Modular Data Serialisation and Mobile Code

Ciarán O'Leary B.Sc.

A dissertation submitted to the University of Dublin in partial fulfilment of the requirements for the degree of Master of Science in Computer Science

September 2000

Declaration

I declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed: _____

Date: 15 September 2000

Permission to lend and/or copy

I agree that Trinity College Library may lend or copy this dissertation upon request.

Signed: _____

Date: 15 September 2000

Acknowledgements

I would like to thank sincerely my supervisor Dr. Simon Dobson for all his help over the past number of months. I would also like to thank Tim Walsh for his assistance when it was required.

Abstract

Since its creation the World Wide Web has revolutionised how we work. Academics have used it as a media to exchange research information, businesses use it to access markets previously unavailable to them and a growing percentage of society at large has been able to use it to access a global database of information. In short it has forced a rethink of the concepts of boundaries and limits.

The aim of the research outlined in this dissertation is to assess the possibility of stretching the functionality of the web a little further, by developing programming languages which exploit the availability of the web as a massive data store. The objective is to use web documents, traditionally HTML documents but in the future, XML documents, as a store of data values and types for programming languages. Values created by programming languages will be saved in available data space on the web. This will in turn lead to the increased interaction between programs on the web which will be able to exchange information in the form of a XML documents. Serialisation of programming language functions and sections of code as well as values will open the door to the development of mobile programming languages. Processing of code can take place wherever sufficient resources are available on the web, or can be moved to an area where the data required is more easily accessible.

Vanilla, a component based programming language development tool will be used to create prototypical programming languages that contain these capabilities.

Table of Contents

Table of Figures

1. Introduction

1.1 Motivation

Jon Bosak [12], one of the developers of the Extensible Markup Language (XML) argues that the revolution started by the combination of the global Internet and hypertext will be completed by the addition of XML. XML will add to the ability of the web to understand data and perform functions where a greater understanding of web content is required.

David Kotx [17] puts forward the same argument for mobility. He maintains that within five years “nearly all major Internet sites will be capable of hosting and willing to host some form of mobile agent”. Mobile code will be a critical near term part of the Internet. It makes new applications possible, improves significantly on the performance of traditional distributed processing techniques, but most importantly it will provide a single general framework to allow the simple and efficient implementation of distributed, information processing applications. This framework will facilitate the division of programming tasks among the various sites on the Internet which are willing to host processing of some sort.

What is required is a prototypical programming language which exploits the suitability of XML as a means for describing data, and which implements a certain degree of mobility in order to demonstrate the future uses of the web. Such uses will be demonstrated in this dissertation to include the addition of processing power to systems through mobility and the opening of file space to allow for sharing and storage of programming language values.

1.2 Proposed Solution

The aim of this dissertation is to produce prototypical programming languages that implement the required functionality to assess the possibility of using the World Wide Web as a store for programming language values and a host for code mobility and migration.

In doing so extensive use will be made of XML, Vanilla and current theory in the area of mobile code and programming language design. Analysis will be made of the various techniques being employed in the area of mobile code to see how the web could be utilised as a global environment for data processing. Also, if data can be serialised to a local disk then why not allow for its serialisation to any available space on the web. If this data is available elsewhere on the web then other programs will be able to access and understand it. If the same serialisation is implemented for programming language code then the processing burden incurred locally can be distributed among the available resources on the web.

XML adds to the contents of web documents. It replaces the *"dumb"* HTML that could only format the public view of the data, with a richer more powerful form of markup. XML insists on the specification of domain specific markup tags. It does so by providing a null set of previously defined tags. If a developer has the power to describe his/her data as (s)he chooses then the tags used to describe his/her data will have been enriched by the addition of semantics. Currently XML is being used in an ever-growing number of domains as a structured data description format. These domains range from Medicare (Health Level Seven http://www.hl7.org) to Real Estate (Real Estate Data Interchange Standard http://www.rets-wg.org/) to the News Industry (News Industry Text Format http://www.nitf.org/).

The use of XML for the purposes of this dissertation will be confined to defining the data content of programming language values and types. These values will then be written as web documents to available space on remote web servers. In effect web content will no longer be simply data for display to be read by humans, web documents will represent programming language values. Programming languages will be able to read the contents of an XML document and extract meaning from it, in much the same way as, say Java, can read a serialised Java object from disk and construct an in memory version of it.

Much has been said about programming languages in the preceding paragraphs. The programming languages of which we spoke will be developed using the Vanilla [1] tool. Vanilla is a completely component based

programming language builder. It separates the various parts of traditional compiler builders (parser, type checker, interpreter) and implements them independently for the different sections of a programming language (functions, objects, conditionals and so on). Currently, any language built using Vanilla will not have disk access. The aim of this dissertation is to add new components to the existing set of Vanilla components in order to provide serialisation functionality.

Serialisation of programming language values will be the primary focus of the dissertation. However, the additional aim of integrating mobility into Vanilla can be provided by exploiting the fact that Vanilla functions are treated as if they are values. If we can successfully serialise programming language functions to the web then will we have ways of making it run. If so, will it be able to run with the same resources it had on its original host, how will it keep its environment consistent with where it was originally located and how will it communicate with its parent to return values.

1.3 Objectives

For clarity the individual objectives of this dissertation can be specified as follows:

1. The development of some standard mechanism for the representation of Vanilla types and values as well as functions as XML documents. At this stage the aim is simply to be able to take in-memory values and display how they may be represented as an XML document.
2. The extension of the current Vanilla framework to add disk access functionality. Initially all that will be allowed for will be the serialisation of simple text strings to the local disk. Following the implementation of this, however, the basic `write(filename, String)` will be extended to allow for functions such as `write(filename, Value)`. The data will be serialised to the disk as an XML web document.
3. The remote serialisation of values. This refers to the ability to write the values as XML documents to file space that is not located on the

machine or possibly not even on the network on which the program that generated the value is running. When this has been successfully completed possible applications for this functionality will be looked at. For example Vanilla based programs could interact over the web by exchanging values as XML documents.

4. Ultimately the ability to serialise code and execute it remotely ought to be implemented. This will lead to the area of code mobility. A number of things need to be examined at this stage. For instance the benefits and drawbacks of weak versus strong migration of code between hosts will be analysed to see what is required to add sufficient power to the programming language.

1.4 Roadmap

The next chapter of this dissertation will be present background information on the various areas explored in the project. Subsequent chapters will outline the design, implementation and evaluation of the system. Here is a clearer outline of the remaining chapters.

- 2. Background
    - In this chapter the following technologies and research areas will be examined in detail
        1. Vanilla and Programming Languages
        2. XML
        3. Serialisation and Communication Mechanisms
        4. Code Mobility
- 3. Design
    - The design of the system. What is required, what approach will be taken, what will the architecture appear like from a high level and so on.
- 4. Implementation

    - What choices were made on how to implement the system, what issues were encountered when implementing it and how was it implemented?
- 5. Evaluation
    - What results did testing of the system produce, what are the possible uses of the system and how do these function at present? What is required to make the system better or more adaptable to other potential uses?
- 6. Conclusion
    - What has the research outlined in this dissertation produced? What future research is possible?
- 7. References
    - A list of all sources

1.5 Summary

It is the aim of this dissertation to produce an extension to the current set of components that comprise the Vanilla system in order to add functionality to allow for access to the disk on local and remote machines. How values that are stored within Vanilla can be represented using the Extensible Markup Language will be assessed. Ultimately it is hoped that a system will be produced to permit languages built using Vanilla to incorporate both local and remote data serialisation as well as code mobility, using the functionality that will have been built as part of the project.

2. Background

2.1 Introduction

In this section the various technologies that will be used in this project will be examined in details as well as what other work is being done in the various research areas relating to these technologies and how these technologies function.
The main sections with be as follows:

- Vanilla
  - In this section on overview of programming language theory will be presented before a more detailed discussion of the Vanilla system, the technology that is core to the research in this dissertation.
- XML
  - The Extensible Markup Language is a subset of the Standard Generalised Markup Language which provides a structure for data so it can be represented in a standard machine understandable manner.
- Serialisation & Communication Mechanisms
  - This section will contain a detailed discussion of two technologies, SOAP and Jiki, that may be used as the basis of the communication mechanism in this project.
- Code Mobility
  - What is code mobility, what different types of mobility exist and what implementations are available. All these questions will be addressed and answered.

2.2 Vanilla & Programming Languages

Computer programs are created using programming language compilers. Programming language compilers are built using compiler compilers. Very often interpreters will be used in place of compilers so that the output of a

program can be seen from processing the source code, rather than machine code. Vanilla is a tool that will build an interpreter for a programming language.

The next section will give a general introduction to compiler design, the lessons from which can just as easily be applied to interpreters. The next section following this will give a complete overview of the Vanilla tool.

2.2.1 Programming Languages

2.2.1.1 Overview

A compiler is a program that translates an program in one language into an executable program in another language. The output produced by the compiler should be better in some way than the original input. For example if some Java code is processed by a Java compiler then the output, Java byte code, will be in a format that is directly understandable by the Java Virtual Machine.

An interpreter is a program that reads an executable program and produces the results of running that program. Typically this will involve executing the source code in some fashion. For example if some Java code is presented to a Java interpreter then the output of running that code will be presented immediately, without the intermediate step of compilation into byte code. Languages such as Perl are usually interpreted rather than compiled into machine code of some sort.

2.2.1.2 Compiler Design

Figure 2.1 below gives a view of the general architecture of a well-designed compiler [3]. A front end reads in the source code and produces some kind of intermediate representation (IR), probably in the form of an Abstract Syntax Tree. The back end will then read the IR and produce the correct machine code for the target machine. Along the way any errors that are encountered will be thrown.

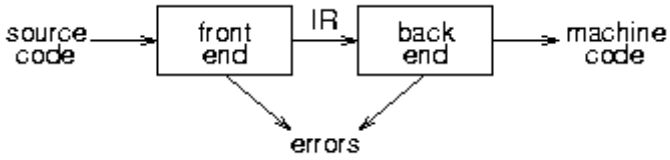


Figure 2.1: Standard architecture for a compiler from [3]

The structure of the front end (Figure 2.2) will be similar for both compilers and interpreters. The source code is scanned and parsed to check for syntactic correctness and to construct the intermediate representation that will be used at the back end, throwing any errors encountered along the way.

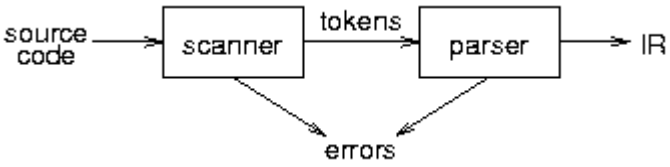


Figure 2.2: Architecture of front end of a compiler from [3]

The processing that takes place for a small piece of code, such as

x + 2 – y

can result in a large parse tree, as seen in Figure 2.3(a), so most compilers prefer to use an Abstract Syntax Tree to represent the piece of code.

Abstract syntax makes a clean interface between the parser and the later phases of the compiler. The Abstract Syntax Tree conveys the phase structure of the source program with all parsing issues resolved but without any semantic interpretation. This tree can be presented to the type checker that can traverse it resolving further issues. In essence, however, it represents a simple intermediate representation for passing a program between the front end of a compiler and the back end.

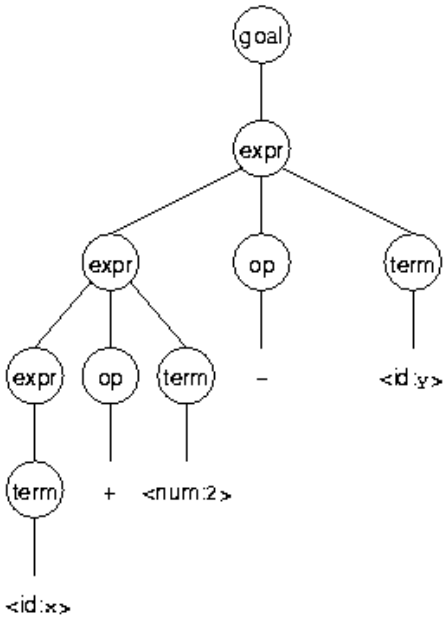


Figure 2.3(a): Example Parse Tree

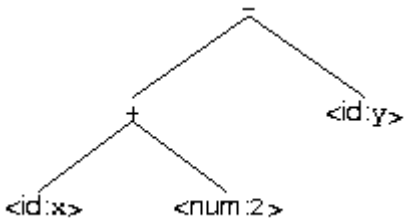


Figure 2.3(b): Example Abstract Syntax Tree

The type checker will usually operate as a recursive function of the Abstract Syntax Tree. Its function is to use entries in Symbol Tables to map symbols to bindings. The main aim is to determine type equivalence, to ensure that types, values and objects are only used where it is appropriate to do so. One of the following approaches is taken when checking type equivalence.

- In *name equivalence*, two types are equivalent if they are named and have the same name.
- In *structural equivalence*, two types are equivalent if they have the same structure.
- In *structural equivalence under naming*, two types are equivalent if they have the same structure and the named components of each structure have the same names.

The back end will relate the intermediate representation onto the environment where the program will be run. If the program is to be interpreted at the back end the output of the program will be generated and displayed. If a compiled

machine code version of the program is required then the necessary code will be generated for the relevant environment.

The three phases of a compiler can therefore be seen as

1. Parser
   - Read in source code and construct Abstract Syntax Tree
2. Type Checker
   - Check the types of the nodes in the Abstract Syntax Tree in order to ensure that the left and right hand sides of expressions evaluate to compatible types, and so on.
3. Interpreter
   - Produce output, whether this is compiled executable code, or simply the output of running the code.

2.2.2 Vanilla

2.2.2.1 Overview

The Jakarta Tool Suite (JTS) [2] is a set of domain independent tools for creating domain specific languages. A language is constructed using JTS by extending an existing programming language, called a *host language.* The benefits of using this suite of tools for language creation are three fold.

1. Common programming constructs need not be re-implemented. The functionality of the host language is re-used in the new language.
2. The extensions of the host language need only be transformed to a point where they can be expressible in the host language
3. The existing infrastructure can be used.

This method of programming encourages the re-use of tried and tested code, and as a result reduces significantly on the time for development of a new language.

The thinking behind the development of the JTS is much akin to the inspiration for the Vanilla language [1]. Vanilla defines a set of components that are used together to define a language. A set of completely independent

components provides all the functionality (parser, type checker, interpreter) for an individual part of a programming language, say loops. Such components are provided for all the basic parts of a language. To alter or extend the language to create a new language only a few components will need to be changed.

This type of programming is a part of the growing area of component based software design. Standard components which provide a certain amount of functionality and that functionality alone can be integrated with other components with confidence that their independent behaviours will not be tied to any other components except for how is specified by the programmer. In this way systems such as JTS and Vanilla can be used to create new languages through the re-use of existing components integrated with a minimal set of newly created components.

2.2.2.2 Vanilla Architecture

Vanilla provides an overall framework (Figure 2.4) into which components for the language can be plugged. It provides a number of components, or subsystems, which mirror the overall architecture of a well-designed interpreter, i.e. parser, type checker, interpreter. Each of these subsystems is implemented as a set of components.

Each individual part of a programming language is implemented as a Vanilla pod. A Vanilla pod will implement a parser, type checker and interpreter for that pod alone. When loaded the different components register themselves with the root component of the subsystem. All pods are loaded at run time. The registry that is constructed for each subsystem is queried each time a parser (or type checker / interpreter / any other subsystem component) is required for an individual piece of code. The registry will then find the appropriate component to perform the operation required.

Vanilla allows the programmer construct simple prototype languages by making it easy to remove components and add in other components in its place. A language can be added to as required, and changed when necessary. If a language is required to handle a number of features that are not included in the basic Vanilla package a developer need only write the

parser, type checker and interpreter for that feature and include the names of those components in the language definition file. In order to change a subsystem, or to include a new one (for say, serialisation), a set of components must be created and included as a new subsystem in the Vanilla properties file.

Since the language is loaded as well as he program at run time a large set of potential languages can co-exist together, with the language definition file for each language specifying what components are required for each language.

2.2.2.3 Vanilla Pods

A Vanilla pod is a set of components that implement a single language feature and only that feature. Separate pods exist for each of the following:

Core Variables Sequences Conditionals Loops	: : : : :	Ground types, arithmetic, sequence Put & get from environment Arrays If-then-else For (…)	Very Common
Functions Universals Existentials µ-recursive types	: : : :	Higher order closures Like C++ Templates Partially abstract types Types containing themselves	The basis of functional programming
Objects Classes	: :	Abadi & Cardelli Style	Very general object model

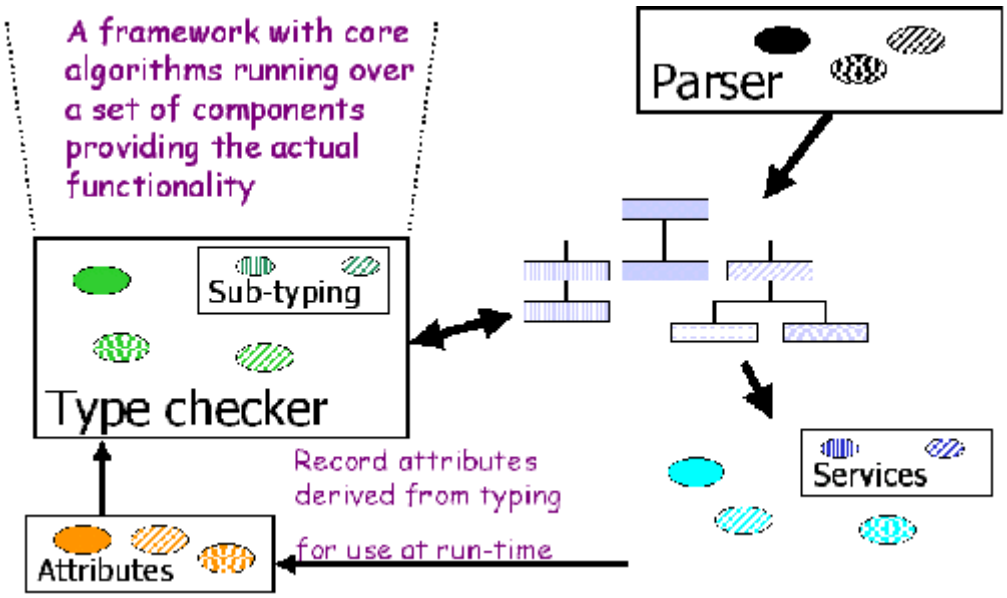


Figure 2.4: Vanilla Architecture from [1]

2.2.2.4 Vanilla Types & Values

All Vanilla `Values` have an associated `Type`. `Type` and `Value` components are provided for all the basic types within a programming language i.e. `Boolean`, `Integer`, `String`, `Record`, `Object`. All `Type` classes extend the root `Type` class and likewise all Vanilla `Values` extend the root `IValue` class. In order to create new `Types` or `Values` these root classes must be extended.

In Vanilla a `Function` is also treated as a `Value`. This was a decision that was made when designing the basic functionality that is provided with Vanilla, but could be easily altered thanks to the decoupled component nature of Vanilla. The research outlined in this dissertation takes advantage of the fact that functions are treated as values.

A `Value` must have an associated `Type`. `Types` may also have a `Kind` associated with them, where the relationship from `Value` to `Type` is similar to the relationship from `Type` to `Kind`. We do not deal with `Kinds` to a sufficiently large degree in the course of this dissertation to necessitate a discussion of them in any great detail.

2.2.2.5 Running Vanilla

Vanilla can be downloaded for free from www.vanilla.ie. The current version is Vanilla 1.6 but the version that will be released following the successful completion of this dissertation will be version 1.7 and will include all the serialisation and mobile code capabilities that will be developed as part of this project.

In order to run Vanilla all the compiled classes must be downloaded and installed and the classpath must be set appropriately. Extra components can then be added or amended as required.

There are a number of tests provided which can then be used to test the installation.

1. `vc`
   - The batch-mode shell. Most commonly used
2. `avc`
   - A more intelligent batch-mode shell
3. `vint`
   - The interactive shell
4. `vprint`
   - The AST print utility
5. `vp`
   - The parser component generator
6. `vpod`
   - The pod compiler
7. `vic`
   - The interpreter compiler

In order to run Vanilla using one of these tools on a file such as `test.v`, run the following command

```
vc test.v
```

2.2.2.6 Summary

In brief, Vanilla is a tool that provides a language designer with a means to *"throw together"* a language in a short time. It saves the designer from the burden of having to implement functionality that is common to most languages, allowing him/her to focus solely on the central part of the language, or the part that they wish to implement differently. Vanilla obeys all rules for language design, and in this respect implements languages in the same way as is done by more conventional systems, i.e. it contains parse, type check, interpret phases, it implements Types, Kinds and Values.
What it does differently (i.e. component based, plug in architecture) is what makes it useful for developing prototypical languages. Because it is mainly supposed to be used for this theoretical work its designers concentrated on getting everything to work correctly, rather than optimising it in terms of speed or performance. As would be expected from a system that requires a large number of components to be loaded at run time, its performance can be quite slow in comparison to conventional systems.

2.3 XML

2.3.1 Overview

Documents containing any sort of meaningful information must have some structure if the content is to be understood by a machine. The structured information should contain both content and some indication of what role that content plays. For machines to understand the content they must be presented with a description of the overall structure of information within the document. A markup language is a mechanism to identify structures in a document.
The Extensible Markup Language was specified in February 1998 by the World Wide Web Consortium [4]. It was proposed as a standard means for the structuring of information and since its specification it has literally spread like wild fire through all industries. XML has been defined as an application friendly cut down version of the Standard Generalised Markup Language previously defined in ISO 8879. SGML is capable of providing the same

functions as XML, but its complexity does not lend itself nicely to simpler tasks, such as representation of web data. XML, as a restricted form of SGML, can easily accomplish this task and much more.

The magic of XML lies in the freedom that is given to the programmer in describing the overall structure and semantics of his/her data within a single document. Although it is a markup language just like HTML it does not specify any tags. Thus any person, or more typically, any industry can specify a set of tags and an overall XML document structure that should be used to represent the information that is used within the industry.

Originally XML was just seen as a natural successor to the afore-mentioned Hyper Text Markup Language (HTML). Indeed it is probably the case that XML in tandem with its sibling XSL will become the standard method for displaying data on the web, but XML is far too rich to be constrained to that use alone.

There are a number of “extras” that come along with XML to add to its power.

1. XSL – The Extensible Stylesheet Language
2. XLink – The XML Linking Language
3. XPointer – The XML Pointer Language

As well as these, the mass of research taking place in this area has also produced numerous APIs for the processing of XML documents in different ways, as well as a wealth of different environments in which XML can be developed.

XML is already becoming a standard for information representation. It will ultimately be the standard means for creating web documents. It will be discussed in detail in the following sections, concentrating mainly but not exclusively on the parts of it that will be used in the course of this dissertation project.

2.3.2 The Extensible Markup Language

2.3.2.1 Document

All XML data is contained within an XML document. An example of a simple XML document (that seemingly has become the “Hello World” standard example in XML tutorials) is given below in Figure 2.5. Note that all XML examples presented below are either from the XML tutorial on XML.com [11] or from the myriad of papers written on XML and available off the World Wide Web Consortium web site [4 – 10]

```
<?xml version="1.0"?>

   <oldjoke>

   <burns>
      Say
      <quote>goodnight</quote>
      ,Gracie.
   </burns>

   <allen>
      <quote>Goodnight, Gracie.</quote>
   </allen>

   <applause/>

</oldjoke>
```

Figure 2.5: A Simple XML Document

All XML documents must begin with the code shown in the first line of Figure 2.5 above. This, quite obviously, is an indication of the version of XML being used in the document. Documents must also be *well formed*, meaning start tags must have corresponding end tags and must not overlap and must be *valid* if a structure is given for the document. This will be explained in more detail below.

2.3.2.2 Markup

An XML document is comprised of both markup and content. There are six different types of markup in XML. They are Elements, Attributes, Entity References, Comments, Processing Instructions and CDATA sections.

Most important for the purposes of this dissertation are Elements and Attributes. These represent the structure of the document and the values that may optionally be associated with them. An example is given below in Figure 2.6.

```
<elementname attribute1=something>
  Blah blah blah
</elementname>
```

Figure 2.6: Common XML markup

2.3.3 Document Type Declarations

As alluded to earlier, one of the great strengths of XML is that it allows the programmer create his/her own tags, thus giving him/her the power to add semantics to their documents by specifying tags that are meaningful to the application in question.

A Document Type Declaration (DTD) describes the Meta data. It gives the structure of the document, saying which tags occur at which point in the document. Documents need not necessarily conform to a DTD but if it does it is said to be *valid*. If present in a document, the DTD must be the first thing in the document. If not located within the document, there may be a reference to an external DTD or there may be both internal and external DTDs associated with a document.

The DTD's main function is in communicating Meta information to the parser to give the machine some "understanding" of what is contained within the document.

An example DTD is given overleaf in Figure 2.7.

```
<!ELEMENT oldjoke  (burns+, allen, applause?)>
<!ELEMENT burns    (#PCDATA | quote)*>
<!ELEMENT allen    (#PCDATA | quote)*>
<!ELEMENT quote    (#PCDATA)*>
<!ELEMENT applause EMPTY>


<!ATTLIST oldjoke name ID label CDATA
   status ( funny | notfunny ) 'funny'>
```

Figure 2.7: Sample DTD Entries

2.3.4 XML Schema

Document Type Declarations are typically used to define the structure of an XML document. However DTDs often lack sufficient power. XML Schema attempt to provide an alternative method for describing data that does not lack the power that DTDs do. An advantage of Schema over DTDs is that they are expressed entirely in XML and do no t require a knowledge of a different syntax (Extended Baukus Naur Form for DTDs) to be constructed.

The purpose of XML Schema, according to its specification [8] is to define a class of XML documents. Schemas have many different parts and are probably best understood through the use of an example. Such an example exists in the W3C spec [8], but is far too complex to deal with here. Suffice it to say that XML schemas appear similar to a DTD expressed in XML. Obviously the power to express data definitions using your own tags can add to the power of the definition. There are a number of built in types and constants available to the document writer. The use of a small subset of these should be apparent from the example piece of code in Figure 2.8.

```
<xsd:complexType name="Address" >
    <xsd:element name="name"   type="xsd:string" />
    <xsd:element name="street" type="xsd:string" />
    <xsd:element name="city"   type="xsd:string" />
    <xsd:element name="state"  type="xsd:string" />
    <xsd:element name="zip"    type="xsd:decimal" />
    <xsd:attribute name="country" type="xsd:NMTOKEN"
                  use="fixed" value="US"/>
</xsd:complexType>
```

Figure 2.8: Use of Simple types and constraints in XML Schema

2.3.5 XSL

The Extensible Stylesheet Language [9] specifies the presentation of a class of XML documents as web documents, by describing how a document in the class can be transformed into a document that conforms to the formatting rules for web presentation. For example, some mechanism may be given to specify how the XML content can be formatted in HTML.

2.3.6 XLink

The XML Linking Language [6] allows elements to be inserted into XML documents in order to create and describe links between resources. A link is a relationship between two resources, made explicit by an XLink linking element. XLink allows XML documents to

- Assert linking relationships among two or more resources.
- Associate Meta data with a link.
- Create link databases that reside in a location separate from the linked resources.

XLink is used to reference documents at URLs in much the same way as pointer variables reference values in a programming language. This may be of relevance to the implementation of this project.

2.3.7 XPointer

The XML Pointer Language [7] is a language that specifies the means for addressing into the internal structures of an XML document. In particular it provides for specific reference to elements, character strings and other parts of XML documents. In short XPointer provides a means for locating elements and other resources within a document, relative to the root, or any other location within the document. The potential use for this would lie in the ability to navigate through documents pulling information from various sections, where a document contains say, two sets of information, possibly type and value.

2.3.8 XML Namespaces

Namespaces in XML [10] provide a simple method for qualifying element and attribute names used in XML documents by associating them with namespaces associated with URI references. According to its specification [10] an XML namespace is "*a collection of names, identified by a URI reference, which are used in XML documents as element types and attribute names*".

An XML namespace is declared using a family of reserved attributes. Such an attribute's name must have either `xmlns:` or `xmlns` as a prefix. An example namespace declaration which associates the namespace `tcd` with the namespace name `http://www.tcd.ie/` is given in Figure 2.9 below.

```
<x xmlns:tcd='http://www.tcd.ie'>
     <!—the "tcd" prefix is bound to http://www.tcd.ie/
     for the x element and contents -->
</x>
```

Figure 2.9: Namespace Declaration for TCD

Once a namespace has been declared then some names in an XML document may be given qualified names. A qualified name begins with the namespace name, followed by a colon, and terminates with the name given to

the element. Consider for example the declaration of `people` as a qualified name for `tcd`

```
<x xmlns:tcd='http://www.tcd.ie'>
     <!—the 'people' element's namespace is
     http://www.tcd.ie/ -->
     <tcd:people type='Staff'>1000</tcd:people>
</x>
```

Figure 2.10: Namespace Usage for TCD

When all sorts of XML Schema are declared then there are likely to be conflicts, where size, for instance, means one thing in one industry and something completely different in another industry. The use of namespaces is aimed at preventing the likely confusion, by qualifying the names according to how they are being used and who is using them.

2.3.9 Parsing XML

Since one of the primary functions of XML is the exchange of data between computer programs, the necessity for a mechanism to parse and hence, understand, XML is paramount. To do so, Sun provide two APIs for parsing XML documents. These APIs represent different approaches to processing XML, and as such are better suited to different types of applications. They closely related to each other, as will be seen in Figure 2.11.

The "Simple API" for XML (SAX) is the event-driven, serial-access mechanism that does element-by-element processing. The API for this mode reads and writes XML to a data repository or the Web. This mode is best suited to server-side and high-performance applications.

The DOM API is also an easier API to use. It hides the intricacies of the SAX API, and provides a relatively familiar tree structure of objects. It also provides a framework to help output the object tree as XML data. On the other hand, constructing the DOM requires reading the entire XML structure and holding the object tree in memory, so it is much more CPU and memory intensive. For

that reason, the SAX API will tend to be preferred for server-side applications and data filters that do not require an in-memory representation of the data. The DOM API builds on the SAX API as shown here in Figure 2.11.

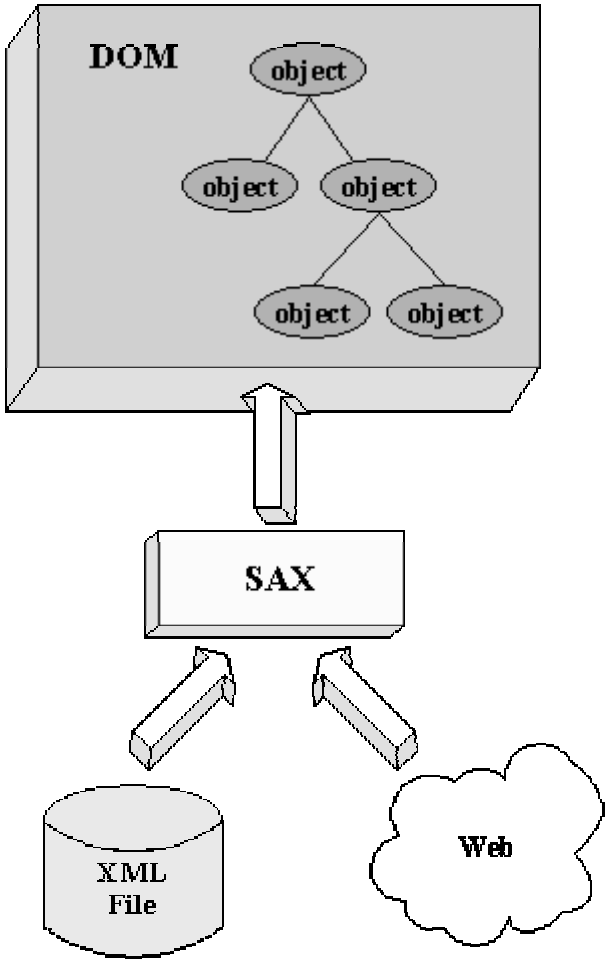


Figure 2.11: DOM and SAX

The essential difference between the two lies in the fact that the DOM [5] produces an actual object for each part of the contents of the XML document. This makes it far easier to use for the programmer. It allows for easy navigation of the tree structure of the document and easy processing of the objects at each stage. However, concerns about efficiency may make the SAX parser more favourable where performance is a large factor.

There are a number of different implementations of the two XML parsers available, including the Apache Xerces parser. This wraps up the Sun implementation with some extra functionality.

2.3.10 Summary

XML has been seen to take its place where it belongs, everywhere. It is destined to become a standard wherever interchange of any sort of structured information is required. This section has presented an outline of how XML documents are constructed, linked, traversed, described and parsed.

2.4 Serialisation & Communication Mechanisms

2.4.1 Overview

The core of this dissertation involves the writing, or serialising of data to disk. This will involve the specification of some mechanism for representing the data, and the development and incorporation of some algorithm that will ensure that the correct data is being collected, to allow for the rebuilding of the data when read from disk.

The Java serialisation mechanism is first looked at to see how Java approaches the issue of serialisation. It is shown why this is insufficient for what is required here.

However, the serialisation in this project takes place in the form of XML documents. In order to look at how data might be represented in XML an examination of the Simple Object Access Protocol (SOAP) is made. This protocol involves the encoding of requests and responses in XML for communication in a distributed programming environment.

Since the data being written will not simply be written out locally, but may also be done remotely ways of communicating between servers will also need to be examined. To see this we will look at web servers and Java Servlets, as well as the Jiki architecture. Jiki is a prototypical web server currently being developed further, which provides fully open access to any web client.

2.4.2 Java Serialisation

Sun Microsystems [23] describes the ability to store and retrieve Java objects as being essential to building all but the simplest applications. The key to storing and retrieving objects in a serialised form is representing the state of objects sufficient to reconstruct the object(s). Objects to be saved in the stream may support either the `Serializable` or the `Externalizable` interface. For Java objects, the serialised form must be able to identify and verify the Java class from which the contents of the object were saved and to restore the contents to a new instance. For serialisable objects, the stream includes sufficient information to restore the fields in the stream to a compatible version of the class. For `Externalisable` objects, the class is solely responsible for the external format of its contents. Objects to be stored and retrieved frequently refer to other objects. Those other objects must be stored and retrieved at the same time to maintain the relationships between the objects. When an object is stored, all of the objects that are reachable from that object are stored as well. The goals for serialising Java objects according to Sun Microsystems [23] given here.

- Have a simple yet extensible mechanism.
- Maintain the Java object type and safety properties in the serialized form.
- Be extensible to support marshaling and unmarshaling as needed for remote objects.
- Be extensible to support simple persistence of Java objects.
- Require per class implementation only for customisation.
- Allow the object to define its external format.

The Java serialisation mechanism is insufficient for a number of reasons. The serialisation of objects by one version of Java may not be understood by subsequent implementations of Java. XML can combat this problem by having sets of DTDs describing data.

2.4.3 SOAP

The Simple Object Access Protocol [13] is a minimal set of conventions for invoking code using XML over HTTP. It was submitted to the IETF as an Internet Draft in December 1999, by DevelopMentor, Microsoft and UserLand. It is an evolution of XML-RPC, which was the first result of the concept of calling remote methods over HTTP using XML.

SOAP works by encoding a method name and its associated parameters in an XML document and sending it over HTTP to the required destination. The amount of information used is minute compared to the requirement of encoding complex values in XML, but for primitive data types it is reasonably closely related.

Figure 2.12(a) below shows a SOAP request. Things to note here are the use of namespaces, the envelope notion for storing the information and the way the method name is encoded in XML.

Since SOAP is a request response based approach, a response must follow a request. The response to the request in Figure 2.12(a) is given below in Figure 2.12(b).

```
<SOAP:Envelope xmlns:SOAP="urn:schemas-xmlsoap-
org:soap.v1">
     <SOAP:Body>
          <m:GetLastTradePrice xmlns:m="Some-Namespace-
          URI">
               <symbol>DIS</symbol>
          </m:GetLastTradePrice>
     </SOAP:Body>
</SOAP:Envelope>
```

Figure 2.12(a): SOAP Request

```
<SOAP:Envelope xmlns:SOAP="urn:schemas-xmlsoap-
org:soap.v1">
     <SOAP:Body>
          <m:GetLastTradePriceResponse xmlns:m="Some-
          Namespace-URI">
               <return>34.5</return>
          </m:GetLastTradePriceResponse>
     </SOAP:Body>
</SOAP:Envelope>
```

Figure 2.12(b): SOAP Response

SOAP operates entirely over HTTP. The chief advantage of this is that web servers whose firewalls permit communication only on port 80 will not stop SOAP data from being communicated. Other protocols such as the IIOP, which work on other ports, can be stopped by suspicious firewalls.
In HTTP headers SOAP uses the Content-Type of "`text/xml`". This is used to specify the body of the HTTP request containing a XML encoded method call. To disambiguate the headers it adds to HTTP, SOAP permits use of the HTTP Extension Framework specification. The extension framework allows use of the HTTP verb `M-POST`. An example HTTP header for the request in Figure 2.12(a) is given below in Figure 2.13.

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml
Content-Length: nnnn
SOAPMethodName: Some-Namespace-URI#GetLastTradePrice
```

Figure 2.13: HTTP Header for SOAP Request

2.4.4 Jiki

Jiki [14] is an open web server architecture that provides access to all documents stored in its file space to any web client. Clients accessing a document stored on a Jiki web server are presented with the opportunity to edit the contents of the page. No security has so far been implemented so anyone who wishes to do so may edit any page on a Jiki web server. This concept of openness is something that could be exploited for the purposes of this dissertation, where a mechanism for writing from one machine to another is required. Although Jiki may not be ideal a lot can be learned from how it is implemented.

1. It is a system written entirely using Java Servlets.
2. It uses simple HTTP POSTs to input all its values, thus avoiding the need for HTTP extensions
3. It keeps everything nice and simple

2.4.5 Summary

Each of Java serialisation, SOAP and Jiki have been introduced to attempt to explain some mechanisms for representing data and sending this data between machines. Java serialisation is a data representation format, SOAP is simply a communications protocol and Jiki is a web server architecture.

2.5 Code Mobility

2.5.1 Overview

The creator of the TCL scripting language, John Ousterhout, once characterised code mobility and mobile agents as a “solution in search of a problem”. Indeed it is true that existing mechanisms for distributed computing are sufficient to do the job that can be performed by mobility. However, in much the same way as CD technology replaced Audio cassettes, which could doubtlessly do the job they were required to do, mobility can provide an

alternative to traditional RPC mechanisms by proving to be a better way to perform the same task.

Code mobility refers to the ability to move code around a network exploiting available resources at the various nodes to provide a quicker, more efficient result, or often, just to enable the computation of some meaningful result. There are three main approaches taken in code mobility, which will be elaborated upon. In brief, however, the main methods of making code mobile are as follows [18].

1. The *Remote Evaluation* Approach
2. The *Mobile Agent* Approach
3. The *Code on Demand* Approach

The three approaches differ in the way the code is moved, how the result is achieved and who performs the computation.

When code is moved, or migrated, the level of detail that accompanies it to its destination determines whether the type of migration is *weak* or *strong*. Weak migration involves the transfer of the code alone. Strong migration requires that all information about state accompany the code. Intermediate levels of migration strength may also be possible, as discussed by Walsh [15].

The next few sections will outline how code mobility with an examination of how mobility contrasts with traditional distributed computing mechanisms such as CORBA, what are the various approaches taken and what systems currently exist.

2.5.2 Mobile Code Systems

Figure 2.14(a) [19] below gives an outline of a traditional distributed system. These types of systems typically employ some sort of Virtual Machine to give the impression of a single machine, rather than a whole network of connected resources. From the diagram it is obvious that the network and lower layers are all made transparent through the *True Distributed System* layer. Any time a service is called the caller will not know of which node on the network is responsible for the provision of the service, but will only know that the service is available in the system. In CORBA systems such calls take place through

an object broker who knows of who provides what services. All that is required of anyone who requires the services is that they know how to locate the object broker.

Fuggetta [19] gives a clear explanation of the differences between this sort of True Distributed System and what he terms a Mobile Code System. Figure 2.14(b) shows the architecture of a Mobile Code System (MCS). The top layer in the True Distributed System is replaced by a set of Computational Environments located at each node. Therefore the underlying computer network is not hidden from the programmer; rather the programmer is given the power to relocate the processing of an application to any of the nodes.

A *Computational Environment (CE)* is an environment on a machine in a network that can host the execution of a unit of code, or *Executing Unit (EU)*. An executing unit is a sequential flow of execution, for example a single threaded process. An executing unit will have a number of components. Its Code Segment will provide a static description of the execution of the EU. Its Data Space represents the set of resources that can be accessed by it. Its Execution State represents the private data that cannot be shared, as well as control information related to the EU's state. CEs can also host what are called *Resources*. Resources represent entities that may be shared among multiple EUs.

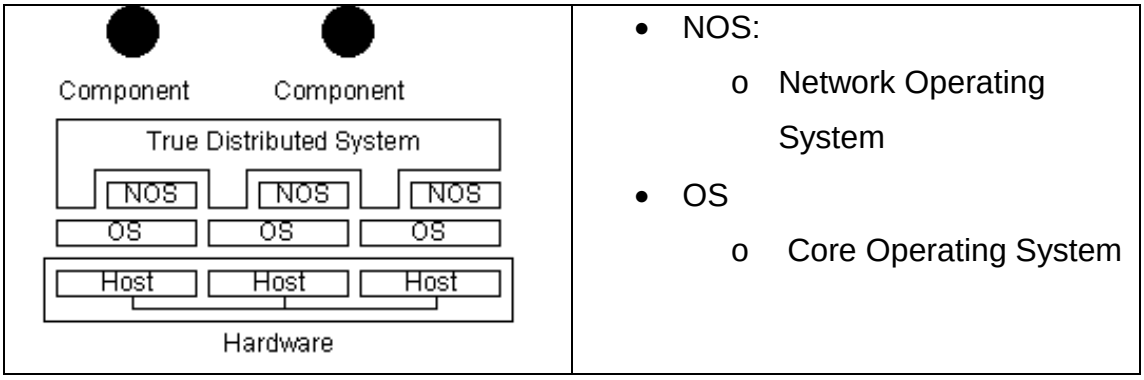
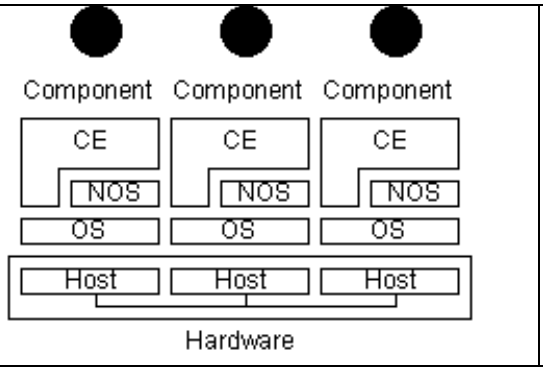


Figure 2.14(a): Traditional Distributed System from [15]



- NOS:
  - Network Operating System
- OS
  - Core Operating System

Figure 2.14(b): Mobile Code System from [15]

In standard, non-mobile systems an EU is tied to a single CE for its entire lifetime. In Mobile Code Systems, however, an EU can be relocated to a different CE. The portion of the EU that needs to be moved is determined by the strength of the migration that is required.

Strong mobility is the ability of an MCS to allow migration of both the code and the execution state of the EU to a different CE. *Weak mobility* is the ability of an MCS to allow code transfer across different CEs, optionally accompanied by some initialisation data, but without any migration of execution state. Mechanisms supporting weak mobility provide the capability to transfer code across CEs and either link it dynamically to a running EU or use it as a code segment for a new EU to be created and initialised.

2.5.3 Design of Mobile Code Systems

2.5.3.1 Introduction

Fuggetta [19], explains that traditional approaches to software design are not sufficient when designing large scale distributed applications that exploit code mobility and dynamic reconfiguration of software components. In these cases, he continues, the concepts of location, distribution of components among locations and migration of components to different locations need to be taken into account at the design stage.

The three main design paradigms exploiting code mobility are

1. Remote Evaluation
2. Code on Demand
3. Mobile Agents

2.5.3.2 Remote Evaluation

A and *B* are two components located respectively at sites S_A and S_B. *A* requires some service to be performed, and has the know how to complete it, but does not have the resources to do so. *A* sends the service know how to *B*. *B* executes the code using the resources available at S_B. *B* then sends the results of the computation back to *A*.

2.5.3.3 Code on Demand

Once again we have two components *A* and *B* are located at sites S_A and S_B. However in this paradigm *A* can access the resources it requires since they are located with it at S_A. *A* interacts with *B* to acquire the know how to perform the service, which is located at S_B. *B* delivers the code to *A*, and *A* executes the code at S_A using the resources available to it.

2.5.3.4 Mobile Agent

A has the know-how and some of the resources, but lacks some other resources vital to the correct computation of the piece of code. Therefore it takes the code and some intermediate results that it has computed at S_A and travels to S_B where it completes the computation.

The first two paradigms deal with the transfer of code. The final one involves the movement of the entire computational component and possible details about its environment and state.

2.5.4 Mobile Agent Technologies

There are many different mobile technologies currently available. A great deal of these systems use the Java technology set, due to its ability to work on any number of different platforms, among other reasons. There follows a

discussion of some of these systems. For a broader analysis and evaluation of some of these systems see Walsh [16] and Kiniry [20].

2.5.4.1 Java

The Java Virtual Machine (JVM) can be viewed as a Computational Environment. The class loader that is provided by Java dynamically links and retrieves classes in a JVM. It is invoked by the Java run time when the code currently in execution contains an unresolved class name. The class loader will retrieve the code from a possibly remote host and load the class into the JVM. For example imagine an applet that is running in a browser. The browser will contain it's own JVM. When the applet calls a class that it has not got locally the class will be retrieved from the host from which the applet was originally downloaded. In this situation we can view the browser with its in built JVM as the computational environment and the applet as the Executing Unit.
We can view this as a simple form of code mobility. It is the weakest form of mobility as the only thing being migrated is stand-alone code that is moving while not running. More complex systems will be dealt with below, but it should be understood that these systems are built using Java and run in a Java Virtual Machine, so they build on the functionality dealt with above.

2.5.4.2 Java Aglets

Aglets [21] are Java classes that can move from one host on the Internet to another. An aglet that executes on one host can suddenly suspend execution, move and then resume execution and another host. Aglets appear to be the most popular of any of the currently available mobility systems, some reasons for which are outlined by Kiniry [20].

1. They are easy to install and use for a Java programmer.
2. The example applications supplied with the installation are flashy and come equipped with attractive APIs (people are easily convinced by aesthetics).
3. They have the strength of the IBM Empire pushing them.

Alongside these reasons comes the fact that the Aglet system has a nice clean design. They take their name from a pun on the two words *applet* and *agent*, and their usage largely reflects the implementation of applets. The aglet system mirrors a great deal of the applet model, for instance

- `dispatch(...)` mirrors `run(...)`
- `deactivate(...)` mirrors `stop(...)`
- `run(...)` mirrors `run(...)`
- `getAgletContext(...)` mirrors `getAppletContext(...)`

Aglets, the Executing Units are threads in a Java Virtual Machine, the Computational Environment. The way in which the Aglet API supports migration is through the primitives `dispatch(...)` and `retract(...)`. `dispatch` performs code shipping of the stand alone code to a new context supplied as a parameter. A context of an aglet is the environment within which it is currently located. Services provided in a context include the facility to list the current set of aglets resident within the context, or add new aglets to the context. The second migration primitive, `retract`, is the same as `dispatch` except for the requirement that the aglet return to where `retract` was called from upon completion of its task.

With both `dispatch` and `retract` the aglet is re-executed from scratch when it is migrated, although the values of its object attributes are retained to provide an initial state for its computation.

Figure 2.15 below gives some sample Aglet code from the Java Aglet API specification [21]. The `run()` method is invoked when the aglet begins. Once `dispatch(url)` is called the aglet relocates to the url supplied. Before it leaves it calls the `onDispatching(...)` method. When it arrives at its new location it calls the `onArrival()` method.

```
import aglet.*;

public class DispatchingExample extends Aglet {

   public void onDispatching(URL url) {
      ...
   }

   public void onArrival() {
      ...
   }

   public void run() {
      ...
      dispatch(url)
      ...
   }
```

Figure 2.15: Sample Aglet Code

2.5.4.3 Voyager

ObjectSpace developed Voyager [22] in 1997. Voyager is a Java based mobile agent system that “exhibits several unique and innovative features” [20]. The chief innovative feature of Voyager is its Virtual Object. The Virtual Object is Voyager’s key communication framework and tool to support inter-agent communication. Conventional RPC based systems such as Java’s Remote Method Invocation (RMI) mechanism, require the developer to go through a series of steps to describe first the interface and then the implementation of an object. Voyager simplifies this by providing the ”vcc” tool (Virtual Code Compiler). This tool will take any existing Java class and modify it to create the Virtual Object mirror of the source class.

Any object that has been processed with the Virtual Class Compiler will then exhibit the properties of an agent. It can be migrated from server to server and accessed remotely by other Virtual Objects in an RPC-like fashion, and it will also have its own life cycle. Unless designed differently a Virtual Object will be a simple passive object, however it is possible to create them, like any Java object, so that they run in their own thread of control.

Another advantage of Voyager over other agent systems is that it can migrate into any Java runtime of another Virtual Object. It is not necessary to have an Agent server located on every node that you wish to use as a Computational Environment.

Voyager increases development time by providing a number of mechanisms that make it simple to move from a simple environment space to a distributed, agent based, mobile processing environment.

2.5.4.4 Agent TCL

This is a simple TCL interpreter extended with support for strong mobility. An Executing Unit or Agent is implemented as a Unix process. The Computational Environment abstraction is implemented by the Operating System. EUs can jump to another CE, fork a new EU at a remote CE or submit some code to a remote CE to process.

2.5.4.5 Obliq

Obliq is an untyped, object based, lexically scoped interpreted language. It allows for remote execution of procedures by means of an execution engine, which implement the CE idea. The EU, implemented as a thread can request the execution of a procedure on a remote execution engine. The code is sent to the destination and executed there in a new EU.

2.5.5 Other Systems

Examples of other Mobile Code Systems include the following:

- M0 from the University of Geneva
- Mole form the University of Stuttgart
- Sumatra from the University of Maryland
- Odyssey from General Magic
- A full list of currently available commercial mobile code systems is available from Kiniry [20].

2.5.6 Summary

Mobile code is code whose execution can take place on a number of nodes, through migration around a network. This chapter has discussed in detail how Mobile Code Systems operate and the different design paradigms to be considered when creating a MCS. It has also presented an overview of some currently available Mobile Code Systems, focussing mainly on IBM's Aglets and ObjectSpace's Voyager.

2.6 Summary

The purpose of this Literature Survey has been to outline the main technologies that will be used and referred to for the duration of the dissertation. The following are the topics that were discussed above:

1. Programming Languages & Vanilla
2. XML
3. Serialisation & Communication Mechanisms
4. Code Mobility

3. Design

3.1 Introduction

The last chapter, entitled *"Literature Survey"* outlined what technologies are available to be used in the course of this dissertation project. Vanilla, the technology that will form the core of the project was discussed in detail, as was XML. Code Mobility was introduced, analysed and assessed for background information on how systems such as the one being developed here are built. In this chapter a detailed design of the system will be presented, making reference back to the technologies discussed above. By then end of the chapter a design of a Data Serialisation and Mobile Code system in Vanilla will have been presented in full.

3.2 Requirements

The requirements of this dissertation can be summarised as follows.

1. Identify the different types/values than should be serialised.
2. Construct an XML representation for each type/value identified.
3. Design a Vanilla subsystem to handle overall serialisation mechanism and an individual component for each of the types/values to be serialised.
4. Design a mechanism for serialising/deserialising functions.
5. Design a mechanism for executing functions delivered as an XML document.
6. Design a mechanism and communications protocol for migrating execution of a function to another machine/environment.

3.2.1 Identification of Types and Values

The complete set of types implemented in Vanilla is as follows

1. **`String`**
2. **`Integer`**
3. **`Boolean`**
4. **`Ok`**
5. **`Record`**
6. **`Function`**
7. `Universals`
8. `Existentials`
9. `Autos`
10. `Mu`
11. **`Top`**
12. **`Bottom`**

The main focus of the serialisation section of this dissertation will be on the types that are central to the functionality of a programming language i.e. the entire set of `core` types; `Strings`, `Integers`, `Booleans`, `Oks`. Serialisation components for `Records` will also be implemented, and `Functions`, as a special type of type/value will also have serialisation components built.
`Top` and `Bottom` represent the different ends of the typing lattice. A `Top` is a type from which all other types inherit. In effect it is a super type. Every other type that exists is a sub type of `Top`. `Bottom` is the opposite. It is a sub type of every other type, from `Top` down. This relationship is shown in Figure 3.1 below. Because of their presence and their simplicity serialisation components will be implemented for these types and their associated values as well.

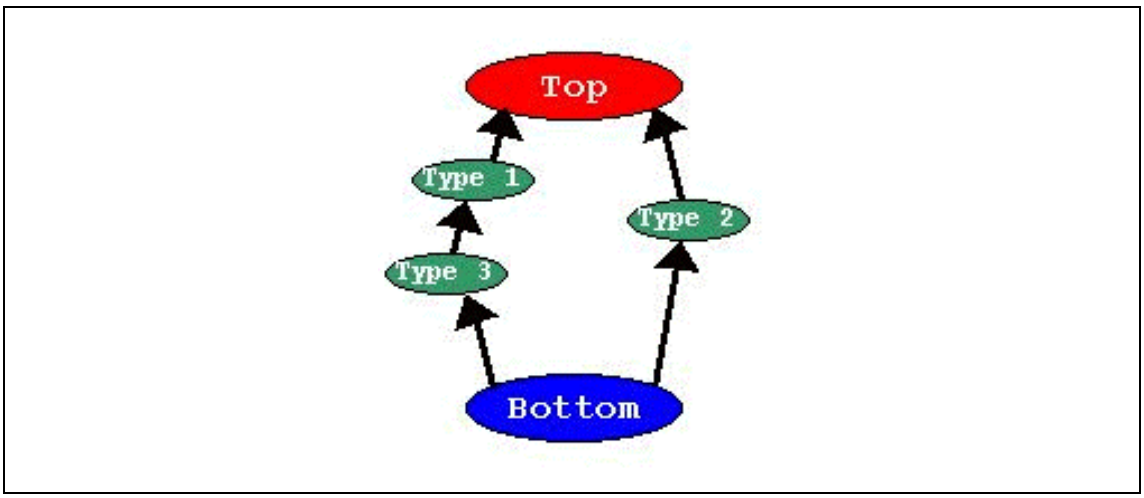


Figure 3.1: Type lattice

3.2.2 XML Representation

The very first decision that is required before the XML structure for each type-value pair is decided upon is how treat types and values should be treated. A number of possibilities existed at this stage.

- Firstly, all information about the Vanilla pods being used for the type/value could have been serialised. This way, when the value was being deserialised the class name of the type could be read in, leaving it up to Vanilla to decide how to handle it.
- Alternatively no information on the type of the serialised value could have been serialised. This could then be decided upon by Vanilla based on the type of the variable to which it was assigned upon deserialisation.
- The decision that was made was to provide type information but keep it separate from the value. For instance, in the first section of the XML document information about the value to be serialised is provided i.e. and indication is given that the value is, say, an `Integer` or a `Record`. All type information would be specified here. In the next section of the document all value information would be specified. The type

information given would be simply an indication of the name of the types, and any attributes it may have, all reduced down to primitive values. This separation of type and value information means that the serialisation and deserialisation of types and values could be handled separately by different Vanilla components.

The overall structure of the XML document that was decided upon is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<vanilla-serialized-value>
    <vanilla-type>
        <!-- all type information -->
    </vanilla-type>
    <vanilla-value>
        <!-- all value information -->
    </vanilla-value>
</vanilla-serialized-value>
```

Figure 3.2: XML representation of type value pair

All the core types will be represented in a manner that will be similar to the `String` example given overleaf in Figure 3.3.

```
<?xml version="1.0" encoding="UTF-8"?>
<vanilla-serialized-value>
    <vanilla-type>
        <String/>
    </vanilla-type>
    <vanilla-value>
        <string-value>This is a String</string-value>
    </vanilla-value>
</vanilla-serialized-value>
```

Figure 3.3: XML representation of a String

Records are more complicated in that they contain a set of values. The example (Figure 3.4) below shows how data describing type and value are kept separate. rather than providing type and value information for each of the fields together. The type and value for field *b* is given in **bold**.

Functions are more complicated still but they are dealt with in further detail below, in the section on *Serialising Functions*.

```
<?xml version="1.0" encoding="UTF-8"?>
<vanilla-serialized-value>
    <vanilla-type>
        <Record>
            <record-field field-name="b">
                <Boolean/>
            </record-field>
            <record-field field-name="s">
                <String/>
            </record-field>
        </Record>
    </vanilla-type>
    <vanilla-value>
        <record-value>
            <record-field field-name="b">
                <boolean-value>false</boolean-value>
            </record-field>
            <record-field field-name="s">
                <string-value>hello world</string-value>
            </record-field>
        </record-value>
    </vanilla-value>
</vanilla-serialized-value>
```

Figure 3.4: XML representation of a Record

3.2.3 Vanilla Subsystem

3.2.3.1 Subsystem architecture

A Vanilla language is constructed by *"plugging-in"* various components to perform the functions of the language. These components may be individual pods to implement aspects of the language such as loops or conditionals, or they may be components that constitute a new subsystem. A subsystem stretches over all pods with an individual component of that subsystem for each pod. An example of a subsystem is a type-checker. A type checker will have to be implemented for each different type within Vanilla, so the type checking subsystem sprawls across all pods implementing the type checker for functions, records, objects etc.

What is required for the purposes of the functionality to be implemented in this project is something similar to this, to implement a serialiser. A root serialiser class will be inherited from to create a serialiser subsystem capable of serialising each of the types and values identified above. An individual serialisation component will be implemented for each type and value and added into the appropriate pod, so that for the, say, `Records` pod we have each of the following components

- Parser
- Type Checker
- Interpreter
- Serialiser

The diagram below in Figure 3.5 shows how these components relate to each other. The root of each subsystem, in red below gets the request to parse, type check, interpret or serialise a specific value. It then goes and locates the component that is willing to do this. For instance if an `Integer` is to be serialised then the root serialiser will get passed the integer value. It will then ask each of the registered components if they are capable of performing the serialisation of an `Integer`. The `Records` pod will reply that it cannot. The

`Core` pod will reply that it is willing to perform this serialisation, and will proceed to do so.

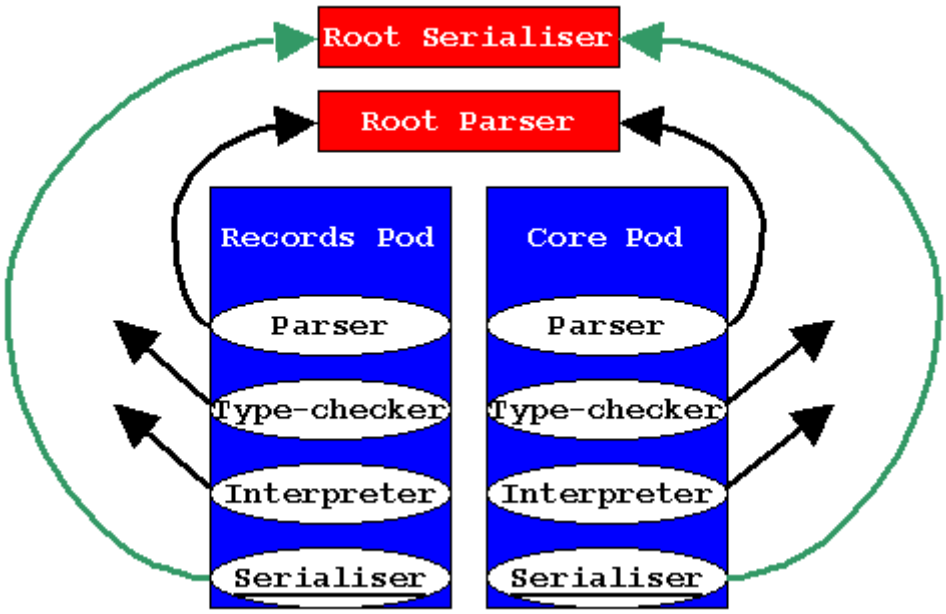


Figure 3.5: Relationship between components

The serialiser subsystem will form the central part of the implementation. The subsystem's root will need to be implemented and then a component for each of the pods that contains a type/value that will be serialised.

3.2.3.2 Individual Components

Each of the components that perform a serialisation of a value will be required to implement the same interface as the root component. They will have to separate Type and Value and Serialisation and Deserialisation. The four functions they must implement are shown in Figure 3.6 below.

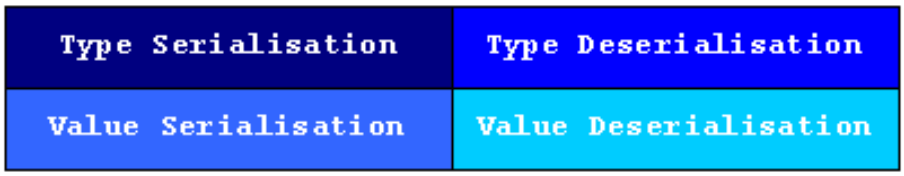


Figure 3.6: Functions required of serialiser components

The naming and implementation of the actual methods within the classes will be dealt with in greater detail in the *Implementation* chapter. Here the intention is only to give a brief overview of what will be performed by each of the methods:

Type Serialisation

- The type will be passed in to the method. The decision is made internally in the method about whether it is possible for this component to perform serialisation of that type. If so then an XML representation of the type will be generated. This will be written to an output stream or an XML document that is also passed as a parameter into the method.

Type Deserialisation

- An input stream or XML document that has been read and parsed by the DOM parser will be passed to the root component's method. This in turn queries all the components to see who can deserialise the type that is within the document. When a successful candidate is found the type is deserialised and returned back to the interpreter.

Value Serialisation

- This is performed in precisely the same manner as type serialisation with the obvious exception that rather than passing a type to the method a value is passed instead.

Value Deserialisation

- To deserialise a value it is necessary to know the type of the value. Therefore prior to deserialisation of a value the type of the value must have been deserialised from the same document or its type must be known. This may be fine for core/primitive types where the type is relatively straightforward, but for more complex types such as `Records` and `Functions` where types contain types and so on, the situation is a great deal trickier. In most cases the type is first deserialised and then passed to the value deserialiser with the XML representation of the value, which can then be deserialised. The decision on which component to use to deserialise a value is made based on the type that is passed in as an argument.

3.2.4 Serialising Functions

Vanilla functions are the same as any Vanilla value in that they implement the `IValue` interface. Therefore functions can be serialised and deserialised in the same way as normal types and values such as the `Core` types and `Records`.

However functions differ from more basic types in each of the ways described below

- Body
  - A function value contains a piece of code that must be serialised, as it is the central part of the function. This is a great deal more complicated than simply writing a value, as is the case for say, `Integers`.
- Environment
  - For a function to operate correctly when it is deserialised it must still have access to all the variables that were in its scope when it was serialised. The variables in its scope form a set that is referred to as its *environment*. Serialising all variables that are in the environment of a function is necessary in order to make them available to the function again when it is deserialised.

3.2.4.1 Body

3.2.4.1.1 Abstract Syntax

In the *Introduction* chapter the basic concepts behind programming language implementations were introduced. In the section on *Compiler Design* the concept behind the internal representation of code was dealt with. In that section Abstract Syntax was first mentioned as a means for representing code. Vanilla, like most other compilers and interpreters forms an abstract syntax tree naming all expressions as nodes. Vanilla considers all parts of code to be expressions with the overall program being an expression itself. Various Abstract Syntax Nodes represent the expressions in the tree. Figure 3.7 below shows a simple program and its representation as an Abstract Syntax Tree.

```
Int year = 2001;

if(year > 2000) (
 println("Future");
)
else
 println("Past");
```

```
ASTSequential
 ASTIntroduce
  ASTIntegerType
  ASTIdentifier <year>
  ASTIntegerLiteral <2001>
 ASTIf
  ASTGreaterThan
   ASTIdentifier <year>
   ASTIntegerLiteral <2000>
  ASTSequential
   ASTPrint
    ASTExpressionList
     ASTStringLiteral <Future>
  ASTPrint
   ASTExpressionList
    ASTStringLiteral <Past>
```

Figure 3.7: A simple program in Abstract Syntax

All code is transformed into abstract syntax as soon as it is parsed. From that point on the type checker and interpreter deal with abstract syntax rather than code.

3.2.4.1.2 Serialising the Body

A function is comprised of the arguments, the body and the return type. The arguments and the return type are typically primitive types and can be handled by the serialiser. The body however is abstract syntax. A means of representing abstract syntax as XML is required before functions can be serialised.

It was decided to maintain the overall tree structure of the Abstract Syntax and simply convert the tree into an XML tree, with each node in the tree associated with the class name of the Abstract Syntax Tree node. Where these nodes may have additional attributes they will also be included as attributes of the nodes in the XML tree, or as child nodes.

An example of a small piece of code is given overleaf in Figure 3.8. It is shown as a Vanilla Abstract Syntax Tree and as an XML tree.

```
Int nextyear = 2000 + 1;
```

```
ASTSequential
   ASTIntroduce
      ASTIntegerType
      ASTIdentifier <nextyear>
      ASTAdd
         ASTIntegerLiteral <2000>
         ASTIntegerLiteral <1>
```

```
<code-node class-name="ie.vanilla.pods.core.ASTSequential">
 <code-node class-name="ie.vanilla.pods.variables.ASTIntroduce">
  <code-node class-name="ie.vanilla.pods.core.ASTIntegerType"/>
  <code-node class-name="ie.vanilla.syntax.ASTIdentifier">
   <identifier-name name-value="nextyear"/>
  </code-node>
  <code-node class-name="ie.vanilla.pods.core.ASTAdd">
   <code-node class-name="ie.vanilla.pods.core.ASTIntegerLiteral">
    <primary-literal-value primary-value="2000"/>
   </code-node>
   <code-node class-name="ie.vanilla.pods.core.ASTIntegerLiteral">
    <primary-literal-value primary-value="1"/>
   </code-node>
  </code-node>
 </code-node>
</code-node>
```

Figure 3.8: Abstract Syntax Tree as XML

This method of converting Abstract Syntax to XML will be used for the serialisation of the body of functions.

3.2.4.2 Environment

Below in Figure 3.9 we can see an example of a Function.

```
Int currentyear = 2000;

Function() Int nextyear;
nextyear = fun() (
     currentyear + 1;
);

Int answer = nextyear();

print("Next year is : ");
println(answer);
```

Figure 3.9: A Function with its Environment

In the above code segment it is seen that the function `nextyear()` uses the variable `currentyear` even though it is not declared within the function itself. This variable is within the *scope* of the function, Such variables are termed *loose variables*. The set of all variables in the scope of a function is termed the *environment* of the function.

When a function is serialised and subsequently deserialised it should be able to operate as it normally would. Therefore it is necessary to write all variables from the environment out to the XML document when the function is being written. When writing out a function the code for the function must be followed by a set of serialised values to represent the environment.

Figure 3.10 overleaf shows the overall structure of an environment in XML. The `parent` is simply the environment in whose scope this environment is located. This is not used in the implementation of this project but is there to accommodate the ongoing work on object serialisation.

```
        <environment-value>
            <parent/>
            <in-scope-variables>
                <in-scope-variable variable-name="squaring">
                    <in-scope-variable-type>
                        <String/>
                    </in-scope-variable-type>
                    <in-scope-variable-value>
                        <string-value>Squaring...</string-value>
                    </in-scope-variable-value>
                </in-scope-variable>
                <in-scope-variable variable-name="squareVal">
                    <in-scope-variable-type>
                        <Function>
                            <arguments-type/>
                            <return-type>
                                <Top/>
                            </return-type>
                        </Function>
                    </in-scope-variable-type>
                    <in-scope-variable-value>
                      <!—and so on -->
                    </in-scope-variable-value>
                </in-scope-variable>
            </in-scope-variables>
        </environment-value>
```

Figure 3.10: An Environment in XML

3.2.4.3 A Function in XML

As already mentioned, a function in XML is fully specified by arguments, body, return and environment components.

The type of a function can be reconstructed once we supply the following information.

- Argument Types & Names
- Return Type

The value of a function can be reconstructed from an XML document once the following is provided.

- Body in Abstract Syntax

In order to do anything useful with the function the following is also required

- Environment

The structure of an XML document containing a function is given overleaf in Figure 3.11.

```
<?xml version="1.0" encoding="UTF-8"?>
<vanilla-serialized-value>
     <vanilla-type>
     <Function>
          <arguments-type>
               <argument argument-name="...">
                    <!-- Argument Type e.g. Integer -->
               </argument>
          </arguments-type>
          <return-type>
               <!-- Return Type e.g. Integer -->
          </return-type>
     </Function>
     </vanilla-type>
     <vanilla-value>
     <body-value>
          <!-- The AST code nodes -->
     </body-value>
     <environment-value>
          <!-- The Environment -->
     </environment-value>
     </vanilla-value>
</vanilla-serialized-value>
```

Figure 3.11: A Function in XML

3.2.5 Executing Functions

Now that there exists a means for representing Vanilla functions cleanly and completely as XML the possibility of executing functions that are stored as XML on the web can become a reality. Here is how this problem was approached.

The section *"Remote Access"* below will deal with writing and reading values onto the World Wide Web. All that is required here is a knowledge of the fact that deserialisation from the Web is as simple as deserialisation from the local disk in terms of what is required by the programmer.
In order to execute a function from the local disk or the web, in fact any XML document regardless of how or where it is stored the following steps must be followed

1. Download the file from its location, be that the web or the local disk.
2. Parse the XML to generate a value. This value will be a Vanilla function value, read in as if it were a normal value being deserialised from an XML document.
3. Call the Vanilla interpreter on the function with appropriate values as arguments.
4. This will return a value if the function returns a value. It is precisely the same as if the function were run normally, except now it is not necessary to run the type checker since this would have been done at the serialisation stage.

The diagram below takes you through the steps graphically.

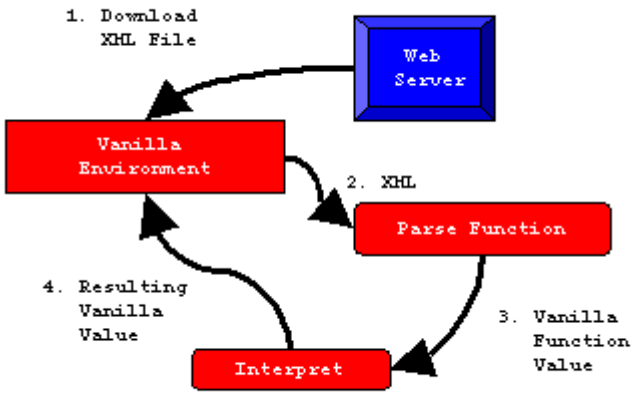


Figure 3.12: Executing XML Function

3.2.6 Migrating Function Execution

Rather than executing a function that is an XML file at some location on the web it may be preferred to wrap your local function up as XML and send it to some other machine. It will attempt to execute itself on that other machine and will then return the value it obtained to the function which first attempted to execute it.

Migration of functions is the mobile section of this dissertation. It has already been shown how the function is wrapped up to include the environment in which it is operating. This will save the state and this provide some degree of strength in terms of where the implementation lies between weak and strong mobility. The means by which the execution of a function is migrated from one machine to another is as follows.

1. Wrap the function and all the parameters to the function up as separate XML documents and HTTP POST them to another machine.
2. A servlet running on that machine will receive the POST and start up its own Vanilla environment.
3. A Vanilla environment is now trying to execute a function that was sent to it as XML. This problem is resolved in the same manner as was demonstrated in the last section *“Executing Functions”*.
4. The resulting value is translated into XML and POSTed back to the client.

This is all shown graphically overleaf.

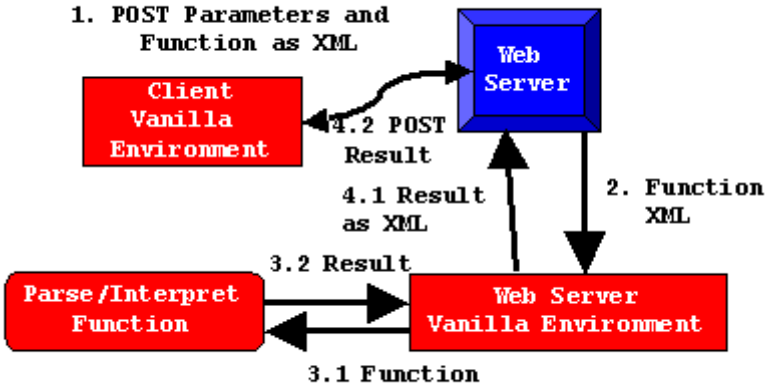


Figure 3.13: Migrating XML Function

Migration of functions while represented as XML is all dependent on the ability to read and write Vanilla values from local and remote disk space. The approach that was taken in order to make file space available on a community of machines is dealt with in the following section. For a greater understanding of the technical issues encountered in this part of the project see the *Implementation* chapter.

3.3 Remote Access

One of the main points of this dissertation was to produce some means of migrating code to another execution environment. The success of this was dependent of two other aspects of the dissertation

1. Correct and accurate serialisation of values including functions to an XML document.
2. A means of writing the XML document to each of the following
    - Local machine
    - Network
    - World Wide Web

The serialisation to XML aspect of the dissertation has already been dealt with in detail and the writing out of these files has been touched on in the discussion of remote execution and function migration. This section is quite technical and will therefore be dealt with in greater detail in the Implementation chapter. However in this section you will be given on overview of what the architecture built for remote access will look like and why this architecture was chosen over alternatives such as Jiki.

3.3.1 Local / Network

In order to read or write a file locally all that is required is some file access mechanism. All file access is handled by Java and so there is no difficulty in reading or writing files to the local disk or any local network drive.

3.3.2 Remote Read

A remote read could also be referred to as a file download. Effectively what is being done is the same as what happens when any web browser accesses a Web page and reads its contents. A file is being downloaded and its content is being interpreter. Reading a file from the web is therefore almost as simple as reading it from the local disk.

3.3.3 Remote Write

A remote write is nowhere near as simple as a local write however. When attempting a remote write, i.e. a write of a value to the web, a number of issues are encountered.

1. First of all some means is required to access file space on another machine.
2. Secondly, all security must be taken care of transparently, if there are any security considerations to be taken into account.
3. Most importantly there needs to be some way of dropping your own files on another machine.

The alternatives that were considered here were the following:

- FTP
  - When a new file is created FTP it to a location on file space elsewhere. For this to work it would be necessary to provide sufficient security rights to each Vanilla client in order to allow it access space on the web server. This had potential security risks.
- Jiki
  - Jiki (www.jiki.org) provides a whole web architecture that allows all files on a web server to be accessed by any client. In much the same way as the web at the moment allows any user read a page, Jiki also allows any user edit the page. This kind of architecture mirrors what is required for the purposes of this dissertation. However the decision was made not to use the Jiki web server as it is still largely untested, and besides it offered little benefit over the eventual approach that was taken.
- Vanilla Enabled Web Server
  - The architecture that was eventually designed is similar to the Jiki architecture in that it allows fully open access to any Vanilla client that wishes to access the Vanilla space on a Vanilla enabled web server. All transactions take place over the web via HTTP POSTs and all files are stored as XML.
  - In order to use space on a Web server as a vanilla repository the following must be done
    1. The Vanilla JAR file must be installed on the server.
    2. The two servlets, `vanilla/run` and `vanilla/write` must be made accessible from your server.
    3. Some file space must be made available on the web server for values to be stored, typically in the directory `http://host:port/vanillavalues`.

The `vanilla/run` servlet is the servlet that will execute a file handed to it. This was discussed above in the section on *Migrating Functions*. A further discussion on this servlet will take place in the next chapter, *Implementation*.

The `vanilla/write` servlet is the servlet that allows a Vanilla client write a Vanilla value as an XML document to the `vanillavalues` directory on a Vanilla enabled web server. Its architecture is rather simple. It is simply a servlet that is constantly running expecting HTTP POST requests with XML content. The XML that is received is written out the `vanillavalues` directory. To read the values back the XML file can then be downloaded from `http://host:port/vanillavalues/filename.xml`. The architecture is show graphically below in Figure 3.14.

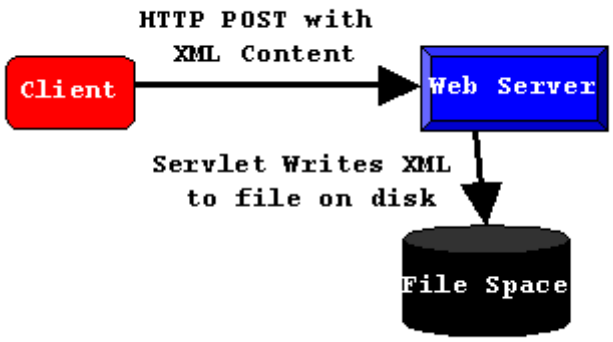


Figure 3.14: The architecture for writing to remote server space

3.4 Other Design Issues

Besides the decisions that were made on the design of the XML documents to represent the Vanilla values and the overall architecture for writing these files the only major design decision came in the choice of XML parser.

3.4.1 XML Parser

It was decided to use the Apache Xerces DOM parser for the purposes of this project. This parser is still being developed and has not fully completed all stages of its implementation. The reason it was chosen was because of its support for XML validation using XML Schema.

Although it is not intended to implement an XML Schema for the various XML documents described above it is hoped that this would be done at some point in the future, when namespaces are being introduced to Vanilla XML documents. Since Xerces parsers already support schema the only change that would be required in order to validate would be to include a URL for a schema in each document. No great change to the design of the system would be required.

DOM is favoured over SAX for the simple reason that memory is not a major concern in this project, and that the benefits of having a document presented in a tree like structure makes the handling of recursive calls to the serialiser mush more easy and less time consuming to implement.

3.5 Summary

In this chapter the overall structure of the system being designed has been outlined. Any decisions that have been made regarding technologies have been described. By now the application and operation of Vanilla Serialisation should be a great deal clearer, at least from a high level.

In this chapter the following has been detailed:

1. How to design a new subsystem to implement a new set of functionality
2. How to structure a document comprised of a type and value
3. What types / values to serialise
4. How to represent each type / value in XML
5. How to migrate / execute functions
6. How to write values locally and remotely
7. How decisions were reached on technologies such as Jiki, Servlets & Xerces.

The next chapter will deal with the same issues but will present a more technical insight into some of the issues encountered while implementing the system.

4. Implementation

4.1 Introduction

The chapter on *Design* will have already presented a reasonably detailed account of how the goals of this dissertation were designed. The implementation of the Data Serialisation and Mobile Code system using Vanilla will be discussed here with the main focus being on technical issues.

4.2 Vanilla Components

A Vanilla subsystem is a set of components that implement a single aspect of the functionality for a programming language. The manner in which they operate is introduced above in the *Design* chapter. Essentially a subsystem will gather together a set of components implemented across a set of pods with a registry. The root of the subsystem will decide which of the registered components will handle each call to the subsystem.

4.2.1 Serialisation Subsystem

Type and Value serialisation is the central part of the project being implemented. A new subsystem called `Serializer` was implemented to handle all aspects of the serialisation functionality.

Any component that forms a part of this subsystem (i.e. any component implemented to serialise any type value pair) must implement the following methods:

1. `boolean componentWriteType (Type type, Environment env, ElementWrapper wrapper)`
2. `boolean componentWriteValue (IValue val, Environment env, ElementWrapper wrapper)`
3. `Type componentReadType (Environment env, ElementWrapper wrapper)`

4. `IValue componentReadValue (Type type, Environment env, ElementWrapper wrapper)`

All of these methods take an `Environment` object and an `ElementWrapper` object as parameters.
The `Environment` object is the set of all values that exist within the same scope as the value that is being serialised.
The `ElementWrapper` object is the means used for passing an XML document between components. For a fuller understanding of this a short description of the XML approach follows.

4.2.1.1 XML Approach

The Apache Xerces DOM parser was used for the purposes of this dissertation. The DOM parser allowed an XML document to be treated as a tree, thus making the serialisation of large amounts of nested levels of values a great deal simpler.
The DOM parser allows the programmer to create a new document that is simply the root node in a tree. Children can be added to the nodes thus extending the tree.
The `ElementWrapper` objects that can be seen as a parameter in the four methods introduced above are simply a means of storing a pair of nodes in an XML tree. The two nodes that are stored are

a. The root of the tree.
b. The node in the tree to which a child must be added.

4.2.1.1.1 Writing

A new XML document is created and put into an `ElementWrapper` object. This is passed between the various components that are serialising values; say for a `Record` where many values are being written. At each serialisation stage a child is added to the tree. When all children have been added the Xerces parser is able to handle writing out the whole XML tree as a document to any output stream.

In order to serialise any value in Vanilla the serialiser is called firstly on the type and secondly on the value. This keeps type and value information separate in line with the design given in the previous chapter.

4.2.1.1.2 Reading

A new DOM object is set up by reading in an XML document. An `ElementWrapper` object is created using the root of the tree and passed between the various deserialisation components which are responsible for extracting sufficient information to be able to create a type or value.
Deserialisation takes place by first calling the (de)serialiser on the type of the value in the document. This will read in the type from the document. This type is then passed to the serialiser method for deserialising values. Using the type the serialiser can read in the value and create a new `IValue` object (further explanation below).

4.2.1.2 Component Methods

The four different methods in the serialiser subsystem are described below.

4.2.1.2.1 ComponentWriteType

This method takes a `Type` object as a parameter. An XML representation of the Type is generated and added as a child node to the node in the `ElementWrapper` object that is being passed around through the components.

4.2.1.2.2 ComponentWriteValue

This method takes an `IValue` object as a parameter. An `IValue` is an interface that must be implemented by any class that represents values in Vanilla. An XML representation of the `IValue` is generated and added as a child node to the node in the `ElementWrapper` object that is being passed around through the components.

4.2.1.2.3 ComponentReadType

The XML document in the ElementWrapper is looked at and the Vanilla type information is extracted. This method will generate and return a Type object.

This method is called recursively for complex types such as `Records` and `Functions`.

4.2.1.2.4 ComponentReadValue

The `Type` is always deserialised, or at the very least known before this method is called. This method takes the type as a parameter and deserialises the value based on this type.

4.2.2 Code Serialisation

The *Design* chapter dealt with the necessity for a means of serialising code. This is necessary for writing `Functions`, as the body of a function is an Abstract Syntax Tree. The code serialisation subsystem provides the two methods listed below. The code serialiser is called from the `FunctionSerialiserComponent`.

1. `boolean componentWrite (Node codeNode, Environment env, ElementWrapper wrapper)`
2. `Node componentRead (Environment env, ElementWrapper wrapper)`

4.2.2.1 ComponentWrite

A `Node` object, which is simply a node in Vanilla's internal Abstract Syntax Tree representation of the code, is passed in. The class of the node is established and written out as a node in an XML tree. The default serialiser simply writes the class name (e.g. `ie.vanilla.pods.functions.ASTFunction`). Where necessary the default serialiser is overridden by a method which can write out additional attributes from the Abstract Syntax Tree, for example for an `ASTIntegerLiteral` class the value of the integer would also have to be serialised. For an example of this see the *Design* chapter.

4.2.2.2 ComponentRead

The reflection API is used to rebuild the AST expression that was serialised. Since class names were written out, instances of these classes can be rebuilt quite easily. Where necessary additional attributes can be added to the AST Expression by over riding the default serialiser.

4.2.3 Environment Serialisation

`Environments`, like abstract syntax, also need to be serialised with `Functions`, for reasons that have been outlined in detail in the *Design* chapter. Although a subsystem has been implemented for this there is still only one version of `Environment` implemented. The idea of using a whole subsystem for serialisation of `Environments` is that new types of `Environments` can be added and have serialisation components added with them.

The methods that are available for `Environment` serialisers are as follows:

1. `boolean componentWrite (Environment env, ElementWrapper wrapper)`
2. `Environment componentRead (ElementWrapper wrapper)`

4.2.3.1 ComponentWrite

Writing an `Environment` consists of calling the basic serialiser on all the various types and values in the `Environment`, before travelling up the Environment tree and serialising its parent.

4.2.3.2 ComponentRead

Similarly, deserialising an `Environment` is no more than calling the serialiser read methods on the types and values, and injecting the retrieved values into a new `Environment` object. If the deserialised `Environment` had a parent deserialised with it then this must be deserialised too and set as the parent of the newly created `Environment` object.

4.2.4 Other Subsystems

Only one other subsystem was implemented. It played a relatively minor but nonetheless necessary role. That was the `TypeMatch` subsystem.

4.2.4.1 Type Matching

Following the type checking stage the types of Vanilla values are often discarded since the interpreter does not require them. However this can prove troublesome for the serialiser that always serialises a type with a value. If the type has been discarded then Vanilla needs some other way of finding out what the type is. The `TypeMatch` components implement one method, which is used to query all registered components to ask if anyone can tell the `Type` of a particular `IValue`.

4.2.5 System Overview

Once all the components were integrated into Vanilla the new view of the Vanilla system was a shown below in Figure 4.1

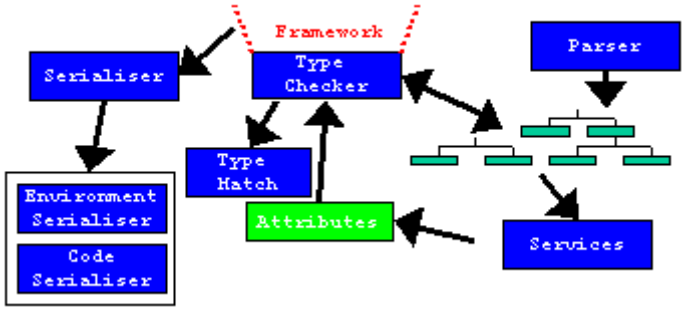


Figure 4.1: Complete view of Vanilla System

4.3 Mobility / Remote Access

4.3.1 Introduction

In order to integrate some type of mobility into the system several different tasks have to be completed. Firstly an extra method needed to be added to the serialiser interface to allow for execution of XML documents downloaded from the web and for running functions on remote machines. Secondly the Xerces parser needs to be modified slightly so that, as is the case with running functions remotely, the XML is needed as a string and not as a file. Thirdly the two servlets required for the implementation of a Vanilla-enabled server need to be written and the changes to the server outlined in the *Design* chapter need to be made.

Changing the Xerces parser was relatively simple. A new output stream writer and a new input stream reader were implemented. Both implemented the interfaces that all writer and readers in Java implement. However they did not take their input from a file or write to a file, instead all writing took place to memory and all reading from memory. This way the XML parser, which could only work when reading from or writing to a file, still operated in the same way and did not require any changes to its own code.

The other tasks were slightly more complicated and are dealt with in some detail below.

4.3.2 Additions to Subsystem

In order to allow execution of an XML file downloaded from the web some extra functionality needed to be added to the serialiser component. Any component that could be executed should implement the extra method that was added. This method is only going to be implemented by the `FunctionSerialiserComponent` for the purposes of this project. However, should there be other components in the future that could be executed directly from XML they need only implement this extra method, the signature for which is given below.

- `IValue componentExecute (ElementWrapper element, IValue [] paramValues, Environment env)`

The `ElementWrapper` object contains the XML for the function that is to be executed. The `paramValues` array contains the set of `IValue` objects that are the parameters to the function. Execution of the function takes place by first parsing the XML to create an `IClosure` object, the Vanilla representation of a function, deserialising the environment, declaring the parameters in the environment and then calling the Vanilla interpreter on the function. An `IValue` object will be returned.

4.3.3 Vanilla Enabled Server

The concept of a Vanilla Enabled server was introduced in the *Design* chapter. Such a server will be able to handle

- Writing a Vanilla value as XML to the file space on the server
- Running a Vanilla function in the Vanilla environment on the server

To allow for this, a number of configuration changes had to be made to the Apache Web server that was used for this implementation. The changes were as follows.

1. A new directory had to be created in the document root of the server. This directory was named `vanillavalues` and serves as the store for all Vanilla values that are written to the server. The location of this directory is specified in the `Vanilla.properties` file.
2. A new Vanilla servlet zone was created. This simply meant that any Vanilla servlets that were written would be located at a URL similar to `http://host:port/vanilla/` rather than at a standard servlet URL which would more likely be `http://host:port/servlets/`.
3. Vanilla was installed on the host machine and the classpath to locate it was added to the configuration files of the web server.

4. Two new servlets were written. These servlets are called write and run and would be located at `http://host:port/vanilla/write` and `http://host:port/vanilla/run` on a Vanilla enabled server. Further explanation of both of these servlets follows below.

4.3.3.1 vanilla/write

This servlet is relatively straightforward. It received two parameters in a HTTP POST. The first parameter is the URL encoded contents of an XML document representing a Vanilla value. The second parameter is the name that will be given to the value when it is stored as a file. As is standard practice with file writes the writing of a Vanilla value as a file overwrites any previous file of the same name in the same directory.

4.3.3.2 vanilla/run

This servlet is a great deal more complicated than the write servlet since execution of this servlet requires the loading up of a new Vanilla environment and execution of a function that is passed to it as an XML document.
In order to execute a function from a servlet the following extra functionality had to be implemented. This is described below.

- Ordinarily Vanilla is run from the command line. A batch file runs which passes in a number of parameters to a Vanilla tool that sets up a Vanilla environment, in which the code is executed. Since Vanilla will not be run in this fashion when it is run from a servlet a new Vanilla tool is required. The new tool that was created was implemented in the `VanillaComponent` class. All loading of properties and so on is handled there.
- The servlet receives a URL encoded XML document representing a Vanilla function in the form of a HTTP POST. All other HTTP parameters that are POSTed represent parameters to the function.
- The operation of this functionality is as follows: The VanillaComponent class is loaded, the function and parameters are converted from XML to Vanilla values, the function is executed in the same way as was described above for execution of remote files, the value is converted

back to XML and then POSTed back to the client. All this is described in greater detail in the *Design* chapter.

4.4 Other Implementation Issues

The only other major part of the implementation was the Vanilla pod to test the functionality. This is described below.

4.4.1 Serialise Pod

In order to test and evaluate the functionality a pod named `serialise` was implemented. This added four extra functions to one of the Vanilla test languages.
These functions were as follows:

- `write (FileName/URL, Value);`
- `read (FileName/URL);`
- `execute (FileName/URL, parameter1, parameter2, ..., parameter n);`
- `run (host, function, parameter1, parameter2, ..., parameter n);`

The first two test the local and remote read and write capabilities. The third one tests the functionality for executing functions that are stored as XML documents and the third one implements the code migration functionality. To see these functions in operation see the *Evaluation* chapter.

4.5 Summary

This chapter has outlined all the issues that were encountered when implementing the project. Rather than concentrate on the system from a high level this chapter has gone into the detail about how everything was implemented at the lowest level deemed appropriate.

The implementation issues, decisions and approaches covered in this chapter include the implementation of the various subsystems and components and the approach taken to mobility using an Apache Web Server.

5. Evaluation

5.1 Introduction

In the preceding chapters the aims, design and implementation of this dissertation have been covered in detail.

In this chapter the testing of the implementation will be described. An assessment of the different aspects of the implementation is presented in order to see what are the conditions under which it performs best, where it does not perform well and where there is room for further work, which will be outlined in full in the *Conclusions* chapter.

5.2 Aims

The aims, as outlined in the introduction were listed below. Each of these will be assessed in turn.

1. The development of some standard mechanism for the representation of Vanilla types and values as well as functions as XML documents.
2. Extending the current Vanilla framework by adding disk access functionality. The data will be serialised to the disk as an XML web document.
3. Values serialised to elsewhere on the web as XML documents.
4. Serialisation of code as XML, leading to the implementation of a mobile code system.

5.3 Types & Values as XML

It has been shown in earlier chapters how types and values have been represented as XML. A structure for documents that separates `Type` and `Value` information for all the basic types including `Records` has been provided. An example of the code for an Integer value is given below in Figure 5.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<vanilla-serialized-value>
    <vanilla-type>
        <Integer/>
    </vanilla-type>
    <vanilla-value>
        <integer-value>1000</integer-value>
    </vanilla-value>
</vanilla-serialized-value>
```

Figure 5.1: A Vanilla Integer as XML

Although an XML DTD or Schema for the types and values in Vanilla has yet to be defined it was found that the data lacks nothing because of this. Since all the XML will be both produced and consumed by the Vanilla components discussed earlier it was not found to be of great importance to provide an outline for how this data should be structured. Should the data be produced by non-Vanilla programs such a description would be necessary. However, for the purposes of what was done here, it was found that the use of a non-validating XML parser was sufficient so no DTD was required. This will be implemented in the future.

5.4 Disk Access

Once a means for describing the data in XML had been provided the next step was to enable disk reads and writes with the values. This functionality was implemented and tested for each of the following Type/Value pairs.

1. Integer
2. String
3. Boolean
4. Ok
5. Record
6. Function

7. Top
8. Bottom

For each of the above Types/Values the XML representation of the data contained sufficient information to be able to reconstruct the value in the deserialisation stage.

Provided below is a sample piece of code and the output produced, showing that records, as an example could be serialised and then deserialised from the local disk. Please note that the language shown below, as with all the examples is simply a Vanilla test language. Any number of languages with any type of syntax can be designed. The language below is the one provided with the Vanilla installation that implements all the basic functionality including serialisation.

```
Record (
     Boolean b;
     String s;
)
rec;
rec.b = false;
rec.s = "I am a string";

println("Writing Record Locally...");
write("record.xml", rec);

Top x = read("record.xml");
println("\"x\" has been deserialized as... ");
println(x);
```

Figure 5.2: Example code to write a record locally

```
Z:\Vanilla>vc example_1.v
Writing Record Locally...
"x" has been deserialized as...
record(b = false; s = I am a string; )
bottom
```

Figure 5.3: Output from above code

5.5 Serialisation to Web

In the same way as the code above writes the data to the local disk, it has been seen that it is possible to write the same data to space available on the web. This has been outlined in full in the design and implementation chapters, so all that is necessary here is to show that is can be done in the same way as for the local disk, simply by substituting a URL for a filename. Note that for writing the URL must be for a server that is Vanilla enabled

```
Record(
     Boolean b;
     String s;
)
rec;
rec.b = false;
rec.s = "I am a string";

println("Writing Record to pc678.cs.tcd.ie...");
WRITE("HTTP://PC678.CS.TCD.IE/VANILLAVALUES/RECORD.XML",REC);

Top x =
     read("http://pc678.cs.tcd.ie/vanillavalues/record.xml");
println("\"x\" has been deserialized as... ");
println(x);
```

Figure 5.4: Example code to write a record locally

```
Z:\Vanilla>vc example_2.v
Writing Record to pc678.cs.tcd.ie...
"x" has been deserialized as...
record(b = false; s = I am a string; )
```

Figure 5.5: Output from code in Figure 5.4

5.6 Functions

Although they are just values there are a number of differences that have to be dealt with when serialising and deserialising functions. Firstly, since functions contain actual code that is stored as Abstract Syntax there must be a means for reconstructing the Abstract Syntax from the serialised XML. Also, for functions to be migrated to other machines there needs to be a mechanism for implementing some degree of strong migration. The migration employed in this case, as is outlined in the preceding chapters, simply wraps up the environment in which the function is operating.
The goals for functions were as follows:

1. Serialise functions as if they are simply values.
2. Read a function back in as if it is a value.
3. Read a function in from an XML document and execute it.
4. Migrate a function to another Vanilla environment, or machine, execute it there and return it to the original Vanilla environment.

All of these goals have been achieved as shown in the figures overleaf

```
Function(Int val) Int square_val;
square_val = fun(Int val) (
     val * val;
);
write("function.xml", square_val);
```

Figure 5.6: Write a function

```
Function(Int val) Int sq;
multiply = read("function.xml");
Int answer = sq(10);

print("Answer : ");
println(answer);
```

Figure 5.7: Read function back in as value

```
Z:\Vanilla>vc example_4.v
Answer : 100
Bottom
```

Figure 5.8: Output from Figure 5.7

```
Int answer = execute("function.xml", 5);
print("Answer : ");
println(answer);
```

Figure 5.9: Execute function from XML file

```
Z:\Vanilla>vc example_5.v
Answer : 25
Bottom
```

Figure 5.10: Output from Figure 5.9

Figure 5.11 shows a piece of code that includes a number of functions and values all contained within an environment. For the function to execute when migrated it requires all the values within the environment. This means that all these values will have to be serialised with the function and inserted into the new environment that will be created in the remote Vanilla environment. It can be seen below that the function operated correctly, proving that the entire environment must also have been serialised along with the function. The values used in the environment are in *italics* and the uses of these variables are shown in **bold**.

```
Boolean square_answer = true;

Function(Int val) Int square_val;
square_val = fun(Int val) (
     val * val;
);

Function(Int m, Int n) Int multiply;
multiply = fun(Int m, Int n) (
     Int ret_val = 0;
     if(square_answer) (
          ret_val = square_val(m * n);
     )
     else
          ret_val = m * n;
     ret_val;
);

Int answer = run("http://pc678.cs.tcd.ie/vanilla/run",
                  multiply, 5, 10);
print("Answer : ");
println(answer);
```

Figure 5.11: Running a function with its environment remotely

```
Z:\Vanilla>vc example_6.v
Answer : 2500
Bottom
```

Figure 5.12: Output from Figure 5.11

5.6.1 Issues

The only issue that was encountered when working with the above values was that the environment constructed from the XML file did not correspond correctly to the original environment that was serialised if the function that was serialised contained multiple `return` statements. This was due to the fact that a hook was placed in the Environment to specify where the function should escape to in the event of encountering a `return` statement. This hook would not be maintained while serialising the environment and since there is no explicit call to the function matching the abstract syntax of the original call a failure occurs. At the time of writing no fix for this issue has been implemented although work is continuing in this area.

All other issues relate to the serialisation of objects. Although not originally part of the requirements for the project it was soon realised that once `Records` and `Functions` were working work on serialising `Objects` could begin since they are effectively a combination of the two. `Object` serialisation is currently being implemented although at the current time it has not been completed. The main issues being dealt with are nested environments and the subtle differences between functions and object methods.

5.7 Applications

What has been implemented here is a simple mobile code system. An existing language development system has been extended to allow languages migrate execution of a function from one machine to another. The primary aim was to prove that this was possible using Vanilla. Further applications of this could

carry on the path that has been started. Obviously some agent technology using Vanilla could be implemented. Such a technology would be similar to Java Aglets by allowing communities of Vanilla environments that could communicate among themselves through the serialisation of values and code. A prototypical version of such a system could easily be implemented using the work that has so far been done.

The fact that a clean XML representation of Vanilla values has been implemented also opens the door to the possibility of some interesting applications in this area. It would be possible to implement a system where a single data repository on the web could be used to store a set of XML documents that all represent Vanilla values. Various applications built using Vanilla could share these values, and thus work in tandem on collaborative projects, where a value's name could be a URL, which by definition would be unique. Such a system could be easily built using the current Vanilla version since URLs of web pages already represent values within Vanilla.

5.8 Mobility Analysis

5.8.1 Code Mobility in Vanilla

The Background chapter has already presented an overview of what is meant by terms such as *Mobile Code System (MCS)*, *Computational Environment (CE)* and *Executing Unit (EU)*.

A MCS is a system that allows a piece of code that is executing on one machine migrate to another location to complete its execution. A distinction is made between this type of system and a *Distributed System (DS)*. A DS also allows execution over a number of nodes but with the appearance of execution on one node, or one environment. A DS hides the boundaries between nodes and distributes processing transparently. A MCS requires the programmer (or some algorithm) to actively tell a piece of code to migrate to another node. In this sense the functionality that has been implemented in Vanilla is truly a MCS. The `run(…)` method that has been added to the test language is the tool that is used to migrate execution between nodes. This needs to be actively called within the program to force migration to another

node. The method of migrating the execution of a Vanilla function makes it a *Remote Evaluation* MCS. A discussion of Remote Evaluation MCSs alone with other types of MCS is presented in the *Background* chapter.

A CE is an environment on a machine the can host the execution of a unit of code. The Implementation chapter above will have outlined what changes need to be made to a standard web server to make it capable of hosting the execution of some Vanilla code. Such a server is referred to as a Vanilla Enabled server and corresponds directly to the definition of a Computational Environment.

An EU is a single sequential flow of execution. Any Vanilla function that is being serialised represents a single flow of execution as it is a normal function. Therefore any Vanilla function is potentially an Executing Unit.

The type of

5.8.2 Strength

Strong migration requires that all information about the state of the code to be migrated is passed with the code to its new location in order for it to execute correctly. Weak migration requires that the code alone is passed. No information about state is sent.

Vanilla functions are accompanied by their environment when they are sent to be executed elsewhere. This stores a good deal of information about the state of the function, as variables would have values associated with them and so on. Therefore the type of migration that takes place in Vanilla has a degree of strength associated with it.

5.8.3 Comparison

The type of code migration that has been shown here is similar in structure to the IBM Aglet [21] system. Aglets allows the programmer to specify that some code should move to another environment and execute there, which is precisely what has been implemented here. The `run(…)` method that has been implemented for Vanilla is similar to the `dispatch(…)` method in Aglets.

Both Aglets and Vanilla Mobility require that a special environment exist on any node on which a piece of migrating code wishes to run. In Vanilla this

means any server to which code is migrates must be Vanilla enabled. This contrasts with a mobile code implementation such as Voyager by ObjectSpace [22]. This system, which is based on the idea of a Virtual Object, allows migration to any Java runtime of another Virtual Object. This is outlined in greater detail in the chapter entitled *Background*.

5.8.4 Mobility Summary

Vanilla is a Mobile Code System based on the Remote Evaluation Design. A Vanilla Enabled server corresponds to a Computational Environment and any function could potentially be an Executing Unit, becoming one when it is actively migrated. The migration in the Vanilla MCS has a degree of strength associated with it due to the serialisation of the environment with the function. A Vanilla MCS is similar to the IBM Aglets system.

5.9 Summary

The four main aims of the dissertation have been achieved and can be seen to be working. The next chapter will briefly discuss where this work could lead to and what further work could be done using the work done for the purpose of this dissertation. Earlier sections alluded to potential applications for what has been achieved thus far. Further development could lead to further applications.

There now exists a system that can allow a language developer create a language to serialise values from his/her programming language to any Vanilla enabled web server’s file space. A basic mobile code system is now a part of Vanilla so should a programmer wish to create a language incorporating this functionality his/her task would be a great deal simpler using Vanilla.

6. Conclusion

6.1 Introduction

The *Evaluation* chapter has shown how the four stated aims of the dissertation have been achieved. It has also outlined a number of applications that are possible using this version of Vanilla. This chapter will summarise what has been achieved and briefly mention the potential for further development of the system.

6.2 Achievements

All the goals have been achieved. Serialisation and deserialisation of Vanilla values will take place without error, with some minor exceptions that will be outlined in the *Further Work* section below. Also all Mobile Code functionality has been tested and seen to be operating correctly, as was shown in the last chapter.

6.3 Further Work

A number of issues have arisen in the course of this dissertation. The resolution of these issues would be a priority for any further work on this version of Vanilla. Issues such as the handling of multiple return statements in a serialised function, the handling of nested environments and the completion of the object serialisation components in the system ought to be the main focus of any future effort. Subsequent to this it would be desirable to implement serialisation components for other values within Vanilla, examples of which are provided below in the section *Remaining Types*.

Vanilla is a very young system and as such is open to work in a number of areas. Many examples of such areas are also given below.

6.3.1 Current Issues

6.3.1.1 Handling of Multiple Returns

Most functions just return a single value, the value contained in the final statement of the function. An example of such a function is given below in Figure 6.1. However other functions return from multiple points within the code in the body of the function, as shown in Figure 6.2.

```
Function (Int x, Int y) Int multiply;
multiply = fun(Int x, Int y) (
     x * y;
);
```

Figure 6.1: Function with single return statement

```
Function (Int x, Int y) Int greater;
greater = fun(Int x, Int y) (
     if(x > y) (
          return x;
     )
     else return y;
);
```

Figure 6.2: Function with many return statements

As outlined in the *Implementation* chapter the serialisation of code, and therefore functions takes place by representing the Abstract Syntax Tree using XML. Deserialisation takes place by reading in the XML representation and reconstructing the Abstract Syntax Tree. When a function is encountered by the Vanilla interpreter a hook is put in the Environment to indicate the point to which the function should return when it is completed. If the function runs through to the end then this hook is not used. However if it escapes at some point within the code the hook is required. The Environment that is serialised does not include such a hook since it is only added by Vanilla when there is a

call to the function. In the mobile case the call to the function takes place with the execute statement, rather than a conventional call. The absence of the hook means that functions with multiple returns will not operate correctly when being executed directly from an XML file. This problem was encountered late in the implementation of the project, so a fix for it has not yet been implemented. It is expected that the problem could be resolved by the insertion of a hook at some point, although early attempts to insert this hook failed to make the functions run correctly.

6.3.1.2 Nested Environments

The current implementation of the Environment serialiser is built to handle the case where Environments have parents. For example should a language be designed where a function is located within another function then the environment of the nested function would be a child of the environment of the outer function. The serialiser that is implemented is built to handle such situations but has not been tested. The closest we have come to such a situation is with the object serialisation components, which are discussed in the next section.

6.3.1.3 Object Serialisation

An object is similar to a record in that both are just collections of fields. In a record the fields are all just primitive values or further records so serialisation of records amounts to little more than simply writing out a set of values. There are no further issues here. Objects differ from records in that the fields in an object can be a method rather than a primitive value. This ought not pose much of a problem since a means for serialising functions has already been implemented, and a method is simply a function that is associated with an object. The difficulty with serialisation of methods lies in the fact that each method must also have a reference to its associated object. This is what distinguishes it from a function.

What is required for the successful serialisation of objects is a means for treating methods and functions differently so that the relationship between a method and the object with which it is associated can be preserved. A good deal of this work has already been completed, and as alluded to in the

previous section we have implemented a means for serialising nested environments. The remainder of the implementation for basic objects will not require an enormous amount of effort. Further issues could arise when we attempt to deal with more complex types of objects.

The object model that as been implemented in Vanilla, according to the documentation provided with Vanilla *"provides a second-order polymorphic object model with covariant self types and imperative semantics. It is modelled directly on the second-order type systems and calculi of Abadi and Cardelli"*. An example of some object code is given below in Figure 6.3, alongside a small segment of XML code showing the current representation of this object in Figure 6.4. Note the inclusion of the `variance` variable and the `self` attribute for when a method includes a return type that matches the object itself. `self` is used in order to avoid recursive loops.

```
Type Object(X)
     {
        Int total;
        Function() X timesTen;
     } O;

O o = object(Y = O)
      {
         total = 0;
         timesTen =
            method( Y self ) fun()
            (
               Y y = clone(self);
               y.total = y.total * 10;
               y;
            );
      };
o.total = 10;
```

Figure 6.3: An Object in a Vanilla test language

```
<?xml version="1.0" encoding="UTF-8"?>
<vanilla-serialized-value>
    <vanilla-type>
        <Object>
            <object-field field-name="total"
              variance="0">
                <Integer/>
            </object-field>
            <object-field field-name="timesTen"
              variance="0">
                <Function>
                    <arguments-type/>
                    <return-type self="1"/>
                </Function>
            </object-field>
        </Object>
    </vanilla-type>
    <vanilla-value>
        <object-value>
          <!—Values as per usual, including environments
          -->
        </object-value>
    </vanilla-value>
</vanilla-serialized-value>
```

Figure 6.4: The above object in XML (edited)

6.3.2 Remaining Types

Although serialisation for the more common types and values that you would expect to find in a programming language has been implemented, there are still a number of extra types that are implemented in Vanilla that, as of yet, do not have any serialisation components implemented for them.
Examples of these are as follows:

1. Universals
    - Universally quantified (parameterised) types
        - These allow functions to take types as parameters, enabling the construction of functions and values that behave uniformly across a family of types.
2. Existentials
    - Existentially quantified (abstract) types
        - May be used to model certain kinds of partially abstract data types.
3. Autos
    - Run-time type information
        - Values that combine a value with a type. The types may be examined at run-time and used to extract the value.
4. Mu pod
    - Recursive types

Extensions to the current Vanilla serialisation model should include the capability to serialise these types and values of these types.

6.4.2 Vanilla

Should all the serialisation components be implemented fully and correctly the next step would be to implement some extra functionality that could for example

1. Allow a client access an arbitrary XML document and treat it as a Vanilla value. For example if some XML document included dental records it could be read in and treated as if it were a Vanilla `Record` value. This could add massive power to Vanilla programming languages in that any web page could conceivably be treated as if it were a value. The whole web is then a set of values.
2. Through associating the current Vanilla architecture with a truly open architecture such as future versions of Jiki we could be able to write any value as a web page to anywhere on the web. This is possible at

the moment on Vanilla enables web servers, where the web server contains a servlet repository for Vanilla servlets. In the future a truly open architecture could allow values to be read and written all over the web as web documents, allowing large amounts of sharing of data and collaboration.

3. If such a repository of objects were to exist checks would need to be made to avoid duplication of objects or values. It may also be necessary to maintain links between documents to specify relationships between objects. In order to do this an XML technology such as XLink or XPointer could be used. An XML namespace for these documents should also be created. This could help distinguish between the semantics of types/values created by different programmers.
4. In the future it is hoped that using Vanilla languages could be downloaded from the web with programs so that as well as writing a program the developer could write a language to run the program. In much the same way as an applet is downloaded the code and language could be downloaded together and run.

Vanilla is a very young system with great potential for the future. There are several different ideas that could be implemented using Vanilla. It is hoped that a large Vanilla community will develop and that the work taking place using Vanilla will increase in order to see how these ideas may or may not work.

6.4 Summary

This chapter has summed up how languages built using Vanilla will now have access to the disk of the machines on which they are running. In addition to this, programs written using these languages will be able to read and write XML documents from around the World Wide Web and treat them as Vanilla values. Vanilla languages will also be able to incorporate a basic level of code

mobility into their functionality. This will allow code written in Vanilla to be migrated between Vanilla enabled machines.

The above paragraph outlines the achievements of this dissertation. Rather than being the end of the work, it only serves to show what can be done with Vanilla. It can be seen from the basic mobile code system that there is great potential for further related work in this area, as detailed in the previous sections.

The research outlined in this dissertation have proven the concept, shown that serialisation of types and values as well as code can be done using Vanilla and examples given have displayed how the achievements, when reached, were used and could be used in the future.

7. References

1. Simon Dobson, Paddy Nixon, Vincent Wade, Sotirios Terzis and John Fuller, *Vanilla: An Open Language Framework.* Generative and Component Based Software Engineering, 1999.
2. Don Batory, Bernie Lofaso, Yannis Smaragdakis, *JTS: A Tool for Implementing Domain Specific Languages.* Proceedings of the 5th International Conference on Software Reuse, 1998.
3. Andrew W. Appel, *Modern Compiler Implementation in Java*. Cambridge, 1998.
4. Tim Bray, Jean Poli, C.M. Sperberg - McQueen, *Extensible Markup Language (XML) 1.0*, W3C. 1998.
5. Jonathan Robie, *What is the Document Object Model?*, W3C. 1998.
6. DeRose, Maler, Orchard, Trafford, *XML Linking Language (XLink) 1.0*, W3C. 2000.
7. Eve Maler, Steve DeRose, *XML Pointer Language (XPointer)*, W3C. 1998.
8. David C. Fallside, *XML Schema*, W3C. 2000.
9. Adler, Berglund, Caruso, Deach, Grosso, Gutentag, Milowski, Parnell, Richman, Zilles, *Extensible Stylesheet Language (XSL) 1.0*, W3C. 2000.
10. Tim Bray, Dave Hollander, Andrew Layman, *Namespaces in XML*, W3C. 1999.
11. Norman Walsh, *What is XML?* www.xml.com. 1998.
12. Jon Bosak, Tim Bray, *XML and the Second Generation Web.* Scientific American. 1999.
13. Box, Kakivaya, Layman, Thatte, Winer, *SOAP: The Simple Object Access Protocol*, IETF. 1999.
14. Jiki, www.jiki.org.
15. Tim Walsh, Paddy Nixon, Simon Dobson, *As strong as possible mobility: An Architecture for stateful object migration on the Internet.* Trinity College Dublin.

16. Tim Walsh, Paddy Nixon, Simon Dobson, *Review of Mobility Systems*, Trinity College Dublin Computer Science Technical Report. 2000.
17. David Kotz, Robert S. Gray, *Mobile Code: The Future of the Internet*, Department of Computer Science / Thayer School of Engineering, Dartmouth College: Hanover, New Hampshire. 1999.
18. Gianpaolo Cugolo, Carlo Ghezzi, Gian Pietro Picco and Giovanni Vigna, *A Characterization of Mobility and State Distribution in Mobile Code Language*, Proceedings of the ECOOP Workshop on Mobile Objects. 1996.
19. Alfonso Fuggetta, Gian Pietro Picco, Giovanni Vigna, *Understanding Code Mobility*, IEEE Transactions on Software Engineering. 1998.
20. Joseph Kiniry, Daniel Zimmerman, *A Hands-On Look at Java Mobile Agents*, IEEE Internet Computing. 1997.
21. Danny B. Lange, *Java Aglet API White Paper*, IBM Tokyo Research Laboratory. 1997.
22. Graham Glass, *Reducing Development Effort using the ObjectSpace Voyager ORB*, ObjectSpace. 1999.
23. Sun Microsystems, *Java Object Serialization Specification.* 1998.