--
    The 'presentation' element specifies the mapping between
    abstract interface parts and platform dependent widgets.
-->

<!ELEMENT presentation (d-class*)>
<!ATTLIST presentation
          id     NMTOKEN                        #IMPLIED
          source CDATA                          #IMPLIED
          base   CDATA                          #REQUIRED
          how    (union|cascade|replace)        "replace"
          export (hidden|optional|required)     "optional">

<!--
```

```
    The 'logic' element specifies the connection between the interface
    and the backend application, including support for scripting.
-->

<!ELEMENT logic (d-component*)>
<!ATTLIST logic
          id     NMTOKEN                          #IMPLIED
          source CDATA                            #IMPLIED
          how    (union|cascade|replace)          "replace"
          export (hidden|optional|required)       "optional">

<!--
    The 'd-component' element maps the name used in a <call> element to
    application logic external to the UIML document (e.g., a class in an
    object oriented language or a function in a scripting langauge).
-->

<!ELEMENT d-component (d-method)*>
<!ATTLIST d-component
          id       NMTOKEN                        #REQUIRED
          source   CDATA                          #IMPLIED
          how      (union|cascade|replace)        "replace"
          export   (hidden|optional|required)     "optional"
          maps-to  CDATA                          #IMPLIED
          location CDATA                          #IMPLIED>

<!--
    Maps class names that can be used for parts and events, as
    well as property and event data names, to UI toolkit.
-->

<!ELEMENT d-class (d-method*, d-property*, event*, listener*)>
<!ATTLIST d-class
          id          NMTOKEN                        #REQUIRED
          source      CDATA                          #IMPLIED
          how         (union|cascade|replace)        "replace"
          export      (hidden|optional|required)     "optional"
          used-in-tag (event|listener|part)          #REQUIRED
          maps-type   (attribute|tag|class)          #REQUIRED
          maps-to     CDATA                          #REQUIRED>

<!--
   Maps a property name to methods in UI toolkit that get and
   set propertys value.
-->

<!ELEMENT d-property (d-method*, d-param*)>
<!ATTLIST d-property
          id          NMTOKEN                                                  #REQUIRED
          maps-type   (attribute|getMethod|setMethod|method|constructor)   #REQUIRED
          maps-to     CDATA                                                    #REQUIRED
          return-type CDATA                                                    #IMPLIED>

<!--
   Maps a method to a callable method or function in the API of
   the application logic.
-->

<!ELEMENT d-method (d-param*, script?)>
<!ATTLIST d-method
          id          NMTOKEN                        #REQUIRED
          source      CDATA                          #IMPLIED
          how         (union|cascade|replace)        "replace"
          export      (hidden|optional|required)     "optional"
          maps-to     CDATA                          #REQUIRED
          return-type CDATA                          #IMPLIED>
```

```
<!--
    Defines a single formal parameter to a <d-method>.
-->

<!ELEMENT d-param (#PCDATA)>
<!ATTLIST d-param
          id    NMTOKEN #IMPLIED
          type  CDATA   #IMPLIED>


<!--
    The 'script' element contains executable script code. The type
    specifies the scripting language (see HTML4.0).
-->

<!ELEMENT script (#PCDATA)>
<!ATTLIST script
          id     NMTOKEN                          #IMPLIED
          type   NMTOKEN                          #IMPLIED
          source CDATA                            #IMPLIED
          how    (union|cascade|replace)          "replace"
          export (hidden|optional|required)       "optional">


<!-- Interface related elements -->

<!--
    The 'structure' element describes the initial organization of the
    parts that comprise the user interface.
-->

<!ELEMENT structure (part*)>
<!ATTLIST structure
          id     NMTOKEN                          #IMPLIED
          source CDATA                            #IMPLIED
          how    (union|cascade|replace)          "replace"
          export (hidden|optional|required)       "optional">

<!--
    Specifies a single abstract part of the user interface.
-->

<!ELEMENT part (style?, content?, behavior?, part*, repeat*)>
<!ATTLIST part
          id         NMTOKEN                          #IMPLIED
          class      NMTOKEN                          #IMPLIED
          source     CDATA                            #IMPLIED
          where      (first|last|before|after)        "last"
          where-part NMTOKEN                          #IMPLIED
          how        (union|cascade|replace)          "replace"
          export     (hidden|optional|required)       "optional">

<!--
    A 'repeat' element encapsulates a sub-tree of the overall interface
    virtual tree to be repeated 0 or more times. Each repeat MUST
    have one 'iterator' child.
-->

<!ELEMENT repeat (iterator ,part*)>


<!--
    An 'iterator' defines how many times a sub-tree should be repeated
    in an interface and serves as a indicator of the current iteration.
-->
```

```
<!ELEMENT iterator (#PCDATA|constant|property|call)*>
<!ATTLIST iterator
          id      NMTOKEN                         #REQUIRED>


<!--
    A 'style' element is composed of one or more 'property' elements,
    each of which specifies how a particular aspect of an interface
    component's presentation is to be presented.
-->

<!ELEMENT style (property*)>
<!ATTLIST style
          id      NMTOKEN                         #IMPLIED
          source  CDATA                           #IMPLIED
          how     (union|cascade|replace)         "replace"
          export  (hidden|optional|required)      "optional">

<!--
    A 'property' element is typically used to set a specified
    property for some interface component (or alternatively,
    a class of interface components), using the element's
    character data content as the value.  If the 'operation'
    attribute is given as "get", the element is equivalent to
    a property-get operation, the value of which may be "returned"
    as the content for an enclosing 'property' element.
-->

<!ELEMENT property (#PCDATA|constant|property|reference|call|op|event|iterator)*>
<!ATTLIST property
          name        NMTOKEN                         #IMPLIED
          source      CDATA                           #IMPLIED
          how         (union|cascade|replace)         "replace"
          export      (hidden|optional|required)      "optional"
          part-name   NMTOKEN                         #IMPLIED
          part-class  NMTOKEN                         #IMPLIED
          event-name  NMTOKEN                         #IMPLIED
          event-class NMTOKEN                         #IMPLIED
          call-name   NMTOKEN                         #IMPLIED
          call-class  NMTOKEN                         #IMPLIED>


<!--
    A 'reference' may be thought of as a property-get operation,
    where the "property" to be read is a 'constant' element defined
    in the UIML document's 'content' section.
-->

<!ELEMENT reference EMPTY>
<!ATTLIST reference
          constant-name   NMTOKEN #IMPLIED
          url-name        NMTOKEN #IMPLIED>

<!--
    The 'content' element is composed of one or more 'constant'
    elements, each of which specifies some fixed value.
-->

<!ELEMENT content (constant*)>
<!ATTLIST content
          id      NMTOKEN                         #IMPLIED
          source  CDATA                           #IMPLIED
          how     (union|cascade|replace)         "replace"
          export  (hidden|optional|required)      "optional">
```

```
<!--
    'constant' elements may be hierarchically structured.
-->

<!ELEMENT constant (constant*)>
<!ATTLIST constant
          id      NMTOKEN                          #IMPLIED
          source  CDATA                            #IMPLIED
          how     (union|cascade|replace)          "replace"
          export  (hidden|optional|required)       "optional"
          model   CDATA                            #IMPLIED
          value   CDATA                            #IMPLIED>

<!--
    The 'behavior' element gives one or more "rule"s that
    specifies what 'action' is to be taken whenever an associated
    'condition' becomes TRUE.
-->

<!ELEMENT behavior (rule*)>
<!ATTLIST behavior
          id      NMTOKEN                          #IMPLIED
          source  CDATA                            #IMPLIED
          how     (union|cascade|replace)          "replace"
          export  (hidden|optional|required)       "optional">


<!ELEMENT rule (condition,action)?>
<!ATTLIST rule
          id      NMTOKEN                          #IMPLIED
          source  CDATA                            #IMPLIED
          how     (union|cascade|replace)          "replace"
          export  (hidden|optional|required)       "optional">

<!--
    At the moment, "rule"s may be associated with two types of
    conditions: (1) whenever some expression is equal to some other
    expression; and (2) whenever some event is triggered and caught.
-->

<!ELEMENT condition (equal|event|op)>

<!ELEMENT equal (event,(constant|property|reference|op))>

<!ELEMENT op (constant|property|reference|call|op|event)*>
<!ATTLIST op
          name    CDATA                            #REQUIRED>

<!ELEMENT action (((property|call|restructure)*,event?)|(when-true?,when-false?,by-default?))>

<!ELEMENT when-true ((property|call)*,restructure?,op?,equal?,event?)>

<!ELEMENT when-false ((property|call)*,restructure?,op?,equal?,event?)>

<!ELEMENT by-default ((property|call)*,restructure?,op?,equal?,event?)>

<!ELEMENT restructure (template)?>
<!ATTLIST restructure
          at-part NMTOKEN                               #IMPLIED
          how     (union|cascade|replace|delete)        "replace"
          where   (first|last|before|after)             "last"
          where-part NMTOKEN                            #IMPLIED
          source  CDATA                                 #IMPLIED>

<!ELEMENT call (param*)>
<!ATTLIST call
```

```
          name   NMTOKEN #IMPLIED
          class  NMTOKEN #IMPLIED>

<!--
    'event' denotes one of three things:
    (1) When a child of <condition> or <op>, denotes that when the named
        event is fired, the condition should be evaluated.
    (2) When a child of <action>, denotes that the named event should
        be fired.
    (3) Inside <d-class>, denotes that the named event can occur for
        the part class named by the <d-class>.
-->

<!ELEMENT event EMPTY>
<!ATTLIST event
          name       NMTOKEN #IMPLIED
          class      NMTOKEN #IMPLIED
          part-name  NMTOKEN #IMPLIED
          part-class NMTOKEN #IMPLIED>

<!--
    'param' denotes a single actual parameter to a call-able routine.
-->

<!ELEMENT param (#PCDATA|property|reference|call|op|event|constant|iterator)*>
<!ATTLIST param
          name NMTOKEN #IMPLIED>

<!--
    'listener' denotes that a name defined with d-class
    used-in-tag="listener" should be attached as a listener to the
    d-class which contains this <listener> element.
-->

<!ELEMENT listener EMPTY>
<!ATTLIST listener
          class    NMTOKEN #IMPLIED
          attacher CDATA   #IMPLIED>
```

Appendix D

UIML 3.0 Hierarchy elements

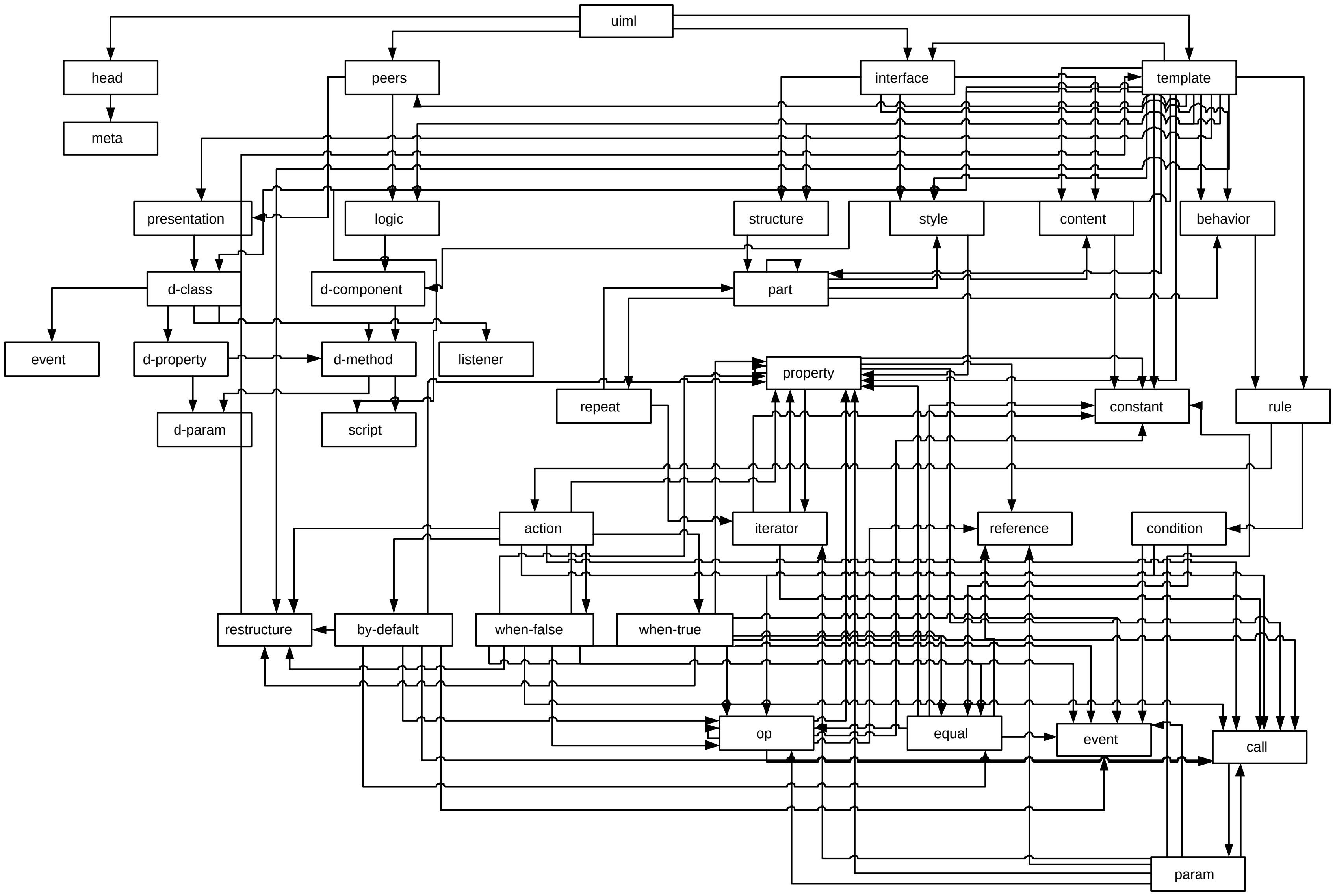


Figure D.1: *UIML 3.0 Reference*

Appendix E

Supporting Tools

E.1 Transcoding $UIML \mapsto VDM\text{-}SL$ - uiml2vdm stylesheet

Listing E.1: *XML Stylesheet to generate* VDM-SL *from UIML*

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!--
    by lufer
    UIML 2 VDM (UIML 2.0)
    05-09-2004: UIML3.0 support
-->

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:fo="http://www.w3.org/1999/XSL/Format"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:uiml="http://www.uiml.org/dtds/UIML2_0a.dtd">
    <xsl:output method="text" encoding="ISO-8859-1" indent="yes"/>

    <xsl:template match="/">
        <xsl:apply-templates/>
    </xsl:template>

<!-- =================== Uiml element ========================= -->

    <xsl:template match="uiml">VDM2UIML`uiml2str(mk_UIMLSpec`Uiml(
        <xsl:if test="count(head)=0">nil,</xsl:if>
        <xsl:if test="count(head) &gt; 0"><xsl:apply-templates select="head"/>,
        </xsl:if>
        <xsl:if test="count(interface | peers | template) = 0">[],</xsl:if>
        <xsl:if test="count(interface | peers | template) &gt; 0">
        [<xsl:for-each select="interface | peers | template">
                <xsl:apply-templates select="."/>
                <xsl:if test="position()!=last()">,</xsl:if>
            </xsl:for-each>]
        </xsl:if>))
    </xsl:template>

<!-- ==================== Head element ======================== -->

    <xsl:template match="head" name="head">
        mk_UIMLSpec`Head( [<xsl:for-each select="meta">
        <xsl:apply-templates select="."/>
        <xsl:if test="position()!=last()">,</xsl:if>
        </xsl:for-each>])
    </xsl:template>
```

```
<!-- ==================== Meta element ======================== -->

    <xsl:template match="meta" name="meta">
    mk_UIMLSpec`Meta(<xsl:call-template name="name"/>,
    <xsl:if test="@content">"<xsl:value-of select="@content"/>"</xsl:if>
    <xsl:if test="not (@content)">nil</xsl:if>)
    </xsl:template>

<!-- ================== Interface element ====================== -->
<!-- Interface = (Structure | Style | Content | Behavior)* + SourceAttributes -->

    <xsl:template match="interface">mk_UIMLSpec`Interface(
        [<xsl:for-each select="(structure | style | content | behavior)">
            <xsl:apply-templates select="."/>
            <xsl:if test="position()!=last()">,</xsl:if>
        </xsl:for-each>],
        <xsl:call-template name="sourceAttributes"/>)
    </xsl:template>

    <xsl:template name="interfaces">
        <xsl:if test="count(interface) = 0">[]</xsl:if>
        <xsl:if test="count(interface) &gt; 0">
            [<xsl:for-each select="interface">
                <xsl:apply-templates select="."/>
                <xsl:if test="not(position()=last())">,</xsl:if>
            </xsl:for-each>],
        </xsl:if>
    </xsl:template>

<!-- ==================== Template element ==================== -->

    <xsl:template match="template">mk_UIMLSpec`Template(
        <xsl:call-template name="sourceElements"/>,
        <xsl:call-template name="id"/>)
    </xsl:template>

<!-- ================== Structure element ======================= -->
<!-- Structure = Part* + SourcesAttributes -->

    <xsl:template match="structure">
    mk_UIMLSpec`Structure(<xsl:call-template name="parts"/>,
        <xsl:call-template name="sourceAttributes"/>)
    </xsl:template>

<!-- ===================== Part element ========================== -->
<!--Part = Style | Content | Behavior | Part* | Repeat*-->

    <xsl:template match="part" name="part">mk_UIMLSpec`Part(
        <!--<xsl:apply-templates select="style | content | behavior "/>-->
        <xsl:if test="style"><xsl:apply-templates select="style"/>,</xsl:if>
        <xsl:if test="not (style)">nil,</xsl:if>
        <xsl:if test="content ">
            <xsl:apply-templates select="."/>,</xsl:if>
        <xsl:if test="not (content )">nil,</xsl:if>
        <xsl:if test="behavior ">
            <xsl:apply-templates select="."/>,</xsl:if>
        <xsl:if test="not (behavior )">nil,</xsl:if>
        <xsl:call-template name="parts"/>,
        <xsl:call-template name="repeats"/>,
        <xsl:call-template name="sourceAttributes"/>,
        <xsl:call-template name="class_attr"/>,
        <xsl:call-template name="where_attr"/>,
        <xsl:call-template name="where_part_attr"/>)
    </xsl:template>

    <xsl:template name="parts">
```

```
        <xsl:if test="count(part) = 0">[]</xsl:if>
        <xsl:if test="count(part) &gt; 0">
            [<xsl:for-each select="part">
                <xsl:apply-templates select="."/>
                <xsl:if test="not(position()=last())">,</xsl:if>
            </xsl:for-each>]
        </xsl:if>
    </xsl:template>

<!-- ==================== Repeat element ====================== -->
<!--Repeat :: Iterator * Part*-->

    <xsl:template match="repeat" name="repeat">mk_UIMLSpec`Repeat(
        <xsl:apply-templates select="iterator"/>,
        <xsl:call-template name="parts"/>)
    </xsl:template>
    <xsl:template name="repeats">
        <xsl:if test="count(repeat) = 0">[]</xsl:if>
        <xsl:if test="count(repeat) &gt; 0">
            [<xsl:for-each select="repeat">
                <xsl:apply-templates select="."/>
                <xsl:if test="not(position()=last())">,</xsl:if>
            </xsl:for-each>]
        </xsl:if>
    </xsl:template>

<!-- ===================== Style element ======================= -->

    <xsl:template match="style" name="style">mk_UIMLSpec`Style(
        <xsl:call-template name="properties"/>,
        <xsl:call-template name="sourceAttributes"/>)
    </xsl:template>

<!-- ====================== Content element ====================== -->

    <xsl:template name="content" match="content">mk_UIMLSpec`Content(
        <xsl:call-template name="constants"/>,
        <xsl:call-template name="sourceAttributes"/>)
    </xsl:template>

<!-- ====================== Behavior element ===================== -->

    <xsl:template name="behavior" match="behavior">mk_UIMLSpec`Behavior(
        <xsl:call-template name="rules"/>,
        <xsl:call-template name="sourceAttributes"/>)
    </xsl:template>

<!-- ===================== Property element ====================== -->

    <xsl:template match="property" name="property">
        <xsl:value-of select="$CRTAB"/>mk_UIMLSpec`Property(
        <xsl:if test="count(text() | constant | property | reference |
            call | op | event | iterator) = 0">[],</xsl:if>
        <xsl:if test="count(text() | constant | property | reference |
            call | op | event | iterator) &gt; 0">
            [<xsl:for-each select="(text() | constant | property |
                reference | call | op | event | iterator)">
                <xsl:apply-templates select="."/>
            <xsl:if test="position()!=last()">,</xsl:if>
            </xsl:for-each>],
        </xsl:if>
            <xsl:call-template name="name"/>,
            <xsl:call-template name="source"/>,
            <xsl:call-template name="how"/>,
            <xsl:call-template name="export"/>,
            <xsl:call-template name="part_name"/>,
            <xsl:call-template name="part_class"/>,
            <xsl:call-template name="event_name"/>,
            <xsl:call-template name="event_class"/>)
    </xsl:template>
    <xsl:template name="properties">
        <xsl:if test="count(property) = 0">[]</xsl:if>
        <xsl:if test="count(property) &gt; 0">
            [<xsl:for-each select="property">
                <xsl:apply-templates select="."/>
                <xsl:if test="position()!=last()">,</xsl:if>
            </xsl:for-each>]
        </xsl:if>
    </xsl:template>

<!-- ================== Constant element ===================== -->

    <xsl:template name="constant" match="constant">mk_UIMLSpec`Constant(
        <xsl:call-template name="constants"/>,
        <xsl:call-template name="sourceAttributes"/>,
        <xsl:call-template name="model"/>,
        <xsl:call-template name="value"/>)
    </xsl:template>

    <xsl:template name="constants">
        <xsl:if test="count(constant) = 0">[]</xsl:if>
        <xsl:if test="count(constant) &gt; 0">
            [<xsl:for-each select="constant">
                <xsl:apply-templates select="."/>
                <xsl:if test="position()!=last()">,</xsl:if>
            </xsl:for-each>]
        </xsl:if>
    </xsl:template>

<!-- ================ Reference element ==================== -->

    <xsl:template name="reference" match="reference">
        mk_UIMLSpec`Reference(
        "<xsl:value-of select="@constant-name"/>",
        "<xsl:value-of select="@url-name"/>")
    </xsl:template>

<!-- =================== Logic element ===================== -->

    <xsl:template name="logic" match="logic">mk_UIMLSpec`Logic(
        <xsl:if test="count(d-component) = 0">[],</xsl:if>
        <xsl:if test="count(d-component) &gt; 0">
            [<xsl:for-each select="d-component">
                <xsl:apply-templates select="."/>
                <xsl:if test="position()!=last()">,</xsl:if>
            </xsl:for-each>],
        </xsl:if>
        <xsl:call-template name="sourceAttributes"/>)
    </xsl:template>

<!-- =============== Presentation element =================== -->

    <xsl:template name="presentation" match="presentation">
        mk_UIMLSpec`Presentation(
        <xsl:if test="count(d-class) = 0">[],</xsl:if>
        <xsl:if test="count(d-class) &gt; 0">
            [<xsl:for-each select="d-class">
                <xsl:apply-templates select="."/>
                <xsl:if test="position()!=last()">,</xsl:if>
            </xsl:for-each>],
        </xsl:if>
        <xsl:call-template name="sourceAttributes"/>,
        <xsl:call-template name="base"/>      )
    </xsl:template>

<!-- ================= Rule element ========================== -->

    <xsl:template name="rule" match="rule">mk_UIMLSpec‘Rule(
        <xsl:if test="condition">
            <xsl:apply-templates select="condition"/>,
            </xsl:if>
        <xsl:if test="not(condition)">nil, </xsl:if>
        <xsl:if test="action">
            <xsl:apply-templates select="action"/>,
            </xsl:if>
        <xsl:if test="not(action)">nil, </xsl:if>
        <xsl:call-template name="sourceAttributes"/>)
    </xsl:template>

    <xsl:template name="rules">
        <xsl:if test="count(rule)  = 0">nil</xsl:if>
        <xsl:if test="count(rule) &gt; 0">
            [<xsl:for-each select="rule">
                <xsl:apply-templates select="."/>
                <xsl:if test="position()!=last()">,</xsl:if>
            </xsl:for-each>]
        </xsl:if>
    </xsl:template>

<!-- ================= Condition element ====================== -->

    <xsl:template name="condition" match="condition">mk_UIMLSpec‘Condition(
        <xsl:apply-templates select="equal | event | op"/>)
    </xsl:template>

<!-- =================== Action element ======================= -->

    <xsl:template name="action" match="action">
        <xsl:choose>
            <xsl:when test="count(property | call | restructure | event) &gt; 0">
                mk_UIMLSpec‘ActionType1([<xsl:for-each select="(property |
                        call | restructure)">
                    <xsl:apply-templates select="."/>
                    <xsl:if test="position()!=last()">,</xsl:if>
                </xsl:for-each>],
                <xsl:if test="count(event) &gt; 0">
                    <xsl:apply-templates select="."/>)
                </xsl:if>
                <xsl:if test="count(event) = 0">nil</xsl:if>)
            </xsl:when>
        </xsl:choose>
        <xsl:choose>
            <xsl:when test="count(when-true | when-false | by-default) &gt; 0">
                mk_UIMLSpec‘ActionType2(<xsl:for-each select="when-true |
                        when-false | by-default">
                    <xsl:apply-templates select="."/>
                    <xsl:if test="position()!=last()">,</xsl:if>
                </xsl:for-each>)
            </xsl:when>
        </xsl:choose>
    </xsl:template>

<!-- =================== Equal element ====================== -->

    <xsl:template name="equal" match="equal">mk_UIMLSpec‘Equal(
        <xsl:apply-templates select="event"/>,
        <xsl:apply-templates select="constant | property | reference | op"/>)
    </xsl:template>
```

```
<!-- ==================== Call element ====================== -->

    <xsl:template name="call" match="call">mk_UIMLSpec`Call(
        <xsl:call-template name="params"/>,
    <xsl:call-template name="name"/>)
    </xsl:template>

<!-- ==================== Event element ====================== -->

    <xsl:template name="event" match="event">mk_UIMLSpec`Event(
        <xsl:choose>
            <xsl:when test="@name">"<xsl:value-of select="@name"/>",</xsl:when>
            <xsl:otherwise>" ",</xsl:otherwise>
        </xsl:choose>
        <xsl:choose>
            <xsl:when test="@part-name">"<xsl:value-of select="@part-name"/>",
                </xsl:when>
            <xsl:otherwise>" ",</xsl:otherwise>
        </xsl:choose>
        <xsl:choose>
            <xsl:when test="@part-class">"<xsl:value-of select="@part-class"/>",
                </xsl:when>
            <xsl:otherwise>" ",</xsl:otherwise>
        </xsl:choose>
        <xsl:choose>
            <xsl:when test="@class">"<xsl:value-of select="@class"/>"</xsl:when>
            <xsl:otherwise>" "</xsl:otherwise>
        </xsl:choose>)
    </xsl:template>

<!-- ==================== Param element ===================== -->

    <xsl:template name="param" match="param">mk_UIMLSpec`Param(
        <xsl:if test="count(text() | property | reference | call |
            op | event | constant | iterator) = 0">nil,</xsl:if>
        <xsl:if test="count(text() | property | reference | call |
            op | event | constant  | iterator) &gt; 0">
            <xsl:for-each select="text() | property | reference | call |
            op | event | constant  | iterator">
                <xsl:apply-templates select="."/>
            </xsl:for-each>,
        </xsl:if>
        <xsl:call-template name="name"/>)
    </xsl:template>

    <xsl:template name="params">
        <xsl:if test="count(param)  = 0"> []</xsl:if>
        <xsl:if test="count(param)  &gt;  0">
            [<xsl:for-each select="param">
                <xsl:apply-templates select="."/>
                <xsl:if test="position()!=last()">,</xsl:if>
            </xsl:for-each>]
        </xsl:if>
    </xsl:template>

<!-- ================= Peer  element ====================== -->

    <xsl:template name="peers" match="peers">mk_UIMLSpec`Peers(
        <xsl:if test="count(presentation | logic) = 0">[],</xsl:if>
        <xsl:if test="count(presentation | logic) &gt; 0">
            [<xsl:for-each select="presentation | logic">
                <xsl:apply-templates select="."/>
                <xsl:if test="position()!=last()">,</xsl:if>
            </xsl:for-each>],
        </xsl:if>
        <xsl:call-template name="sourceAttributes"/>)
    </xsl:template>

<!-- ================== Op element ========================= -->

    <xsl:template name="op" match="op">mk_UIMLSpec`Op(
        <xsl:if test="count(constant | property | reference | call |
            op | event)=0">[],</xsl:if>
        <xsl:if test="count(constant | property | reference | call |
            op | event) &gt; 0">
        [<xsl:for-each select="constant | property | reference | call |
            op | event">
                <xsl:apply-templates select="."/>
                <xsl:if test="position()!=last()">,</xsl:if>
            </xsl:for-each>],
        </xsl:if>
        <xsl:call-template name="name"/>)
    </xsl:template>

<!-- ================ Restructure element ===================== -->

    <xsl:template name="restructure" match="restructure">mk_UIMLSpec`Restructure(
        <xsl:if test="template">
            <xsl:apply-templates select="template"/>,</xsl:if>
        <xsl:if test="not (template)">nil,</xsl:if>
        <xsl:call-template name="at-part"/>,
        <xsl:call-template name="how"/>,
        <xsl:call-template name="where_attr"/>,
        <xsl:call-template name="where_part_attr"/>,
        <xsl:call-template name="source"/>)
    </xsl:template>

<!-- ================== Iterator element ====================== -->

    <xsl:template name="iterator" match="iterator">mk_UIMLSpec`Iterator(
            <xsl:apply-templates select="(text() | constant | property |
            call)"/>,<xsl:call-template name="id"/>)
    </xsl:template>

<!-- ================ when-true element ======================= -->

    <xsl:template name="when-true" match="when-true">mk_UIMLSpec`When_true(
        <xsl:if test="count( property | call) =0">[],</xsl:if>
        <xsl:if test="count( property | call) &gt; 0">
            [<xsl:for-each select="property | call">
                <xsl:apply-templates select="."/>
                <xsl:if test="position()!=last()">,</xsl:if>
            </xsl:for-each>],
        </xsl:if>
        <xsl:if test="restructure">
            <xsl:apply-templates select="restructure"/>,</xsl:if>
        <xsl:if test="not (restructure)">nil,</xsl:if>
        <xsl:if test="op">
            <xsl:apply-templates select="op"/>,</xsl:if>
        <xsl:if test="not (op)">nil,</xsl:if>
        <xsl:if test="equal">
            <xsl:apply-templates select="equal"/>,</xsl:if>
        <xsl:if test="not (equal)">nil,</xsl:if>
        <xsl:if test="event">
            <xsl:apply-templates select="event"/>
        </xsl:if>
        <xsl:if test="not (event)">nil</xsl:if>)
    </xsl:template>

<!-- ================= when-false element ===================== -->
```

```
    <xsl:template name="when-false" match="when-false">mk_UIMLSpec`When_false(
        <xsl:if test="count( property | call) =0">[],</xsl:if>
        <xsl:if test="count( property | call) &gt; 0">
            [<xsl:for-each select="property | call">
                <xsl:apply-templates select="."/>
                <xsl:if test="position()!=last()">,</xsl:if>
            </xsl:for-each>],
        </xsl:if>
        <xsl:if test="restructure">
            <xsl:apply-templates select="restructure"/>,</xsl:if>
        <xsl:if test="not (restructure)">nil,</xsl:if>
        <xsl:if test="op">
            <xsl:apply-templates select="op"/>,</xsl:if>
        <xsl:if test="not (op)">nil,</xsl:if>
        <xsl:if test="equal">
            <xsl:apply-templates select="equal"/>,</xsl:if>
        <xsl:if test="not (equal)">nil,</xsl:if>
        <xsl:if test="event">
            <xsl:apply-templates select="event"/>
        </xsl:if>
        <xsl:if test="not (event)">nil</xsl:if>)
    </xsl:template>

<!-- ================= by-default element ======================== -->

    <xsl:template name="by-default" match="by-default">mk_UIMLSpec`By_default(
        <xsl:if test="count( property | call) =0">[],</xsl:if>
        <xsl:if test="count( property | call) &gt; 0">
            [<xsl:for-each select="property | call">
                <xsl:apply-templates select="."/>
                <xsl:if test="position()!=last()">,</xsl:if>
            </xsl:for-each>],
        </xsl:if>
        <xsl:if test="restructure">
            <xsl:apply-templates select="restructure"/>,</xsl:if>
        <xsl:if test="not (restructure)">nil,</xsl:if>
        <xsl:if test="op">
            <xsl:apply-templates select="op"/>,</xsl:if>
        <xsl:if test="not (op)">nil,</xsl:if>
        <xsl:if test="equal">
            <xsl:apply-templates select="equal"/>,</xsl:if>
        <xsl:if test="not (equal)">nil,</xsl:if>
        <xsl:if test="event">
            <xsl:apply-templates select="event"/>
        </xsl:if>
        <xsl:if test="not (event)">nil</xsl:if>)
    </xsl:template>

<!-- ================ d-component element ====================== -->

    <xsl:template name="d-component" match="d-component ">mk_UIMLSpec`D_component(
        <xsl:if test="count(d-method) = 0">[],</xsl:if>
        <xsl:if test="count(d-method) &gt;  0">
            [<xsl:for-each select="d-method">
                <xsl:apply-templates select="."/>
                <xsl:if test="position()!=last()">,</xsl:if>
            </xsl:for-each>],
        </xsl:if>
        <xsl:call-template name="sourceAttributes"/>,
        <xsl:call-template name="maps-to"/>,
        <xsl:call-template name="location"/>)
    </xsl:template>

<!-- ================== d-class element ===================== -->

    <xsl:template name="d-class" match="d-class ">mk_UIMLSpec`D_class(
        <xsl:if test="count(d-method) = 0">[],</xsl:if>
        <xsl:if test="count(d-method) &gt; 0">
            [<xsl:for-each select="d-method">
                <xsl:apply-templates select="."/>
                <xsl:if test="position()!=last()">,</xsl:if>
            </xsl:for-each>],
        </xsl:if>
        <xsl:if test="count(d-property) = 0">nil,</xsl:if>
        <xsl:if test="count(d-property) &gt; 0">
            [<xsl:for-each select="d-property">
                <xsl:apply-templates select="."/>
                <xsl:if test="position()!=last()">,</xsl:if>
            </xsl:for-each>],
        </xsl:if>
        <xsl:if test="count(event) = 0">nil,</xsl:if>
        <xsl:if test="count(event) &gt; 0">
            [<xsl:for-each select="event">
                <xsl:apply-templates select="."/>
                <xsl:if test="position()!=last()">,</xsl:if>
            </xsl:for-each>],
        </xsl:if>
        <xsl:if test="count(listener) = 0">nil,</xsl:if>
        <xsl:if test="count(listener) &gt; 0">
            [<xsl:for-each select="listener">
                <xsl:apply-templates select="."/>
                <xsl:if test="position()!=last()">,</xsl:if>
            </xsl:for-each>],
        </xsl:if>
        <xsl:call-template name="sourceAttributes"/>,
        <xsl:call-template name="maps-to"/>,
        <xsl:call-template name="maps-type"/>,
        <xsl:call-template name="used-in-tag"/>)
    </xsl:template>

<!-- ================== d-property element ==================== -->

    <xsl:template name="d-property" match="d-property ">mk_UIMLSpec`D_property(
        <xsl:if test="count(d-method) = 0">[],</xsl:if>
        <xsl:if test="count(d-method) &gt; 0">
            [<xsl:for-each select="d-method">
                <xsl:apply-templates select="."/>
                <xsl:if test="position()!=last()">,</xsl:if>
            </xsl:for-each>],
        </xsl:if>
        <xsl:if test="count(d-param) = 0">nil,</xsl:if>
        <xsl:if test="count(d-param) &gt; 0">
            [<xsl:for-each select="d-param">
                <xsl:apply-templates select="."/>
                <xsl:if test="position()!=last()">,</xsl:if>
            </xsl:for-each>],
        </xsl:if>
        <xsl:call-template name="id"/>,
        <xsl:call-template name="maps-type"/>,
        <xsl:call-template name="maps-to"/>,
        <xsl:call-template name="return-type"/>)
    </xsl:template>

<!-- ================= d-method element ======================= -->

    <xsl:template name="d-method" match="d-method ">mk_UIMLSpec`D_method (
        <xsl:if test="count(d-param) = 0">[],</xsl:if>
        <xsl:if test="count(d-param) &gt; 0">
            [<xsl:for-each select="d-param">
                <xsl:apply-templates select="."/>
                <xsl:if test="position()!=last()">,</xsl:if>
            </xsl:for-each>],
        </xsl:if>
        <xsl:if test="count(sript) = 0">nil,</xsl:if>
        <xsl:if test="count(script) &gt; 0">
            <xsl:apply-templates select="script"/>,
        </xsl:if>
        <xsl:call-template name="sourceAttributes"/>,
        <xsl:call-template name="maps-to"/>,
        <xsl:call-template name="return-type"/>)
    </xsl:template>

<!-- =================== d-param element ======================== -->

    <xsl:template name="d-param" match="d-param ">mk_UIMLSpec`D_param (
        <xsl:apply-templates select="text()"/>,
        <xsl:call-template name="id"/>,
        <xsl:call-template name="type"/>)
    </xsl:template>

<!-- =================== SourceElements ========================== -->

    <xsl:template name="sourceElements">
        <xsl:if test="behavior|d-class|d-component|constant|content|
             interface|logic|part|peers |presentation|property|restructure|
             rule|script|structure|style">
            <xsl:apply-templates select="child::*"/></xsl:if>
        <xsl:if test="not(behavior|d-class|d-component|constant|content|
             interface|logic|part|peers |presentation|property|restructure|
             rule|script|structure|style)">nil</xsl:if>
    </xsl:template>

<!-- ================== SourceAttributes ========================= -->
<!--     SourcesAttributes= id + src + how + export
         id=String
         src=String
         how=Append | Cascade | Replace
         export= Hidden | Optional | Required
 -->
    <xsl:template name="sourceAttributes">
        <xsl:call-template name="id"/>,<xsl:call-template name="source"/>,
        <xsl:call-template name="how"/>,<xsl:call-template name="export"/>
    </xsl:template>

    <xsl:template name="source">
        <xsl:if test="@source">"<xsl:value-of select="@source"/>"</xsl:if>
        <xsl:if test="not(@source)">" "</xsl:if>
    </xsl:template>

    <xsl:template name="how">
        <xsl:if test="@how">&lt;<xsl:value-of select="@how"/>&gt;</xsl:if>
    </xsl:template>

    <xsl:template name="export">
        <xsl:if test="@export">&lt;<xsl:value-of select="@export"/>&gt;
        </xsl:if>
    </xsl:template>

    <xsl:template name="id" match="id">
        <xsl:if test="@id">mk_UIMLSpec`ID("<xsl:value-of select="@id"/>")
        </xsl:if>
        <xsl:if test="not(@id)">nil</xsl:if>
    </xsl:template>

<!-- =================== Other Attributes ======================== -->

    <xsl:template name="class_attr">
        <xsl:if test="@class">"<xsl:value-of select="@class"/>"
        </xsl:if>
        <xsl:if test=”not(@class)”>” ”l</xsl:if>
    </xsl:template>

    <xsl:template name=”class”>
        <xsl:if test=”@class”>”<xsl:value-of select=”@class”/>”
        </xsl:if>
        <xsl:if test=”not(@class)”>” ”</xsl:if>
    </xsl:template>

    <xsl:template name=”name”>
        <xsl:if test=”@name”>”<xsl:value-of select=”@name”/>”
        </xsl:if>
        <xsl:if test=”not(@name)”>” ”</xsl:if>
    </xsl:template>

    <xsl:template name=”where_attr”>
        <xsl:if test=”@where”>&lt;<xsl:value-of select=”@where”/>&gt;
        </xsl:if>
        <xsl:if test=”not(@where)”>&lt;last&gt;</xsl:if>
    </xsl:template>

    <xsl:template name=”where_part_attr”>
        <xsl:if test=”@where_part”>”<xsl:value-of select=”@where_part”/>”
        </xsl:if>
        <xsl:if test=”not(@where_part)”>” ”</xsl:if>
    </xsl:template>

    <xsl:template name=”part_name”>
        <xsl:if test=”@part-name”>”<xsl:value-of select=”@part-name”/>”
        </xsl:if>
        <xsl:if test=”not(@part-name)”>” ”</xsl:if>
    </xsl:template>

    <xsl:template name=”part_class”>
        <xsl:if test=”@part-class”>”<xsl:value-of select=”@part-class”/>”
        </xsl:if>
        <xsl:if test=”not (@part-class)”>” ”</xsl:if>
    </xsl:template>

    <xsl:template name=”event_name”>
        <xsl:if test=”@event-name”>”<xsl:value-of select=”@event-name”/>”
        </xsl:if>
        <xsl:if test=”not (@event-name)”>” ”</xsl:if>
    </xsl:template>

    <xsl:template name=”event_class”>
        <xsl:if test=”@event-class”>”<xsl:value-of select=”@event-class”/>”
        </xsl:if>
        <xsl:if test=”not (@event-class)”>” ”</xsl:if>
    </xsl:template>

    <xsl:template name=”model”>
        <xsl:if test=”@model”>”<xsl:value-of select=”@model”/>”
        </xsl:if>
        <xsl:if test=”not (@model)”>” ”</xsl:if>
    </xsl:template>

    <xsl:template name=”value”>
        <xsl:if test=”@value ”>”<xsl:value-of select=”@value ”/>”</xsl:if>
        <xsl:if test=”not (@value )”>” ”</xsl:if>
    </xsl:template>

    <xsl:template name=”constant_name”>
        <xsl:if test=”@constant-name”>”<xsl:value-of select=”@constant-name”/>”
        </xsl:if>
```

```
        <xsl:if test="not (@constant-name)">" "</xsl:if>
    </xsl:template>

    <xsl:template name="url_name">
        <xsl:if test="@url-name">"<xsl:value-of select="@url-name"/>"
        </xsl:if>
        <xsl:if test="not (@url-name)">" "</xsl:if>
    </xsl:template>

    <xsl:template name="at-part">
        <xsl:if test="@at-part">"<xsl:value-of select="@at-part"/>"
        </xsl:if>
        <xsl:if test="not (@at-part)">" "</xsl:if>
    </xsl:template>

    <xsl:template name="base">
        "<xsl:value-of select="@base"/>"
    </xsl:template>

    <xsl:template name="type">
        <xsl:if test="@type">"<xsl:value-of select="@type"/>"</xsl:if>
        <xsl:if test="not (@type)">" "</xsl:if>
    </xsl:template>

    <xsl:template name="maps-to">
        <xsl:if test="@maps-to">"<xsl:value-of select="@maps-to"/>"</xsl:if>
        <xsl:if test="not (@maps-to)">" "</xsl:if>
    </xsl:template>

    <xsl:template name="return-type">
        <xsl:if test="@return-type">"<xsl:value-of select="@return-type"/>"
        </xsl:if>
        <xsl:if test="not (@return-type)">" "</xsl:if>
    </xsl:template>

    <xsl:template name="used-in-tag">
        <xsl:if test="@used-in-tag">&lt;<xsl:value-of select="@used-in-tag"/>
        &gt;</xsl:if>
        <xsl:if test="not (@used-in-tag)">" "</xsl:if>
    </xsl:template>

    <xsl:template name="maps-type">
        <xsl:if test="@maps-type">&lt;<xsl:value-of select="@maps-type"/>&
         gt;</xsl:if>
        <xsl:if test="not (@maps-type)">" "</xsl:if>
    </xsl:template>

    <xsl:template name="location">
        <xsl:if test="@location ">"<xsl:value-of select="@location "/>"
        </xsl:if>
        <xsl:if test="not (@location )">" "</xsl:if>
    </xsl:template>

<!-- =================== Method element ======================= -->

    <xsl:template name="method" match="method">mk_UIMLSpec`Method(
         [<xsl:call-template name="params"/>],
             <xsl:if test="returns">
             <xsl:apply-templates select="returns"/>,</xsl:if>
        <xsl:if test="not(returns)">nil,</xsl:if>
        <xsl:if test="script">
             <xsl:apply-templates select="script"/>,</xsl:if>
        <xsl:call-template name="sourceAttributes"/>,
             <xsl:if test="@maps-to">"<xsl:value-of select="@maps-to"/>",
             </xsl:if>
        <xsl:if test="@type=(input | output | inout | none)">
```

```
                mk_UIMLSpec`TypeOption("<xsl:value-of select="@type"/>")
            </xsl:if>
        <xsl:if test="not(@type=(input | output | inout | none))">
                mk_UIMLSpec`TypeOption("inout")</xsl:if>)
    </xsl:template>
    <xsl:template name="methods">
        <xsl:for-each select="method">
            <xsl:apply-templates select="."/>
            <xsl:if test="position()!=last()">,</xsl:if>
        </xsl:for-each>
    </xsl:template>

<!-- ================= Returns element ==================== -->

    <xsl:template name="returns" match="returns">mk_UIMLSpec`Returns(
        <xsl:if test="@name">"<xsl:value-of select="@name"/>"</xsl:if>
        <xsl:if test="not(@name)">nil</xsl:if>)
    </xsl:template>

    <xsl:template name="script" match="script">mk_UIMLSpec`Script(
        <xsl:if test="string-length() &gt;0">"<xsl:value-of
         select="normalize-space(.)"/>",
        </xsl:if>
        <xsl:call-template name="sourceAttributes"/>,
        <xsl:call-template name="type"/>)
    </xsl:template>

    <xsl:template match="text()">
        <xsl:if test="string-length() &gt; 0">
            "<xsl:value-of select="normalize-space(.)"/>"
        </xsl:if>
    </xsl:template>

<!-- =================== Variables ===================== -->

    <xsl:variable name="CR" select="'&#13;'"/>
    <xsl:variable name="CRTAB" select="'&#13;&#9;'"/>
    <xsl:variable name="TAB" select="'&#13;&#9;'"/>

</xsl:stylesheet>
```

E.2 Verifier $VDM \mapsto UIML$ - vdm2uiml

This Appendix presents an excerpt of *vdm2uiml VDM-SL* module, responsible for the second phase of our architecture. It allows the generation of *UIML* from the *VDM-SL* specification. The complete source code is available on [Luí04].

```
module $VDM2UIML$
    imports
      from $IO$ all ,
      from $UIMLSpec$ all
    exports all
definitions
```

E.2.1 Auxiliar Data Types

```
    $FileName = UIMLSpec`String;$
    $PropertyTypes = UIMLSpec`String \mid UIMLSpec`Constant \mid UIMLSpec`Property \mid UIMLSpec`Reference \mid UIMLSpec`Call \mid UIMLSpec`Iterator;$
    $ConstantType = UIMLSpec`String \mid UIMLSpec`Constant;$
    $ActionType = UIMLSpec`Property \mid UIMLSpec`Call \mid UIMLSpec`Restructure$
```

```
values
    $PI : UIMLSpec`String =$ " $<?xmlversion = '1.0'encoding = 'ISO\text{-}8859\text{-}1'? >$ ";
    $end\text{-}doc : UIMLSpec`String =$ " $< /uiml >$ ";
    $end\text{-}doc\text{-}html : UIMLSpec`String =$ " $< peers >< presentationhow = 'replace'source = 'HTML_3.2_H armonia_1.0.uiml\#vocab'base = 'HTML_3.2_H armonia_1.0'/ > \backslash n < /peers >< /uiml >$ "
```

E.2.1.1 Function $uiml2str$

Specification:

```
    $uiml2str : UIMLSpec`Uiml \rightarrow UIMLSpec`String$
    $uiml2str\,(ui) \triangleq$
      $PI \curvearrowright$
      " $< uiml >$ " $\curvearrowright$
      $head2str\,(ui.head) \curvearrowright$
      $members2str\,(ui.members) \curvearrowright$
      $end\text{-}doc\text{-}html;$
```

Description:

Converts a UIML *element into String.*

Calls:

$head2str$,$members2str$

E.2.1.2 Function $interf2str$

Specification:

```
$interf2str : [UIMLSpec`Interface] \to UIMLSpec`String$
$interf2str\,(i) \triangleq$
  if $i$ = nil
  then ""
  else "< $interface$" $\frown$ $id2str\,(i.id)$ $\frown$
       $source2str\,(i.source)$ $\frown$
       $how2str\,(i.how)$ $\frown$
       $export2str\,(i.export)$ $\frown$
       " > " $\frown$
       $inteles2str\,(i.intele)$ $\frown$
       " < /$interface$ > ";
```

Description:

Converts a Interface element into String.

Calls:

Standard VDM-SL only

E.2.1.3 Function $member2str$

Specification:

```
$member2str : [UIMLSpec`Member] \to UIMLSpec`String$
$member2str\,(m) \triangleq$
  if $m =$ nil
  then ""
  else cases $m$ :
        mk-$UIMLSpec`Peers$ (-, -, -, -, -) $\to peer2str\,(m)$,
        mk-$UIMLSpec`Interface$ (-, -, -, -, -) $\to interf2str\,(m)$,
        mk-$UIMLSpec`Template$ (-, -) $\to templ2str\,(m)$,
        others $\to$ ""
      end;
```

Description:

Converts a Member type into String.

Calls:

$peer2str$,$interf2str$,$templ2str$

E.2.1.4 Function $members2str$

Specification:

```
$members2str : [UIMLSpec`Member^*] \to UIMLSpec`String$
$members2str\,(s) \triangleq$
  if $s = []$
  then ""
  else let $x =$ hd $(s)$ in
      $member2str\,(x) \curvearrowright members2str\,($tl $(s))$;
```

Description:

Converts a Member type sequence into String.

Calls:

$member2str$

E.2.1.5 Function $inteles2str$

Specification:

```
    $inteles2str : UIMLSpec`InterfaceElements^* \to UIMLSpec`String$
    $inteles2str\,(s) \triangleq$
      if $s = []$
      then ""
      else let $x =$ hd $(s)$ in
          cases $x$ :
            mk-$UIMLSpec`Structure$ (-, -, -, -, -) $\to stru2str\,(x)$ $\curvearrowright$
$inteles2str$ (tl $(s)$),
            mk-$UIMLSpec`Style$ (-, -, -, -, -) $\to style2str\,(x)$ $\curvearrowright$
$inteles2str$ (tl $(s)$),
            mk-$UIMLSpec`Content$ (-, -, -, -, -) $\to content2str\,(x)$ $\curvearrowright$
$inteles2str$ (tl $(s)$),
            mk-$UIMLSpec`Behavior$ (-, -, -, -, -) $\to behav2str\,(x)$ $\curvearrowright$
$inteles2str$ (tl $(s)$)
          end;
```

Description:

Converts an InterfaceElements set into String.

Calls:

$stru2str$,$style2str$,$content2str$,$behav2str$

E.2.1.6 Function $stru2str$

Specification:

```
    $stru2str : UIMLSpec`Structure \to UIMLSpec`String$
    $stru2str\,(s) \triangleq$
      " $<$ $structure$" $\curvearrowright$ $id2str\,(s.id)$ $\curvearrowright$ $source2str\,(s.source)$ $\curvearrowright$
$how2str\,(s.how) \curvearrowright export2str\,(s.export) \curvearrowright$ " $>$ " $\curvearrowright$
      $parts2str\,(s.parts) \curvearrowright$
      " $<$ /$structure$ $>$ ";
```

Description:

Converts a Structure element into String.

Calls:

$srcatt2str$,$parts2str$

E.2.1.7 Function $parts2str$

Specification:

```
$parts2str : UIMLSpec`Part^* \to UIMLSpec`String$
$parts2str\,(s) \triangleq$
  if $s = []$
  then ""
  else let $x =$ hd $(s)$ in
      $part2str\,(x) \curvearrowright parts2str\,($tl $(s));$
```

Description:

Converts a Part sequence into String.

Calls:

$part2str$

E.2.1.8 Function $part2str$

Specification:

```
$part2str : [UIMLSpec`Part] \to UIMLSpec`String$
$part2str\,(p) \triangleq$
  if $p =$ nil
  then ""
  else "$< part$" $\curvearrowright id2str\,(p.id) \curvearrowright$
      $source2str\,(p.source) \curvearrowright$
      $how2str\,(p.how) \curvearrowright$
      $export2str\,(p.export) \curvearrowright$
      $class2str\,(p.class) \curvearrowright$
      $where2str\,(p.where) \curvearrowright$
      $wherepart2str\,(p.where\text{-}part) \curvearrowright$
      "$>$" $\curvearrowright$
      $style2str\,(p.style) \curvearrowright$
      $content2str\,(p.content) \curvearrowright$
      $behav2str\,(p.behavior) \curvearrowright$
      $parts2str\,(p.parts) \curvearrowright$
      $repeats2str\,(p.repeats) \curvearrowright$
      "$< /part >$";
```

Description:

Converts a Part element into String.

Calls:

$srcatt2str$,$classs2str$,$style2str$,$content2str$,$behav2str$,$parts2str$

E.2.1.9 Function $toFileHTML$

Specification:

$$
\begin{array}{l}
toFileHTML : UIMLSpec`Uiml \times UIMLSpec`String \to \mathbb{B} \\
toFileHTML\,(ui, f) \triangleq \\
\quad IO`fecho\,(f, PI, \textsc{start}) \wedge \\
\quad IO`fecho\,(f, \texttt{"} <!DOCTYPEuimlPUBLIC\text{'-}//UIT//DTDUIML\texttt{"} \curvearrowright \\
\qquad\qquad\qquad \texttt{"}2.0Draft//EN\text{'}\backslash n\texttt{"}, \\
\qquad\qquad\qquad \textsc{append}) \wedge \\
\quad IO`fecho\,(f, \texttt{"}\text{'}UIML2_0g.dtd\text{'} > \backslash n\texttt{"}, \textsc{append}) \wedge \\
\quad IO`fecho\,(f, \texttt{"} < uiml > \texttt{"}, \textsc{append}) \wedge \\
\quad IO`fecho\,(f, head2str\,(ui.head), \textsc{append}) \wedge \\
\quad IO`fecho\,(f, members2str\,(ui.members), \textsc{append}) \wedge \\
\quad IO`fecho\,(f, end\text{-}doc\text{-}html, \textsc{append});
\end{array}
$$

Description:

Output to HTML file an uiml *element.*

Calls:

$head2str$,$members2str$

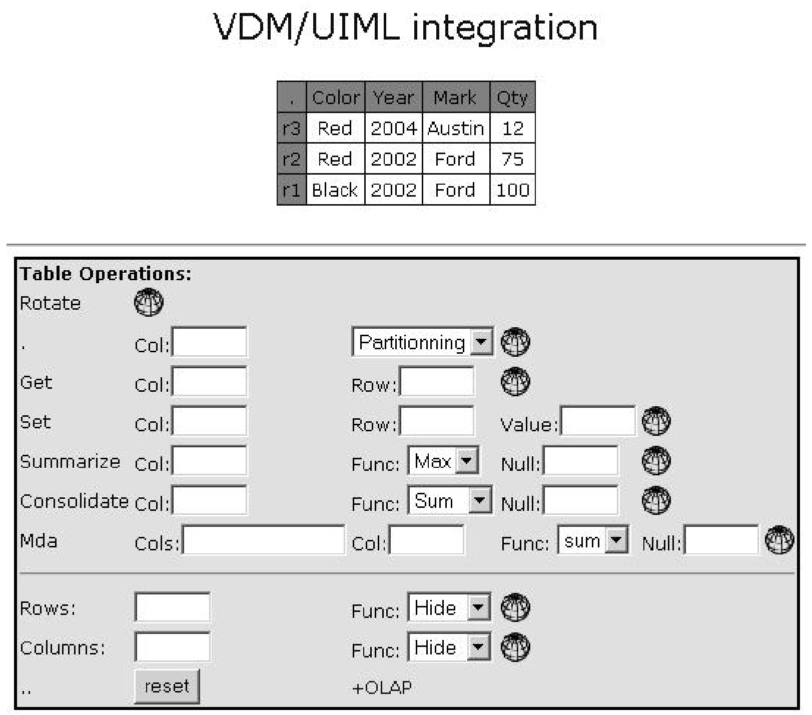
---

end *VDM2UIML*

Appendix F

Prototype

The following Figures depict some of the *OLAP* implemented features. The prototype was developed using a CGI mechanism, with PHP technology and Java Applets. The whole source code is on the technical report [Luí04].



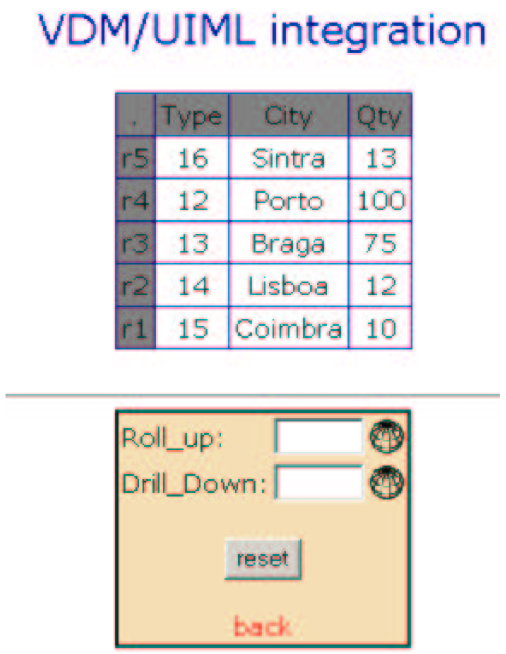
VDM/UIML integration

.	Color	Year	Mark	Qty
r3	Red	2004	Austin	12
r2	Red	2002	Ford	75
r1	Black	2002	Ford	100

Table Operations:

Rotate				
.	Col:	Partitionning		
Get	Col:	Row:		
Set	Col:	Row:	Value:	
Summarize	Col:	Func: Max	Null:	
Consolidate	Col:	Func: Sum	Null:	
Mda	Cols:	Col:	Func: sum	Null:
Rows:		Func: Hide		
Columns:		Func: Hide		
..	reset	+OLAP		

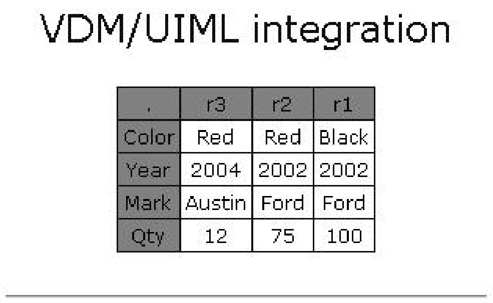
Figure F.1: *VDM/UIML integration test case*



VDM/UIML integration

.	Type	City	Qty
r5	16	Sintra	13
r4	12	Porto	100
r3	13	Braga	75
r2	14	Lisboa	12
r1	15	Coimbra	10

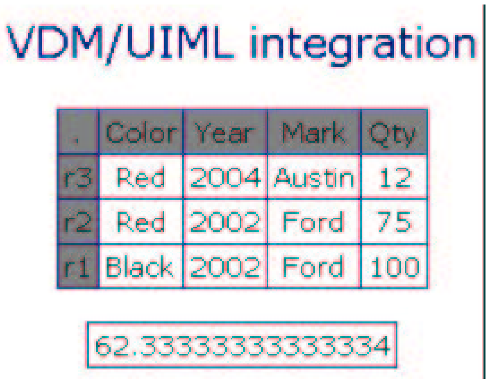
Figure F.2: *VDM/UIML integration test case - OLAP*



VDM/UIML integration

.	r3	r2	r1
Color	Red	Red	Black
Year	2004	2002	2002
Mark	Austin	Ford	Ford
Qty	12	75	100

Figure F.3: *Prototype: Rotation operation*

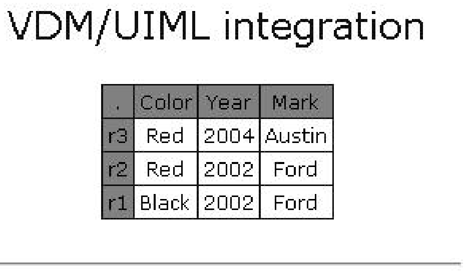


VDM/UIML integration

.	Color	Year	Mark	Qty
r3	Red	2004	Austin	12
r2	Red	2002	Ford	75
r1	Black	2002	Ford	100

62.33333333333334

Figure F.4: *Prototype: Consolidation operation*



VDM/UIML integration

.	Color	Year	Mark
r3	Red	2004	Austin
r2	Red	2002	Ford
r1	Black	2002	Ford

Figure F.5: *Prototype: Hiding column operation*

Appendix G

UIML Code examples

G.1 Stack *UIML* Code

Listing G.1 shows the *UIML* code to specify our *Stack* case study.

Listing G.1: *Complete* UIML Stack *user interface specification*

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC "-//Harmonia//DTD UIML 2.0
 Draft//EN" "UIML2_0g.dtd">

<uiml>

  <!--
    Test of Stack's on a JList:
    This uiml should be called from Stack.java,
    not started on its own.
  -->

  <head>
        <meta name="lufer" content="UIML Simple Stack"/>
        <meta name="Date" content="July, 2003"/>
        <meta name="Description" content= "This is an example
        of how to use the UIML to specify real problems."/>
  </head>

  <interface>

    <structure>
      <part id="Top" class="JFrame">
        <style>
                <property name="layout_hgap">10</property>
                <property name="layout_vgap">25</property>
        </style>
        <part id="Label" class="JLabel"/>
        <part id="ScrollPane"    class="JScrollPane">
          <part id="List"        class="JList"/>
        </part>
        <part id="ButtonPanel" class="JPanel">
          <part id="AddButton"     class="JButton"/>
          <part id="RemoveButton" class="JButton"/>
          <part id="TopButton" class="JButton"/>
          <part id="Clear" class="JButton"/>
        </part>
       </part>
    </structure>
    <style>
      <property part-name="Top" name="size">
        380,230
      </property>
      <property part-name="Top" name="location">
        100,100
      </property>
      <property part-name="Top" name="layout">
        javax.swing.BoxLayoutY
      </property>
      <property part-name="Top" name="title">
        Stack manipulation
      </property>


      <property part-name="ButtonPanel" name="layout">
        javax.swing.BoxLayoutX
      </property>
      <property part-name="ButtonPanel" name="alignmentX">
        LEFT_ALIGNMENT
      </property>

      <property part-name="AddButton" name="text">
        Push
      </property>
      <property part-name="RemoveButton" name="text">
        Pop
      </property>
      <property part-name="TopButton" name="text">
        Top
      </property>
      <property part-name="Clear" name="text">
        Clear
      </property>

      <property part-name="Label" name="text">
        Values in Stack
      </property>


      <property part-name="ScrollPane" name="alignmentX">
        LEFT_ALIGNMENT
      </property>

      <property part-name="List" name="selectionMode">
        SINGLE_SELECTION
      </property>
      <property part-name="List" name="content">
        <constant model="list">
          <constant id="1"     value="1"/>
          <constant id="2"     value="2"/>
          <constant id="3"     value="3"/>
          <constant id="4"     value="4"/>
        </constant>
      </property>

    </style>

    <behavior>

      <rule>
        <condition>
          <event part-name="AddButton" class="actionPerformed"/>
        </condition>
        <action>
          <call name="stack.addElement"/>
        </action>
      </rule>

      <rule>
        <condition>
          <event part-name="TopButton" class="actionPerformed"/>
        </condition>
        <action>
          <call name="stack.Top"/>
        </action>
      </rule>

      <rule>
        <condition>
          <event part-name="Clear" class="actionPerformed"/>
        </condition>
        <action>
          <call name="stack.Clear"/>
        </action>
      </rule>


      <rule>
        <condition>
          <event part-name="RemoveButton" class="actionPerformed"/>
        </condition>
        <action>
          <call name="stack.removeElement"/>
        </action>
      </rule>

           <rule>
        <condition>
          <event part-name="List" class="intervalAdded"/>
        </condition>
        <action>
          <property part-name="Label" name="text">
                An element was Pushed.
          </property>
        </action>
      </rule>

      <rule>
        <condition>
          <event part-name="List" class="intervalRemoved"/>
        </condition>
        <action>
          <property part-name="Label" name="text">
                An element was Poped.
          </property>
        </action>
      </rule>

      <rule>
        <condition>
          <event part-name="List" class="contentsChanged"/>
        </condition>
        <action>
          <property part-name="Label" name="text">
                Contents changed.
          </property>
        </action>
      </rule>

    </behavior>
  </interface>

  <peers>
    <logic>
        <d-component id="stack" maps-to="stack">
            <d-method id="removeElement"
                maps-to="removeTopElement"/>
            <d-method id="addElement" maps-to="Push"/>
            <d-method id="Top" maps-to="Top"/>
            <d-method id="Clear" maps-to="Clear"/>
        </d-component>
    </logic>

    <presentation base="Java_1.3_Harmonia_1.0"/>
  </peers>

</uiml>
```

G.2 Table *UIML* Code

Listing G.2: UIML *"template" for table definition*

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC "-//UIT//DTD UIML 2.0 Draft//EN"
    "UIML2_0g.dtd">
<uiml>
    <interface id="HTMLTableTest">
        <structure>
            <part id="top" class="Html">
              <part id="body" class="Body">
                <part id="table" class="Table">
                    <style>
                        <property name="background-color">yellow</property>
                        <property name="align">CENTER</property>
                    </style>
                    <part id="tr1" class="Tr">
                        <part class="Th">
                            <style>
                                <property name="content">ENGLISH</property>
                            </style>
                        </part>
                        <part class="Th">
                            <style>
                                <property name="content">SPANISH</property>
                            </style>
                        </part>
                        <part class="Th">
                            <style>
                                <property name="content">GERMAN</property>
                            </style>
                        </part>
                    </part>
                    <part id="tr2" class="Tr">
                        <part class="Td">
                            <style>
                                <property name="content">one</property>
                            </style>
                        </part>
                        <part class="Td">
                            <style>
                                <property name="content">uno</property>
                            </style>
                        </part>
                        <part class="Td">
                            <style>
                                <property name="content">eins</property>
                            </style>
                        </part>
                    </part>
                    <part id="tr3" class="Tr">
                        <part class="Td">
                            <style>
                                <property name="content">two</property>
                            </style>
                        </part>
                        <part class="Td">
                            <style>
                                <property name="content">dos</property>
                            </style>
                        </part>
                        <part class="Td">
                            <style>
                                <property name="content">zwei</property>
                            </style>
                        </part>
                    </part>
                    <part id="tr4" class="Tr">
                        <part class="Td">
                            <style>
                                <property name="content">three</property>
                            </style>
                        </part>
                        <part class="Td">
                            <style>
                                <property name="content">tres</property>
                            </style>
                        </part>
                        <part class="Td">
                            <style>
                                <property name="content">drei</property>
                            </style>
                        </part>
                    </part>
                    <part id="caption" class="Caption">
                        <style>
                            <property name="align">BOTTOM</property>
                            <property name="content">A Table of Numbers.</property>
                        </style>
                    </part>
                </part>
              </part>
            </part>
        </structure>
    </interface>
    <peers>
        <presentation how="replace" source="HTML_3.2_Harmonia_1.0.uiml#vocab"
         base="HTML_3.2_Harmonia_1.0"/>
    </peers>
</uiml>
```

Appendix H

Stack $VDM\text{-}SL$ Specification

This excerpt comes from [Oli02] and, as the author of the paper said, represents a purely functional specification of a Stack.

Data types

$$\begin{array}{l} \mathit{Stack} = A^*; \\ A = \text{token} \end{array}$$

Functions

H.0.0.10 Function $push$

Specification:

$$\begin{array}{l} \mathit{push} : A \times \mathit{Stack} \to \mathit{Stack} \\ \mathit{push}\,(a, s) \triangleq \\ \quad [a] \curvearrowright s; \end{array}$$

Description:

Add a new element to the Stack

Calls:

Standard VDM-SL only

H.0.0.11 Function pop

Specification:

```
$pop : Stack \to Stack$
$pop\,(s) \triangleq$
    tl $s$
pre $\neg\, empty\,(s)$
    ;
```

Description:

Remove the Top element of the Stack

Calls:

$empty$

H.0.0.12 Function top

Specification:

```
$top : Stack \to A$
$top\,(s) \triangleq$
    hd $s$
pre $\neg\, empty\,(s)$
    ;
```

Description:

Get the first element of the Stack

Calls:

$empty$

H.0.0.13 Function $empty$

Specification:

```
$empty : Stack \to \mathbb{B}$
$empty\,(s) \triangleq$
    $s = [\,];$
```

Description:

Test if the Stack is empty

Calls:

Standard VDM-SL only

Function/Method Cross-Reference Index

Glossary

Notation	Description	
AIO	Abstract Interaction Object	27
API	Application Program Interface	7
AST	Abstract Syntax Tree	201
AUIML	Abstract User Interface Markup Language	66
B2B	Business to Business	2
CBD	Component Based Software Development	42
CGI	Common Gateway Interface	38
CIO	Concrete Interaction Object	27
CMS	Content Management Systems	37
CSS	Cascading Style Sheet	14
DFD	Data Flow Diagrams	34
DIWG	W3C Device Independence Working Group	68
DMI	Direct Manipulation Interfaces	33
DOM	Document Object Model	56, 253
DSL	Domain Specific Language	10
DTD	Document Type Definition	250
EAI	Enterprise Application Integration	2, 33
FME	Formal Methods Europe	25
GPL	General-Purpose Language	10
GUI	Graphical User Interface	6
HCI	Human-Computer Interaction	2
IAI	Inter-Enterprise Application Integration	33
IDSS	Intelligent Decision Support System	17
IO	Interaction Objects	44
IT	Information Technology	41
JFC	Java Foundation Classes	37

Notation	Description	
MB-UIDE	Model-based User Interface Development Environments	34
MDOLAP	Multidimensional OLAP	50
MFC	Microsoft Foundation Classes	37
MIM	Meta-Interface Model	20
MSC	Message Sequence Charts	43
MVC	Model View Controller	17
NFR	Non Functional Requirements	23
OASIS	Organization for the Advancement of Structured Information Systems	53, 65
OLAP	Online Analytical Processing	44
OMG	Object Management Group	34
PAC	Presentation Abstraction Control	17
RDBMS	Relational Database Management Systems	1
RDF	Resouces Description Framework	58
ROLAP	Relacional OLAP	50
SAX	Simple API for XML Parsing	253
SDL	System Description Languages	42
STD	State Transaction Diagrams	34
UI	User Interface	32
UIML	User Interface Markup Language	60
UML	Unified Modelling Language	34
UMLi	Unified Modelling Language for Interactive Applications	34
VCL	Visual Component Library	2
VDM	The Vienna Development Method	8
VDM-SL	VDM Specification Language	10, 29
VoiceXml	Voice Extensible Markup Language	38
W3C	World Wide Web Consortium	53
XIML	eXtensible Interface Markup Language	54
XML	eXtended Markup Language	248
XML	eXtensible Markup Language	38
XPath	XML Path Language	254
XSL	Extensible Stylesheet Language Family	253
XSLFO	XML Formatting Objects	254

Notation	Description	
XSLT	XSL Transformations	254
XUL	XML-based User Interface Language	56