Department of Computer Science
University of Pretoria
Pretoria
South Africa

A VIRTUAL MACHINE FRAMEWORK FOR DOMAIN-SPECIFIC LANGUAGES

by

David Fick

February 2007

Submitted in partial fulfillment of the requirements for the degree
Master of Science (Computer Science) in the Faculty of
Engineering, Built Environment and Information Technology

University of Pretoria
Pretoria, South Africa

Supervisor: Professor B. W. Watson
Co-supervisor: Professor D. G. Kourie

Summary

A Virtual Machine Framework for Domain-Specific Languages

By David Fick

Supervisor: Professor B. W. Watson

Co-supervisor: Professor D. G. Kourie

Department of Computer Science

Master of Science (Computer Science)

Experts in a field regularly apply a defined set of rules or procedures to carry out a problem-solving task or analysis on a given problem. Often the problem can be represented as a computer model, be it mathematical, chemical, or physics based, and so on. It would certainly be advantageous for a domain expert who is not proficient in software development to express solutions to problems in a domain-specific notation that can be executed as a program. Many new ideas aim to make software development easier and shift the development role closer to the end-user. One such means of development is the use of a small, intuitive programming language called a Domain-Specific Language (DSL.)

This dissertation examines a generic approach to constructing a Virtual Machine (VM) to provide the runtime semantics for a particular DSL. It proposes a generic, object-oriented framework, called a VM Framework, in which to build a VM by subtyping abstract instruction and environment classes that are part of the VM Framework. The subtyped classes constitute an environment and an interface called an instruction set architecture and the instructions can access and operate on the environment in a deterministic way to provide the runtime semantics of a DSL program. Both instruction classes and environment classes encapsulate functionality of an existing domain, represented programmatically as a namespace construct. The namespace is home to related classes that provide the various concepts inherent of a domain. These are concepts understood by a domain expert and in this dissertation it is shown how they are exposed as DSL constructs.

With the use of compiler writing tools, a compiler can be created for a DSL that generates an appropriate instruction sequence that can be executed by the VM. The grammar of the DSL is shown to feature constructs that allow a domain expert to express concepts of the underlying domain in an intuitive manner. The dissertation details how a VM is configured for a specific set of instructions and an environment. Instruction sets and environments can be extended creating VMs with additional semantics for DSLs that are similar, or contain subsets of semantics of other DSLs. The languages are intended to be intuitive and it is shown using examples how a specific DSL program is mapped to an instruction sequence with the

instruction set architecture and environment in mind. Comparative performance in relation to other DSL implementations, including a hard-coded approach of a VM and an interpreted approach are also provided. The VM Framework is proven to be most effective in rapidly prototyping a DSL for a particular problem domain. The dissertation also provides examples of DSLs such as a real-valued expression language and a scene description language that uses a ray-tracer for rendering geometric objects onto a canvas. It is shown how the scene description language is an extension to the real-valued expression language in terms of their underlying VMs. All DSL grammars are provided.

Keyterms: domain expert, domain-specific language, virtual machine, instruction set architecture, environment, language prototyping, generic framework, syntax, semantics, comparative performance.

Abstract

Software engineering professionals are frequently faced with the challenges of applying good design principles in an environment that is often lacking in consistency and discipline. Those experienced in industry will argue that it is mostly due to processes that are not always followed and problems that are rather demanding on the developer's knowledge gained from experience and general expertise. Also, developers with a lesser degree of experience who are required to improve upon, or implement a new set of requirements specified by system level designers are faced with steep learning curves. While system designers play very little role in the actual implementation of systems, it is up to the developers to wade through copious amounts of source code and documentation, first understanding a system's behaviour, before any further coding can commence.

It is not just software practitioners, but also *end-users* like experts in a particular scientific or engineering domain who experience similar problems. These experts are not necessarily versed in software design, yet may still require a scientific means to express solutions in a particular problem domain, and it would certainly be to their advantage if they did not concern themselves with the seemingly limitless amount of complexities in software.

Many new ideas aim to make software development easier and shift the development role closer to the end-user. This dissertation investigates a software development strategy, which entails the design of a Domain-Specific Language (DSL) that is tailored towards a particular problem domain, and a Virtual Machine (VM) to support execution of a program written in the DSL. Since DSLs have the characteristic of being highly end-user oriented, they are favourable to experts who have a thorough understanding of their problem domain since they provide an elegant means to express their solutions easily. Different techniques to support DSLs are examined, namely an interpreter, a hard-coded VM, and a VM Framework. The VM Framework is proven to be most effective in rapidly prototyping a DSL for a particular problem domain.

Throughout this dissertation, example DSLs are used to illustrate some of the concepts that are discussed. For each DSL example some details are shown, such as their grammars and results of executing a program written in the DSL. The DSL grammars and example programs are all included in the appendix. The implementation of the interpreters, hard-coded VMs and VMs based on the VM Framework for each DSL are all included on the accompanying CD.

Acknowledgements

My sincere thanks to:

- My parents who believed in me, and made me believe in myself.

- All my friends and family for their support and encouragement.

- My supervisor, Prof. B. W. Watson, for sharing his tremendous technical wisdom, and his enthusiasm in assisting me.

- My co-supervisor, Prof. D. G. Kourie, for all his time spent correcting my grammar and providing me with invaluable criticisms, advice and support.

Introductory artwork by M. C. Escher.

Table of Contents

1 Introduction 12
1.1 Development Language 13
1.2 A Look at Domain-Specific Languages 14
1.2.1 The Concept of a Domain in a DSL Context 14
1.2.2 The Concept of a Language in a DSL Context 15
1.2.3 The Language Syntax 16
1.2.4 The Language Semantics 17
1.3 Criteria for Evaluation 18
1.3.1 Scalability 18
1.3.2 Performance 18
1.3.3 Modularity 19
1.4 Related Work 19
1.5 Summary 20

2 Development Environments for Domains 21
2.1 CASE Tools of Today 22
2.2 A VM as Part of the CASE Software 22
2.2.1 Languages in a DSL-Supporting IDE 23
2.2.2 A DSL Development IDE 26
2.3 Summary 30

3 Domain-Specific Languages and Interpretation 31
3.1 DSL Translation 32
3.2 DSL Interpretation 32
3.3 Shlaer-Mellor Domains 33
3.4 Configuring Shlaer-Mellor Domains with Bridges 33
3.5 View of the Interpretation Approach 34
3.6 The Ray-Tracer DSL 35
3.6.1 The RT Namespace 36
3.6.2 The RT Language 39
3.7 Scaling up the RT DSL for the Interpreter Approach 42
3.7.1 Adding a Construct to an Interpreter 42
3.7.2 Components Requiring Alteration 43
3.8 Modularity 44
3.9 Summary 45

4 Domain-Specific Virtual Machines 46
4.1 VM vs. Code Generator 46
4.2 Virtual Machine Instruction Set Architecture 47
4.3 System Virtual Machines versus Process Virtual Machines 48
4.4 View of the Virtual Machine Approach 50
4.5 Scaling up the RT DSL for the VM Approach 53
4.5.1 Adding a Construct to a VM 54
4.5.2 Components Requiring Alteration 54
4.6 Modularity 57
4.7 Summary 57

5 Domain-Specific Virtual Machines: A More General Approach 59
5.1 Compiled Program Parsing and Loading 60
5.2 Instruction Set and Environment Configuration 60
5.3 Program Execution 61
5.4 Output Results 62
5.5 View of the Virtual Machine Framework Approach 63
5.6 Virtual Machine Framework Detailed Design Description 64
5.6.1 The EVM Class 65
5.6.2 The Config Class 66
5.6.3 The Loader Class 67
5.6.4 The Inst Abstract Class 68
5.6.5 The Env Abstract Class 68
5.7 Environments 69
5.8 Instructions 70
5.8.1 Instructions with No Operands 71
5.8.2 Instructions with a Branch Label 72
5.8.3 Instructions with a Temporary 73
5.8.4 Instructions with a String Parameter 73
5.8.5 Instructions with a Number Parameter 74
5.9 Summary 74

6 Extending the Virtual Machine Framework 76
6.1 The Configuration File 77
6.2 Extending the Environments 78
6.3 Extending the Instruction Sets 78
6.4 The extended DSL VM 80
6.5 Building a DSL 82
6.5.1 The EXP DSL 83
6.5.1.1 The EXP Domain 83

6.5.1.2 The EXP Language 83
6.5.2 The Little Quilt DSL 86
6.5.2.1 The LQ Domain 86
6.5.2.2 The LQ Language 87
6.6 Scaling up the RT DSL for the VM Framework Approach 90
6.6.1 Extending EXP to suite the RT language 91
6.6.2 Adding a Construct 92
6.6.3 Components Requiring Alteration 92
6.7 Modularity 93
6.8 Summary 94
6.8.1 What the VM Framework is Not Intended to Provide 94
6.8.2 What the VM Framework Does Provide 95

7 Comparative Results 97
7.1 Summary 100

8 Future Work 102
8.1 Graphical User Interface for the Virtual Machine Framework 102
8.2 Multiple Environments and Namespaces 103
8.3 Architectural DSLs 104

Conclusion 106

Appendix 107

References 121

Figures

Figure 1. High-level view of a DSL-supported IDE. 24
Figure 2. Generic VM used in the context of a DSL IDE. 28
Figure 3. Conceptual view of the Interpretation Approach. 35
Figure 4. Information Model of the RT namespace. 36
Figure 5. Determining the type associated with an AST node. 38
Figure 6. Actual interpretations of the AST nodes. 39
Figure 7. Lexer input for RT. 40
Figure 8. Parser input for RT. 40
Figure 9. Example scene described in RT. 41
Figure 10. Result of rendering the scene defined in RT. 42
Figure 11. Appending the lexer definition for keyword ambient. 43
Figure 12. Altering the grammar definition for the new *ambient* construct. 43
Figure 13. Altering the functions for the new *ambient* construct. 44
Figure 14. Instruction Set Architectures – System ISA and User ISA. 48
Figure 15. A System VM. Virtualizing software between hardware and OS. 49
Figure 16. A Process VM. Virtualizing software supports an individual process. 50
Figure 17. Broad classification of VMs. 50
Figure 18. The DSL VM relative to the HLL VM on which it was written. 51
Figure 19. Conceptual view of the VM Approach. 52
Figure 20. Lexer definition for the RT VM. 52
Figure 21. Parser definition for the RT VM. 53
Figure 22. Altering the grammar for the new *ambient* construct. 55
Figure 23. Adding the Ambient instruction to the VM's lexer definition. 55
Figure 24. Parsing the new Ambient instruction. 56
Figure 25. The semantics for the new Ambient instruction in function RT_AMB() 56
Figure 26. Main loop for executing loaded instructions. 62
Figure 27. The GetResult() method of the EVM class. 63
Figure 28. Conceptual view of the VM Framework Approach. 64
Figure 29. Information Model of the VM Framework namespace. 65
Figure 30. Configuration mappings. 66
Figure 31. Example EnvExp class. 70
Figure 32. Token definitions and grammar for the instructions. 71
Figure 33. The Inst_OpCode abstract class. 71
Figure 34. The Add instruction. 72
Figure 35. The Inst_OpCode_Br abstract class. 73
Figure 36. The Inst_OpCode_ID abstract class. 73
Figure 37. The Inst_OpCode_Str abstract class. 74

Figure 38. The Inst_OpCode_Num abstract class. 74
Figure 39. Example configuration file for EXP. 77
Figure 40. The EnvRT class. 79
Figure 41. Configuration file to setup RT, borrowing instructions from EXP. 80
Figure 42. Defining an environment and instruction set for a DSL VM. 81
Figure 43. ISA_{LQ} is a subset of ISA_{EXP}, with new instructions. 81
Figure 44. ISA_{RT} is a subset of ISA_{EXP}, with new instructions as before. 82
Figure 45. Lexer definition of EXP. 84
Figure 46. Parser definition of EXP. 84
Figure 47. Programmatic representation of summation expression (1). 85
Figure 48. Compiled program of the summation expression (1) 85
Figure 49. The namespace of the LQ. 87
Figure 50. The configuration file for LQ. 87
Figure 51. The lexer definition of LQ. 88
Figure 52. The grammar definition for LQ. 88
Figure 53. A function defined in LQ. 89
Figure 54. Translation of a function defined in LQ. 89
Figure 55. LQ using the VM Framework in the context of a GUI. 90
Figure 56. The EnvExp environment. 91
Figure 57. The EnvRT environment as an extension to EnvExp. 92
Figure 58. The Ambient instruction extends Inst_OpCode (no operands.) 93
Figure 59. Adding the Ambient instruction to the instruction set. 93
Figure 60. Comparative performance results of the DSL implementations. 99
Figure 61. Communicating DSL infrastructure. 103

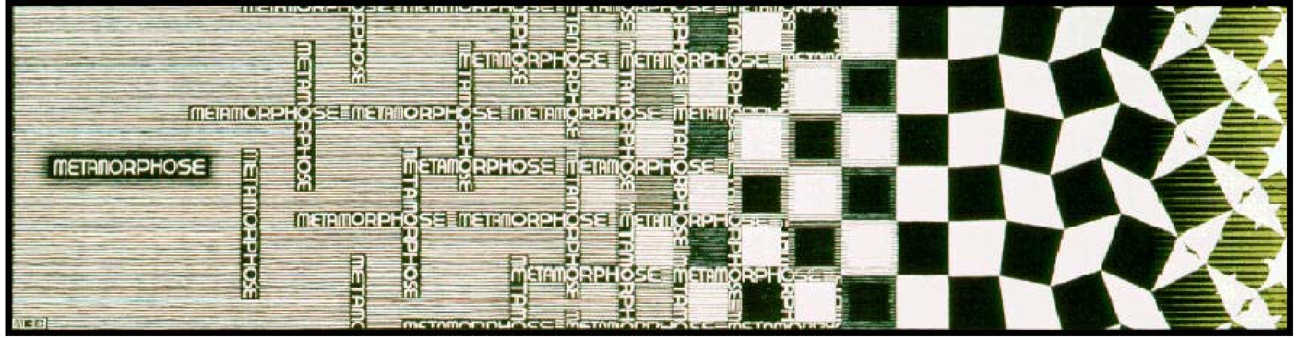
Tables

Table 1. Comparing IDEs and their use of language types. ... 27
Table 2. Components for the Ray-Tracer Interpreter... 35
Table 3. Source files for the RT namespace. ... 37
Table 4. Summary of criteria. ... 100

Source Listings

Listing 1. RT DSL complete lexer definition. ... 107
Listing 2. RT DSL complete grammar definition... 107
Listing 3. RT example. ... 113
Listing 4. EXP DSL complete lexer definition... 114
Listing 5. EXP DSL complete grammar definition. ... 114
Listing 6. EXP example... 117
Listing 7. LQ DSL complete lexer definition. ... 117
Listing 8. LQ DSL complete grammar definition... 117
Listing 9. LQ example. ... 120

1 Introduction



Programming languages were invented to make machines easier to use [Set97], but the word *invented* should perhaps be elaborated on slightly. Scientists and others regularly reuse sound pieces of functionality in defining a concept or a solution to a problem, whether it is an iterative algorithm, an inductive proof or a mathematical formula. One of the first industrially accepted programming languages was Fortran [Bac81, Wex81], widely embraced by the scientific and engineering community since it allowed formulas to be written in the traditional mathematical sense, instead of coding them directly at the machine level. These formula expressions were automatically translated into machine level instructions. Over the years, other recurring concepts have been encapsulated, either in the form of a library or a language, producing languages aimed at the next generation of developers. Good examples of both include the OpenGL API, a graphical library that is highly portable and available for most Windows and UNIX platforms, and the Prolog language, which does a successful job of encapsulating first-order predicate logic into concise formalisms that support logical inference.

The most logical next step would be a software development process that facilitates or aids encapsulating commonly recurring functionality into a language, and, over time, lets the language evolve toward a more mature state. This is essentially what this dissertation is all about: the provision of a platform or framework, that will support the development of Domain-Specific Languages (DSLs) and that will easily accommodate the natural tendency for them to evolve over time.

This chapter begins by identifying the programming language and development environment used for the experimentation of DSL development. The notion of a DSL is examined in some detail in Section 1.2, while Section 1.3 proposes the criteria that will be used in this study for

evaluating DSL development methods. Related work is mentioned in Section 1.4, and the matters are summarized in Section 1.5. Chapter 2 illustrates DSLs in the context of some development environments. Chapters 3 to 6 examine the different techniques to provide runtime semantics of a DSL-program: Chapter 3 looks at interpreters, Chapter 4 looks at hard-coded Virtual Machines (VMs) and Chapters 5 and 6 present a framework-oriented approach. The framework, referred to as the VM Framework, provides ease of creation and modification of various software building blocks for the semantics of some DSL definition. It also allows for the creation of a compiler for a defined DSL that is targeted towards these semantics. Chapter 7 provides some comparative results between DSL development techniques, i.e. interpreters, hard-coded VMs and the VM Framework. Chapter 8 provides some work yet to be explored and follows with a conclusion to this dissertation.

1.1 Development Language

This study is centered not merely on the design of DSLs. It also represents a quest for an effective approach to support DSL development and DSL modifications in a particular software environment. Of course, any existing imperative programming language can be used for DSL development, but to further justify the usefulness of DSLs, the illustrating language should also be mature enough to allow a developer to build language constructs quickly and easily. Constructs simply refer to any syntactical features offered by the language itself, such as loops or use-defined declarations of functions or types, and are often identified by the occurrence of *keywords* of the language. To that end, the illustrating platform chosen is the Microsoft .NET platform, and the illustrating language used is the C# language. An excellent introduction to C# is provided in [Lib02]. This is not to say that other platforms could have been used with equal success in this study. A somewhat desirable characteristic of the .NET platform is the exposure of the language compiler itself in the form of an Application Programming Interface (API). This makes language and compiler development a fairly streamlined task.

Central to the .NET platform is the Common Language Runtime (CLR) system. Any language compiling to the CLR can invoke other modules compiled with a different programming language. Thus, later chapters that discuss building software for instruction extensions, and environment extensions are not necessarily restricted to C#, but may also include Visual Basic .NET, C++ .NET, and more domain oriented languages such as the .NET variant of COBOL.

1.2 A Look at Domain-Specific Languages

Before attempting to justify what is meant by *language* in DSLs, a justification on what is meant by *domains* should be fairly appropriate at this stage, since the DSL designer needs to keep a clear picture of the domain in mind, before attempting to expose it as a language. The DSL designer also needs to perform a thorough analysis of the domain to identify the *features* of the domain, and the dependencies among them [vDeu01a]. While it is also possible to describe the definition of a domain both pragmatically and scientifically, the DSL designer is more interested in defining domain concepts that map to sequences of instructions in a conventional programming language. Ultimately, the DSL designer would like to quickly and easily build up a prototypical language for end-users, just as much as end-users would like to quickly and easily get started using a DSL to solve their problems in their respective domain. In this dissertation, the actual DSL-program source code, written in the language of the DSL, is referred to as the DSL-program. The definition of a DSL that follows is taken from [vDeu98c].

DSL : A Domain-Specific Language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to a particular domain.

DSL-program : A program, written in the language of the DSL, i.e. the actual DSL program *source code*.

There is a natural choice between using an industrial language, and resorting to using a DSL. In this dissertation, through some examples and empirical experimentation with DSLs and the design of the underlying execution mechanisms, the text will attempt to expose their usefulness in a scientific context, as well as their use as a software-engineering tool. While industry always has a myriad of complexities in software design and maintenance, industrial strength languages will always be part of the software engineering process. There is no reason why the two disciplines cannot be merged together and used in harmony to form a more effective means of software design.

1.2.1 The Concept of a Domain in a DSL Context

Simply put, a domain is characterized by a set of related concepts, which need to be understood, before they can be applied to real world problems. For example, Chemistry is a domain in which the concepts of *elements*, *compounds*, and their respective *reactions*, are

among those that are to be understood by chemists if they are to be considered experts in the domain. An example of a DSL for dealing with financial concepts, specifically financial products, is the DSL *RISLA*, described in [Arn95, vDeu97]. This notion of a domain is reflected in the Shlaer-Mellor design methodology [Mon95, Shl92, Shl96]. In keeping with a more pragmatic mindset, developers who regularly exercise the UML [Ale01, Bjö02, Gur03], would normally group related classes in packages, and again in [vDeu97] a discussion on an object-oriented frameworks is provided, which is analogous to this concept of a domain. Developers who are more favorable to the Shlaer-Mellor design methodology, group all related classes within an explicit domain. The Shlaer-Mellor design approach evolved out of UML in an attempt to solidify some of the rules or policies of UML, which either didn't exist or were still not well understood by many developers. It can be said that Shlaer-Mellor is a summarized variant of UML, that boasts a fair amount more user friendliness than UML and that is well suited for asynchronous systems. While asynchronous systems are beyond the scope of this dissertation, what will be of use is Shlaer-Mellor's definition of a domain.

> **Domain** : A domain is a set of classes, all connected via relationships of some multiplicity. No two domains use the same classes, ie. Classes do not overlap between domains. A domain can be dependant on, or make use of services of other domains.

In essence then, a domain, from a developer's perspective, is simply a home where related concepts can live happily together. The C# `namespace` construct groups together related classes, interfaces and enumerations. The C# namespace is well aligned to the concept of a Shlaer-Mellor domain. Later in this dissertation, when the possibility of constructing DSLs for various domains will be explored, each domain will be associated programmatically with a C# namespace and thus will represent the domain of the DSL.

1.2.2 The Concept of a Language in a DSL Context

Delving into the language side of DSLs, this is where the end-user requirements start to play a role. Before embarking on designing a DSL it is important to first note, and understand, two separate concerns centered on the design of a DSL. They are the *syntax*, and the *semantics* of the language. One obvious requirement of a DSL is that the syntax of such a language should be simple and intuitive. The main driving point behind end-user software is to relieve a developer or expert from the burden of developing around existing software that either he or she has little knowledge about or is cumbersome and shouldn't concern the individual at all [vDeu98a, vDeu98c]. Very often a DSL's simple constructs map to a large amount of

underlying processing. This mapping determines the meaning or semantics of the DSL's constructs. Many DSL-compilers merely generate code and therefore provide the semantics in a high-level language. In such instances, the intention is that the generated code again be compiled along with other existing code, a library, or a software framework, for a complete application to be produced [Cza99, vDeu01b].

In contrast, in the current work, a VM was incorporated as a means of executing a variety of DSL-programs. The VM exposes an existing namespace as a set of instructions and an environment, and once a particular DSL has been defined along with a corresponding compiler, then a program written in that DSL can be translated into a sequence of instructions that the VM can execute. For this dissertation, any reference of a VM created using the VM Framework shall be called a VM instance since it amounts to an object instance from a programmatic viewpoint. A more detailed description of how this is accomplished, as well as the finer details of VMs, will be discussed in Chapters 4 to 6. It will be seen that this will support solutions to the problem of effectively designing and implementing DSLs in such a way that an expert can later reconfigure them. Effective DSL design is therefore not seen as a once-off activity. Instead two distinct development roles are investigated: that of designing the original form of the DSL; and later, the domain expert role of redefining the language.

1.2.3 The Language Syntax

As mentioned before, the syntax needs to be intuitive. Syntax design should take place in the context of a few consultations with an expert, because the design shouldn't be taken for granted. Though the use of a specific construct might seem fairly obvious to a DSL designer, it may be too cumbersome or verbose for the expert. Language syntax should also be designed with the aim of reducing programmatic errors and to have increased programmatic error detection. This implies keeping the syntax concise and doing away with elements of syntax that cause unnecessary confusion in the readability of a program. Thus syntax design needs to be loosely coupled from the semantics and may well be reworked a few times before the end-user is wholly satisfied. The syntax design of the DSL, in this dissertation, is kept simple, and, in fact, doesn't need to be anymore complicated. The requirement here is to derive a very simplistic compiler capable of translating a DSL source into a target machine language. A lexer generator, and parser generator are needed that are able to generate C# code. The two tools, which can provide this functionality, were chosen for the .NET environment. They are aptly named LexerGenerator (LG) and ParserGenerator (PG) [Cro03]. It is important also to note that static semantic checks, such as static type checking, are a concern of the compiler and any static semantic checking routines would need to be implemented in the compiler.

1.2.4 The Language Semantics

Runtime semantics, as opposed to static semantics, refer to the noticeable effect of executing a DSL-program. At the formal level there are different methods to specify the language semantics. They include informal semantics, attribute grammars, operational semantics, denotational semantics and natural semantics [Set97].

For the VM Framework described later in Chapters 5 and 6, specification of the semantics is a combination of informal semantics and operational semantics. Since the VM Framework already deals with compiled C# code, the semantics are more illustrative at the operational level. All semantics is encapsulated in instructions and executed by the VM instance.

Instructions that are executed can update an environment that is registered with the VM instance. The environment referred to here is not the conventional environment that contains a mapping of variables to values. Rather it is a custom data structure whose internals are known only by the instructions themselves, and not to the VM instance itself. However, it is also possible to include a separate environment in the compiler if the language is to cater for local variables.

As will be seen, it is the obligation of the DSL designer to extend the VM Framework with an instruction set and a related environment, and to further configure the VM instance using a separate configuration file that links each instruction to the environment. The configuration allows the DSL designer to reuse instructions with different environments, and to extend environments to have additional functionality. Each instruction and environment is stored as a .NET Dynamically Linked Library (DLL) on disk. The instruction DLLs and environments are loaded at runtime depending on the VM instance's configuration.

Tying the two concepts of syntax and semantics together is the challenge behind rapidly prototyping DSLs, and it is in the VM Framework presented in this dissertation that the approach is simplified. The internal structure of the VM Framework shall unfold in later chapters.

To place DSLs in their context, this dissertation shall assume the design and implementation of DSLs in a software-engineering context. The terms *developer*, *practitioner* or *domain expert* will sometimes be used interchangeably, but they simply refer to the end-user of the DSL who is eager to make use of the resultant DSL built by the DSL designer. Thus development moves away from programming in the small, to programming in the large. The aim is not to restrict a detailed discussion on DSL design purely for a single use, i.e. building a product for an end-user [Arn95], but also to allow developers in industry to embrace some of the techniques

discussed and apply them to real world software engineering practices. In the latter case, the DSL that results is not targeted to a specific end-user, but also to be reapplied back into a developer's own working environment, whether they be designing software for embedded systems, databases, visual applications, mathematical modeling applications, and any other domain inherent in a particular field. Most of these applications are based on well-defined concepts, and so they can indeed benefit from DSLs, and in so doing improve the developer's software development practice.

1.3 Criteria for Evaluation

To make any significant comparison between the various DSL development techniques, a certain set of criteria is needed. Three useful criteria are identified and defined in this section; they are *scalability*, *performance* and *modularity*. They correspond to important properties that go hand in hand with good software design. Another somewhat important property associated with languages is *portability*. However this dissertation assumes a .NET development context, and therefore it only makes sense to consider all platforms that support the .NET environment. The three previously mentioned properties are studied further at the DSL level.

1.3.1 Scalability

Scalability describes the readiness for a system to grow, without significant loss in efficiency and effectiveness. In the context of DSL design, such growth could include new constructs that are added to the language itself, as well as new functional code that is added to the underlying namespace, such as new functions, classes etc. It is often difficult to focus on building scalable software in the time allocated to a given project. Even when developers do manage to build scalable software, it is often at a performance cost, so there tends to be a tradeoff between scalability and performance.

1.3.2 Performance

Performance can either refer to the formally derived polynomial or non-polynomial complexity of a computation or algorithm or simply the time elapsed for a computation or algorithm to complete. Throughout the dissertation, when a measure for performance is required, a simple time elapsed will be checked at corresponding points of execution.

1.3.3 Modularity

Modularity will be described in terms of the DSL itself. This refers to the interpreter, the hard-coded VM and a VM instance created from within the context of the VM Framework. One can ask how reusable is the interpreter or the VM? Supposing a DSL is developed, and the environment defined for the DSL has a certain set of inputs and outputs. It may be a desirable property to be able to treat the VM as a component, just like a UML object, and to allow it to interact with other components of a bigger system.

Quite often, in object-oriented languages, it is considered good practice to design and to build classes as reusable entities. Thus in object-oriented languages, use of constructs such as abstract classes and polymorphism are encouraged to add extra degrees of reusability. Many times though, it is not easy or even possible for that matter to design all classes to be reusable, because many class have very specific functionality or behaviour and are perhaps only suited in a single context.

In the Shlaer-Mellor context it is the namespaces, and not the classes within the namespaces, that are considered the reusable entities. Ideally, a domain can be used over and again in various contexts without changing the code or classes found inside the namespace, but only making alterations to the namespace's so-called bridge, discussed in more detail on Chapter 3.

1.4 Related Work

There is an interest in building languages rapidly, and consequently there is a need for rapid prototyping of language syntax and semantics, and more so, the ease and efficiency at which this can be achieved in an industrial environment. High-level languages such as C#, C++ or Java have a standard syntax, and are not prototypical languages as such. Though powerful in their nature, the need to build DSLs is driven by a necessity to have smaller languages that are more expressive. Environments such as the ASF+SDF MetaEnvironment [vdBra01] allow a user to freely alter a prototypical language's syntax, and provide semantic rules for the language, and in so doing allow a rapid form of language prototyping, well suited for building DSLs. However, the mapping from syntax to semantics in the case of the MetaEnvironment is at a more formal level, and equations are used to define the mapping. The usefulness of DSLs is further explored in Aspect Oriented Programming (AOP) [Irw97, Kic96], and consequentially, their affiliation to aspect weavers. They don't rely on a VM. Rather, they fuse a set of DSL-programs together and generate code in a target language, such as Java.

1.5 Summary

This chapter provided an overview of DSLs and briefly discussed the Shlaer-Mellor programming methodology, since it forms a good basis onto which DSLs can be conceptualized, i.e. a custom DSL built on top of an existing domain or a domain that is yet to be designed. The domain in the context of an expert is analogous to the knowledge in which he or she is proficient, and which he or she wishes to describe in a syntactic, communicative manner. Used in the context of a DSL, the domain is associated programmatically to a namespace where all the concepts, in the form of modules (functions, classes etc) are housed. Also, DSLs were discussed in terms of their syntax, and in terms of the semantics. The semantics forms the link between the syntax and the underlying namespace. Three criteria for evaluating a DSL were described and will be used throughout this dissertation; they are scalability, performance and modularity. Later, in Chapters 3, 4 and 5, as three different types of DSL techniques are described, these criteria will provide us with some measure of how suitable each of the different types of techniques are, especially when a DSL is required to be altered to suite new requirements.

2 Development Environments for Domains

Most state-of-the-art software development suites that are available at present enable a fairly streamlined means in which to develop software, especially in a Windows environment, using Rapid Application Development (RAD) techniques. Aspects of these RAD techniques have been encapsulated and made available to developers in specialized software known as Computer-Aided Software Engineering (CASE) tools [Pre97, Whi98]. A concept associated with the notion of CASE tools is that of Integrated Development Environments (IDEs). This refers to the visual appearance or layout of the CASE tool. The layout impacts the usability of the CASE tool, as well as the tool's ability to expose its intended software engineering features in a practical manner that simplify software development further. CASE tools have been used with great success in industry. These tools have also been widely adopted because of the ease that they offer in building subcomponents of applications. The IDE of a tool, however, refers more to the development environment exposed to the developer, as opposed to other aspects of a CASE tool that support aspects such as design, documentation etc. IDEs are usually unique to CASE tools. They offer a variety of features, mainly visual in nature, to build parts of a system that seem to warrant a more interactive approach to its construction. The rest of the chapter touches on CASE tools, as there is much room for incorporating the forthcoming concepts, particularly DSLs and VMs, into CASE tools to further improve their interactive approach to software development. The chapter starts by briefly introducing some existing CASE software, and Section 2.2 identifies some key areas as suitable candidates for incorporating DSLs and VMs. Section 2.3 provides the summary.

2.1 CASE Tools of Today

It is important to note that CASE is not necessarily a systems engineering methodology. Rather it is the application of software technology to support the inner processes, strategies and disciplines of *various* such methodologies. The definition of CASE, adopted from [Whi98], is provided below. The use of a VM in the context of such technologies is provided in the next section.

> **CASE** : CASE is the application of information technology to systems development activities, techniques and methodologies. CASE tools are programs that automate or support one or more phases of a system's development life cycle. The technology is intended to accelerate the process of developing systems and to improve the quality of the resulting system.

Most CASE tools enable rapid ways to build visual components such as Windows Frames or database table layouts. Some go so far as to visualize the design-time structure of a program or application. Development environments such as BridgePoint and Artisan allow a high-level view of an application under development using Shlaer-Mellor or UML diagrams. For example, they might represent graphically an active object's state machine using appropriate UML symbols instead of the corresponding source code for that object, though it is natural to enquire how the programmatic representation appears. Once a program has been developed fully, or perhaps even at early stages of development, it is possible to run the program within a simulated context with a fixed set of inputs and verify the correctness of the program's logic through repeated testing using a predefined set of test cases. It is mainly the industrial strength languages, such as Delphi, C++ and Java, that have had vendors create IDEs that expose the language in a RAD fashion, resulting in CASE tools such Borland Delphi, Borland C++Builder, Borland JBuilder, and Microsoft's Visual C++.

2.2 A VM as Part of the CASE Software

Presented in this chapter is the concept of a CASE tool into which a VM can be used. The chapter illustrates the conceptual use of a VM to build an IDE. Such an IDE is generic with regard to the type of DSL required by an end-user, and the semantics of the DSL. Thus, it is a CASE tool that provides an IDE having the following property: the IDE allows one to invoke a VM that will run a DSL-program, where the DSL's syntax and semantics have also been

provided by facilities within the IDE. Indeed, the facilities for writing the DSL-program are also part of the IDE. In this sense, (i.e. of being able to define and support many DSLs) the IDE is generic, and in the same sense, the associated VM is generic. This implies that the tool itself will be in two separate instances or parts, each part being used at different stages of software development.

The first part will allow a developer to custom build a DSL, with syntax and semantics, and allow linking into a namespace written in a platform such as the .NET CLR. The second part will allow the developer to write programs using the DSL that has been specifically customized for his or her particular need.

The VM Framework presented in this dissertation will not cater for every feature mentioned above, such as visualizing components in design-time or runtime or translation of a high-level language to a low-level language such as C or assembler. A VM instance of the VM Framework will instead, form the core component responsible for executing a DSL-program within a simulated environment. The VM is generic, as will be discussed in Chapter 5, in that the language is customizable to suite a wide range of end-user domains.

2.2.1 Languages in a DSL-Supporting IDE

Figure 1 demonstrates a hypothetical IDE that does not feature any specific development language of a software application, but rather illustrates five key areas where a particular language can be a used. Such an IDE does not exist in industry, though certain IDEs that are in existence today do expose one or more of the mentioned areas with a language of some kind, and examples are mentioned in this and the next section. To simplify the illustration, the notion of having a real-valued functional expression language as a development language is shown. One of the areas is indeed the syntax definition (grammar) of the development language, and thus the grammar definition in Figure 1 is the grammar of such a functional language. Furthermore, it should be understood that not all of these five languages are used at the same instance, but instead some are used before the design of an application, some are used during the design of an application, and some are used for testing and post-development. The way to understand this hypothetical IDE would be to ask what five key areas of building a software application could be useful if they were to be exposed as a language. In so doing, could they provide a means to configure the IDE to develop the required software application in an optimal fashion, depending on the type of application to be built? Naturally, at least one language is required – the development language itself, but the aim is to be able to configure more of the IDE's development facilities. The five key areas are described in more detail below.

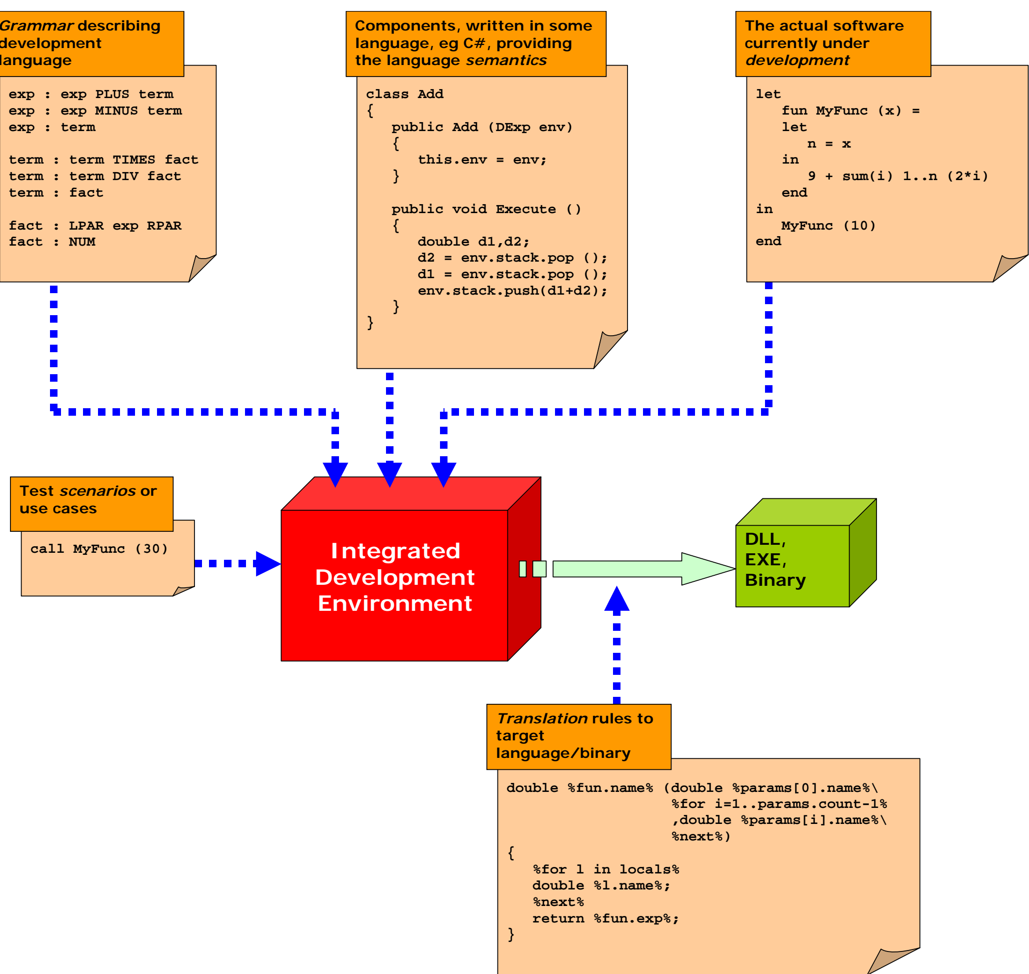


Figure 1. High-level view of a DSL-supported IDE.

The *grammar* describes the syntax of the language, usually in a standard form such as Backus-Naur Form (BNF). In Figure 1 above, the top left-hand box illustrates the first language of a DSL-supporting IDE: the language indicating the BNF which defines the syntax of some DSL – in this case, a DSL that has arithmetic operators as part of its syntax. Grammar is typically not changeable by a user, especially in industrial strength languages, like C++, as it would be senseless to allow the actual syntax to be altered. However, in such environments as in the ASF+SDF MetaEnvironment, in addition to the target application currently under development, it is also the language that is a concern to the developer. Thus the DSL is rapidly prototyped. In the case of the MetaEnvironment, it is necessary to provide a means for a language designer to let language syntax evolve until it suites a particular need, as is the intention with prototyping a DSL. The present study has a similar objective.

The *semantics* of a particular development language is usually encapsulated in the compiler provided by a traditional IDE. The sequence of machine code instructions generated by the compiler represents the semantics of the program written in the development language. The sequence of instructions generated by the compiler is also not changeable in standard compilers since it is implemented as part of the compiler's internal translation pipeline. The

compiler is responsible for generating machine code instructions for a target machine type, such as a Pentium processor. Without making too many direct low-level references to the actual instruction set of the target machine at the hardware level, it is sufficient to state more simply that the semantics of a program amounts to the actual observable behaviour of an executing program. This does not necessarily mean that one can observe the behaviour as the program is being executed. Rather it means one expects some useable result from executing the program, be it a once off execution, or execution that is continuous until some condition is met. Moving away from a traditional IDE towards its hypothetical counterpart, the semantics for the development language is allowed to be configurable. It is possible to use any existing development language to build a set of semantics for a particular DSL development language, and in Figure 1 above C# is used. The top middle box in Figure 1 shows how the semantics of real-valued addition is encapsulated in an external C# library class called Add. The Add class will form an instruction of the instruction set that a VM instance will ultimately be able to interpret, and thus defines runtime behaviour for the developing DSL. Having a set of classes in this way and having them supported by the IDE means they need to comply to some specific interface and this interface is indeed what is prescribed by the VM Framework discussed later.

The *development* language itself, briefly mentioned before, is the language exposed to the end-user – i.e. in the present study, it is the DSL exposed to the domain expert. This is typically the primary tool of the application developer, and is what the designer will make use of to develop the required application. As mentioned, even in the case where there are features of an IDE that allow a user to develop interactively (such as a feature for designing GUIs, or features such as those found in BridgePoint that provide a notation to describe state machines for active objects) there will still be a fair amount of underlying code that is generated automatically by these features, targeted to the development language. Some well known such examples of development languages (including any generated code from the abovementioned features) are C#, C++, Java, Delphi. In Figure 1, the top right-hand box illustrates some functional language that is being used as a development language.

Some environments include some kind of test *scenario* language that is used to simulate various conditions on an application. This enables earlier testing of the application being developed. An example of such testing on embedded devices is given in [Oli95]. Essentially the language allows the definition of a sequence of inputs or events to occur within a simulated context. These inputs may either occur in some predefined order or as a random sequence of events. Such events include interrupts, messages that are generated from external devices, data from a LAN or other communication mechanism or information from sensors attached to the device being programmed with the software. These types of inputs are especially useful in embedded systems, where it may not necessarily be practical to test the software on a target platform. An example of such an environment is the BridgePoint IDE that allows an application to be isolated and executed in a simulated context, and responds to inputs described by a

particular type of test case scenario. These inputs can be programmed to occur at certain times or to occur in some sequence inside a loop. This allows for rapid testing of an application. Though this allows for rapid elimination of logic errors in the application, it will not expose any errors that are ascribable to some specific characteristic of the target embedded platform. Such a scenario language is represented as input to the IDE in Figure 1 in the middle left-hand side.

The last language that warrants elaboration describes the *translation* of a program written in the *development* language (a DSL-program) into some other target representation, such as an imperative language like C. Such a translation language is merely a set of textual rules that specify how one construct represented in the development language manifests as a corresponding construct for some other target language. The actual semantics of the original DSL-program source needs to be retained after it has been translated into the target representation, thus the mapping between the two different representations need to be clearly understood, so the translation rules can be specified without any ambiguity. For example, a summation construct in a real-valued expression language would ideally translate to a `for`-loop of some segment of code that represents the expression being summed. In the above example in Figure 1, the translation rules are described in the form of an *archetype*, which is essentially a representation of a textual source code target, in this case, a straightforward C function, with placeholders for names of functions, parameter lists, locals, etc. This is an approach taken by the BridgePoint CASE tool. When the translation occurs, the placeholders are replaced with names defined within the DSL-program source code. In the above example, when a C-function is to be produced, the name of the C-function is derived from the name of a function declared within the DSL-program, and is queried by the occurrence of `%fun.name%`, and is valid for whatever function is currently under translation. Any occurrence of `%fun.exp%` will generate the translated form of the expression defined by function (declared using the `fun` keyword) in the DSL-program. This will in turn invoke other specified rules for translating real-valued expressions such as addition, multiplication, summation and so on.

2.2.2 A DSL Development IDE

As mentioned, not all IDEs make use of each of the languages described above. Some may not be relevant to the particular IDE or they might simply not be provided as a feature of the IDE. The table below relates a few language IDEs, and includes the VM framework proposed in this dissertation. Though the VM Framework is not an IDE in itself, it has a vital role in the design of a VM (resulting in a VM instance) that can be used in some DSL-supported IDE. This role of the VM Framework in such a context will be discussed shortly in this section, and it is this idea of a DSL-supported IDE to which the remaining three IDEs are compared. The table indicates

whether the IDE or DSL-supported IDE (based on the VM Framework) facilitates the particular language in any way, as far as the developer's point of view is concerned.

	Delphi/C++ Builder	**ASF+SDF**	**BridgePoint**	**DSL-supported IDE**
Grammar		✓		✓
Semantics				✓
Development	✓	✓	✓	✓
Scenario			✓	✓
Translation			✓	

Table 1. Comparing IDEs and their use of language types.

Borland's Delphi and C++Builder are fundamentally identical in terms of features provided by the tool, the development language is really the only difference. In both cases, the development language is the only means to build an application, with a few visual tools to aid in designing GUIs and database sources. The ASF+SDF MetaEnvironment additionally allows the formal definition of a development language's grammar, as well as semantic rule mappings from a provided grammar. The semantics, however, is not checked on the table. The reason is that the semantics in the context of the table refers to observable behaviour, rather than formal denotational semantics specified by the ASF+SDF MetaEnvironment compilation equations. This is where the power of the VM Framework will become apparent. It has a more practical application of specifying semantics in that it will allow the use of externally defined library classes to provide a rich set of functional runtime behaviour. A grammar definition will make the invocation of these semantics a simple matter of mapping a construct into a sequence of instructions. This will be discussed with reference to Figure 2 shortly. BridgePoint supports the specification of translation rules from its development language (an intuitive query-like language) to a target language. Often C is used as the target language as BridgePoint is well suited to embedded hardware systems. Building a scenario language into a DSL-supported IDE will not be difficult. The task would entail the execution of a DSL-program, having the VM instance execute the sequence of instructions. Invocation of the DSL-program can be done with some specific parameters that are used for testing purposes, and also, it may even be useful to build *debug* instructions as part of the instruction set. Such instructions could print values to a console, or provide immediate program termination. To simplify the intention of this section, it is worth mentioning that the generation of the VM and DSL-compiler together (shown as two blue opaque boxes in Figure 2) is intended to be an automated process. This automation is still speculative at this point, however the VM Framework that has been written for this dissertation enables a user of the VM Framework to manually craft such a VM component, while using suitable compiler writing tools to (almost completely) automate the task of building a DSL-compiler. Such tools will automate the generation of the front-end

processes, such as the lexer and parser, but the user still needs to supply the correct logic for translation of a DSL-program into a sequence of instructions. This translation specification forms part this example.

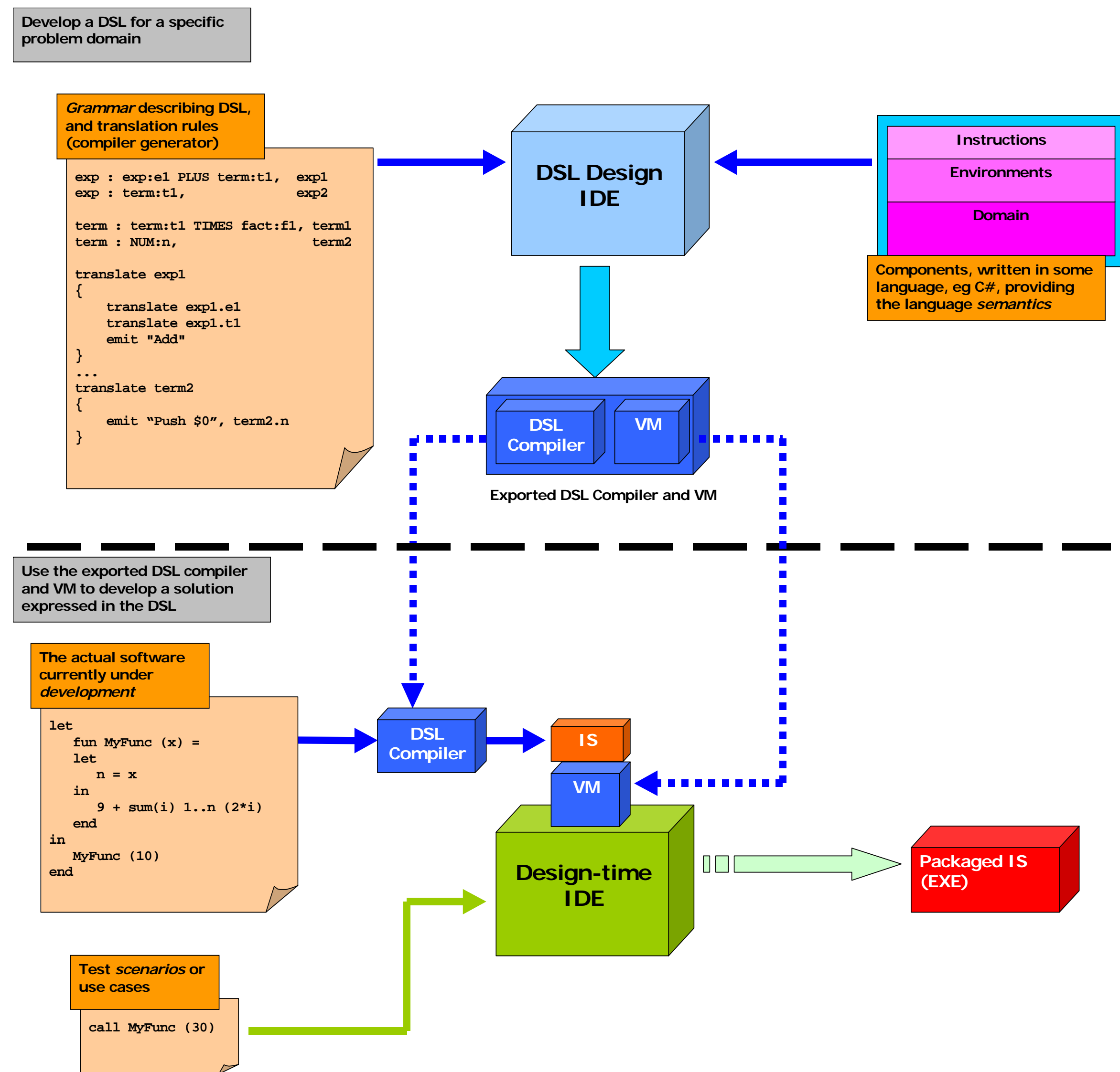


Figure 2. Generic VM used in the context of a DSL IDE.

This dissertation is centered on a framework that aids in developing DSLs. It is aimed at producing a software system by means of which the syntax and semantics of some DSL can be defined. The software should then generate a compiler that translates this DSL code to an executable Instruction Sequence (IS in Figure 2). The software should also be capable of producing a VM that can read in this sequence of instructions, and then translate them into some low-level language that can be executed on the platform on which the overall system is running. Here, the overall flow of activity in developing and running DSL code in such an environment is discussed with reference to Figure 2 above.

A VM Framework has been built and will more formally be described in later chapters. A product of this Framework will be a VM instance and will serve as the underlying execution mechanism of a CASE tool. A VM instance will be based on some custom built namespace during the design of a DSL. Figure 2 illustrates the two *views* of such a CASE tool, and the flow of information to arrive at the resultant application under development.

The first view in the upper half of the Figure 2, illustrates how the VM Framework supports the design of a particular DSL. From this perspective, the software constitutes a DSL design IDE. The upper left-hand box shows that the VM Framework relies on a language to build a DSL grammar. This language can also specify translation rules to generate sequences of instructions from syntactically correct DSL constructs, and consequently it defines the rules to produce a sequence of instructions for some complete DSL-program. The box on the top right of Figure 2 indicates that the DSL-Design IDE also requires instructions, environments and domains. These are third-party components or libraries written in any conventional development language and are imported into the DSL-Design IDE. The domains are simply namespaces that exist as legacy code, or are written specifically for some DSL. The ray-tracer namespace, described in Chapter 3, is one such example of a domain that will be exposed as a DSL. The DSL-Design IDE will then generate a DSL-compiler *and* a VM from the provided specification.

The second view below the dotted line in Figure 2 is of the Development IDE, so-called because it is used to support the design and implementation of systems to be written in the DSL defined in the first view. It imports the DSL-compiler generated in the first view into the Design-time IDE, to allow the developer to freely design software in the specified DSL. Any time that the program is to be executed within the Design-time IDE, the DSL-compiler is invoked to generate a compiled sequence of instructions, which is then executed by the VM instance that had been generated for this DSL.

To have the compiled program execute outside of the Design-time IDE, either the VM instance can be packaged and provided as a library that is invoked when the program is executed, or a platform specific executable can be generated, that inherently embeds the program along with the VM instance required to execute the instructions, into the executable. It is more favourable to have the VM instance packaged as a separate library, as it may be invoked programmatically as part of a larger piece of software.

2.3 Summary

This chapter identified five different languages used either in the development of a language, or in the configuration of an IDE. It includes the actual language used by a developer. The five languages are supportive of each the following: the syntax, underlying semantics, end-user development, test scenarios, and translations to target languages. The syntax of a DSL is usually coded in a format similar to BNF, and in the context of an IDE that supports DSL design, it can allow for the translation of certain constructs to a sequence of instructions. The underlying semantics refers to the implementation of behaviour that is representative of the constructs expressed in the DSL. The VM Framework allows a set of instructions and environments to be subclasses of certain classes of the VM Framework. These subclasses embody (encapsulate) the semantics of the DSL that has been defined.

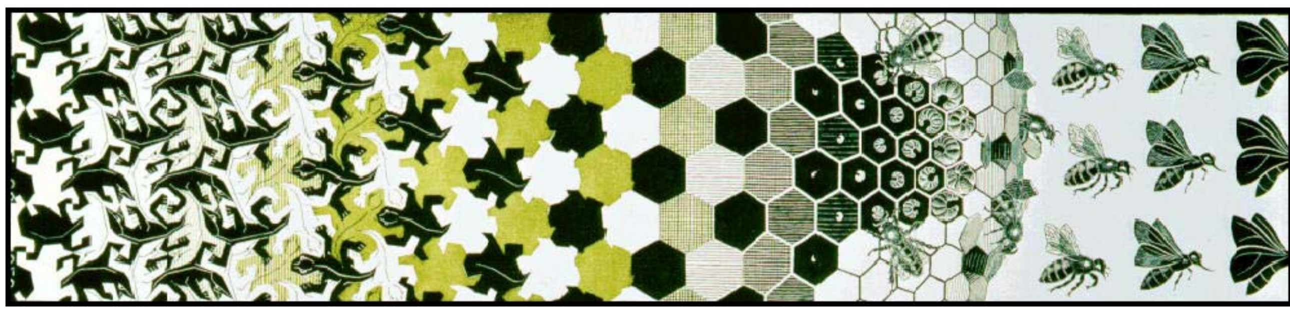
The DSL will ultimately be used by the end-user of the DSL to express solutions elegantly and concisely within their problem domain. Test scenarios define external behaviour, that essentially run an instance of the end-user's program. Such a test scenario provides parameterized data to the executing instance for testing purposes. Asynchronous systems can often be in one of a large number of states. Having these scenarios to generate events into the executing instance can greatly aid in verifying the correctness of such systems and eliminate any unforeseen deadlock or starvation situations which are prone to arise in asynchronous systems. Finally, translations simply map the DSL-program into a target language such as C or Java.

The chapter also suggests an idea in which a VM Framework will be of particular use, that is, to fit within the context of an IDE featuring two views of a developer's interests – that of the design issues of a DSL, and an end-user view that makes use of the DSL.

The dissertation does not take the notion of embedding the VM framework within an IDE any further. Rather, it focuses on describing the design and implementation of a VM Framework for DSLs. This framework has been developed in C#. It facilitates the construction of a VM and a compiler from the given syntax specification, which can then be packaged and exported to provide the runtime behaviour of a program written in the DSL.

It should be noted that the notion of developing a VM Framework evolved out of a number of preliminary experiments aimed at providing an environment to support the use of DSLs. It will be of interest to discuss the most important of these experiments and to indicate how and why, in each case, an alternative suggested itself. The next chapter discusses the first of these experiments.

3 Domain-Specific Languages and Interpretation



This chapter begins the historical account of an investigation into various ways of supporting DSL development. The first DSL development approach to consider is that of using an interpreter. Two questions can be asked in regard to this interpreter-based technique. First, how much effort is involved in building an interpreter for a particular DSL, in the event that the DSL has some formal specification in terms of syntax and the required functionality to be exposed to the expert? Second, is the development process for designing an interpreter streamlined enough to scale up the DSL in the event of an ever-growing domain? Later, similar questions will be raised in having a VM as the mechanism responsible for providing observable semantics from a DSL-program being executed.

It will be shown that difficulties arise when scalability starts to become an issue. It is likely to be the case that at some future stage, an end-user needs further amendments to a DSL that has already been provided. Scalability is a very real part of software design, and it should be incorporated into the design process. On the negative side, scalability often comes at a performance cost, but in a prototyping environment, developers and designers alike are willing to make that sacrifice. Because scalability was perceived from the outset as an important matter, it was decided to rely on the Shlaer-Mellor software development methodology, which is well suited to support the development of scalable systems.

Sections 3.1 and 3.2 make the distinction between *translation* and *interpretation*, and sections 3.3 and 3.4 provide more information of the Shlaer-Mellor methodology, and relate it to the DSL problem domain. Sections 3.5 and 3.6 provide an overview of the interpreter approach, and rely on a ray-tracer DSL for rendering geometric shapes as an illustrative DSL example. Section 3.7 demonstrates the scaling up of the DSL in terms of syntax and semantics. Finally Section 3.8 briefly discusses modularity of the DSL and Section 3.9 provides the summary.

3.1 DSL Translation

Most industrial strength DSLs currently available, use (source) code generation techniques [Cza99, vDeu01b, Kic96]. Certain areas of concern can be isolated, for example, concurrency, and appropriate DSLs can be designed for expressing concurrency-related issues with primitive concurrency constructs. A *process* could, for example, be one such construct expressed in terms of its constituent states as well as constructs that enquire on the correctness of such declared processes [Mag99]. After expressing a set of concurrent processes along with a set of formal verification tests in a DSL-program, the user can execute the DSL-program with an interpreter to determine the correctness of the concurrent system. The output of executing this type of DSL-program on an interpreter will simply be a set of possible state transitions that lead to incorrect behaviour or deadlock situations, if at all. This feedback on the correctness of the prototyped concurrent system might prompt for the design of the system to be revisited. Once the logic of the system has been determined to be suitably correct, then conventional programming language source code can be generated using a separate tool to perform the translation of the DSL-program, or this may be a feature of the interpreter itself. The generated source code can either be in the form of a skeleton framework where further functional code can be added or as a separate component (namespace, package or class, depending on the target language) that is generated and compiled in alongside existing functional code. Essentially the tool translates the DSL-program into a language, such as C, which can further be compiled and ready to run on a native processor, such as an embedded system. In this way it is easier to verify the concurrency of the system before it gets further engineered with functional code and debugged on the target system as a C program.

3.2 DSL Interpretation

In contrast to DSL translation, DSLs can also be interpreted. In this case, the DSL is a directly executable programming language, and in this context the DSL forms a scientific tool for a domain expert. Often with interpretation, the semantic analysis forms part of the same code that is already responsible for parsing the actual DSL source. When a certain construct is being parsed and ready to be reduced, there is enough information gathered by the parser at that point to make a function call, and that is the very function that interprets the currently parsed construct.

3.3 Shlaer-Mellor Domains

When discussing software construction at the language or compiler level, there is an obvious assumption that there is some core-underlying principle, which does not change throughout the development life cycle. This is the rationale of the language. Modules that do not change are readily reusable, and for the most part, it is often the class that forms a piece of reusable code. To drive the idea of a reusable rationale, that is configurable at the language level, the Shlaer-Mellor software construction method has been adopted as it serves as a good base onto which to build DSLs. Like most object oriented analysis methods, Shlaer-Mellor encourages the formation of information models to depict static relationships between classes, as well as to depict any dynamic behaviour associated with relevant classes. A fundamental difference between Shlaer-Mellor and other methods is the separation of subject matters into separate domains as was discussed in Section 1.2.1.

Most systems are the result of a collaboration to address overlapping concerns. The concerns could relate to, for example, graphical user interfaces (GUIs), databases, concurrency, mathematical models and so on. Building a domain for each area of concern allows developers to specialize, in parallel, on each different subject matter. This is called *horizontal partitioning*. This opposes *vertical partitioning* where a developer needs to understand most of the concerns, as the design is decomposed recursively.

3.4 Configuring Shlaer-Mellor Domains with Bridges

Domains exist as separate reusable frameworks available to project teams that have similar requirements. Thus it is the domain itself that is the main reusable component, and not the classes within the domains. Although classes that have been designed with reusability in mind, and can quite freely be used in more than one domain, classes are usually very specific to its particular domain, and thus limited to one domain. Often it is not possible to build a domain that's absolutely cast in stone and reuse it amongst other systems and so some level of configurability is required for each domain.

In Shlaer-Mellor, configuring domains is done via *bridges*. Bridges form the links between domains. They connect the domains in a logical network resulting in the final system. They contain any glue code necessary for the communication between domains to be possible. Bridges do not normally consist of object-oriented code, rather it is mainly procedural function calls. Domains can be reused in different systems with different contexts and the intention is to alter the bridge to allow the domain to fit within the context of a different system.

The configuration language of a bridge is the same language used to build the domains themselves. Pursuing the Shlaer-Mellor idea further, this dissertation involves using C# as the development language for the domains, and then exposing any configurable functionality, as was done with Shlaer-Mellor bridges, as a DSL. This invariably makes configuring a domain much easier if a suitable DSL does exist. It may still be necessary though, to apply an intermediate step to renovate the domain to improve efficiency and reuse of the existing code [vDeu98b]. Another intermediate step may require the rewriting of a large number of procedural legacy methods or functions, into suitable classes for an object-oriented language [vDeu99]. This idea of configuring a domain has proved to form a useful basis for the design and construction of DSLs for domain experts.

3.5 View of the Interpretation Approach

In the view of the interpretation approach, no compilation or translation to an intermediate language is performed and therefore no executable file is generated. A simple illustration of such a view is shown Figure 3. In most interpreted languages it is possible to encounter a syntax error midway through program execution.

In the illustrating software for this chapter, a syntax error will be caught before any interpretation takes place because the parser generator used, PG (see Section 1.2.3), generates code that always builds an *Abstract Syntax Tree* (AST) from all of the code, before the AST is traversed and interpreted. An AST is a tree data structure representing a program that has been parsed. This representation is irrespective of such parts of the grammar like semicolons and braces (the *concrete* syntax), and is simply an abstract representation linking all the constituent constructs that make up the program. Since nodes in an AST represent constructs they may contain additional information pertinent to that construct. For example, a node may represent a *`while`*-loop construct and contain information as to the condition upon which to exit the *`while`*-loop. This information could indeed be a node representing another construct, such as a boolean expression.

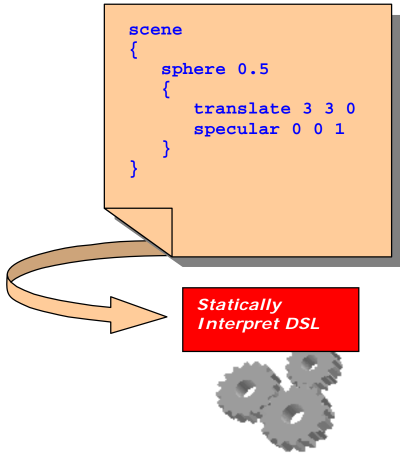


Figure 3. Conceptual view of the Interpretation Approach.

Apart from the actual underlying ray-tracer domain that will be used to illustrate the interpretation approach below, there are only three user-defined components, namely the program's main entry point, a grammar definition for the parser's input and a keyword definition for the lexer's input. The three components are listed in the following table. They are all found on the accompanying CD.

Source File	Description
`rtc.lexer`	The keyword definition for the ray-tracer lexer's input, used by LG
`rtc.parser`	The grammar definition for the ray-tracer parser's input, used by PG
`rt.cs`	File containing the program entry point for the interpreter. Home to functions that translate nodes of the AST after the DSL-program has been parsed.

Table 2. Components for the Ray-Tracer Interpreter.

3.6 The Ray-Tracer DSL

One of the DSL examples used throughout this dissertation to illustrate some of the concepts is a simple graphical scene description language, called Ray-Tracer, otherwise referred to as the RT DSL. The language has been devised, and the ray-tracing software upon which it relies has been written, specifically for the purpose of this present study. The underlying mechanism uses a recursive ray-tracer to render scenes of geometric objects with various forms of lighting. Later, when scalability and modularity are improved in the DSL design, other example languages, with their corresponding namespaces will be shown.

The RT language serves as a good example since it is fairly illustrative in more than one sense. Also, ray tracing in general is a fairly well understood technique, and its domain is well defined and almost obligates a DSL to be wrapped around it. The primitive objects used in ray-tracers are usually defined by mathematical equations, like spheres, planes, and cones, so ray interceptions can be calculated on the objects. By adjusting some of the coefficients of the equations, the size and shape of the primitive objects can be altered. Already, it is easy to start realizing some of the concepts that form part of the ray-tracer namespace: rays, vectors, geometric objects and scenes etc. For a more detailed and mathematical account of ray-tracers and ray casting equations see [Wat00].

3.6.1 The RT Namespace

At this point it is appropriate to start fleshing out a real instance of a domain, or rather from a programmatic viewpoint, a namespace. As mentioned above some of the concepts that developers encounter in a ray-tracer are realized as classes within the RT namespace and shown in Figure 4.

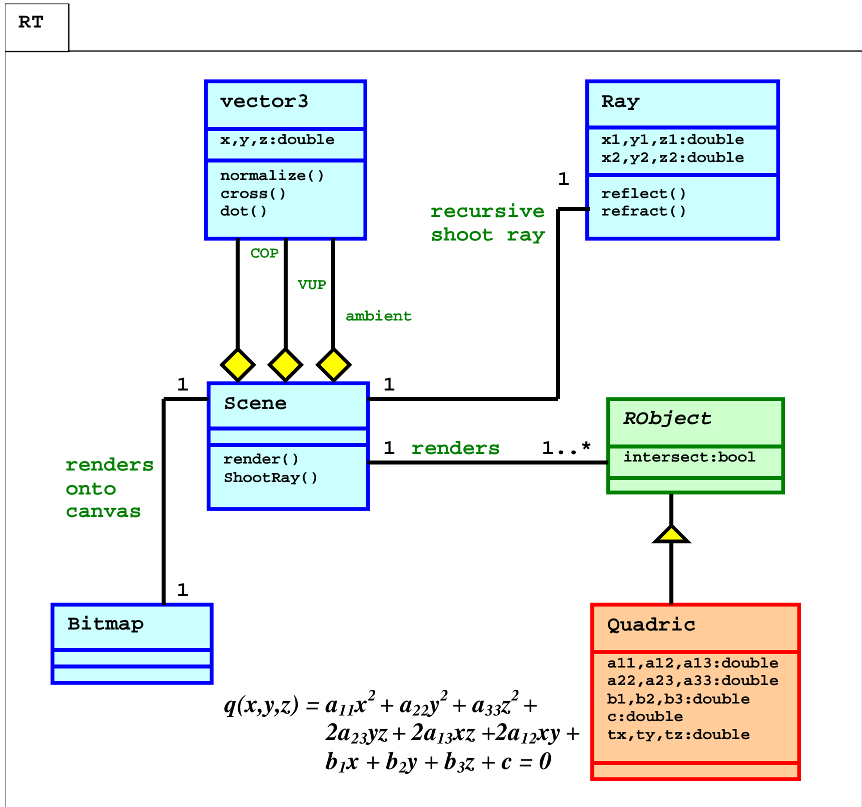


Figure 4. Information Model of the RT namespace.

The following list of source files is given along with a brief description of each file and its purpose. Later extensions to the language can be compared with this list to show fewer files being modified and correspondingly, less coding needed.

Source File	Description
`vector3.cs`	A simple 3-space vector including typical vector operators like cross products, dot products and some primitive reflection operators. Also includes functions to normalize and calculate the magnitude.
`Ray.cs`	A higher level to the vector class describing a single ray in space. Includes functionality to generate reflected rays, refracted rays and pass-through rays. Also include functionality to mitigate rounding errors.
`RGBStruct.cs`	Structure representing the red, green and blue components of a single pixel colour.
`RObject.cs`	An abstract class forming the base class of ray-tracer primitive object. Thus there can be a generic list of RObjects and the intersection test is dynamically bound depending on the type of object intersected by a ray.
`Scene.cs`	Simple scene class encapsulating all the described objects. Includes a list of `RObject` instances that form the scene to be rendered. Once all the objects are in place, then a scene can be rendered to a bitmap image file.
`Quadric.cs`	Class encapsulating the mathematics for the intersection of a ray on a generic quadric object (sphere, cone, plane) and its coefficients, and is defined by the following equation: $q(x,y,z) = a_{11}x^2 + a_{22}y^2 + a_{33}z^2 + 2a_{23}yz + 2a_{13}xz + 2a_{12}xy + b_1x + b_2y + b_3z + c = 0$

Table 3. Source files for the RT namespace.

A primitive declaration in the RT DSL implies that a primitive geometric object is being declared (a quadric, such as a sphere, a cone or a plane). As shown in Figure 8, `element2` is the name of the node on the AST representing a primitive declaration, and as such will be representing internally by a class called `element2` that lives in the parser's namespace since it is auto-generated code from PG. Note that in the figure, `element2` refers to the second `element` node. The name of the node, `element2` in this case, is not shown alongside the second occurrence of `element` to due space constraints, however the complete grammars are all found on the CD. Figures 5 and 6 shows example code that performs the interpretation of

an `element2` node on the AST. The first block shows the code responsible for determining the types associated with each node, while the second block shows the code performing actual interpretation of an identified node. Translating this primitive declaration implies first translating the primitive itself by calling `trans_prim()`. This function determines the type associated with the `prim` node, as it could be a `prim1` node representing a sphere or a `prim2` node representing a plane, and translates the primitive accordingly by creating a quadric object, and adjusts the necessary coefficients for the quadric. Then the primitive's list of properties is translated in a similar fashion, by first calling `trans_prim_prop_list()` which translates each property in the list, determining the type associated with each property. In Figure 6 only translation of the specular property of a primitive object is shown. Finally the primitive object is added to the scene's objects list.

```
public static void trans_element2 (RtcParser.element2 node)
{
  trans_prim (node.p);
  trans_prim_prop_list (node.ppl);
  scene.addObject (robject);
}

public static void trans_prim (RtcParser.prim node)
{
  if (node is RtcParser.prim1)
  {
    trans_prim1 ((RtcParser.prim1) node);
  }
  else if (node is RtcParser.prim2)
  {
    trans_prim2 ((RtcParser.prim2) node);
  }
}

public static void trans_prim_prop_list (RtcParser.prim_prop_list node)
{
  if (node is RtcParser.prim_prop_list1)
  {
    trans_prim_prop_list1 ((RtcParser.prim_prop_list1) node);
  }
}

public static void trans_prim_prop_list1 (RtcParser.prim_prop_list1 node)
{
  if (node.ppl != null) trans_prim_prop_list (node.ppl);
  if (node.pp != null) trans_prim_prop (node.pp);
}

public static void trans_prim_prop (RtcParser.prim_prop node)
{
  if (node is RtcParser.prim_prop1)
  {
    trans_prim_prop1 ((RtcParser.prim_prop1) node);
  }
  else if (node is RtcParser.prim_prop2)
  {
    trans_prim_prop2 ((RtcParser.prim_prop2) node);
  }

  ...
  else if (node is RtcParser.prim_prop6)
  {
    trans_prim_prop6 ((RtcParser.prim_prop6) node);
  }
}
```

Figure 5. Determining the type associated with an AST node.

```
public static void trans_prim1 (RtcParser.prim1 node)
{
  robject = new Quadric ();
  robject.alterCoeff_a11 (1.0 / (node.d1 * node.d1));
  robject.alterCoeff_a22 (1.0 / (node.d1 * node.d1));
  robject.alterCoeff_a33 (1.0 / (node.d1 * node.d1));
  robject.alterCoeff_c (-1.0);
}

public static void trans_prim2 (RtcParser.prim2 node)
{
  robject = new Quadric ();
  robject.alterCoeff_b1 (node.d1);
  robject.alterCoeff_b2 (node.d2);
  robject.alterCoeff_b3 (node.d3);
  robject.alterCoeff_c (node.d4);
}

public static void trans_prim_prop2 (RtcParser.prim_prop2 node)
{
  robject.alterSpecular (node.d1, node.d2, node.d3);
}
```

Figure 6. Actual interpretations of the AST nodes.

3.6.2 The RT Language

The RT scene description DSL is a simple declarative language. Once a primitive object is declared, for example, a sphere, it is automatically allocated in memory. Once all primitives have been declared, the ray-tracer will render the scene as a bitmap. The object used for the bitmap is the .NET class `System.Drawing.Bitmap`, which, once instantiated can be either rendered onto a canvas or saved to disk.

Primitives are defined by specifying their geometry and chromatic characteristics. All primitives reside in a scene declaration. A scene is defined to have a window width and height, and a set of other characteristics, such as the camera position, light position and so on. The lexer's input consists of regular expression definitions that are given in a format supported by LG, and the parser's input are grammar definitions in a variation of Backus-Naur Form (BNF) that is supported by PG.

```
%lexer
%token                    DOUBLE {public double val;}
"{"                       %LBRACE
"}"                       %RBRACE
"scene"                   %SCENE
"window"                  %WINDOW
"light"                   %LIGHT
"cop"                     %COP
"lookat"                  %LOOKAT
"viewup"                  %VIEWUP
"geometry"                %GEOM
"specular"                %SPECULAR
"diffuse"                 %DIFFUSE
"reflective"              %REFLECTIVE
"refractive"              %REFRACTIVE
"sphere"                  %SPHERE
"plane"                   %PLANE
"translate"               %TRANSLATE
[-+]?[0-9]+(\.[0-9]+)?    %DOUBLE {val = Double.Parse (yytext);}
[ \n\r\f\t]               ;
"(*"                      {yybegin ("COMMENT");}
<COMMENT>"*)"             {yybegin ("YYINITIAL");}
<COMMENT>.                ;
<COMMENT>\n               ;
```

Figure 7. Lexer input for RT.

Integer and double tokens have been combined into a single `DOUBLE` token that matches any signed or unsigned integer or double. Internally it is represented as a `double` C# primitive type. Also, the regular expressions recognized by the lexer provide for Pascal-like comments in the DSL.

```
scene_desc : SCENE LBRACE element_list RBRACE

element_list :
element_list : element_list element

element : scene_prop
element : prim LBRACE prim_prop_list RBRACE

scene_prop : WINDOW DOUBLE DOUBLE
scene_prop : LIGHT DOUBLE DOUBLE DOUBLE
scene_prop : COP DOUBLE DOUBLE DOUBLE
scene_prop : LOOKAT DOUBLE DOUBLE DOUBLE
scene_prop : VIEWUP DOUBLE DOUBLE DOUBLE
scene_prop : AMBIENT DOUBLE DOUBLE DOUBLE

prim_prop_list :
prim_prop_list : prim_prop_list prim_prop

prim_prop : GEOM DOUBLE DOUBLE DOUBLE DOUBLE
prim_prop : SPECULAR DOUBLE DOUBLE DOUBLE
prim_prop : DIFFUSE DOUBLE DOUBLE DOUBLE
prim_prop : REFLECTIVE DOUBLE DOUBLE DOUBLE
prim_prop : REFRACTIVE DOUBLE DOUBLE DOUBLE
prim_prop : TRANSLATE DOUBLE DOUBLE DOUBLE

prim : SPHERE DOUBLE
```

Figure 8. Parser input for RT.

Figure 9 is an example of how the RT DSL can be used to describe a scene. This will be the running example used throughout the dissertation in reference to using RT as a DSL. The example will also be used for the performance measurements.

```
(* Render a red, a blue and a green sphere, and an orange plane. *)
scene
{
	(* Define the settings for the scene. *)
	window 300.0 200.0
	light -3.5 0.5 2.0
	cop 0.0 2.5 1.0
	lookat 0.0 0.0 0.0
	viewup 0.0 0.0 1.0

	sphere 0.5
	{
		translate 1.0 0.0 0.0
		specular 0.7 0.0 0.0
		diffuse 0.7 0.0 0.0
		reflective 0.1 0.1 0.1
	}

	sphere 0.5
	{
		translate 0.0 -1.0 0.2
		specular 0.0 0.7 0.0
		diffuse 0.0 0.7 0.0
		reflective 0.2 0.3 0.2
	}

	sphere 0.5
	{
		translate -1.0 -0.5 0.0
		specular 0.0 0.0 0.7
		diffuse 0.0 0.0 0.7
		reflective 0.1 0.1 0.1
	}

	plane -0.1 0.0 1.0 1.0
	{
		specular 0.0 0.0 0.0
		diffuse 1.0 0.5 0.0
		reflective 0.25 0.25 0.25
	}
}
```

Figure 9. Example scene described in RT.

The language is fairly intuitive, and nowhere does the end-user of the language require any knowledge of the underlying mechanics of ray-tracers, neither at the implementation level nor at the mathematical level. The end-user's only concern is modeling a scene, and this end-user requirement is what the RT DSL has exposed.

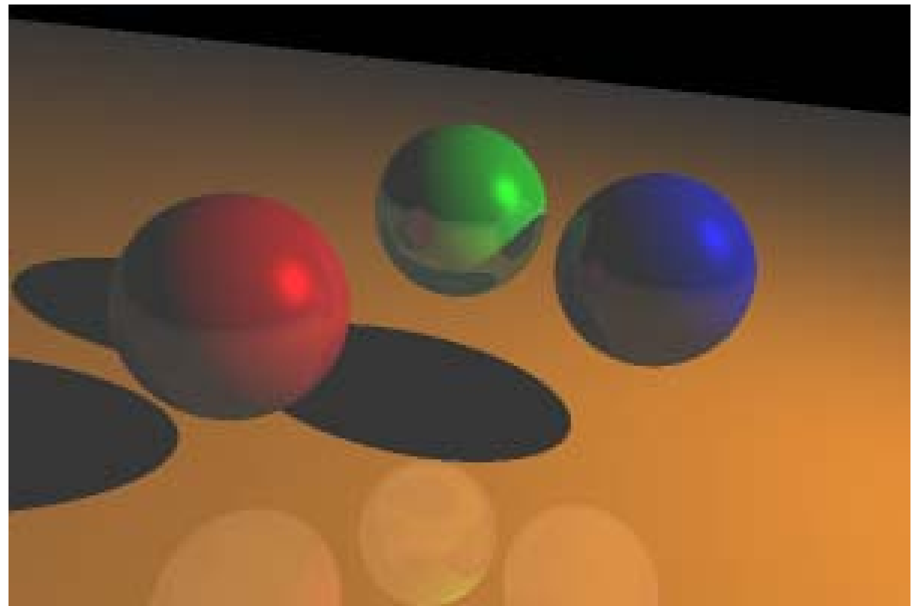


Figure 10. Result of rendering the scene defined in RT.

The end result of interpreting the example RT program is a bitmap image on disk of the described model, shown in Figure 10.

3.7 Scaling up the RT DSL for the Interpreter Approach

This section explores the implications of scaling up the RT DSL. Suppose that the user of the DSL would like to be able to alter the ambient light within the scene being rendered. The user would like the ability to adjust the red, green and blue colour components of the scene. A new *`ambient`* construct needs to be added to the language syntax. The Scene class within the RT domain already provides this functionality, so no additional code is needed in the namespace to facilitate the semantics of ambient light. The difficulty will come in the interpretation of the AST node representing the ambient component once the construct has been parsed.

3.7.1 Adding a Construct to an Interpreter

Adding an *`ambient`* construct to an interpreter implies modifying the way the interpreter translates nodes on the AST. It is desirable to devise a means to retain such modifications in a small, isolated region throughout the DSL pipeline, thus minimizing chances of introducing new bugs into the software that ultimately processes the DSL-program. First off, the grammar needs altering to add the *`ambient`* construct to the language itself, and hence the lexer also needs a minor modification to accept a new keyword `ambient`. The code that performs the

translations of the AST nodes also need to be modified and here one needs to add the necessary code to provide the correct interpretation of such a construct.

3.7.2 Components Requiring Alteration

It is trivial to amend the new keyword `ambient` to the existing keyword set. It is simply a matter of appending its regular expression to the RT DSL's lexer definition as highlighted in the Figure 11.

```
%lexer
%namespace RtcLexer

%token                  DOUBLE {public double val;}
"{"                     %LBRACE
"}"                     %RBRACE
"scene"                 %SCENE
...
"viewup"                %VIEWUP
"ambient"               %AMBIENT
"geometry"              %GEOM
...
```

Figure 11. Appending the lexer definition for keyword `ambient`.

The grammar should now support the addition of the *ambient* construct itself, highlighted in bold below, as well as the AST representation structure of the ambient node after it has been parsed, also highlighted in bold below. The new node is called `scene_prop6()`.

```
...
%node scene_prop6 : scene_prop
{
       public double d1, d2, d3;
       public scene_prop6 (double d1,double d2,double d3)
             {this.d1=d1;this.d2=d2;this.d3=d3;}
}
...

element : scene_prop
element : prim:p LBRACE prim_prop_list:ppl RBRACE

scene_prop : WINDOW DOUBLE:d1 DOUBLE:d2
scene_prop : LIGHT DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
scene_prop : COP DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
scene_prop : LOOKAT DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
scene_prop : VIEWUP DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
scene_prop : AMBIENT DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
             %scene_prop6 (d1.val,d2.val,d3.val);

prim_prop_list :
prim_prop_list : prim_prop_list:ppl prim_prop:pp
...
```

Figure 12. Altering the grammar definition for the new *ambient* construct.

The modification to the code that traverses the AST and performs the interpretation does have a certain degree of risk, as these modifications need to be carefully integrated so as not to alter the semantics of any existing constructs. In the listing shown below, the translation of a scene property, `trans_scene_prop()` needs adjustment to call the translation of the ambient property by invoking `trans_scene_prop6()`, and in so doing providing the necessary interpretation of adjusting the ambience of the scene.

```
public class rtc
{
   public static void Main (string [] argv)
   {
      ...
   }

   ...

   public static void trans_scene_prop (RtcParser.scene_prop node)
   {
      if (node is RtcParser.scene_prop1)
      {
         trans_scene_prop1 ((RtcParser.scene_prop1) node);
      }
      ...
      else if (node is RtcParser.scene_prop6)
      {
         trans_scene_prop6 ((RtcParser.scene_prop6) node);
      }
   }

   ...
   public static void trans_scene_prop6 (RtcParser.scene_prop6 node)
   {
      scene.alterAmbient (node.d1, node.d2, node.d3);  // ambient
      scene.setupAxes ();
   }

   ...
   public static Scene scene;
   public static RObject robject;
   public static Bitmap bitmap;
   public static string filename;
}
```

Figure 13. Altering the functions for the new *`ambient`* construct.

3.8 Modularity

In the interpretation approach to DSLs, issues of parsing and issues of semantics form part of the same piece of software. Thus it is not easy to reuse, say, just the semantic component of the language, for example, if no syntax is required for a particular solution. Ideally, one would like to approach the Shlaer-Mellor route, in that there be a clear separation between the issues of the semantics for the namespaces rationale or the functionality that the namespace provides, and the issues of allowing an end-user the ability of configuring the components (or classes) of the namespace to provide a resultant system that is better suited to their requirement. It is not necessarily the case that the configurable portion of the system be in the

form of a DSL, though that is the main focus of this dissertation, and as mentioned above, in the case of a developer using a Shlaer-Mellor architecture, the configuration language is usually the same language used to build the components or classes of the namespace itself.

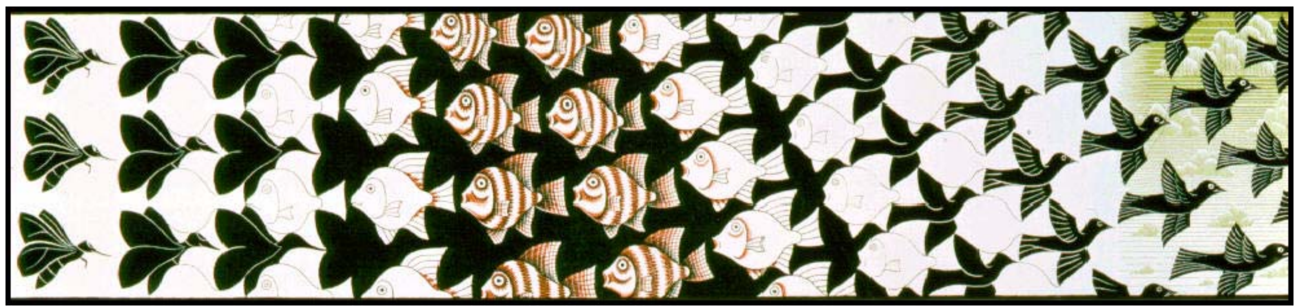
3.9 Summary

In this chapter an interpreter for a ray-tracer DSL is examined. Interpreters are easy to build and if the required DSL is small enough, and if the end-user foresees minimal modifications to the DSL's construct or underlying namespace, then the DSL interpreter is a simple enough solution. However as far as the language scalability is concerned, an interpreter poses problems. This was illustrated in this chapter by showing the required modifications to add an *`ambient`* construct to the RT DSL.

Since the semantics are interpreted directly from the nodes of the AST, it was not an easy task to alter the code that traverses the AST. Though the generation of a parser is automated, the code that forms the translation rules of each node is not, and regeneration of a parser potentially requires rearranging segments of code.

In this chapter the RT DSL was introduced. It shall serve as the example language throughout the dissertation, and is further used to illustrate how it forms an extension to a much simpler expression language later in Chapter 5. The RT namespace has a well-defined set of classes, including a `Quadric` class, with the responsibility of providing the mathematics to render any quadric (a sphere, a plane or a cone) onto a canvas.

4 Domain-Specific Virtual Machines



This chapter heads into the realm of VMs as the core mechanism responsible for providing the runtime execution of a DSL-program. It should be noted here that it is possible to build a complete taxonomy of VMs, each with varying objectives and implementations and differing contextual uses. The idea of a VM is not new, and conceptually rather simple. Interests in VMs are mainly invested in their ability to offer a high degree of flexibility in larger systems. This flexibility characteristic will be important later when portability of DSLs is discussed.

Virtual Machines are used often in computing to solve problems, but they are usually transparent to an end-user because they essentially consist of software running inside other programs or other operating systems. The concept of VMs has proven to be a powerful tool in shaping computations, and due to their popularity they provide another option from which to investigate the prototyping of DSLs.

Section 4.1 discusses some of the reasons for using VMs instead of code generation techniques to provide the semantics. Section 4.2 and 4.3 describe VMs from two different perspectives, and Section 4.4 describes the VM approach when applied to a DSL. Section 4.5 provides the details on scaling up a VM in the face of a new instruction and it's corresponding modularity is discussed in Section 4.6. This chapter concludes with a summary.

4.1 VM vs. Code Generator

There are two paths that a language designer can follow in building the software that processes a source program written in the designed source language. In the first case, a *translator* can be designed that will translate the source program directly into a program in

some target language. This is the code generation approach, and it essentially comes down to representing source programs in a different language. In the second case, a *compiler* can be designed that translates the source program into an instruction sequence, where the instructions are part of some instruction set. In some instances the instructions can only be natively executed on the platform. However in other instances the instructions are platform-independent. In the platform-independent case, a VM is used to interpret each instruction in the instruction sequence that the compiler generated on the platform on which the VM is running. The compiler in this case is platform independent, but a platform dependent VM is needed to execute the instructions that the compiler generates. The advantage of this is that multiple compilers can generate instructions for the same VM.

Why would a language designer opt to use a VM at all? Although code generation is certainly useful [vDeu01b, Cza99a], a more modular and more flexible means to create the software responsible for executing a DSL-program is sought. Allowing a user to build a language that requires additional work to target another language (or perhaps even to target a specific architecture built in that language) further complicates the software engineering process, as the user needs to think about the DSL, and the target language, and the complexities of matching the two together. A VM in this regard is thus more suitable because the user needs to only be concerned with the DSL, its syntax and semantics, and the raw execution of its constructs in the form of an instruction sequence that execute.

When scalability of DSLs is a concern, designing a suitable set of instructions may prove to be a more modular approach than designing a translator, since designing a finite set of instructions and any data structures that these instructions may modify, could lend itself to a more modular design. This current study aims to pursue this approach in building the software that will be responsible for providing the runtime semantics of DSLs. Language designers could potentially ask how easy it is for them to build or scale up a VM for a particular DSL. Will it be a simple case of adding constructs to the language that translate to instructions that already exist as part of the VM's instruction set, or will new instructions need to be added? How easy will is the task of adding new instructions to an existing VM?

4.2 Virtual Machine Instruction Set Architecture

The VM executes a sequence of instructions (i.e. a compiled program) and each instruction, as it is executed, performs some *atomic* operation. The word atomic simply means that the instruction represents an elementary operation that cannot be broken down any further in terms of the instruction set. As instructions are executed, the VM's internal data structures are continuously modified according to these operations, for example, one instruction may push a real value onto a stack, while another may pop off the top two values and push their sum back

on. The instructions thus access the VM's internal data structures in different ways. Naturally, a correct sequence of these instructions will leave the VM's data structures in predictable, consistent states and it is possible to verify the correctness of a sequence of instructions if the behaviour of each instruction is understood. This notion of executing the sequence of instructions is called a *process*, and there is a notion that the VM performing the execution of the instructions can understand or identify these instructions. In other words, the VM supports the instruction specification or the *Instruction Set Architecture* (ISA). Attempting to execute an instruction that is not part of the ISA will result in an error condition or graceful program termination if the VM is able to trap unrecognizable instructions. Figure 14 below is a simple illustration of this concept. It is also shown that part of the ISA contains a more privileged set of instructions that are used by an operating system while user instructions are used by typical application software.

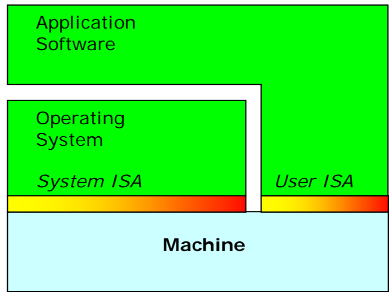


Figure 14. Instruction Set Architectures – System ISA and User ISA.

4.3 System Virtual Machines versus Process Virtual Machines

As mentioned before, though there exists a detailed hierarchy of VMs, it suffices at this point to classify two broad types of VMs at the top of this hierarchy: *process VMs*, and *system VMs*, [Smi05], each offering two different levels of support to the software running on top of them. As will later be seen, a software program created by a DSL will be regarded as a process and will slot comfortably into a process VM type of architecture. Here, both process- and system-VMs will briefly be discussed to understand the differences.

System VMs, simply put, exist to support Operating Systems (OSs), and hence are required to provide a full system environment for a particular OS to run on, that is, an environment with hardware resources, networking IO, and related subcomponents. *Virtualizing software*, by which is meant the software that constitutes the system VM itself, sits between an existing machine (machine, in an abstract sense, since the machine itself can be virtualized), and an OS, as depicted in Figure 15. In a *hosted* system VM, the virtualizing software is installed as a typical application running on an OS. *VMware* is one well-known example, and it provides an ISA that is identical to the one of the system onto which it was installed. In contrast, in a

codesigned system VM the ISA differs from that of the underlying system, in essence providing such flexibility to run, for example, a Windows OS on a non IA-32 machine.

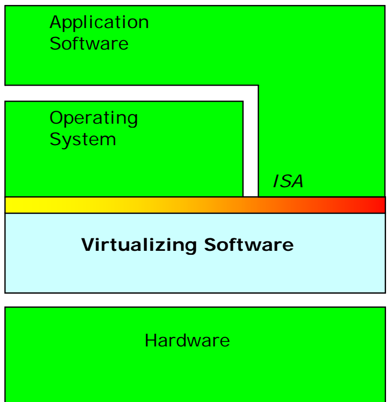


Figure 15. A System VM. Virtualizing software between hardware and OS.

Process VMs are of more interest to language designers because a process VM hides even system calls into the OS. It thus relieves the programmer from even thinking about the details of a specific OS. Instead, it allows the programmer to just be aware that one does indeed exist, and that it has features typical of an OS, such as GUIs and threads.

Emulation is one use of process VMs. In this case, the virtualizing software allows a program that has been compiled for some ISA to run on another ISA – i.e. the virtualizing software emulates the second ISA. Internally, the process VM performs per-instruction translation of the original sequence of instructions so that the process results in the same intended behaviour. An example is the Digital FX!32 system. Windows programs are intended to run on PCs based on 32 bit Intel Architecture (x86 or IA32) microprocessors, such as a Pentium processor. Using the Digital FX!32 system these programs are able to run on PCs featuring Alpha microprocessors which are a non-Intel based platforms. Furthermore, it is not uncommon to have profiling software embedded in process VMs, collecting statistical information about a program's runtime behaviour, and thereby allowing a programmer to identify sections of code that are called frequently and perhaps warrant an optimization effort.

For a developer though, portability is still a major concern, and so it is useful to have a process VM that does not necessarily reflect a real platform as such, but rather reflects the features of a high-level language. In particular, having a VM that reflects features of a DSL would be useful, and even more so a process VM that takes on a more active approach and allows these features to be scalable and configurable by a language designer. These types of process VMs are known as *high-level language VMs* (HLL VMs), illustrated in Figure 16. The .NET CLR and the Java VM are both examples of such HLL VMs and, incidentally, both use a stack based ISA. Operands, of whatever type they may be, are implied by their position in the stack, thus resulting in executable binaries of a much smaller footprint.

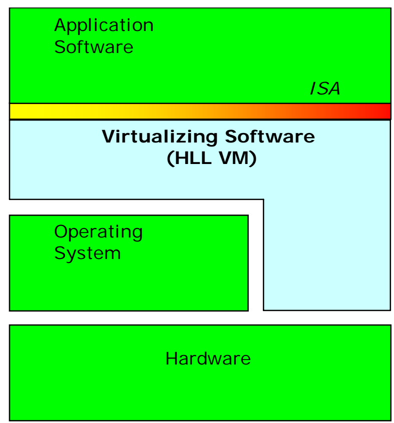


Figure 16. A Process VM. Virtualizing software supports an individual process.

For completeness sake, the summarized VM hierarchy of the afore-mentioned VM types is shown in Figure 17 below. The classification is taken from [Smi05]. Also, it will be the assumption that some form of hardware does indeed exist lower down the various layers of abstraction, so it is not important to stress that part of the system. The interest is at the level of the HLL VM's ISA, and more to the point, that there will be some kind of useful ISA for the designer of the DSL VM. The terminology will change slightly as the view of the different approaches change, but will be clearly defined as the approaches are discussed.

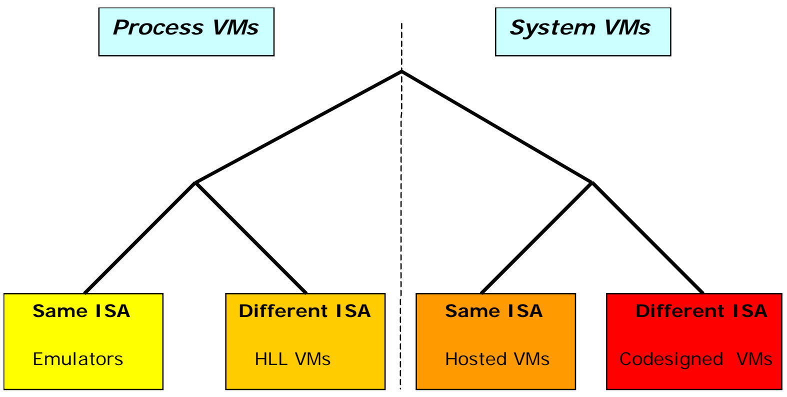


Figure 17. Broad classification of VMs.

4.4 View of the Virtual Machine Approach

In the proceeding sections the discussion centered around VMs that facilitate an ISA for a DSL. This type of HLL VM shall be named a *DSL VM*. Such a process VM runs as a program written in a high-level object oriented language, in this case C# on the .NET platform. The ISA provided by the DSL VM will undoubtedly be much smaller than that of the underlying HLL VM in which it was written, in terms of the sheer number of instructions, though this does not necessarily have to be the case, depending on the level of complexity or flexibility that the DSL provides (or will provide in future iterations).

> **DSL VM** : A concrete instance of a HLL VM, providing an ISA capable of supporting the semantics of some DSL, i.e. the semantics of a single instruction or combination of some sequence of instructions, will facilitate the semantics of the DSL, and thus will reflect the application domain for which the DSL was designed.

As defined in Chapter 1, a DSL-program is simply the source code of a program written in the language of the DSL. To take that definition a step further, it is necessary here to define the compiled state of that program, referred to as the DSL-compiled program, The DSL-compiled program will be executed by the DSL VM.

> **DSL-compiled program** : The DSL-program, once it has been compiled into a suitable ordered sequence of instructions ready to be executed on a DSL VM.

The position of the DSL VM relative to the all-important HLL VM on which it itself executes, and of course the DSL-programs written by the domain expert, is clearly shown in Figure 18.

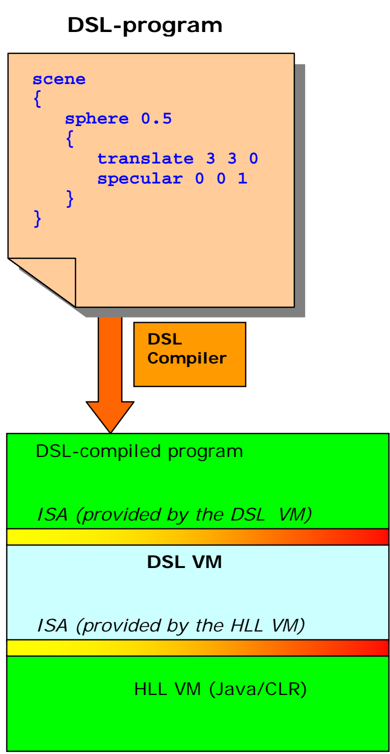


Figure 18. The DSL VM relative to the HLL VM on which it was written.

The same RT DSL previously introduced will be used to illustrate aspects of the design of the VM approach. In this approach, a set of instructions has been hard coded in the VM. A DSL-compiler first translates the RT DSL-program (a scene description) into an ordered list of instructions that is understood by the RT VM, shown in Figure 19. It is relevant to note at this point that a DSL-compiler is required for each DSL. There will be a DSL-compiler that will translate a program written in RT into a sequence of instructions that will be executed by the RT VM. Similarly there will be a DSL-compiler that will translate each of the other example DSL-programs into a sequence of instructions targeted to the relevant DSL VM. The theme of the next section will make note of some pitfalls in using this typical VM approach, with particular attention to making further additions to the RT DSL.

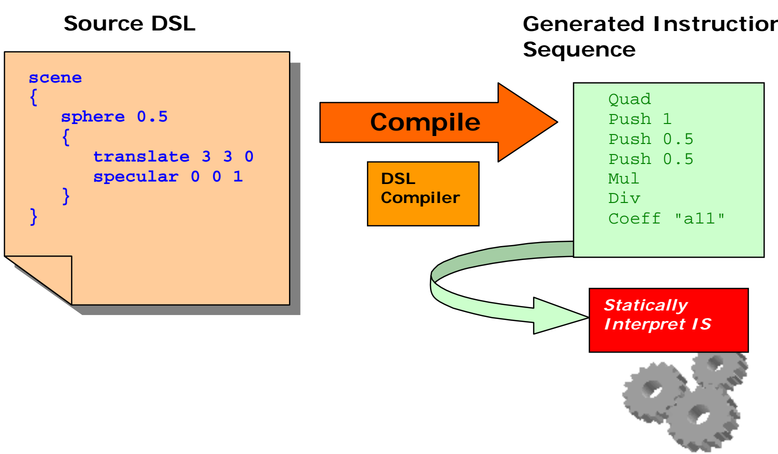


Figure 19. Conceptual view of the VM Approach.

Since the VM needs to interpret instruction sequence opcodes and operands, a separate parser and lexer is needed in the VM itself to read the DSL-compiled program. Figures 20 and 21 give the regular expression definition and grammar definition of the lexer and parser.

```
%lexer
%token                    DOUBLE {public double val;}
"PUSH"                    %PUSH
"SPHERE"                  %SPHERE
"PLANE"                   %PLANE
"WIN"                     %WIN
"LIGHT"                   %LIGHT
"COP"                     %COP
"LOOK"                    %LOOK
"VUP"                     %VUP
"AMB"                     %AMB
"GEOM"                    %GEOM
"SPEC"                    %SPEC
"DIFF"                    %DIFF
"REFL"                    %REFL
"REFR"                    %REFR
"AXES"                    %AXES
"REN"                     %REN
[-+]?[0-9]+(\.[0-9]+)?    %DOUBLE {val = Double.Parse (yytext);}
[ \n\r\f\t]               ;
```

Figure 20. Lexer definition for the RT VM.

At the outset, it can be seen that the grammar for the parser is very basic, as it simply matches any one of a set of predefined opcodes defined as keywords in the lexer definition. In the next chapter this grammar will be revisited, as this basic grammar will not be sufficient. It does not make provision for labels or for branching instructions, and only the `Push` instruction supports an operand.

```
asm_list :
asm_list : asm_list asm

asm : PUSH DOUBLE
asm : SPHERE
asm : PLANE
asm : WIN
asm : LIGHT
asm : COP
asm : LOOK
asm : VUP
asm : AMB
asm : GEOM
asm : SPEC
asm : DIFF
asm : REFL
asm : REFR
asm : AXES
asm : REN
```

Figure 21. Parser definition for the RT VM.

4.5 Scaling up the RT DSL for the VM Approach

As before, an attempt is made to scale up the RT DSL by adding an *`ambient`* construct. Because the process that parses the DSL-program (the DSL-compiler) is now separate from the process performing the logic to provide the semantics (the RT VM), the modifications are needed in both the VM and the compiler. The compiler's lexer will need to incorporate a new keyword `ambient` as before, and the grammar for the compiler's parser needs to be altered to produce a valid node on the AST. This amounts to the same difficulty as that involved in amending the interpreter described in Chapter 3.

However, this additional AST node will be handled differently by the DSL-compiler. Instead of providing new functions to be called by an interpreter and changing the main program of the interpreter to make these function calls (as was illustrated in Section 3.7.2), the DSL-compiler now has to generate a sequence of instructions that alter the ambient lighting of the scene. The VM will thereafter execute these instructions to adjust the ambient lighting in the scene.

A new instruction needs to be added to the instruction set, so a modification is needed to the VM. The lexer and parser for the VM needs to match on the new instruction, and the VM needs to have a function that provides the implementation for executing the new instruction. These

modifications will be shown in the next section. In Chapter 6, a more efficient means of appending a new instruction to the VM will be discussed and compared to the one mentioned shortly.

4.5.1 Adding a Construct to a VM

The changes to be made to the DSL-compiler that has just been mentioned are fairly trivial: one merely has to ensure that the lexer and grammar cater for the new *`ambient`* construct. Given the set of instructions, it is necessary also to modify the VM to include a new instruction, `Ambient`, with no operands. There is less risk in modifying the VM, than modifying the DSL-compiler, as the VM only requires one additional function to interpret the new instruction, whereas the interpreter approach needs an additional function that will be called during the traversal of the AST. Adding these functions is a less trivial task in the case of the interpreter approach, as it involves changes to the logic of the interpreter code itself.

4.5.2 Components Requiring Alteration

The alterations to the RT-compiler are similar to the alterations made to the interpreter front-end outlined in Section 3.7.2, insofar as the lexer is concerned. The RT-compiler's parser though, needs to handle the `ambient` node on the AST differently. The translation of the `ambient` node on the AST is required to generate a sequence of instructions that will alter the ambient light within the scene that is to be rendered. The translation of this node is handled by a new the function `scene_prop6()`, shown in bold in Figure 22. The sequence of instructions that are generated for the *`ambient`* construct is shown in the body of this function.

```
...
%node scene_prop6 : scene_prop
{
        public double d1, d2, d3;
        public scene_prop6 (double d1,double d2,double d3)
              {this.d1=d1;this.d2=d2;this.d3=d3;}
        public override void translate (StreamWriter asmFile)
        {
                asmFile.WriteLine ("Push\t{0}", d1);
                asmFile.WriteLine ("Push\t{0}", d2);
                asmFile.WriteLine ("Push\t{0}", d3);
                asmFile.WriteLine ("Ambient");
                asmFile.WriteLine ("Axes\n");
        }
}

...

element : scene_prop
element : prim:p LBRACE prim_prop_list:ppl RBRACE

scene_prop : WINDOW DOUBLE:d1 DOUBLE:d2
scene_prop : LIGHT DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
scene_prop : COP DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
scene_prop : LOOKAT DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
scene_prop : VIEWUP DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
scene_prop : AMBIENT DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
             %scene_prop6 (d1.val,d2.val,d3.val);

prim_prop_list :
prim_prop_list : prim_prop_list:ppl prim_prop:pp
...
```

Figure 22. Altering the grammar for the new *ambient* construct.

One of the instructions to be emitted by the compiler is the `Ambient` instruction which is to be newly defined in the VM as a function call. The red, green and blue component operands to the `Ambient` opcode are pushed on the stack prior to its execution. The `Push` instruction already exists as part of the RT VM's instruction set. Furthermore, the VM's lexer and parser need to match on the new `Ambient` instruction. The modifications are highlighted in bold in Figures 23 and 24 below.

```
%token                     DOUBLE {public double val;}
...
"Window"                   %WIN
"Light"                    %LIGHT
"COP"                      %COP
"LookAt"                   %LOOK
"VUP"                      %VUP
"Ambient"                  %AMB
...
```

Figure 23. Adding the `Ambient` instruction to the VM's lexer definition.

The parser definition simply matches on the keyword `Ambient`, and calls the appropriate translation rule. In this case it calls the function `RT_AMB()`, defined in the `RTMachine` class in the `RTVM` namespace.

```
...
%node asm9_amb : asm
{
        public override void interpret (RTVM.RTMachine rtm)
        {
                rtm.RT_AMB ();
        }
}

...

asm_list :                      %asm_list1 (null, null);
asm_list : asm_list:al asm:a    %asm_list1 (al, a);

asm : PUSH DOUBLE:d             %asm1_push (d.val);
asm : SPHERE                    %asm2_sphere ();
asm : PLANE                     %asm3_plane ();
asm : WIN                       %asm4_win ();
asm : LIGHT                     %asm5_light ();
asm : COP                       %asm6_cop ();
asm : LOOK                      %asm7_look ();
asm : VUP                       %asm8_vup ();
asm : AMB                       %asm9_amb ();
asm : GEOM                      %asm10_geom ();
...
```

Figure 24. Parsing the new `Ambient` instruction.

The definition for `RT_AMB()` is to simply pop the three red, green and blue component operands off the stack in reverse order, and apply them to the scene, as shown in Figure 25.

```
public class RTMachine
{
        public RTMachine (int rd, int stackSize)
        {
                rtstack = new RTStack (stackSize);
                scene = new Scene ();
                render_depth = rd;
        }

        ...

        // Alter the current ambient colour vector.
        public void RT_AMB ()
        {
                double amb = (double) rtstack.pop ();
                double amg = (double) rtstack.pop ();
                double amr = (double) rtstack.pop ();
                scene.alterAmbient (amr, amg, amb);
        }

        ...
        private asm_list ast;
        public RTStack rtstack;
        public Scene scene;
        public int render_depth;
}
```

Figure 25. The semantics for the new `Ambient` instruction in function `RT_AMB()`.

4.6 Modularity

The VM approach makes far better use of modularity. The VM is not a compiler but simply interprets instructions, so issues of syntax are handled by separate compiler software. The VM approach is closer to that of the Shlaer-Mellor method in that issues of semantics are exposed in the form of an instruction set. A sequence of instructions or a DSL-compiled program thus dictates a specific runtime execution of the domain, and these instructions are simply executed by the VM itself.

However, a developer would still like to be able to use the VM as a reusable component of a much larger system, but the VM has shortfalls for being used in this context. For example, suppose the images generated by the RT DSL are to be rendered immediately onto a visible canvas, instead of writing the images out to a file. Though possible, it is cumbersome to integrate this type of modification into the instruction set.

4.7 Summary

In this chapter, VMs are examined as the core functional mechanics used to provide the operational semantics of a DSL-program, or more specifically, of the DSL-compiled program produced by the DSL-compiler. A VM hierarchy was discussed in this chapter, with descriptions of the two main classes of VMs, namely process VMs and system VMs. One of the classes, namely the class of HLL VMs, was identified as being particularly relevant in the context of this study. This class of VM includes what has been termed the DSL VM, which runs as a single process program on another underlying HLL VM. (It may be noted that multi-process DSL VMs are indeed a possibility but the complexity of concurrent processes are beyond the scope of this dissertation.) The underlying HLL VM may be a Java VM or in this case, the .NET CLR.

The DSL-compiled program is in the form of a sequence of instructions and their operands. The VM loads the compiled program and begins execution from the first instruction. The implementation of the VM is built on top of the existing namespace (the domain) and the namespace is exposed via the instruction set. Thus, execution of these instructions implies some calls into the namespace. Logical interpretation of a DSL-program is done in two steps – first the DSL source needs to be compiled into a DSL-compiled program. For this a compiler is needed and a set of translation rules that map the DSL syntax into segments of instruction sequences. Then the instructions are then executed by the VM. A separate parser is needed to load the compiled program into memory before execution.

Scaling up the DSL, as in the case of adding an *`ambient`* construct to the RT DSL, requires modification on both the compiler and the VM. The VM needs to be modified to support a new instruction to change ambient values; the compiler needs to be modified to parse the new *`ambient`* construct, and to generate the necessary sequence of instructions, including the generation of the `Ambient` instruction as part of the sequence.

5 Domain-Specific Virtual Machines: A More General Approach

This chapter introduces the abstract VM architecture into the design of DSLs. It is abstract, in the sense that there are no instructions, and a VM instance cannot execute any given DSL-compiled program until it has been configured. It will appear to a developer as a VM with an empty instruction set and no environment. This will give the developer complete freedom in designing an instruction set and an environment appropriate for a particular namespace. For further VM design architectures, see [Jac96, Rie01].

Once a computation or program execution has completed, an instance of a result data type can be retrieved if necessary. This retrieval of a result makes the VM Framework versatile from a modularity perspective, as the result type can be user-defined. The instantiated VM object will feature methods that allow configuration of the VM itself. It will also feature appropriate methods to parse and load a DSL-compiled program, to execute a loaded program and to retrieve the result of a program execution. These will all be discussed in detail.

The VM Framework proposed in this section is described in detail in Section 5.1 through to Section 5.4. Section 5.1 explains the use of parsing DSL-compiled programs at a textual level, and not a binary level, i.e. having the DSL-compiled programs human readable. Section 5.2 provides a brief overview of the instruction classes, and configuring the VM instance with each instruction class, and an environment. Section 5.3 describes the internal control loop responsible for executing each instruction of a loaded DSL-compiled program and Section 5.4 describes how a result of a computation can be retrieved after a loaded program has finished executing. Section 5.5 provides a view of the VM Framework as the core mechanics providing the semantics for the DSL, and Section 5.6 provides a more detailed design description.

Environments and instructions and their respective classes are discussed in Sections 5.7 and 5.8 respectively, followed by a summary of the chapter.

5.1 Compiled Program Parsing and Loading

Section 4.4 gave a high-level description of the instructions and the grammar for parsing the different instructions. In this chapter, each instruction has been classified according to one of five categories and each category is so defined by the type of operand following the instruction. These categories will be fully fleshed out in Section 5.8. The VM program loader has a parser that will match on the appropriate instruction as it loads a DSL-compiled program and hence it will be able to identify to which category it belongs. Extending the Framework to support an instruction set will not require any modification to the program loader's grammar, but only the creation of specific instructions from one of the five categories. Each category of instruction is represented in code by an abstract class that is subtyped into a concrete subclass, representing an actual user-defined instruction. This means after a user-defined instruction class has been created and registered with the VM instance using the configuration file, the VM's loader will automatically be able to parse an occurrence of the specified instruction during program load time.

5.2 Instruction Set and Environment Configuration

To use the VM Framework, classes have to be created that extend the instruction classes and environment classes. This means that each user-defined instruction class must extend one of the five abstract instruction classes, and there must be at least one user-defined environment class extending the abstract environment class. Such a collection of classes defines a *configuration* to support a given DSL. It is possible to have multiple configurations by setting up various configuration files for different types of DSLs. In this way an instantiated VM can be used as part of a DSL prototyping environment. Each time a different language needs to be prototyped a new configuration file is loaded by calling the appropriate method on the instantiated VM object, and in turn, the VM will create a new pool of instruction classes (the user-defined classes, specified by the configuration file). Each user-defined instruction class from this pool will be instantiated once a specific occurrence of an instruction is loaded during the loading of a DSL-compiled program, i.e. an instruction object instance of one of these instruction classes encapsulates the instruction's operand, if any. An array of instruction instances thus represents a loaded program. There is no pool of environment classes, simply a pool of instances of the user-defined environments and as such the environments are simply instantiated during the loading of the configuration file.

5.3 Program Execution

Once the sequence of instructions defined in the DSL-compiled program have been parsed and loaded into memory as described in the previous section, they can be executed. The main loop that is responsible for the execution of the loaded program is shown in Figure 26. This shows how execution of an instruction is handled according to its class. Using simple reflection (i.e. using the C# boolean operator, `is`) it is possible to identify the class of the current instruction before it is executed.

Before entering the loop, the first instruction is fetched from the memory. This simply amounts to retrieving the first instruction in an array of instructions that represents a loaded DSL-compiled program. Once in the loop, a check is done to determine if the instruction is of a type that has an ID operand. These instructions will be described later in Section 5.8, however for the sake of explanation an instruction with an ID operand is simply an instruction that references one of the VM's internal temporaries. The reference is simply an index value. These temporaries can be an object of any user-defined type. They can be instantiated by the instruction and later retrieved again. The first block in Figure 26 handles these types of instructions. It retrieves the appropriate temporary from the `temps[]` array and supplies it to the instruction before it is executed, if such a temporary exists.

If the instruction is one of the four remaining instruction types, the instruction is executed immediately, however, a further nested check is performed to determine if a branch instruction is being executed at the current point of execution or if a previous point of execution should resume. These types of instructions may be used, for example, to jump to a subroutine and return from a subroutine.

```
if (loader.hasProgram ()) {
        // Grab the first instruction for execution.
        current_ins = loader.FetchNext ();

        // Keep executing until there are no more instructions to execute.
        while (current_ins != null) {
                if (current_ins is Instruction_OpCode_IDOperand) {
                        id_ins = (Instruction_OpCode_IDOperand) current_ins;

                        // Get temp value and supply it to the instruction.
                        if (temps [id_ins.ID] == null) {
                                // Doesn't exist in list, initialize it to 0.
                                id_ins.temp = new object ();
                                temps.Add (id_ins.ID, id_ins.temp);
                        }
                        else {
                                id_ins.temp = temps [id_ins.ID];
                        }

                        // Execute the instruction with latest temp value.
                        current_ins.Execute ();

                        // Store the temp value back into the local list.
                        temps [id_ins.ID] = id_ins.temp;
                }
                else {
                        current_ins.Execute ();

                        if (current_ins is Instruction_OpCode_BrOperand) {
                                br_ins = (Instruction_OpCode_BrOperand) current_ins;

                                // If we must save the program counter, we must do
                                // it before we branch to another point of execution.
                                if (br_ins.SaveProgramCounter) {
                                        loader.SavePC ();
                                }

                                // OK, now its safe to branch.
                                if (br_ins.BranchCond) {
                                        loader.Branch (br_ins.label);
                                }
                        }
                        else if (current_ins is Instruction_OpCode) {
                                ins = (Instruction_OpCode) current_ins;
                                if (ins.LoadProgramCounter) {
                                        loader.RecallPC ();
                                }
                        }
                }

                // After updating the program counter, grab the next instruction.
                current_ins = loader.FetchNext ();
        }
}
```

Figure 26. Main loop for executing loaded instructions.

5.4 Output Results

Figure 27 shows the `GetResult()` method that is to be found in the `EVM` class (Section 5.6.1). It uses a string parameter that is the name of an environment that was registered with the VM instance. One of the encapsulated objects of the `EVM` class is the `config` object of type `Config`, which simply manages the pool of instruction classes and environment instances, as well as the mapping that links an environment to an appropriate instruction class. The `Config` class encapsulates an array `environments[]` that is the pool of instantiated environments for the

current VM configuration. The string parameter of GetResult() simply uses the string name of a registered environment as the index to the array environments[]. Indexing arrays with a string is an elegant feature of C#. This retrieves a specific environment instance, env, and in turn the function GetResult(), in this case, now a member function of Env, is then called on the object env. The function GetResult() is abstractly declared in Env and it is the obligation of the implementer of a subclass of Env to decide what should be returned as a result. More about this inner call to GetResult() will be given in Section 5.6.5 and an example will be presented in later sections. The method GetResult() is called after the VM instance has completed execution of a loaded program to retrieve any result of a computation performed by the execution of the loaded program. If no such result is expected, the function needn't be called.

While the result is returned generically as an object, its actual data type forms part of the namespace and essentially describes an object that makes sense as a result of executing the loaded program. For example, in a simple real-valued expression DSL, the results can be a simple C# float data type, while an RT DSL may return a System.Drawing.Bitmap instance as a result of rendering object primitives within a scene. The VM Framework needn't have any knowledge of this data type and hence it does not form part of the configuration in any way.

```
public object GetResult (string Environ)
{
        Env env;

        env = (Env) config.environments [Environ];
        return env.GetResult ();
}
```

Figure 27. The GetResult() method of the EVM class.

It is possible to switch between different environments and instructions at runtime. This makes the VM well suited in a language prototyping application, where the same VM can be reused, and there is no need to recompile a specification to generate a new machine for a new language.

5.5 View of the Virtual Machine Framework Approach

The raw execution rationale of an instantiated VM of the VM Framework is similar to that of the VM approach discussed in Chapter 4, in that an compiled program is first loaded and then each instruction is executed in sequence. However, there is an initial phase to set up the VM Framework for a particular DSL. This is done via the configuration file. An instantiated VM reads the configuration file, as depicted in Figure 28, and creates the necessary instructions and environment instances accordingly. As in the VM approach, the source DSL-program is still compiled. However in the VM Framework approach the semantics of the target language is

encapsulated in instruction classes that are subtypes of abstract instruction classes, which are part of the VM Framework. Also, in the VM Framework approach, a result data type can be setup by the user. After the instantiated VM has executed a sequence of instructions defined in the loaded program, then the user of the VM can retrieve the instance of the result data type.

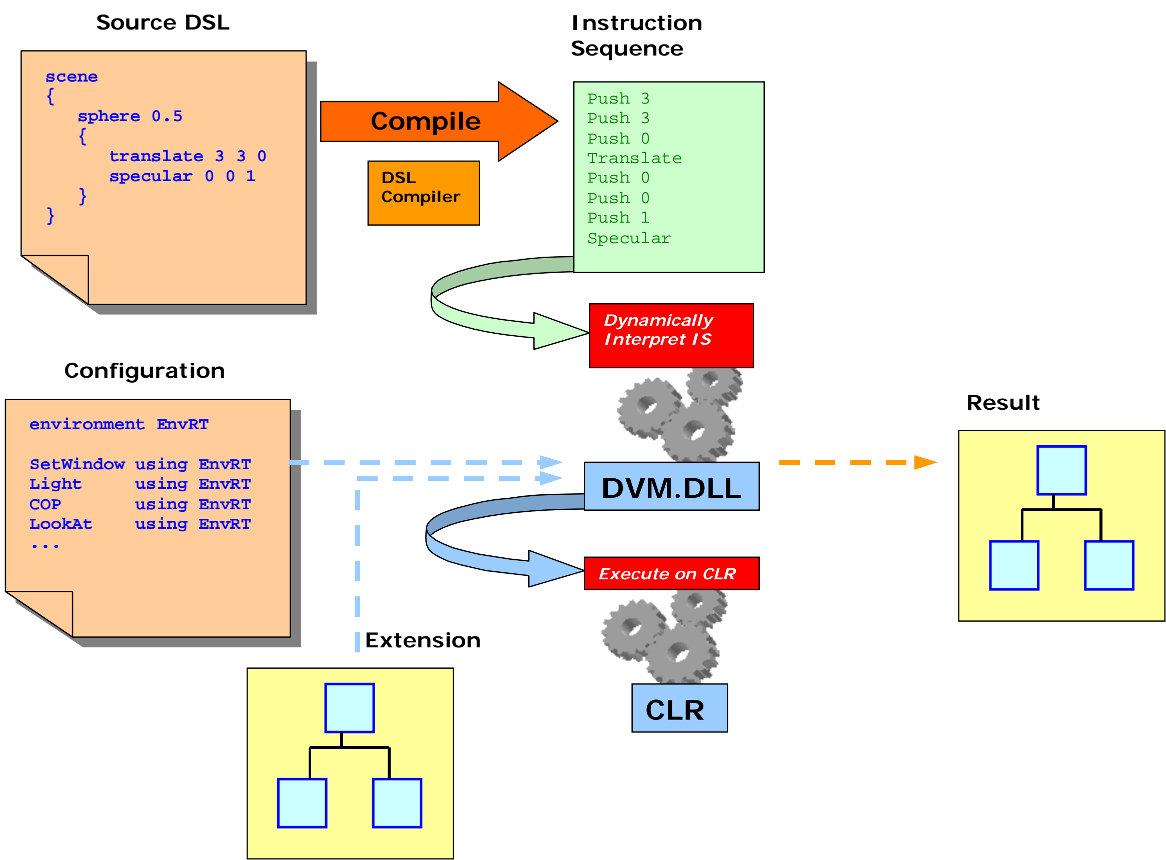


Figure 28. Conceptual view of the VM Framework Approach.

5.6 Virtual Machine Framework Detailed Design Description

The VM Framework provides the basic functionality of a typical VM, including a DSL-compiled program loader, a program counter, internal temporary values, and conditions on which to build branching instructions. A top-level class (the `EVM` class) is provided through which to start up and configure an instance of a VM. It also provides the interface for the loading and running of a program. The actual configuration of the instantiated VM, i.e. the environment and instruction set setup, is done through the configuration file discussed in Section 5.6.2. Thus no modification to the VM Framework itself is required and its component classes can consequently be compiled and saved as a library. The VM Framework consists of five main classes, (excluding the .NET `Type` class) each discussed in the following sections. They are the `EVM` class discussed in Section 5.6.1, the `Config` class discussed in Section 5.6.2, the `Loader`

class discussed in Section 5.6.3, the `Inst` abstract instruction class discussed in Section 5.6.4 and the `Env` abstract environment class discussed in Section 5.6.5. They all belong to the `DVM` (Dynamic VM) namespace, and their inter-relationship is depicted in Figure 29. The VM Framework, in its compiled form, amounts to a single .NET DVM.DLL file. One of the classes, the `Inst` class, has five abstract instruction subclasses. They are not shown in Figure 29, but they will be discussed later in Section 5.8.

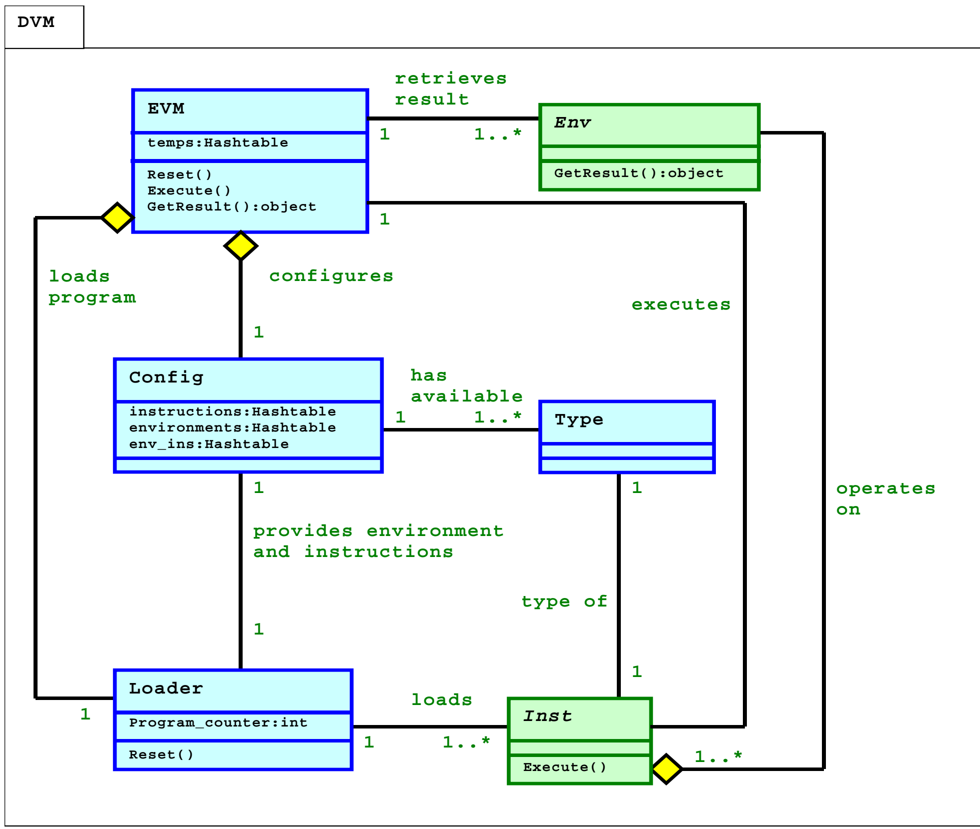


Figure 29. Information Model of the VM Framework namespace.

5.6.1 The `EVM` Class

The `EVM` class (Extendable Virtual Machine) is the top-level class of the VM Framework. An instance of this class is a reference to an instantiated VM object. Configuration of the instantiated VM, loading of DSL-compiled programs, execution of loaded programs etc., is performed on this object. Once instantiated, the object represents an instance of a configurable VM, with an empty instruction set and no environment, i.e. an instance of this object represents the VM itself. As such, this implies that an instance of the `EVM`, once configured, is in fact a DSL VM, providing some DSL ISA as laid out in the configuration file.

A specific configuration can then be assigned to the VM instance, and a program can be loaded and executed. When a program is executed, then the VM, through the use of dynamic binding,

will invoke the correct instruction instance. These instruction instances are created at program load time and their corresponding classes (all subclasses of `Inst`) are defined in the configuration file.

The `EVM` class also encapsulates the internal temporary values in the `temps` hashtable. As mentioned in Section 5.3, temporaries are simply any type of object that the user may wish to instantiate. Having access to temporaries, referenced with an ID, is a feature of the VM Framework that is provided for convenience. One instruction may create a value, while another may make use of the value as it was created by an earlier instruction. They are useful in representing variables of some DSL (such as variables of an integer or double type), though the `EVM` allows them to be of any type. It is up the compiler to make sure the temporaries are accessed correctly by the respective instructions. Each internal temporary value has a unique ID, and instructions with ID operands can gain read and write access to them. The temporary values have an object type, so they can be assigned values most convenient to the DSL being constructed.

Internally, the `EVM` class contains a loop in its `Execute()` method (Figure 26), that iterates through each instruction stored by the `Loader` class. The very next instruction to be executed is first fetched, and then the `Execute()` method of its class is invoked. This `Execute()` method will vary according to which of the five instructions discussed in Section 5 has been fetched. This may entail accessing an internal temporary value or handling a branch instruction and saving the current program counter value, if need be. Some branch instructions do not require the program counter to be saved.

5.6.2 The `Config` Class

The `Config` class is responsible for the configuration setup of an instantiated VM. The `Config` class encapsulates three mappings: `instructions`, `environments` and `ins_env`. They are hashtables, and their respective mappings are shown in Figure 30 below.

```
instructions : string → Inst Type
environments : string → Env
ins_env      : string → string
```

Figure 30. Configuration mappings.

The `instructions` mapping maps the string name of an instruction to an `Inst` *type*. Note carefully that the mapping is not from the string name to an `Inst` *instance*. Instead it is from

the name to one of the five `Inst` classes defined in Section 5. Instances of the `Inst` class are only created upon program loading.

The `environments` mapping, maps the string name of an environment, to an actual instance of `Env`. This is in contrast to the `instructions` mapping, which does not map to instances, but rather to classes (subclasses of `Inst`) that are yet to be instantiated. As indicated below in Section 5.6.5, an `Env` instance will typically encapsulate some Abstract Data Type (ADT) such as a runtime stack, or any other data structure that proves to be sufficiently useful for facilitating the language of the DSL that will ultimately make use of the VM instance.

The last mapping, `ins_env`, maps the string name of an instruction to the string name of the environment that the instruction is to use. This `ins_env` mapping is an associative mapping since more than one instruction can use a singe environment. This will be the environment that the instruction will access and update during the execution of the loaded program. It is noted at this point that the `Config` class is a not an associative object that maps a single string name (of an instruction, or environment) to its respective class, or instance. It is a class that encapsulates the mappings themselves, using hashtables. Thus there is only one instance of the `Config` class and it is instantiated once the `EVM` class has been instantiated. The hashtables are populated after the VM instance (an instance of the `EVM` class) has been configured with an appropriate ISA using the configuration file.

5.6.3 The `Loader` Class

The `Loader` class encapsulates a loaded program and the program counter. The loaded DSL-compiled program is an array of instruction instances, each a subclass of `Inst`. The `Loader` class also maintains a hashtable that maps labels of any labeled instructions encountered during the loading of a DSL-compiled program to that instruction's index in the array representing the loaded program. Instructions that have labels are usually the target of a branching instruction. It is convenient to view a DSL-compiled program in human readable form and thus each label is simply an integer number preceded by the symbol `@`, and is placed before the instruction being labeled. Upon loading the DSL-compiled program, the mapping is updated with a valid index entry for each label.

When a program is loaded, an instance of each instruction is created for each instruction in the program. The parser of the instructions in the VM only has the string name of instructions. It uses the `instructions` mapping defined in the `Config` class to retrieve the actual class that is used to create the instruction instance. Thus when a program is fully loaded, the array will contain instances of instruction classes, each a subclass of `Inst`, and each instruction

encapsulating its own operand ready for execution. The program counter is reset to the beginning of the array.

5.6.4 The `Inst` Abstract Class

The abstract `Inst` class encapsulates a reference to an instance of a subclass of the `Env` class, discussed in Section 5.6.5. This will be an environment object that is updated by each instruction during the execution of the loaded program. Thus all instructions in a loaded program will have a reference to the same `Env` instance because all instructions in the ISA defined by the configuration file are subclasses of `Inst`.

The `Inst` class does not define how the updates to the environment object are to be performed by a given instruction. Instead the `Inst` class provides an abstract `Execute()` method, that concrete subclasses are obligated to override. The `Inst` class is subtyped by a further five abstract classes, one class for each type of instruction supported by the VM Framework. Concrete instruction classes are classes that are subclasses from one of these five abstract classes, and should not be subtyped from `Inst` directly. Each of these five abstract classes will be discussed in Section 5.8.

Execution of a program implies an invocation of the `Execute()` method on each instruction instance that is referenced by the program counter, while there are still instructions to be executed by the loaded program. When the program counter has run through each instruction instance, the program has completed execution and the result of the execution can be retrieved. When a program is loaded by the program loader (an instance of the `Loader` class) the program counter is set to index 0 which is the first instruction in the program. As each instruction is executed the program counter is incremented by 1. Thus, the `Execute()` method of a given instruction instance can be viewed as encapsulating the semantics of the associated instruction. The sequence of Execute() methods performed in running a program then provides the semantics of the program as a whole.

5.6.5 The `Env` Abstract Class

The abstract `Env` class encapsulates some ADT or even a number of ADTs that form the central data storage mechanism for the DSL. The abstract `Env` class does not dictate the type of ADT that is encapsulated, and thus does not define any member ADT. It merely provides an abstract `GetResult()` method that subclasses of `Env` are obligated to override. In Figure 27 above, the call to `env.GetResult()` whose outcome is returned to the outer `GetResult()`

method, itself a method of the `EVM` class, illustrates where the `GetResult()` method of `Env` is actually used.

The `GetResult()` method of a subclass of `Env` should return the result of a computation, if the DSL-program was indeed written to perform some computation. This is especially the case in functional language DSLs where the result of a computation can be retrieved after the program has completed execution, i.e. after a call to `Execute()` on the instantiated VM. It may also be the case that a final result is not necessarily expected. Such may be the case if the user-defined instructions are printing intermediate values to calculations onto a console or window at the time they are being executed. In this case `GetResults()` still needs to be implemented so as to yield a concrete class, however it simply needs to return `null`.

5.7 Environments

The purpose of the abstract `Env` class is to have an ADT that is updated during runtime. It is intended to aid in the processing of the semantics of the DSL in an appropriate or convenient manner. For example, in a simple real-valued expression language, a runtime stack can be used as an environment, where operands are first loaded onto the stack and then an arithmetic operation is performed on the most recently pushed values, depending on the number of operands for a given operation. In a ray-tracer [Wat00a] scene description language, the main data structure may be a runtime stack, for any arithmetic calculations, and a bitmap image data type that is incrementally updated as the image information is processed. Thus it is possible to extend the environment built for an expression language, into one that is suitable for a ray-tracer language.

Classes that extend the abstract `Env` class, are obligated to override the method `GetResult()`. The method `GetResult()` returns an instance of an object. When an instance of a VM has completed execution, the user can call `GetResult()` to retrieve the result of the execution. In the example of an expression language, this would typically be a `double` value, while for a ray-tracer language this result would be an instance of a `System.Drawing.Bitmap` image type. Since VM Framework users will be aware of the data type that they are using for the result, a simple type cast to narrow the returned instance to the user's own result type is sufficient.

An example of `EnvExp`, a concrete extension to `Env` for an expression language, is provided in Figure 31. It encapsulates a real-valued stack, and its `GetResult()` method returns the last entry on the stack or –1 if the stack is empty. If all operations on the stack are consistent, there should be only one remaining value on the stack, which will be the result of evaluating some expression. The `stack` object itself can be declared as `public` or `protected` allowing

subclasses access to it for any environments that may later subtype EnvExp and still need a stack. New methods can also be added to the subclass. In this example the method GetStack() is added so instructions that have a reference to an object of type EnvExp can get a reference to the stack itself. Another alternative would be to expose all the methods of the Stack class in the EnvExp class.

```
class EnvExp : Env
{
   public EnvExp () {
      stack = new Stack (100);
   }
   public override object GetResult () {
      object result;
      if (stack.isEmpty ()) {
         result = -1.0;
      }
      else {
         result = stack.peek ();
      }
      return result;
   }
   public Stack GetStack(){return stack;}
   protected Stack stack;
}
```

Figure 31. Example EnvExp class.

5.8 Instructions

The instructions form the basic operational building blocks to a DSL-compiled program. There are five classes of instructions, each represented by an abstract class extending the abstract class Inst. The user creates their own instruction by extending one of the classes of the five abstract instructions provided, depending on the required operand of the instruction.

In the current implementation of the VM Framework, each instruction is allowed to take at most one operand. This design decision was taken to keep the VM Framework relatively simple. However, there is no inherent reason for not allowing more than one operand per instruction. The implications of accomplishing it are beyond the scope of this dissertation.

Being limited to one operand per instruction directly influences the flexibility of the instruction set, as well as its use in combination with an appropriate environment. The five abstract classes of instructions are defined in terms of their operand types. They are Inst_OpCode, Inst_OpCode_BrOperand, Inst_OpCode_IDOperand, Inst_OpCode_StrOperand and Inst_OpCode_NumOperand, and they form part of the VM Framework namespace, DVM. Each of the five abstract classes of instructions will be explained in Sections 5.8.1 to 5.8.5. Any of the

five instructions can be labeled, if they are targeted by any branching instructions. The token and grammar definition for parsing a DSL-compiled program code is shown in Figure 32.

```
LABEL     : [lL][aA][bB][eE][lL]
INS       : [_a-zA-Z][_a-zA-Z0-9]*
BRANCH    : @[1-9][0-9]*
TEMP      : $[1-9][0-9]*
DOUBLE    : [-+]?[0-9]+(\.[0-9]+)?
STR       : \".*\"

ir_list   :
ir_list   : ir_list ir_instr

ir_instr : ir_label INS
ir_instr : ir_label INS STR
ir_instr : ir_label INS TEMP
ir_instr : ir_label INS DOUBLE
ir_instr : ir_label INS LABEL BRANCH

ir_label :
ir_label : BRANCH
```

Figure 32. Token definitions and grammar for the instructions.

5.8.1 Instructions with No Operands

An instruction with no operands is created by subtyping the abstract class `Inst_OpCode` and must have a constructor with the signature, `public` *`UserDefinedInst`* `(`*`UserDefinedEnv`* `env)`. This new concrete instruction class still has to provide an `Execute()` method, which is inherited as an abstract method from `Inst`. It also inherits a boolean property from the immediate parent abstract class `Inst_OpCode`. This boolean property, `LoadProgramCounter`, in Figure 33, is used to determine whether or not to recall the last saved program counter. This allows the creation of instructions that return from a branch into a subroutine, provided that the last branch instruction did indeed save the program counter. The `Execute()` method for such an instruction will then simply test the `LoadProgramCounter` property and change the program counter if necessary.

```
abstract class Inst_OpCode : Inst
{
   protected bool
   LoadProgramCounter = false;
}
```

Figure 33. The `Inst_OpCode` abstract class.

As a further example of an instruction without an operand, consider the `Add` instruction presented in Figure 34. It could typically be used in a simple expression language. The instruction `Add` simply pops the two topmost operands off a stack and pushes the sum back on.

Again, these stack operations and the addition operation would be invoked in the `Execute()` method of the `Add` instruction.

```
class Add : Inst_OpCode
{
   public Add (EnvExp env)
   {
      this.env = env;
   }

   public override void Execute ()
   {
      double d1, d2, r;

      d2 = (double)
          ((EnvExp)env).GetStack().pop();

      d1 = (double)
          ((EnvExp)env).GetStack().pop();

      r = d1 + d2;

      ((EnvExp)env).GetStack().push(r);
   }
}
```

Figure 34. The `Add` instruction.

5.8.2 Instructions with a Branch Label

An instruction with a branch label operand is created by subtyping the abstract class `Inst_OpCode_Br` and must have a constructor with the signature, `public` *`UserDefinedInst`* `(`*`UserDefinedEnv`* `env, string label)`. Instructions with a branch label are used for conditional or unconditional branching. Two properties are used to implement branching semantics depending on the requirement of the branch condition. The first property, `BranchCond`, is the actual condition for branching. This property should be assigned to true in the overridden `Execute()` method for unconditional branching. For conditional branching it is assigned according to the evaluation of a boolean expression inside the body of the `Execute()` method as per the required semantics of the condition. The second property, `SaveProgramCounter`, dictates whether the program counter should be saved for a corresponding return call into a subroutine. The branch label is supplied as a string parameter by the loader to the constructor of the instruction when it is instantiated. It is used internally by the `EVM` instance when the instruction is executed. The class is shown in Figure 35.

```
abstract class Inst_OpCode_Br : Inst
{
   protected string label;

   protected bool
   BranchCond = false;

   protected bool
   SaveProgramCounter = false;
}
```

Figure 35. The `Inst_OpCode_Br` abstract class.

5.8.3 Instructions with a Temporary

An instruction with a temporary ID operand is created by subtyping the abstract class `Inst_OpCode_ID` and must have a constructor with the signature, `public` *`UserDefinedInst`* `(`*`UserDefinedEnv`* `env, string label)`. Internally, so-called *temporaries* are stored in a hashtable that maps temporary names (IDs) to object references. They are akin to conventional microprocessor registers, but a temporary can be treated as any object type as illustrated in Figure 36. Instructions have full access to a temporary, so it is the obligation of the `EVM` instance to supply the necessary temporary object value to the current instruction before it is executed. Thereafter, the instruction can modify the temporary by typecasting the object to the required usable type. The temporary ID is supplied as a string parameter by the loader to the constructor of the instruction when it is instantiated. It is used internally by the `EVM` instance when the instruction is executed. In the EXP and LQ examples shown later in the next chapter, both use temporaries as temporary storage for variables. EXP uses temporaries for real-valued variables, and LQ uses temporaries for Quilt objects.

```
abstract class Inst_OpCode_ID : Inst
{
   protected string ID;
   protected object temp;
}
```

Figure 36. The `Inst_OpCode_ID` abstract class.

5.8.4 Instructions with a String Parameter

An instruction with a string operand is created by subtyping the abstract class `Inst_OpCode_Str` and must have a constructor with the signature, `public` *`UserDefinedInst`* `(`*`UserDefinedEnv`* `env, string str)`. Instructions with string parameters are useful in string processing applications such as those that deal with regular expressions. Alternatively, if variables are declared in the DSL, the string parameter can be used as a reference to a

variable name. The class is shown in Figure 37. The string value is supplied as a string parameter by the loader to the constructor of the instruction when it is instantiated and stored in the variable `str`.

```
abstract class Inst_OpCode_Str : Inst
{
   protected string str;
}
```

Figure 37. The `Inst_OpCode_Str` abstract class.

5.8.5 Instructions with a Number Parameter

An instruction with a number operand is created by subtyping the abstract class `Inst_OpCode_Num` and must have a constructor with the signature, `public` *`UserDefinedInst`* `(`*`UserDefinedEnv`* `env, double num)`. Similarly to the above string parameter, the number value is supplied as a parameter by the loader to the constructor of the instruction when it is instantiated. The class facilitates instructions that provide any intermediate arithmetic calculation. Real or integer numbers can be parsed and stored. However, internally they are treated as C# `double` values, and once parsed, the double parameter is stored in the variable `num`, shown in Figure 38.

```
abstract class Inst_OpCode_Num : Inst
{
   protected double num;
}
```

Figure 38. The `Inst_OpCode_Num` abstract class.

5.9 Summary

In this chapter the VM Framework is first presented and some of the details of its design are outlined further. The VM Framework provides a top-level class from which a VM is instantiated. The five instruction classes are all abstract, so any instantiated VM has an empty instruction set. Instructions and environments are created and compiled separate from the VM Framework and are a set of user-defined classes that can be compiled as DLLs. Instructions perform some operation based on a given environment and may or may not alter an environment's state.

The VM Framework provides a mechanism to parse instructions of a DSL-compiled program. The compiled program contains an ordered list of instructions and is human readable. The

parsed instructions are automatically classified as one of the five instructions supported by the VM Framework, based on the type of operand of the instruction in the compiled program. Environments also form extensions to the VM Framework and are also created and compiled separately as DLLs. The instantiated VM is configured via a configuration file that links the instructions and environments together. Initially the instantiated VM reads the configuration file and after initialization the VM is ready to execute instructions of a loaded program.

6 Extending the Virtual Machine Framework

Using the VM Framework entails designing a set of instructions and environments, without modifying the VM Framework itself in any way. This design is then implemented as a set of classes that subtype the classes that constitute the `DVM` namespace. This chapter is intended to point to the overall activity of defining a set of classes tailored toward some particular DSL.

The collection of user-defined classes constitutes an architectural layer of software. This layer encompasses the set of concrete instructions that can be instantiated at program load time, as well as an environment on which these instructions will operate. The configuration file is responsible for connecting the VM Framework to the user-defined layer.

This chapter provides an example of extending the RT DSL using the VM Framework approach. Also, two smaller DSLs are examined to see how different languages, with vastly differing uses, can be built. These two languages are a simple expression language (EXP), and a tile manipulation language called Little Quilt (LQ). The environments of EXP and RT are very similar and hence these two languages provide a simple and elegant illustration of how the basis for one DSL can be extended, or scaled up further to provide enough functionality to cater for the semantics of another language. It suffices to incorporate EXP's runtime stack to aid in any arithmetic operations that may be needed in performing calculations during the setting up of quadric equations in RT.

Section 6.1 starts the chapter by describing the configuration file, and Sections 6.2 and 6.3 continue by describing the use of the file to extend the instruction sets and the environments, respectively, for a particular VM configuration. Section 6.4 lists the three main ways to extend a VM in terms of the VM definition described in Chapter 4. Section 6.5 provides some example DSLs, and illustrates how these DSLs are built using the VM Framework, and Section 6.6

continues by providing an example of scaling up the RT language described in Chapter 3. Finally, modularity is discussed in Section 6.7, before providing a chapter summary.

6.1 The Configuration File

Before an instantiated VM can execute instructions in a loaded compiled program, the VM needs to be configured as a specific VM type. This is achieved through a configuration file that is initially loaded (or read) by the VM instance. Once the VM has been configured, a compiled program can be loaded and executed.

The configuration file defines an ISA for a particular VM instance and an environment used by the specified instructions themselves. More accurately, the file specifies the name of each user-defined instruction class together with the name of the user-defined environment class that each instruction will use during program execution. Thus the configuration file should enumerate the names of the complete instruction set for a particular VM. Figure 39 gives an example configuration file for the EXP DSL.

```
(* Create an instance of the *)
(* expression environment. *)
environment EnvExp

(* Register the following expression
(* instructions with the DVM. *)
Push  using EnvExp
Store using EnvExp
Load  using EnvExp
Sub   using EnvExp
Add   using EnvExp
Mul   using EnvExp
Br    using EnvExp
Brgz  using EnvExp
Nop   using EnvExp
Div   using EnvExp

(* Some generic instructions *)
Call  using EnvExp
Ret   using EnvExp
Print using EnvExp
```

Figure 39. Example configuration file for EXP.

The keyword `environment` is followed by the name of a class that subtypes `Env`, and one environment instance will be instantiated for that class. Thereafter, a list of clauses defines the actual instructions to be used in the current VM configuration, by the `using` keyword. The environment class name following the `using` keyword refers to the singleton environment instance whose creation was specified near the top of the configuration file after the `environment` keyword. The VM instance creates that environment instance as part of the configuration process, or simply put, when the configuration file loaded. Later, when the

program is read by the VM instance, then the environment instance will be used as the parameter to the various instruction constructors.

The user-defined environment class and the instruction classes are all compiled separately as DLLs, thus each DLL library file will be home to a single class. DLLs are loaded by the CLR dynamically, or at runtime. The advantage here is that the classes are not statically compiled in along with the VM Framework when a new VM is being built or prototyped. So there is no need for a complete recompile of the VM Framework. Rather, the VM Framework is compiled once, and only user-defined classes are compiled as separate DLLs and are loaded on demand by the CLR. The configuration file specifies the names of these DLLs, which is the same name of the class that each DLL contains. The configuration file thus uses the class names to load a specific DLL from disk.

6.2 Extending the Environments

Suppose a new language is required, be it similar to an existing language or one that features an entirely new syntax. If an existing language uses an environment with an appropriate data structure, then the new language can extend the existing environment to suite its own needs. An RT language needs to render a scene comprising geometric objects onto some canvas, but may also require a means to perform numeric calculations for setting up quadric objects. Thus the `EnvExp` environment of EXP can be extended with two extra data structures; a scene and a bitmap, giving rise to an `EnvRT` environment suitable for a ray-tracer, shown in Figure 40.

6.3 Extending the Instruction Sets

Instructions are extended from one of the five instruction classes mentioned in the previous chapter, to a set of concrete instruction classes instantiated whenever a program is read. It should be noted that these five instructions provide enough complexity to build a fairly versatile VM. As previously discussed, the VM Framework makes provision for instructions for branching; for accessing of primitive data types that can be written in text, namely numbers and strings; and for internal storage in the form of temporaries. If more complex data types such as classes or arrays are needed as part of the DSL, then it is the obligation of the DSL-compiler to provide a mapping of such complex data types and their associated operations onto the primitive date types and the set of instructions that use these primitive data types. Depending on how the DSL semantics are to be implemented, the user of the VM Framework can choose to create instructions specifically for the task of handling complex data types. For example, it should be possible to handle array indexing by defining a set of instructions for

handling arrays. An array construct defined as part of the DSL, will then be translated into a sequence of these instructions by the DSL-compiler.

While it is possible to subtype individual instructions further, it is not always necessary. However, it useful to extend instruction sets, and have environments that are subclasses of previously defined environment classes. This allows the VM Framework to be scalable, and aids in tailoring a VM for a particular DSL. The RT DSL serves as an example. The `EnvRT` environment was defined as a subclass of `EnvExp` in Figure 40. This means that any one of the instructions operating on an `EnvExp`, can also operate on an `EnvRT`. The configuration file for the RT DSL, depicted in Figure 41 shows this to be the case.

```
class EnvRT : EnvExp
{
   public EnvRT () : base ()
   {
      scene = new Scene ();
   }

   public override object GetResult ()
   {
      scene.render (4);
      return scene.GetBitmap ();
   }

   protected Scene scene;
   protected Bitmap bitmap;
}
```

Figure 40. The `EnvRT` class.

The configuration file for the RT DSL is shown below in Figure 41. Note that some instructions are *borrowed* from the EXP DSL. It enumerates not only RT related instructions such as `Light`, `COP`, etc, whose class definitions are specifically associated with ray tracing; it also enumerates a number of the instructions such as `Push`, `Pop`, etc. that are related to EXP.

```
(* Create an instance of the *)
(* ray-tracer environment.   *)
environment EnvRT

(* These instructions were part of    *)
(* the EnvExp environment.            *)
Push      using EnvRT
Add       using EnvRT
Sub       using EnvRT
Mul       using EnvRT
Div       using EnvRT

(* Ray-tracer specific instructions. *)
SetWindow using EnvRT
Light     using EnvRT
COP       using EnvRT
LookAt    using EnvRT
VUP       using EnvRT
Geometry  using EnvRT
Specular  using EnvRT
Diffuse   using EnvRT
Reflect   using EnvRT
Translate using EnvRT
Axes      using EnvRT
Coeff     using EnvRT
Quad      using EnvRT
```

Figure 41. Configuration file to setup RT, borrowing instructions from EXP.

6.4 The extended DSL VM

As one extends the environments and instruction sets, essentially one is creating new versions of ISAs, either for different DSLs with similar or same environments or with different environments and instructions altogether (ergo, different semantics entirely). It is obvious that the designer has the freedom to extend existing ISAs by extending the number of instructions with new instruction classes, and extending the environments using subtypes of existing environments. In the discussion below, subscripts will be used to identify the environment, VM, etc that is associated with a given DSL. Thus ISA_{EXP} represents an ISA associated with EXP, and VM_{EXP} is the VM associated with EXP, etc.

In Figure 42, the configuration file shown on the right defines the DSL VM. Thus an environment, along with a corresponding instruction set classes defines that DSL VM. The instructions listed in the configuration file, naturally define ISA_{EXP} in this case.

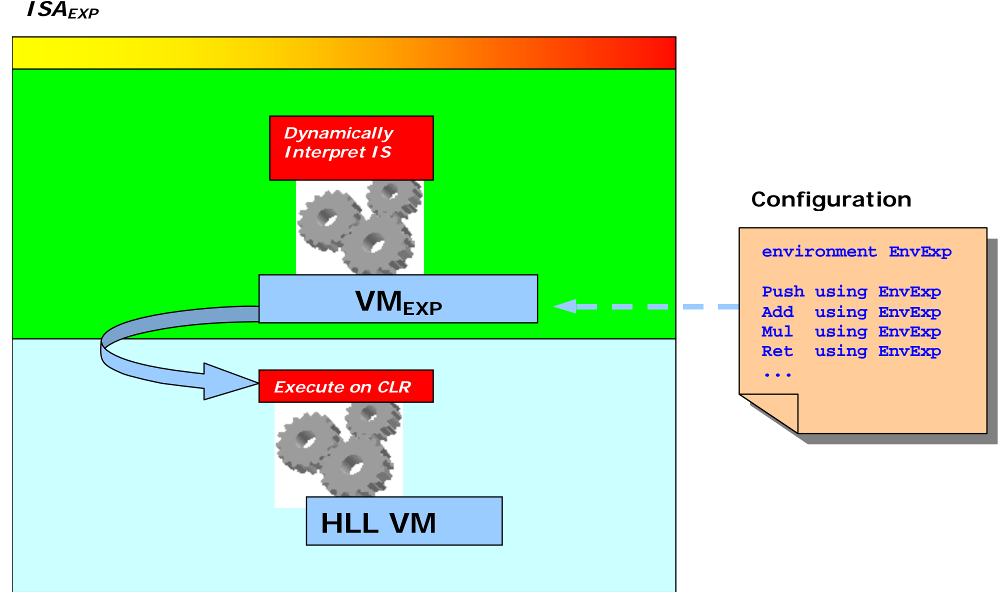


Figure 42. Defining an environment and instruction set for a DSL VM.

Figure 43 depicts the same environment for VM_{LQ} as that used for VM_{EXP}, by placing the two VMs at the same level, but the instructions sets differ slightly. In fact, there is an *overlap* as the instructions `Store`, `Load`, `Br`, `Call` and `Ret` are all used as they are defined in their respective classes, i.e. no modifications are required for them and they can be used *as is* in the context of VM_{LQ}. Four new instructions are added, namely `QuiltA`, `QuiltB`, `Turn` and `Sew`, and they continue to rely on EXP's existing environment class, `EnvExp`. The darker shade of the VM_{EXP} simply indicates that VM_{EXP} is the not the current VM in question. The overlap signifies the subset of EXP's instructions that are to be used by VM_{LQ}.

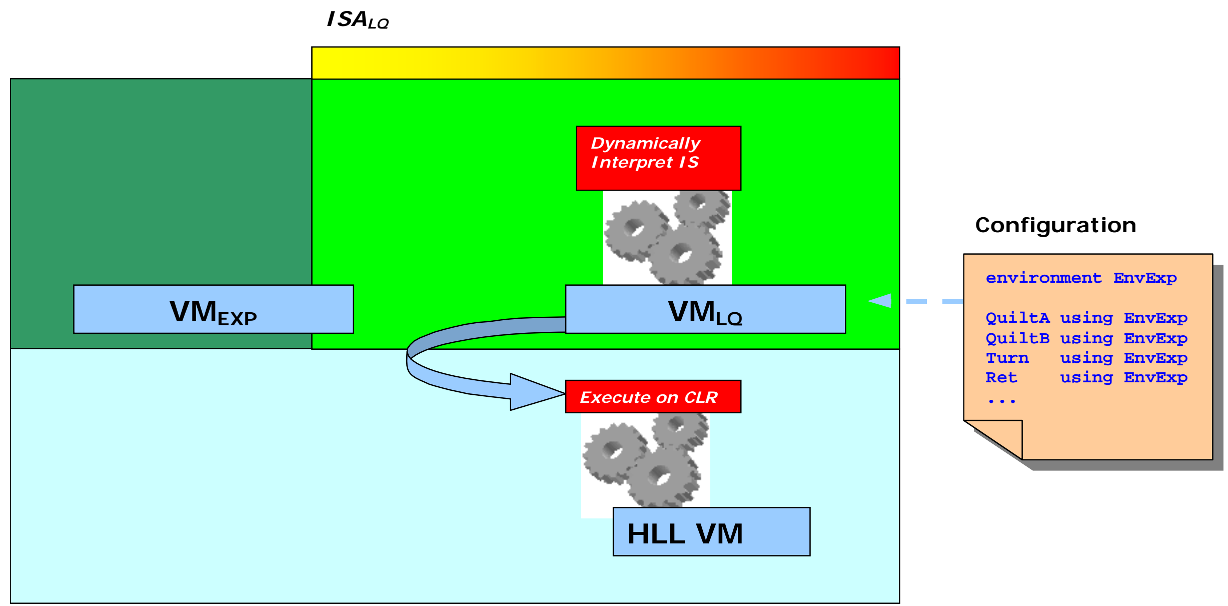


Figure 43. ISA_{LQ} is a subset of ISA_{EXP}, with new instructions.

Figure 44 depicts the environment of the VM_{RT} as a subclass of the one used for VM_{EXP}, by placing VM_{RT} at a higher level to indicate this intent. The instruction sets again differ slightly, as there is still an overlap with the real-valued expression instructions `Sub`, `Add`, `Mul`, `Div`, and the `Push` instruction, that work with the `EnvExp`. However, there is additional RT specific instructions, indicated by the portion of the ISA above VM_{RT} in Figure 44. This set of instructions, along with the real-valued expression instructions together form ISA_{RT}. Those RT specific instructions, such as `COP`, `LookAt`, `VUP`, `Specular` and so forth (as listed in the RT configuration file shown in Figure 41), work with RT's environment class, `EnvRT`. The darker shade of VM_{EXP} again gives a clue that a certain subset of instructions that was used with VM_{EXP} is no longer needed for VM_{RT}, except for those instructions indicated by the overlapping shade on the right hand side. There is no notion of a VM_{EXP} being executed by the underlying HLL VM, hence the linking arrow between only VM_{RT} and the HLL VM.

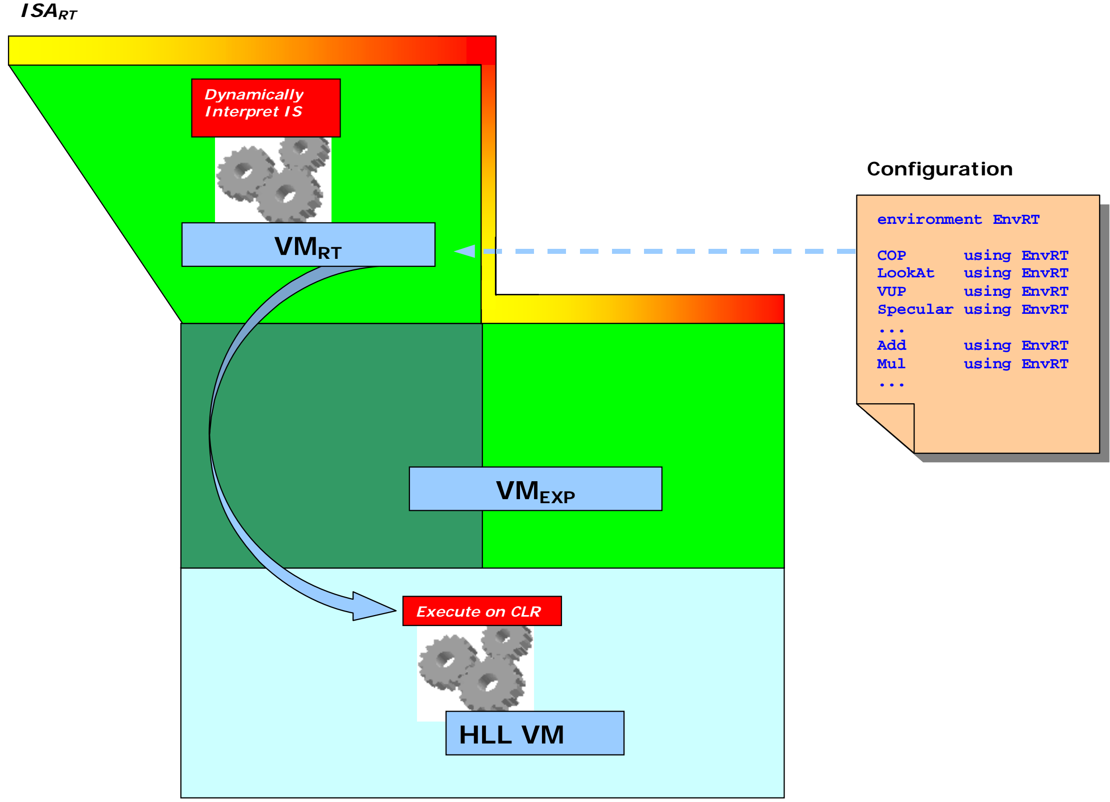


Figure 44. ISA_{RT} is a subset of ISA_{EXP}, with new instructions as before.

6.5 Building a DSL

Once a defined environment and instruction set are in place, a front-end for the DSL needs to be developed. Essentially this is the task of writing a simple compiler for the DSL. This entails designing syntax for the language using compiler tools. Translation rules are also needed to map syntactically correct statements onto a sequence of instructions that can be executed by

the VM. It is beyond the scope of this text to discuss how each of these tasks is performed, but some of the translations from DSL-program source code segments to DSL-compiled code are shown (refer to Figures 47 and 48, and 53 and 54) to illustrate the simplicity of some of the translation for small languages. The example DSLs in this section were built using the compiler tools LG and PG. PG allows the translation rules to be defined as part of the grammar. Both tools have a familiar syntax to most commonly used industry compiler tools. The complexity of the required DSL will influence the complexity of the compiler needed to parse the DSL-program source and to generate the target DSL-compiled program code. Keeping the language simple and intuitive keeps the translations simple.

6.5.1 The EXP DSL

The EXP DSL is an intuitive, real-valued expression evaluation language that includes typical real-valued constructs for addition, subtraction, multiplication and division. The language is functional, and such a language suggests that a runtime stack should be used as the underlying data structure when evaluating an expression. The runtime stack will be encapsulated in the environment. Also, local variables can be assigned and scoping rules are applied in a strict, functional form. Having real-valued addition as part of the EXP language almost persuades inclusion of a construct like summation to be incorporated into the EXP DSL as well. Summation simply makes use of repeated addition of an expression as shown in equation (1). Summation is part of the EXP language syntax, and thus in this example DSL there is indeed a construct for summation.

6.5.1.1 The EXP Domain

In the case of the RT DSL, the RT domain was a fairly fleshed-out namespace (recall the C# *namespace* construct, Sections 1.2.1, 1.3.3 and 3.1) with classes that encapsulate scenes, rays, and object primitives and so on. However, for a DSL as simple as EXP, no namespace needs to exist as all the semantics for EXP are provided by the real-valued operators of the C# language, namely `+`, `-`, `*` and `/`, and these operators essentially define the EXP domain.

6.5.1.2 The EXP Language

The EXP language is a simple, intuitive, functional language, with much resemblance to the functional language ML [Mil84, Ull94]. Expressions can be written verbatim, forming the DSL-program to be compiled. Because of its usefulness, EXP also borrows ML's *let-in-end*

construct for declaring variables, and for assigning values to variables before their use in evaluating an expression. The language is simple and nowhere does the user need to make use of cumbersome *for*-loops for summation. Neither does the user need to worry about variable types, as the language always treats variables as a C# `double` type. The lexer's regular expression definition is illustrated next, with Pascal-like comments defined.

```
%lexer
%namespace ExpcLexer

%token                      DOUBLE {public double val;}

"let"                       %LET
"in"                        %IN
"end"                       %END
"sum"                       %SUM
".."                        %DOTDOT
"="                         %EQU
"+"                         %PLUS
"-"                         %MINUS
"*"                         %TIMES
"/"                         %DIV
"("                         %LBRACKET
")"                         %RBRACKET
[_a-zA-Z][_a-zA-Z0-9]*      %ID
[-+]?[0-9]+(\.[0-9]+)?      %DOUBLE {val = Double.Parse (yytext);}
[ \n\r\f\t]                 ;
"(*"                        {yybegin ("COMMENT");}
<COMMENT>"*)"               {yybegin ("YYINITIAL");}
<COMMENT>.                  ;
<COMMENT>\n                 ;
```

Figure 45. Lexer definition of EXP.

The grammar definition is shown, again with integer and doubles combined into a single `DOUBLE` token that matches any signed or unsigned integer or double.

```
FExp     : LetExp

LetExp   : LET var_list IN exp END

var_list :
var_list : var_list var

var      : ID EQU DOUBLE

exp      : exp PLUS term
exp      : exp MINUS term
exp      : term

term     : term TIMES fact
term     : term DIV fact
term     : fact

fact     : LetExp
fact     : ID
fact     : DOUBLE
fact     : LBRACKET exp RBRACKET
fact     : SUM LBRACKET ID RBRACKET DOUBLE DOTDOT ID LBRACKET exp RBRACKET
```

Figure 46. Parser definition of EXP.

The translation rules of the small, functional expression language, EXP, can be intuitively understood by the following illustrative example. For a more detailed description on functional language compilers see [App98]. The program in Figure 47 evaluates the expression,

$$9+\sum_{i=1}^{n} 2i, \ where \ n=5 \qquad \text{...(1)}$$

in a functional manner. Section 5.8 gives the token and grammar definitions for each of the five instruction types. The translated compiled code for this program is shown in Figure 48. It is a concrete example that demonstrates the use of temporaries (as storage for variables `n` and `i`) and also branching instructions for the actual implementation of the summation construct.

```
let
   n = 5
in
   9 + sum (i) 1..n (2 * i)
end
```

Figure 47. Programmatic representation of summation expression (1).

The generated instructions perform operations on a runtime stack, which is part of `EnvExp` discussed earlier. The result of the expression will be the only remaining value on the stack.

```
     Push 5
     Store $1
     Push 9
     Push 1
     Store $2
     Push 0
@100 Load $2
     Load $1
     Sub
     Brgz label @200
     Push 2
     Load $2
     Mul
     Add
     Load $2
     Push 1
     Add
     Store $2
     Br label @100
@200 Nop
     Add
```

Figure 48. Compiled program of the summation expression (1).

6.5.2 The Little Quilt DSL

The Little Quilt language (LQ) [Set97] is a functional language, where the primitive types are that of quilts. Two primitive quilts are defined, A and B, and are illustrated later in the following section. All quilts are constructed from only these two primitive quilts. The idea is that there are two primitive operations that can be applied to quilts to yield a new quilt. Either a quilt can be turned 90 degrees clockwise (using the `turn` operator) or it can be sewn horizontally together with another quilt (using the `sew` operator). Using this small set of primitives, it is possible to derive more complex functions and thus the DSL can generate patterns of various complexities. Since the language is functional, it suffices to make use of the EXP language environment, i.e. the runtime stack to express its semantics.

6.5.2.1 The LQ Domain

The LQ namespace consists of a single class `Quilt` (or more precisely `LittleQuilt.Quilt` in its fully qualified form as `LittleQuilt` is the namespace to which it belongs). Instances of `Quilt` represent an immutable quilt of arbitrary dimension. Internally, a quilt is simply represented as two matrices called `symbols`, of enumerator type `Symbol`, and `quilt`, of enumerator type `Orientation` shown in Figure 49. The `symbols` matrix simply represents the primitive pieces that make up an instance of `Quilt`. As suggested in Figure 49, a single elementary quilt piece is either one of `Symbol.A` or `Symbol.B` of the `Symbol` enumerator type. When a `Quilt` instance is rendered onto a canvas, the appropriate glyph for each of the two symbols is used. They are respectively shown alongside the namespace as glyphs A and B. The Orientation matrix, `quilt`, represents the orientation of each single elementary quilt piece, the orientation being indicated by an element `Orientation.N`, `Orientation.S`, `Orientation.E`, or `Orientation.W` of the `Orientation` enumerator type. The resultant data structure returned by the `GetResult()` method of the LQ environment is an instance of `Quilt`. It can then be rendered onto a canvas by iterating though the `quilt` and `orientation` matrices, and rendering the appropriate glyph, A or B, at the appropriate orientation.

The domain is somewhat simplistic. It is only necessary to think about the representation of a quilt, and the result of either the `turn` or the `sew` operation on a quilt to derive a new quilt. The flexibility of manipulating quilts will become apparent later when an instruction set is designed exposing the namespace, and the compiler generates the proper sequence of instructions to allow a VM instance of the VM Framework to perform the manipulations on the quilts.

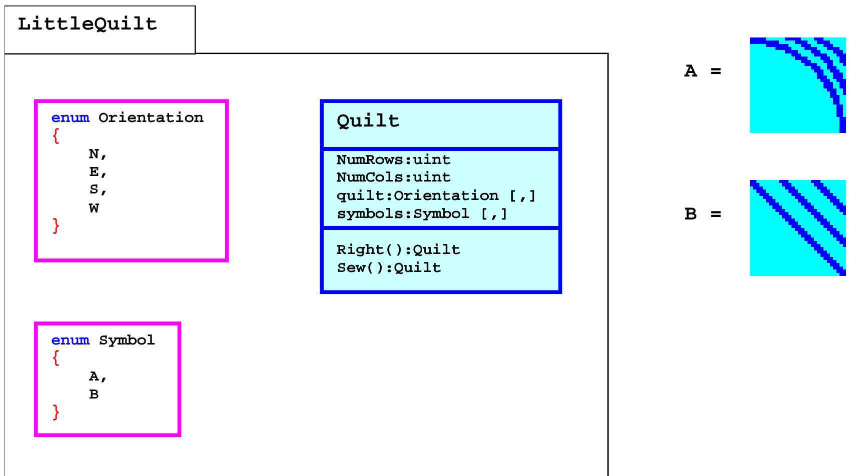


Figure 49. The namespace of the LQ.

6.5.2.2 The LQ Language

Since the LQ language is centered around operations on quilts to build more complex quilts, the language has a functional nature, and as such can use the same runtime stack as that used in EXP. However, this time it is `Quilt` instances that are pushed on and popped off the stack, instead of `double` values as was the case in EXP. Thus the environment `ExpEnv` can be used for LQ. The instructions will differ, however, in that they will perform operations on Quilts. The configuration file for LQ is shown in Figure 50.

```
environment EnvExp

QuiltA using EnvExp
QuiltB using EnvExp
Load   using EnvExp
Store  using EnvExp
Turn   using EnvExp
Sew    using EnvExp
Call   using EnvExp
Ret    using EnvExp
Br     using EnvExp
```

Figure 50. The configuration file for LQ.

The lexer definition for LQ is shown in Figure 51. Two keywords `turn` and `sew` expose the operations defined in the `Quilt` class, namely the two methods `Right()` and `Sew()` respectively.

```
%lexer
%namespace LqcLexer

"let"                        %LET
"in"                         %IN
"end"                        %END
"turn"                       %TURN
"sew"                        %SEW
"fun"                        %FUN
"val"                        %VAL
"a"                          %A
"b"                          %B
"="                          %EQU
","                          %COMMA
"("                          %LBRACKET
")"                          %RBRACKET
[_a-zA-Z][_a-zA-Z0-9]*       %ID
[ \n\r\f\t]                  ;
"(*"                         {yybegin ("COMMENT");}
<COMMENT>"*)"                {yybegin ("YYINITIAL");}
<COMMENT>.                   ;
<COMMENT>\n                  ;
```

Figure 51. The lexer definition of LQ.

LQ also includes a *let-in-end* construct, shown in Figure 52, to allow the definition of local variables (in this case, variables of type `Quilt`), as well as functions, parameterized by local variables.

```
LQExp     : exp

exp       : A
exp       : B
exp       : TURN LBRACKET exp RBRACKET
exp       : SEW LBRACKET exp COMMA exp RBRACKET
exp       : LET decls IN exp END
exp       : ID LBRACKET exp RBRACKET
exp       : ID LBRACKET exp COMMA exp RBRACKET
exp       : ID

decls     : decl
decls     : decl decls

decl      : FUN ID LBRACKET ID RBRACKET EQU exp
decl      : FUN ID LBRACKET ID COMMA ID RBRACKET EQU exp
decl      : VAL ID EQU exp
```

Figure 52. The grammar definition for LQ.

Figures 53 and 54 below show a simple LQ program and the corresponding compiled program. The translation of the functions `turn()` and `sew()`, and the user-defined functions `unturn()` and `pile()` can be seen. It illustrates the use of branching instructions to implement function calls, with all necessary quilt parameters loaded into temporaries before a function call. Note in Figure 54 the very first instruction is a branch instruction, `Br label @199`, that skips the first user-defined function, `unturn()`. Following the first user-defined function is yet another branch instruction, `Br label @299`, forcing program flow to skip the second user-defined function, `pile()`, as well. Those functions are only to be executed via the `Call` instructions. Each of

these two functions has a `Ret` instruction that will return program flow to the instruction just after the corresponding `Call` instruction that invoked the function.

```
let
    fun unturn(x)=turn(turn(turn(x)))
    fun pile(x,y)=unturn(sew(turn(y),turn (x)))
in
    pile(unturn(a),turn(a))
end
```

Figure 53. A function defined in LQ.

The `Load $n` instruction pushes temporary number *n* onto the stack. This is in contrast to the `Store $n` instruction that pops the top `Quilt` instance off the stack and saves it in temporary number *n*. The temporaries used in this example are used to pass local parameters (quilts) to functions. The instructions `QuiltA` and `QuiltB` simply push one of either of the elementary quilts A or B onto the stack. The instructions `Turn` and `Sew` pop the top quilt off the stack, perform a turn or sew operation, and push the resultant quilt back on.

```
     Br label @199

    (* unturn($1) *)
@100
     Load $1
     Turn
     Turn
     Turn
     Ret
@199
     Br label @299

     (* pile($1,$2) *)
@200
     Load $2
     Turn
     Load $1
     Turn
     Sew
     Store $1
     Call label @100
     Ret

     (* program start *)
@299
     QuiltA
     Store $1
     Call label @100
     QuiltA
     Turn
     Store $2
     Store $1
     Call label @200
```

Figure 54. Translation of a function defined in LQ.

Example software for LQ, developed as part of this study, is illustrated in Figure 55. The software makes use of a canvas to render the desired quilt onto a canvas, and features a text editor alongside to define the pattern. The menu option *Compile* will perform a compile of the

displayed LQ program and then execute the compiled program in one step, without any further user intervention. As a result, a quilt will be rendered on the canvas alongside.

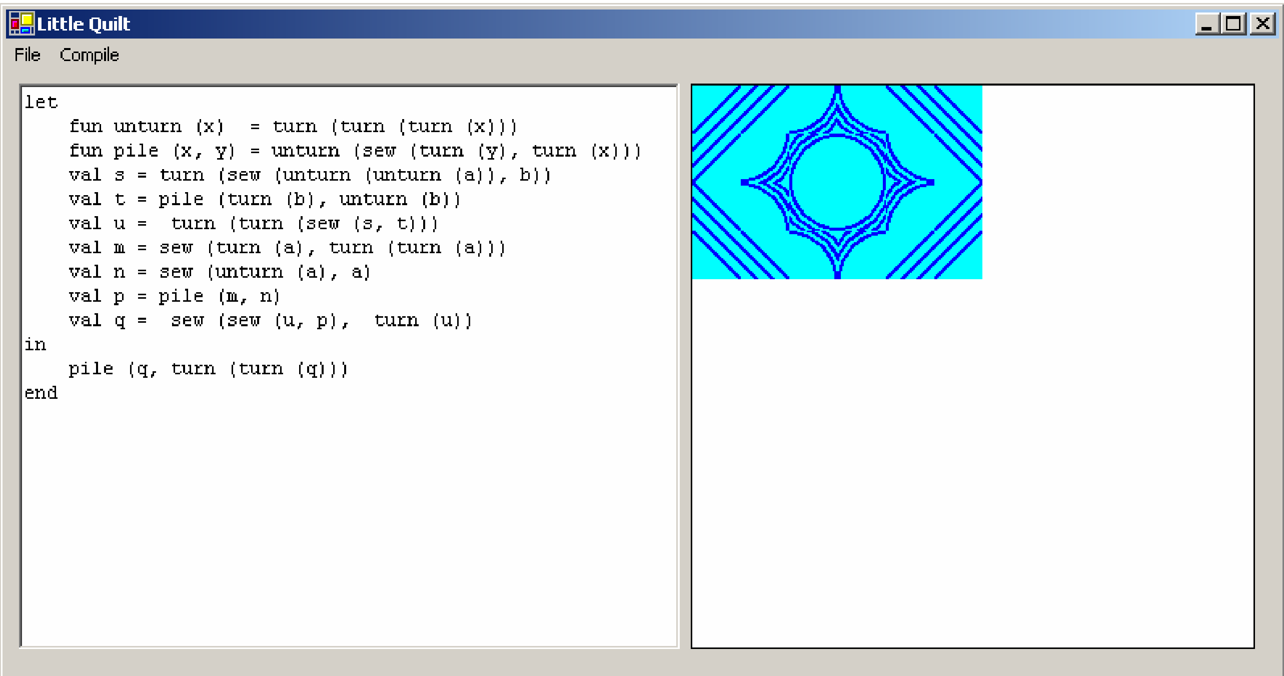


Figure 55. LQ using the VM Framework in the context of a GUI.

6.6 Scaling up the RT DSL for the VM Framework Approach

Again the RT DSL is scaled up by adding an *ambient* construct. As in the hard-coded VM approach, modifications are required in both the VM and the compiler. The compiler's lexer will need to incorporate a new keyword `ambient` and the grammar for the compiler's parser needs to be altered to match on this new keyword and produce a node on the AST representing the *ambient* construct.

This additional AST node of the *ambient* construct will be handled in the same way as it was handled in the modification to the DSL-compiler described in Chapter 4, Section 4.5, as it amounts to generating the same sequence of instructions that alter the ambient lighting of the scene.

A new instruction needs to be added to the instruction set, as was the case in Section 4.5, but this time no risky modification is needed for the VM. Nor does the lexer and parser for the VM need to be modified in any way to support this new instruction. Adding this new instruction simply entails creating an instruction class, `Ambient`, which subtypes one of the five abstract instruction classes. The `Ambient` instruction has no operands, and thus subtypes `Inst_OpCode` and is shown in Figure 58. Once this class is compiled to a DLL, it can be added to the

instruction set by adding it to the list of instructions specified in the configuration file. This is shown in Figure 59.

6.6.1 Extending EXP to suite the RT language

The environment `EnvExp` features a runtime stack, of C# `object` instances. For its use in the EXP environment, the `object` instances will be of type `double`. A quadric is defined by the equation *q(x,y,z)* in Figure 4. By supplying the appropriate set of coefficients, namely a_{11}, a_{22}, a_{33}, a_{23}, a_{13}, a_{12}, b_1, b_2, b_3 and c, any quadric can be represented. However, consider for example the definition of a sphere, where the coefficients; a_{23}, a_{13}, a_{12}, b_1, b_2, b_3 and c, are all 0. Then the definition for a sphere simply amounts to

$$q(x, y, z) = a_{11}x^2 + a_{22}y^2 + a_{33}z^2 \qquad ...(2)$$

Typically, a radius parameterizes a sphere, so the coefficients in the above case would be defined as

$$a_{11} = a_{22} = a_{33} = \frac{1}{r^2} \qquad ...(3)$$

Thus it is certainly helpful, in setting up a quadric, to have some arithmetic operations in assigning values to the necessary coefficients. In the above case, equation (3) prompts for a multiplication, and a division of real values. Figure 56 and 57 shows the environment `EnvExp`, and the environment `EnvRT` as a subclass of `EnvExp`, which includes the runtime stack.

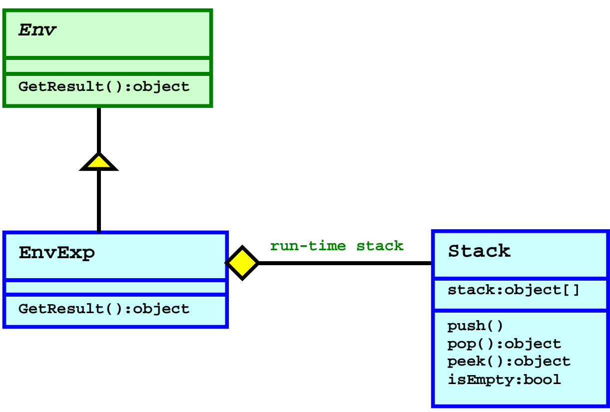


Figure 56. The `EnvExp` environment.

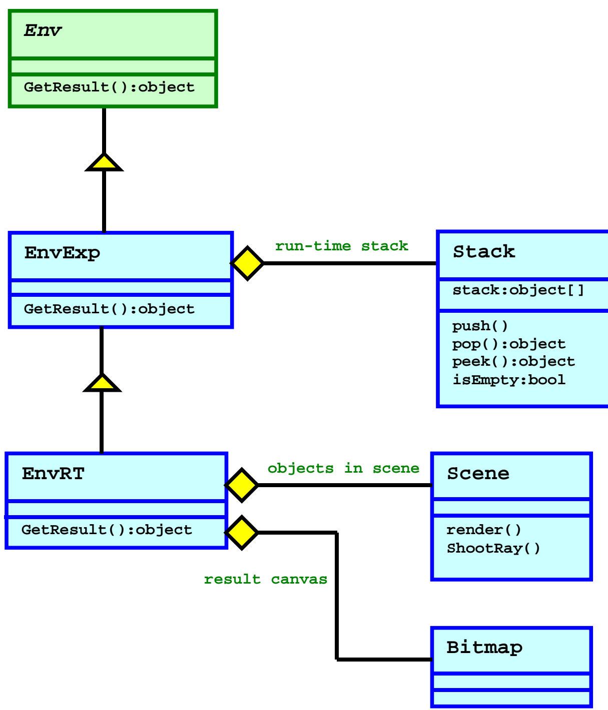


Figure 57. The `EnvRT` environment as an extension to `EnvExp`.

6.6.2 Adding a Construct

Recall the discussion in Chapter 3, Section 3.7, which illustrated the requirements for adding a construct in the hard-coded VM approach. For comparative purposes, the implications of adding the same *`ambient`* construct to the RT DSL in the context of the VM Framework will be discussed. The changes made to the RT DSL-compiler are identical, as the same sequence of instructions is to be produced for the compiled program. However it will be shown in the following section that the changes made on the VM side are far simpler. Because of this simplicity, there is a low risk of error when adding a new instruction to the instruction set. It is simply a matter of coding a new instruction as a class, and registering it as an instruction via the configuration file.

6.6.3 Components Requiring Alteration

The compiler is modified as outlined in Section 4.5.2, the lexer and grammar are modified to accept the new keyword `ambient`, and translation of the *`ambient`* construct involves the generation of the same sequence of instructions. On the VM side, no alteration to the VM's lexer or parser is necessary. An appropriate instruction class simply subtypes the abstract instruction class `Inst`, and the new instruction is registered with the instruction set via the configuration file. Parsing this new instruction from the DSL-compiled program is performed

automatically according to the name of the instruction given in the configuration file, and the appropriate operands being inferred from the type of instruction that the new class subtypes. In the example `Ambient` instruction, the instruction class `Ambient` extends class `Inst_OpCode`, and hence no opcode will be expected by the parser when the DSL-compiled program is read by the VM's loader. The `Ambient` instruction itself will be parsed as such, since it has been included in the instruction set defined by the configuration file. The `Ambient` instruction is shown below in Figure 58. Note its similarity and differences in expressing the semantics to the hard-coded VM discussed in Chapter 4.

```
public class Ambient : Inst_OpCode
{
        public Ambient (EnvRT env)
        {
                this.env = env;
        }

        public override void Execute ()
        {
                double amz = (double) ((EnvRT) env).stack.pop ();
                double amy = (double) ((EnvRT) env).stack.pop ();
                double amx = (double) ((EnvRT) env).stack.pop ();
                ((EnvRT) env).scene.alterAmbient (amx, amy, amz);
        }
}
```

Figure 58. The `Ambient` instruction extends `Inst_OpCode` (no operands.)

The last change required is to the configuration file, adding the instruction `Ambient` using the environment class `EnvRT`.

```
(* Create an instance of the ray-tracer environment. *)
environment EnvRT

...
COP       using EnvRT
LookAt    using EnvRT
VUP       using EnvRT
Ambient   using EnvRT
Geometry  using EnvRT
Specular  using EnvRT
Diffuse   using EnvRT
...
```

Figure 59. Adding the `Ambient` instruction to the instruction set.

6.7 Modularity

The VM Framework is shown to be highly modular, in that the work required to scale up or alter a language, is encapsulated in either an environment or a set of instructions. The environments and instructions needn't warrant any amendment at all. They simply comply to the rules of the VM Framework, i.e. certain instructions need to be subtypes of the appropriate

instruction classes of the VM Framework and the environments need to be subtypes of the environment class of the VM Framework and override its abstract method `GetResult()`. Only the translation rules of the compiler will need attention.

6.8 Summary

This chapter has provided a detailed description of the process involved for building or customizing a DSL for a particular use. This entails thinking about the instructions that are necessary to facilitate the semantics of the DSL constructs, though as described, the VM Framework allows easy adaptation for instructions that require alteration at a later stage and also allow for increasing the size of the instruction set itself. Once a set of instructions is in place, a simple translator is required to map the DSL syntax, into sequences of instructions that execute on the VM instance. The instruction set is configured for the VM via a configuration file. This process was demonstrated using two example languages – EXP, a simple real-valued expression language, and Little Quilt or LQ, a small language for manipulating and compounding patterns. Both use a runtime stack to perform operations on their respective primitives (numbers or quilts respectively) and both illustrate the use of the VM in its intended context; that of streamlining the development of a DSL for a particular end-user, and leaving the DSL open for future enhancements and even allowing the DSL to grow into another useful language, if they are based on the same environments and similar instructions.

6.8.1 What the VM Framework is Not Intended to Provide

This section reaffirms what the VM Framework is not; or rather it serves to warn the user of the VM Framework of how not to approach the use of it. The following list itemizes preconceptions of how a developer might potentially perceive the VM Framework to be:

1. It should not be treated as a precompiled, standalone VM of some predefined instruction set.

2. It should not be treated as a system that provides an automated process for the creation of a language or a standalone VM.

3. The VM Framework is intentionally not a code generation tool and does not provide any functions to process archetypes that define code translation rules. However, there is no reason why a translation DSL cannot be created using the Framework, and as such, the

instructions provided by the user could potentially be used as functions to emit sequences of translated code to a file.

4. Regarding the VM Framework as a mechanism for building DSLs, it does not facilitate solely the creation of functional languages as prescribed, though functional languages tend to be easier to get up and running, and serve as intuitive examples.

5. Once a VM has been created that sufficiently implements the semantics of a DSL, and prototyping of the DSL is at its completion, then it is not necessary to make use of the configured VM as the target runtime system if a higher degree of performance is required. It may be useful to use a second software engineering step to hard-code the instruction set, and the environment, into a more compact module that executes the compiled program at a more native level.

6.8.2 What the VM Framework Does Provide

It is necessary at this point to highlight what the VM Framework provides to the developer intending to extend it or put another way, to indicate how it differs from a traditional VM that is not extendable and has a hard-coded instruction set. The following is a list, itemizing the roles of the VM Framework for a developer:

1. It provides a means to take an existing domain or namespace, and expose that namespace in such a way that it appears to an end-user as a DSL, simplifying the task of reusing operations and data structures within the namespace.

2. It provides the opportunity to design readable, and intuitive DSL syntax, and implement DSL translators to invoke the reusable operations and data structures within the namespace.

3. The VM Framework includes a top-level class, `EVM`, which is essentially representative of a VM with an empty instruction set.

4. It provides a single interface, in terms of an abstract environment class, for any data structure or environment that is suitable for facilitating the semantics of the DSL, and that is updated during DSL-program execution.

5. It provides five interfaces, in terms of abstract instruction classes, for any operations that are suitable for updating the environment during program execution, leaving the

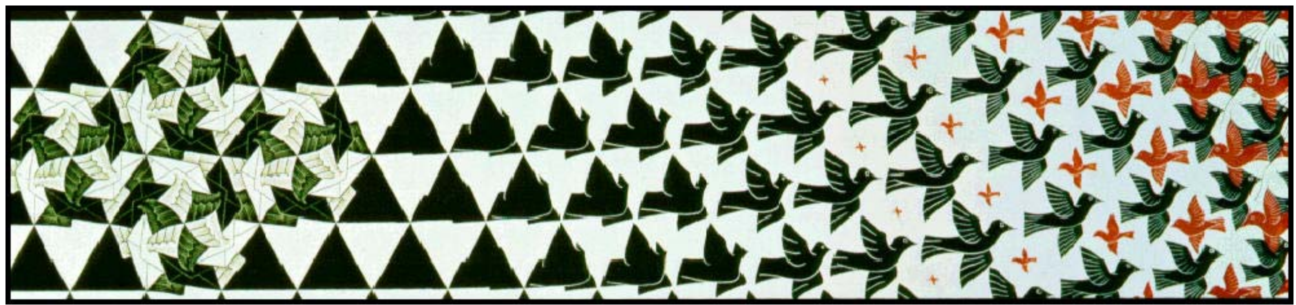
environment in a consistent state, i.e. the semantics can be specified in terms of pre- and post-conditions on the operations.

6. It provides a mechanism to configure an instance of a VM, using a configuration file, to register instructions and environments for a VM instance of a particular DSL. This implies the existence of a pool of instructions and environments, that through careful selection of instructions and environments from this pool, a developer can easily create an efficient and effective VM for a DSL.

7. It provides a parser for loading program source files of a DSL-compiled program, whose instructions are defined within the configuration file, and it also provides the parser to parse the configuration file.

8. The VM Framework, being of an object-oriented nature, allows environments to subtype existing environments that already provide partial functionality that is likely to be suitable for a particular DSL being developed.

7 Comparative Results



This chapter provides a summary of the performance of the execution of a DSL-compiled program. The performance test is done for each of the three DSL development techniques described in the dissertation, namely the interpreter, the hard-coded VM, and a VM instance built using the VM Framework. To measure the performance of DSL-compiled program execution, a simple time measurement is taken. There are two time intervals that are measured between three different types of information. The machine used for the performance test features a Pentium III, 450 MHz processor, with 128 Mbytes of RAM, running the Windows 2000 OS. The time measurements are consistent over numerous runs. The three types of information are named, and described next.

SRC:
This refers to an example DSL-program containing the source code of a program written in the language of the DSL.

IS:
This is the instruction sequence that is the output of the compiler's effort to translate the source DSL-program into the instructions ready to be executed by the VM.

SEM:
This refers to a complete piece of information or a populated data structure or some form of observable output, which is the result of executing the instruction sequence. Thus SEM is will be an instance of a data structure holding the result of a computation.

The two time intervals measured are the time taken to generate an instruction sequence from a given source file (SRC → IS) and the time taken to compute the final results of a DSL-compiled program execution (IS → SEM).

SRC → IS:
This is the time taken to parse a program written in the language of the DSL, to build an AST, and further translate the nodes of the AST into executable instructions. The example DSL used for the experiment is the RT DSL, and the example program used is shown in Figure 9, named `test1.scene` and is also found on the accompanying CD.

IS → SEM:
This refers to the time taken to parse and load the instructions of a DSL-compiled program, and further generate a data structure, which is the result of the execution of the compiled program. For the RT DSL, the result returned by the `GetResult()` method on the environment will be an instance of a `System.Drawing.Bitmap`. Execution of the instructions incrementally builds an RT data structure, composed of quadrics and will only be rasterized once a call to the `GetResult()` method is made. Thus the rasterization process of the image is not part of the time measurement, only the creation of the RT data structure. The image is saved as a file after the time measurement is taken.

Clearly, in the interpretation approach, there is no intermediately generated instruction sequence. However, there is nonetheless a flow of information between SRC and SEM, and thus only the time between SRC and SEM will be noted.

Comparative performance results were done for the three different DSL implementations described in the dissertation: an interpreter, a hard-coded VM and the VM Framework. Two time intervals are compared for each implementation: compiling DSL-program source code to an instruction sequence (SRC→IS); and executing the instructions to observe the semantics (IS→SEM). The total time (SRC→SEM) is also calculated. The resolution of the timing mechanism is 100 nanoseconds, as this is the default resolution offered by the Windows 2000 operation system. Under the .NET environment, each 100-nanosecond time interval is called a *tick*. The interpreter does not generate any intermediate instruction sequence, as the semantics are interpreted directly from the AST, and thus only the total time (SRC→SEM) is relevant for the interpreter. The hard-coded VM has a predefined set of instructions and the VM Framework is similarly configured with the same set of instructions. For the purpose of the experiment, the same DSL was used for each implementation.

From the performance results in Figure 60, it can be seen that using some of the reflection properties of .NET does not necessarily impede the compiled program's execution speed, and in this case it is actually shown to perform better than its hard-coded counterpart. Naturally,

the interpreter is quickest to deliver observable results. However, it suffers from a lack of scalability, and modifying the interpreter to fit above an existing namespace is a cumbersome task. The hard-coded VM suffers less from scalability problems, as it is easier to add new instructions as part of the VM core, and amend the grammar to accept a wider range of instructions. But instructions are not reusable by another DSL, as they are executed as functions with parameters from within the VM core. The VM Framework treats environments and instructions that access these environments as separate external libraries or DLLs and they do not form part of the VM Framework's namespace. Rather, these DLLs are configured together as a set of building blocks to yield a customized VM instance for a particular DSL. Furthermore, the VM Framework easily accommodates the scaling up of the DSL with new constructs as the need arises.

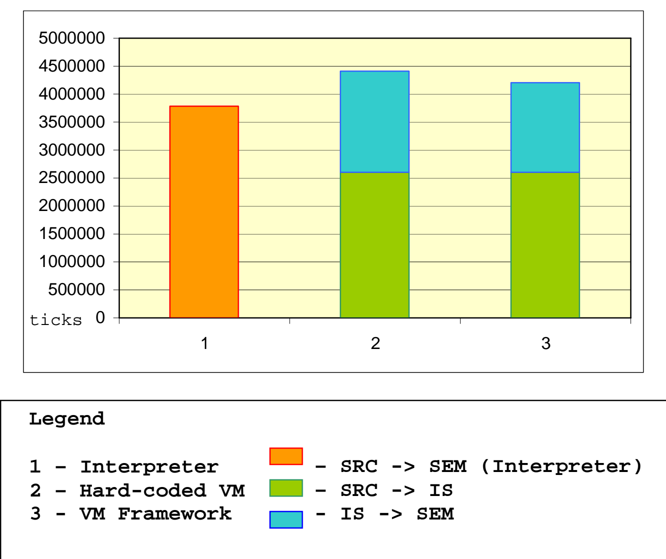


Figure 60. Comparative performance results of the DSL implementations.

The illustrating software in this dissertation was developed in C# which naturally prompts for further investigation of DSL VMs in languages other than those that execute on top of the .NET CLR. That in itself lends to the scalability nature of the VM Framework, as different instructions can be implemented in different languages, provided those languages have compilers that generate code that runs on the .NET CLR.

A summary of the different criteria used to evaluate the 3 types of semantic techniques is shown below. For simplicity, 3 degrees of indication are depicted and are briefly described in the legend.

Legend:

☀ : Good
☁ : Average
🌧 : Poor

	Scalability	**Performance**	**Modularity**
Interpreter	🌧 Difficult and cumbersome to modify a language and alter semantics.	☀ Interpretation of AST done in same program, and there's no IS.	🌧 Syntax and semantics issues are fairly dependant on one another.
Hard-coded VM	☁ Instructions are still required to be hard-coded as part of the core.	☁ Compiled program needs to be parsed before it can be executed.	☁ Minimal effort required adding new instructions to the VM core.
VM Framework	☀ Easily add new instructions and environments as the language grows.	☁ Comparable performance to the hard-coded VM, but slightly faster for compiled program execution.	☀ Instructions and environments form the link between the VM and the underlying namespace, and are easily modified as the namespace expands.

Table 4. Summary of criteria.

7.1 Summary

This section compares the relative performance of the three different types of ways to process a DSL-program. This section does not aim to promote any particular means over the other, as it is noted in the above table that all three cases have valid arguments in their favour. Rather, it was necessary to show how a VM developed for a particular DSL, using the VM Framework, compares relative to other means of DSL-program execution. The performance criteria used were scalability, performance and modularity. Scalability concerns measuring just how easy a language can be extended to include new constructs, and concepts in the underlying namespace. Performance concerns the actual time taken to execute a DSL-program using one of the aforementioned VMs and measurement was taken in 100-nanosecond ticks. Lastly,

modularity relates more to the risk of making a modification to the DSL software, i.e. it addresses questions such as how well compartmentalized is the software? To add a construct, does a change need to be made somewhere deep in the code or can a new class simply be extended or added?

8 Future Work

The VM Framework is used for building DSL VMs to execute on a HLL VM. In this study, the HLL VM used was the .NET CLR. This idea naturally prompts for further investigation of DSL VMs written in languages other than those that execute on an HLL VM such as the .NET CLR. This requires isolating the essential components that were used in this study, such as dynamic loading of classes, and examining how these features are used in other languages. Embedded systems, for example, would not typically be written in a .NET CLR language such as C#, but are more likely to be built using C/C++ or an assembler to generate native machine code with much smaller footprints as storage is often a limitation in such systems. However, DSLs may still be useful in that area if embedded system software designers can isolate separate aspects of the software.

Other future works might include focusing on the usability aspects of creating DSLs, such as GUIs or an interactive DSL development environment. Some obvious next steps are listed below and further elaborated upon, and may prove to be useful investigations. Section 8.1 proposes a GUI for the VM Framework, Section 8.2 proposes multiple environment and namespaces, and lastly Section 8.3 discusses architectural DSLs.

8.1 Graphical User Interface for the Virtual Machine Framework

Currently the VM Framework is purely a set of classes written in C#, the rationale behind which is to allow for rapid prototyping of new languages as specified by an end-user. End-users and experts alike expect a DSL to streamline their daily routines, and so one can also envision streamlining the actual task of developing the DSL itself in a RAD environment. This would entail the design of an IDE. The necessary tools for creating the DSL would all be

combined into one development environment. The system should feature a text editor for defining the syntax of a DSL, the operational semantics of the DSL VM and test programs to be executed by the defined DSL VM. The system's output should feature an open interface so as to allow a developer to *plug-in* a component, such as a separate console for textual outputs or even a canvas to display any graphs or images that are part of the test program's output. The system should also have a scratchpad environment for any VM currently under test. A system using some of the reflection features of .NET can generate a VM as a DLL. A similar application is outlined in Klint's ASD MetaEnvironment and allows the user to build syntax and semantics for a particular DSL [vdBra01].

8.2 Multiple Environments and Namespaces

It has been shown that the VM Framework is modular and treated as an object once the top-level `EVM` class is instantiated, and thus no assumption is made regarding its context of execution. Expanding on this idea, it may be interesting to examine how efficient the development process can be made by allowing many different VM instances to interact with one another, and hence allowing each VM instance to be responsible for the execution of a different DSL-program. The interaction of the constituting VMs can be controlled via a common interface as shown in Figure 61. The DSLs should all feature a syntax (either static such as having predefined types or keywords, or dynamic such as the ability to create user-defined types) that facilitates this communication in a meaningful way.

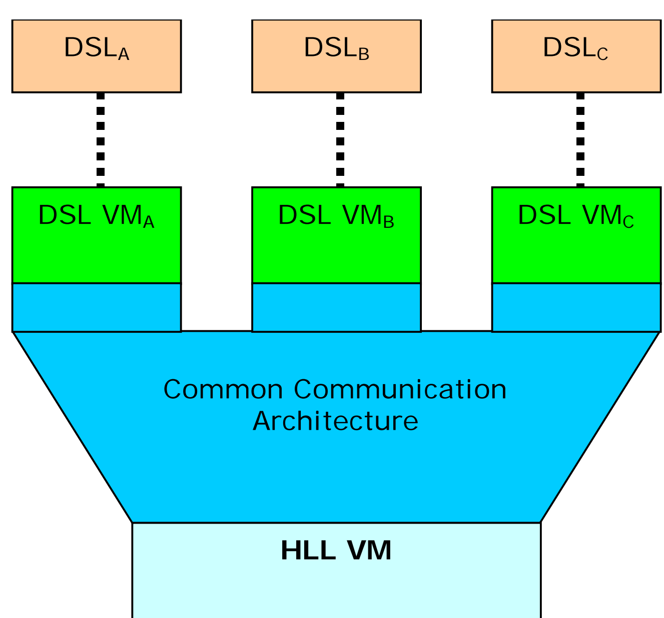


Figure 61. Communicating DSL infrastructure.

Each DSL will configure a separate concern of the system. This is similar to Aspect Oriented Programming (AOP) [Kic96]. AOP makes use of an Aspect Weaver to *fuse* each program

together at certain points called *joins*, to yield a single program, be it in Java, C# or C++. Using VMs there will be no generated code that needs compiling, but instead there will be code that can be reinserted back into a system at runtime. Again, this is useful in prototyping environments as mentioned above, but this time more than one language is considered. Thus it should be possible to have a prototyping environment with a language to control a set of test cases, another language to control concurrent processes or temporal logic, and perhaps a further language to reconfigure the GUI for a particular end-user requirement.

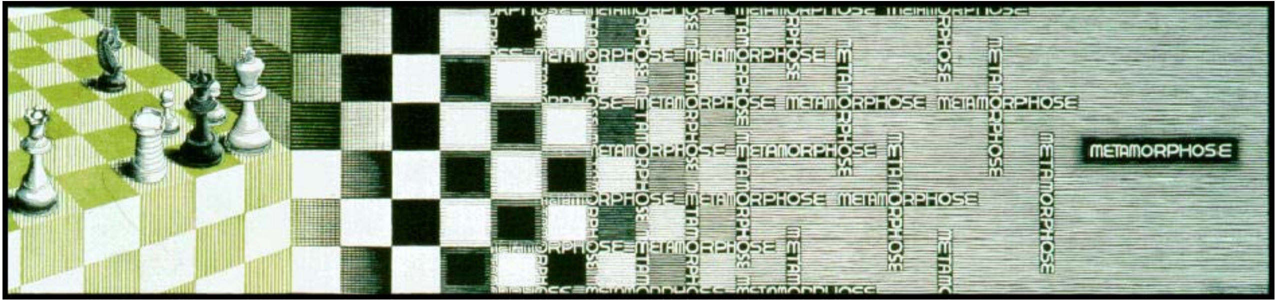
8.3 Architectural DSLs

Earlier in the dissertation it was mentioned that the DSL design and development should not be restricted to a particular end-user, but rather that seasoned software practitioners should also design their own DSL suited for their own needs. This is all very well if the practitioner can identify a layer of software that exhibits a set of rules or is somehow exposed as an API, and no further modification will be done to that software layer. The example used earlier was the OpenGL API, which is exposed as a graphical library accessible through most platforms supporting C/C++, Delphi, Visual Basic and the like. It is possible to treat OpenGL as a namespace in itself and provide some wrapper classes to aid in further building a complete configurable namespace. Other layers of software suitable for DSLs include ports and sockets, databases and GUIs. However, the problem arises when the developer also wishes to consider non-standardized software layers or their own *in-house* software layers, where modifications are done on a regular basis. In that case, the developer does not want to be restricted to constructs defined by a high-level DSL, and it would be far more practical for the developer to use a low-level language like Java or C++ for greater flexibility.

That said, suppose the developer would, nonetheless, still like to take advantage of a language, tailored to his or her needs, yet still have a high degree of flexibility like typical object-oriented languages. In this case it is worth considering the actual *architecture* of the software. Architecture can usually be narrowed down to a set of rules or even standards that are decided upon before further software design commences. Software design is then based on the decided architecture. It may be feasible to encapsulate the architecture, and expose some of the architecture as constructs, as well as the relevant object-oriented constructs. The language itself, though, will be far less concise than the likes of RT and others mentioned, but it serves a more industrial purpose. It is clear that this is a more daunting task for DSL designers. A good example of a software architecture is the architecture of Shlaer-Mellor based systems. These systems are designed around an *event kernel*, which facilitates instantiation of objects, polymorphism and message passing for active (state machine driven) objects and so on. The event kernel is written in the language of the domains, such as C, and using it in practice is cumbersome. Clearly, building a language to expose some of the event kernel

functionality in a syntactically elegant manner, while still leaving the relevant C language constructs, such as loops and arrays, will make for improved efficiency. Thus it will be beneficial to examine more closely these architectural aspects as extensions to DSL design.

Conclusion



The focus of this research has been to develop a framework in which to easily and cost effectively build DSLs on behalf of an expert who has domain knowledge, and who requires a textual, programmatic means to represent solutions to problems inherent within their domain of expertise. The intuitiveness of a language in this case is paramount, as it is not the job of the expert to apply, nor understand an industrialized approach to software development. This task consumes valuable time and it is more to an expert's advantage if she had a computer language that captures the fundamentals of her domain knowledge. The aim of this dissertation is to provide a VM Framework that allows rapid development of such DSLs, with emphasis on language scalability. Scalability often comes at a performance cost, but it is shown in this study that performance results are less of a concern, in that the VM Framework's performance is comparable to similar hard-coded VM variants. The VM Framework also relies on certain features offered by .NET platform to configure an instantiated VM at runtime, and .NET DLLs are used extensively to aid in flexibility and modularity. For the VM Framework to serve any use it must be extended by a set of concrete classes that form the instruction set and environments suitable for a particular DSL. Typically, a domain expert will work alongside a software practitioner to collaboratively tailor a DSL to the expert's or end-user's needs. Thus the syntax of constructs needs to be refined to be as intuitive as possible, while the practitioner needs to decide what type of instructions are necessary to facilitate the semantics of the constructs. This may involve a few iterations but a flexible VM Framework will aid in the development lifecycle of the DSL.

Appendix

The following listings are the complete LG and PG (lexer and parser) definitions for each of the languages mentioned in the dissertation; the Ray-Tracer language (RT), the real-valued expression language (EXP) and the Little Quilt Language (LQ), including an example DSL-program of each language. The listing titles are positioned above the listing for legibility. The complete source code of the VM Framework, and all example programs are on the accompanying CD.

Listing 1. RT DSL complete lexer definition.

```
%lexer
%token                     DOUBLE {public double val;}
"{"                        %LBRACE
"}"                        %RBRACE
"scene"                    %SCENE
"window"                   %WINDOW
"light"                    %LIGHT
"cop"                      %COP
"lookat"                   %LOOKAT
"viewup"                   %VIEWUP
"ambient"                  %AMBIENT
"geometry"                 %GEOM
"specular"                 %SPECULAR
"diffuse"                  %DIFFUSE
"reflective"               %REFLECTIVE
"refractive"               %REFRACTIVE
"sphere"                   %SPHERE
"plane"                    %PLANE
"translate"                %TRANSLATE
[-+]?[0-9]+(\.[0-9]+)?     %DOUBLE {val = Double.Parse (yytext);}
[ \n\r\f\t]                ;
"(*"                       {yybegin ("COMMENT");}
<COMMENT>"*)"              {yybegin ("YYINITIAL");}
<COMMENT>.                 ;
<COMMENT>\n                ;
```

Listing 2. RT DSL complete grammar definition.

```
%parser rtc.lexer

%{
using System.IO;
%}

%symbol scene_desc
{
      public virtual void translate (StreamWriter asmFile) {}
}

%symbol element_list
```

```
{
      public virtual void translate (StreamWriter asmFile) {}
}

%symbol element
{
      public virtual void translate (StreamWriter asmFile) {}
}

%symbol scene_prop
{
      public virtual void translate (StreamWriter asmFile) {}
}

%symbol prim_prop_list
{
      public virtual void translate (StreamWriter asmFile) {}
}

%symbol prim_prop
{
      public virtual void translate (StreamWriter asmFile) {}
}

%symbol prim
{
      public virtual void translate (StreamWriter asmFile) {}
}



%node scene_desc1 : scene_desc
{
      public element_list el;
      public scene_desc1 (element_list el) {this.el = el;}
      public override void translate (StreamWriter asmFile)
      {
            el.translate (asmFile);
      }
}

%node element_list1 : element_list
{
      public element_list el;
      public element e;
      public element_list1 (element_list el, element e) {this.el = el; this.e =
e;}
      public override void translate (StreamWriter asmFile)
      {
            if (el != null) el.translate (asmFile);
            if (e != null) e.translate (asmFile);
      }
}

%node element1 : element
{
      public scene_prop sp;
      public element1 (scene_prop sp) {this.sp = sp;}
      public override void translate (StreamWriter asmFile)
      {
            sp.translate (asmFile);
      }
}
```

```
%node element2 : element
{
      public prim p;
      public prim_prop_list ppl;
      public element2 (prim p, prim_prop_list ppl) {this.p = p; this.ppl = ppl;}
      public override void translate (StreamWriter asmFile)
      {
            p.translate (asmFile);
            ppl.translate (asmFile);
      }
}

%node scene_prop1 : scene_prop
{
      public double d1, d2;
      public scene_prop1 (double d1,double d2) {this.d1=d1;this.d2=d2;}
      public override void translate (StreamWriter asmFile)
      {
            asmFile.WriteLine ("     Push {0}", d1);
            asmFile.WriteLine ("     Push {0}", d2);
            asmFile.WriteLine ("     SetWindow");
            asmFile.WriteLine ("     Axes\n");
      }
}

%node scene_prop2 : scene_prop
{
      public double d1, d2, d3;
      public scene_prop2 (double d1,double d2,double d3)
{this.d1=d1;this.d2=d2;this.d3=d3;}
      public override void translate (StreamWriter asmFile)
      {
            asmFile.WriteLine ("     Push {0}", d1);
            asmFile.WriteLine ("     Push {0}", d2);
            asmFile.WriteLine ("     Push {0}", d3);
            asmFile.WriteLine ("     Light");
            asmFile.WriteLine ("     Axes\n");
      }
}

%node scene_prop3 : scene_prop
{
      public double d1, d2, d3;
      public scene_prop3 (double d1,double d2,double d3)
{this.d1=d1;this.d2=d2;this.d3=d3;}
      public override void translate (StreamWriter asmFile)
      {
            asmFile.WriteLine ("     Push {0}", d1);
            asmFile.WriteLine ("     Push {0}", d2);
            asmFile.WriteLine ("     Push {0}", d3);
            asmFile.WriteLine ("     COP");
            asmFile.WriteLine ("     Axes\n");
      }
}

%node scene_prop4 : scene_prop
{
      public double d1, d2, d3;
      public scene_prop4 (double d1,double d2,double d3)
{this.d1=d1;this.d2=d2;this.d3=d3;}
      public override void translate (StreamWriter asmFile)
      {
            asmFile.WriteLine ("     Push {0}", d1);
            asmFile.WriteLine ("     Push {0}", d2);
```

```
            asmFile.WriteLine ("     Push {0}", d3);
            asmFile.WriteLine ("     LookAt");
            asmFile.WriteLine ("     Axes\n");
      }
}

%node scene_prop5 : scene_prop
{
      public double d1, d2, d3;
      public scene_prop5 (double d1,double d2,double d3)
{this.d1=d1;this.d2=d2;this.d3=d3;}
      public override void translate (StreamWriter asmFile)
      {
            asmFile.WriteLine ("     Push {0}", d1);
            asmFile.WriteLine ("     Push {0}", d2);
            asmFile.WriteLine ("     Push {0}", d3);
            asmFile.WriteLine ("     VUP");
            asmFile.WriteLine ("     Axes\n");
      }
}

%node scene_prop6 : scene_prop
{
      public double d1, d2, d3;
      public scene_prop6 (double d1,double d2,double d3)
{this.d1=d1;this.d2=d2;this.d3=d3;}
      public override void translate (StreamWriter asmFile)
      {
            asmFile.WriteLine ("     Push {0}", d1);
            asmFile.WriteLine ("     Push {0}", d2);
            asmFile.WriteLine ("     Push {0}", d3);
            asmFile.WriteLine ("     Ambient");
            asmFile.WriteLine ("     Axes\n");
      }
}

%node prim_prop_list1 : prim_prop_list
{
      public prim_prop_list ppl;
      public prim_prop pp;
      public prim_prop_list1 (prim_prop_list ppl, prim_prop pp) {this.ppl = ppl;
this.pp = pp;}
      public override void translate (StreamWriter asmFile)
      {
            if (ppl != null) ppl.translate (asmFile);
            if (pp != null) pp.translate (asmFile);
      }
}

%node prim_prop1 : prim_prop
{
      public double d1, d2, d3, d4;
      public prim_prop1 (double d1,double d2,double d3,double d4)
{this.d1=d1;this.d2=d2;this.d3=d3;this.d4=d4;}
      public override void translate (StreamWriter asmFile)
      {
            asmFile.WriteLine ("     Push {0}", d1);
            asmFile.WriteLine ("     Push {0}", d2);
            asmFile.WriteLine ("     Push {0}", d3);
            asmFile.WriteLine ("     Push {0}", d4);
            asmFile.WriteLine ("     Geometry\n");
      }
}
```

```
%node prim_prop2 : prim_prop
{
      public double d1, d2, d3;
      public prim_prop2 (double d1,double d2,double d3)
{this.d1=d1;this.d2=d2;this.d3=d3;}
      public override void translate (StreamWriter asmFile)
      {
            asmFile.WriteLine ("     Push {0}", d1);
            asmFile.WriteLine ("     Push {0}", d2);
            asmFile.WriteLine ("     Push {0}", d3);
            asmFile.WriteLine ("     Specular\n");
      }
}

%node prim_prop3 : prim_prop
{
      public double d1, d2, d3;
      public prim_prop3 (double d1,double d2,double d3)
{this.d1=d1;this.d2=d2;this.d3=d3;}
      public override void translate (StreamWriter asmFile)
      {
            asmFile.WriteLine ("     Push {0}", d1);
            asmFile.WriteLine ("     Push {0}", d2);
            asmFile.WriteLine ("     Push {0}", d3);
            asmFile.WriteLine ("     Diffuse\n");
      }
}

%node prim_prop4 : prim_prop
{
      public double d1, d2, d3;
      public prim_prop4 (double d1,double d2,double d3)
{this.d1=d1;this.d2=d2;this.d3=d3;}
      public override void translate (StreamWriter asmFile)
      {
            asmFile.WriteLine ("     Push {0}", d1);
            asmFile.WriteLine ("     Push {0}", d2);
            asmFile.WriteLine ("     Push {0}", d3);
            asmFile.WriteLine ("     Reflect\n");
      }
}

%node prim_prop5 : prim_prop
{
      public double d1, d2, d3;
      public prim_prop5 (double d1,double d2,double d3)
{this.d1=d1;this.d2=d2;this.d3=d3;}
      public override void translate (StreamWriter asmFile)
      {
            asmFile.WriteLine ("     Push {0}", d1);
            asmFile.WriteLine ("     Push {0}", d2);
            asmFile.WriteLine ("     Push {0}", d3);
            asmFile.WriteLine ("     Refract\n");
      }
}

%node prim_prop6 : prim_prop
{
      public double d1, d2, d3;
      public prim_prop6 (double d1,double d2,double d3)
{this.d1=d1;this.d2=d2;this.d3=d3;}
      public override void translate (StreamWriter asmFile)
      {
            asmFile.WriteLine ("     Push {0}", d1);
```

```
            asmFile.WriteLine ("     Push {0}", d2);
            asmFile.WriteLine ("     Push {0}", d3);
            asmFile.WriteLine ("     Translate\n");
      }
}

%node prim1 : prim
{
      public double d1;
      public prim1 (double d1) {this.d1=d1;}
      public override void translate (StreamWriter asmFile)
      {
            // Generate Sphere code.
            asmFile.WriteLine ("(* Build a sphere. *)");
            asmFile.WriteLine ("     Quad");
            asmFile.WriteLine ("     Push 1");
            asmFile.WriteLine ("     Push {0}", d1);
            asmFile.WriteLine ("     Push {0}", d1);
            asmFile.WriteLine ("     Mul");
            asmFile.WriteLine ("     Div");
            asmFile.WriteLine (@"     Coeff ""a11""");
            asmFile.WriteLine ("     Push 1");
            asmFile.WriteLine ("     Push {0}", d1);
            asmFile.WriteLine ("     Push {0}", d1);
            asmFile.WriteLine ("     Mul");
            asmFile.WriteLine ("     Div");
            asmFile.WriteLine (@"     Coeff ""a22""");
            asmFile.WriteLine ("     Push 1");
            asmFile.WriteLine ("     Push {0}", d1);
            asmFile.WriteLine ("     Push {0}", d1);
            asmFile.WriteLine ("     Mul");
            asmFile.WriteLine ("     Div");
            asmFile.WriteLine (@"     Coeff ""a33""");
            asmFile.WriteLine ("     Push -1");
            asmFile.WriteLine (@"     Coeff ""c""");
            asmFile.WriteLine ();
      }
}

%node prim2 : prim
{
      public double d1, d2, d3, d4;
      public prim2 (double d1,double d2,double d3,double d4)
{this.d1=d1;this.d2=d2;this.d3=d3;this.d4=d4;}
      public override void translate (StreamWriter asmFile)
      {
            // Generate Plane code.
            asmFile.WriteLine ("(* Build a plane. *)");
            asmFile.WriteLine ("     Quad");
            asmFile.WriteLine ("     Push {0}", d1);
            asmFile.WriteLine (@"     Coeff ""b1""");
            asmFile.WriteLine ("     Push {0}", d2);
            asmFile.WriteLine (@"     Coeff ""b2""");
            asmFile.WriteLine ("     Push {0}", d3);
            asmFile.WriteLine (@"     Coeff ""b3""");
            asmFile.WriteLine ("     Push {0}", d4);
            asmFile.WriteLine (@"     Coeff ""c""");
            asmFile.WriteLine ();
      }
}



scene_desc : SCENE LBRACE element_list:el RBRACE           %scene_desc1 (el);
```

```
element_list :                                              %element_list1 (null,
null);
element_list : element_list:el element:e                    %element_list1 (el,
e);

element : scene_prop:sp                                     %element1 (sp);
element : prim:p LBRACE prim_prop_list:ppl RBRACE           %element2 (p, ppl);

scene_prop : WINDOW DOUBLE:d1 DOUBLE:d2                     %scene_prop1
(d1.val,d2.val);
scene_prop : LIGHT DOUBLE:d1 DOUBLE:d2 DOUBLE:d3            %scene_prop2
(d1.val,d2.val,d3.val);
scene_prop : COP DOUBLE:d1 DOUBLE:d2 DOUBLE:d3              %scene_prop3
(d1.val,d2.val,d3.val);
scene_prop : LOOKAT DOUBLE:d1 DOUBLE:d2 DOUBLE:d3           %scene_prop4
(d1.val,d2.val,d3.val);
scene_prop : VIEWUP DOUBLE:d1 DOUBLE:d2 DOUBLE:d3           %scene_prop5
(d1.val,d2.val,d3.val);
scene_prop : AMBIENT DOUBLE:d1 DOUBLE:d2 DOUBLE:d3          %scene_prop6
(d1.val,d2.val,d3.val);

prim_prop_list :                                            %prim_prop_list1
(null, null);
prim_prop_list : prim_prop_list:ppl prim_prop:pp            %prim_prop_list1
(ppl, pp);

prim_prop : GEOM DOUBLE:d1 DOUBLE:d2 DOUBLE:d3 DOUBLE:d4    %prim_prop1
(d1.val,d2.val,d3.val,d4.val);
prim_prop : SPECULAR DOUBLE:d1 DOUBLE:d2 DOUBLE:d3          %prim_prop2
(d1.val,d2.val,d3.val);
prim_prop : DIFFUSE DOUBLE:d1 DOUBLE:d2 DOUBLE:d3           %prim_prop3
(d1.val,d2.val,d3.val);
prim_prop : REFLECTIVE DOUBLE:d1 DOUBLE:d2 DOUBLE:d3        %prim_prop4
(d1.val,d2.val,d3.val);
prim_prop : REFRACTIVE DOUBLE:d1 DOUBLE:d2 DOUBLE:d3        %prim_prop5
(d1.val,d2.val,d3.val);
prim_prop : TRANSLATE DOUBLE:d1 DOUBLE:d2 DOUBLE:d3         %prim_prop6
(d1.val,d2.val,d3.val);

prim : SPHERE DOUBLE:d1                                     %prim1 (d1.val);
prim : PLANE DOUBLE:d1 DOUBLE:d2 DOUBLE:d3 DOUBLE:d4        %prim2
(d1.val,d2.val,d3.val,d4.val);
```

Listing 3. RT example.

```
(* Program to render 3 spheres floating above a plane. *)
scene
{
      (* Define the settings for the scene. *)
      window 300.0 200.0
      light -3.5 0.5 2.0
      cop 0.0 4.0 1.0
      lookat 0.0 0.0 0.0
      viewup 0.0 0.0 1.0
      ambient 0.3 0.3 0.3

      (* Define a bunch of spheres and a plane. *)
      sphere 0.5
      {
            translate 1.0 0.0 0.0
            specular 0.7 0.0 0.0
```

```
            diffuse 0.7 0.0 0.0
            reflective 0.1 0.1 0.1
      }

      sphere 0.5
      {
            translate 0.0 -1.0 0.2
            specular 0.0 0.7 0.0
            diffuse 0.0 0.7 0.0
            reflective 0.2 0.3 0.2
      }

      sphere 0.5
      {
            translate -1.0 -0.5 0.0
            specular 0.0 0.0 0.7
            diffuse 0.0 0.0 0.7
            reflective 0.1 0.1 0.1
      }

      plane -0.1 0.0 1.0 1.0
      {
            specular 0.0 0.0 0.0
            diffuse 1.0 0.5 0.0
            reflective 0.25 0.25 0.25
      }
}
```

Listing 4. EXP DSL complete lexer definition.

```
%lexer
%namespace ExpcLexer

%token                    DOUBLE {public double val;}

"let"                     %LET
"in"                      %IN
"end"                     %END
"sum"                     %SUM
".."                      %DOTDOT
"="                       %EQU
"+"                       %PLUS
"-"                       %MINUS
"*"                       %TIMES
"/"                       %DIV
"("                       %LBRACKET
")"                       %RBRACKET
[_a-zA-Z][_a-zA-Z0-9]*    %ID
[-+]?[0-9]+(\.[0-9]+)?    %DOUBLE {val = Double.Parse (yytext);}
[ \n\r\f\t]               ;
"(*"                      {yybegin ("COMMENT");}
<COMMENT>"*)"             {yybegin ("YYINITIAL");}
<COMMENT>.                ;
<COMMENT>\n               ;
```

Listing 5. EXP DSL complete grammar definition.

```
%parser expc.lexer
%namespace ExpcParser
```

```
%{
using ExpcLexer;
%}



%symbol FExp;
%symbol LetExp;
%symbol var_list;
%symbol var;
%symbol exp;
%symbol term;
%symbol fact;



%node FExp1 : FExp
{
      public LetExp le;
      public FExp1 (LetExp le) {this.le=le;}
}

%node LetExp1 : LetExp
{
      public var_list vl;
      public exp e1;
      public LetExp1 (var_list vl, exp e1) {this.vl=vl;this.e1=e1;}
}

%node var_list1 : var_list
{
      public var_list vl;
      public var v;
      public var_list1 (var_list vl, var v) {this.vl=vl;this.v=v;}
}

%node var1 : var
{
      public string n;
      public double d;
      public var1 (string n, double d) {this.n=n;this.d=d;}
}

%node exp1 : exp
{
      public exp e1;
      public term t1;
      public exp1 (exp e1,term t1) {this.e1=e1;this.t1=t1;}
}

%node exp2 : exp
{
      public exp e1;
      public term t1;
      public exp2 (exp e1,term t1) {this.e1=e1;this.t1=t1;}
}

%node exp3 : exp
{
      public term t1;
      public exp3 (term t1) {this.t1=t1;}
}

%node term1 : term
```

```
{
      public term t1;
      public fact f1;
      public term1 (term t1,fact f1) {this.t1=t1;this.f1=f1;}
}

%node term2 : term
{
      public term t1;
      public fact f1;
      public term2 (term t1,fact f1) {this.t1=t1;this.f1=f1;}
}

%node term3 : term
{
      public fact f1;
      public term3 (fact f1) {this.f1=f1;}
}

%node fact1 : fact
{
      public LetExp le;
      public fact1 (LetExp le) {this.le=le;}
}

%node fact2 : fact
{
      public string n;
      public fact2 (string n) {this.n=n;}
}

%node fact3 : fact
{
      public double d;
      public fact3 (double d) {this.d=d;}
}

%node fact4 : fact
{
      public exp e1;
      public fact4 (exp e1) {this.e1=e1;}
}

%node fact5 : fact
{
      public string n1;
      public double d;
      public string n2;
      public exp e1;
      public fact5 (string n1,double d,string n2,exp e1)
      {
            this.n1=n1;
            this.d=d;
            this.n2=n2;
            this.e1=e1;
      }
}



FExp     : LetExp:le                         %FExp1 (le);

LetExp   : LET var_list:vl IN exp:e1 END     %LetExp1 (vl, e1);
```

```
var_list :                                  %var_list1 (null, null);
var_list : var_list:vl var:v                %var_list1 (vl, v);

var      : ID:n EQU DOUBLE:d                %var1 (n.yytext, d.val);

exp      : exp:e1 PLUS term:t1              %exp1 (e1, t1);
exp      : exp:e1 MINUS term:t1             %exp2 (e1, t1);
exp      : term:t1                          %exp3 (t1);

term     : term:t1 TIMES fact:f1            %term1 (t1, f1);
term     : term:t1 DIV fact:f1              %term2 (t1, f1);
term     : fact:f1                          %term3 (f1);

fact     : LetExp:le                        %fact1 (le);
fact     : ID:n                             %fact2 (n.yytext);
fact     : DOUBLE:d                         %fact3 (d.val);
fact     : LBRACKET exp:e1 RBRACKET         %fact4 (e1);
fact     : SUM LBRACKET ID:n1 RBRACKET DOUBLE:d DOTDOT ID:n2 LBRACKET exp:e1
RBRACKET   %fact5 (n1.yytext, d.val, n2.yytext, e1);
```

Listing 6. EXP example.

```
let
    n = 5
in
    9 + sum (i) 1..n (2 * i)
end
```

Listing 7. LQ DSL complete lexer definition.

```
%lexer
%namespace LqcLexer

"let"                    %LET
"in"                     %IN
"end"                    %END
"turn"                   %TURN
"sew"                    %SEW
"fun"                    %FUN
"val"                    %VAL
"a"                      %A
"b"                      %B
"="                      %EQU
","                      %COMMA
"("                      %LBRACKET
")"                      %RBRACKET
[_a-zA-Z][_a-zA-Z0-9]*   %ID
[ \n\r\f\t]              ;
"(*"                     {yybegin ("COMMENT");}
<COMMENT>"*)"            {yybegin ("YYINITIAL");}
<COMMENT>.               ;
<COMMENT>\n              ;
```

Listing 8. LQ DSL complete grammar definition.

```
%parser lqc.lexer
%namespace LqcParser
```

```
%{
using LqcLexer;
%}



%symbol LQExp;
%symbol exp;
%symbol decls;
%symbol decl;



%node LQExp1 : LQExp
{
      public exp e1;
      public LQExp1 (exp e1) {this.e1=e1;}
}

%node exp1 : exp
{
      public exp1 () {}
}

%node exp2 : exp
{
      public exp2 () {}
}

%node exp3 : exp
{
      public exp e1;
      public exp3 (exp e1) {this.e1=e1;}
}

%node exp4 : exp
{
      public exp e1;
      public exp e2;
      public exp4 (exp e1,exp e2) {this.e1=e1;this.e2=e2;}
}

%node exp5 : exp
{
      public decls dl;
      public exp e1;
      public exp5 (decls dl,exp e1) {this.dl=dl;this.e1=e1;}
}

%node exp6 : exp
{
      public string n;
      public exp e1;
      public exp6 (string n,exp e1) {this.n=n;this.e1=e1;}
}

%node exp7 : exp
{
      public string n;
      public exp e1;
      public exp e2;
      public exp7 (string n,exp e1,exp e2) {this.n=n;this.e1=e1;this.e2=e2;}
}
```

```
%node exp8 : exp
{
      public string n;
      public exp8 (string n) {this.n=n;}
}

%node decls1 : decls
{
      public decl d;
      public decls1 (decl d) {this.d=d;}
}

%node decls2 : decls
{
      public decl d;
      public decls dl;
      public decls2 (decl d,decls dl) {this.d=d;this.dl=dl;}
}

%node decl1 : decl
{
      public string n1;
      public string n2;
      public exp e1;
      public decl1 (string n1,string n2,exp e1)
{this.n1=n1;this.n2=n2;this.e1=e1;}
}

%node decl2 : decl
{
      public string n1;
      public string n2;
      public string n3;
      public exp e1;
      public decl2 (string n1,string n2,string n3,exp e1)
{this.n1=n1;this.n2=n2;this.n3=n3;this.e1=e1;}
}

%node decl3 : decl
{
      public string n1;
      public exp e1;
      public decl3 (string n1,exp e1) {this.n1=n1;this.e1=e1;}
}



LQExp    : exp:e1                                                           %LQExp1
(e1);

exp      : A                                                                %exp1 ();
exp      : B                                                                %exp2 ();
exp      : TURN LBRACKET exp:e1 RBRACKET                                    %exp3
(e1);
exp      : SEW LBRACKET exp:e1 COMMA exp:e2 RBRACKET                        %exp4
(e1,e2);
exp      : LET decls:dl IN exp:e1 END                                       %exp5
(dl,e1);
exp      : ID:n LBRACKET exp:e1 RBRACKET                                    %exp6
(n.yytext,e1);
exp      : ID:n LBRACKET exp:e1 COMMA exp:e2 RBRACKET                       %exp7
(n.yytext,e1,e2);
exp      : ID:n                                                             %exp8
(n.yytext);
```

```
decls    : decl:d                                                     %decls1
(d);
decls    : decl:d decls:dl                                            %decls2
(d, dl);

decl     : FUN ID:n1 LBRACKET ID:n2 RBRACKET EQU exp:e1               %decl1
(n1.yytext,n2.yytext,e1);
decl     : FUN ID:n1 LBRACKET ID:n2 COMMA ID:n3 RBRACKET EQU exp:e1   %decl2
(n1.yytext,n2.yytext,n3.yytext,e1);
decl     : VAL ID:n1 EQU exp:e1                                       %decl3
(n1.yytext,e1);
```

Listing 9. LQ example.

```
let
    fun unturn (x)  = turn (turn (turn (x)))
    fun pile (x, y) = unturn (sew (turn (y), turn (x)))
    val s = turn (sew (unturn (unturn (a)), b))
    val t = pile (turn (b), unturn (b))
    val u =  turn (turn (sew (s, t)))
    val m = sew (turn (a), turn (turn (a)))
    val n = sew (unturn (a), a)
    val p = pile (m, n)
    val q =  sew (sew (u, p),  turn (u))
in
    pile (q, turn (turn (q)))
end
```

References

[Ale01] Alexander, I. Visualizing Requirements in UML. Telelogic White Paper, 2001

[App98] Appel, A. W. Modern Compiler Implementation in Java. Cambridge, 1998

[Arn95] Arnold, B. R. T., Deursen, A. v., and Res, M. An Algebraic Specification of a Language for Describing Financial Products. Proceedings of the ICSE-17 Workshop on Formal Methods Application in Software Engineering Practice, 1995

[Bac81] Backus, J. W. The History of Fortran I, II and III. History of Programming Languages [Wex81], 25-74, 1981

[Bjö02] Björkander, M. Model-Driven Development and UML 2.0. Telelogic White Paper, 2002

[vdBra01] Brand, M. G. J. v. d., et al. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. Wilhelm, R., Ed. Compiler Construction 2001. pp.365-370. Springer-Verlag

[Cro03] Crowe, M. K. Compiler Writing Tools Using C#, 2003, http://cis.paisley.ac.uk/crow-ci0/

[Cza99] Czarnecki, K., and Eisenecker, U. Generative Programming: Methods, Techniques, and Applications. Addison-Wesley, pp.273-291, 1999

[vDeu97] Deursen, A. v. Domain-Specific Languages versus Object-Oriented Frameworks: A Financial Engineering Case Study

[vDeu98a] Deursen, A. v., and Klint, P. Little Languages: Little Maintenance? Journal of Software Maintenance, volume 10, 1998

[vDeu98b] Deursen, A. v., Klint, P., and Verhoef, C. Research Issues in the Renovation of Legacy Systems. Fundamental Approaches to Software Engineering, Springer-Verlag, 1999

[vDeu98c] Deursen, A. v., Klint, P., and Visser, J. Domain-Specific Languages. ACM Computing Classification System: D.3

[vDeu99] Deursen, A. v., and Kuipers, T. Identifying Objects using Cluster and Concept Analysis. Proceedings of the 21st International Conference on Software Engineering, ICSE-99, ACM, 1999

[vDeu01a] Deursen, A. v., and Klint, P. Domain-Specific Language Design Requires Feature Descriptions. ACM Computing Classification System: D.2.2, D.2.9, D.2.11, D.2.13. Journal of Computing and Information Technology, 2001

[vDeu01b] Deursen, A. v. Generative Programming. Review article of Generative Programming: Methods, Tools, and Applications, by Krzysztof Czarnecki and Ulrich W. Eisenecker. Addison-Wesley, 2000, ISBN 0-201-30977-7. IEEE Software, March/April 2001

[Gur03] Gurd, A. Using UML 2.0 to Solve Systems Engineering Problems. Telelogic White Paper, 2003

[Irw97] Irwin, J., Loingtier, J., Gilbert, J. R., Kiczales, G., Lamping, J., and Mendhekar, A. Aspect-Oriented Programming of Sparse Matrix Code. Proceedings of the International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE), Springer-Verlang, 1997

[Jac96] Jacobsen, E. E., and Nowack, P. A Pattern Language for Building Virtual Machines, 1996

[Kic96] Kiczales, G. Aspect-Oriented Programming, 1996, Xerox PARC, http://www.parc.xerox.com/spl/projects/aop/position.htm

[Lib02] Liberty, J. Programming C#. O'Reilly and Associates, Inc., 2002

[Mag99] Magee, J., and Kramer, J. Concurrency: State Models and Java Programs. John Wiley and Sons, 1999

[Mil84] Milner, R. A Proposal for Standard ML. ACM Symposium on Lisp and Functional Programming, 184-197, 1984

[Mon95] Montrose, R. C. Object-Oriented Development Using The Shlaer-Mellor Method. Project Technology, Inc., 1995. http://www.projtech.com

[Oli95] Olivier, P. A. Embedded System Simulation: Testdriving the ToolBus. Programming Research Group, University of Amsterdam

[Pre97] Pressman, R. S. Software Engineering: A Practitioner's Approach, Fourth Edition. McGraw-Hill, 1997

[Rie01] Riehle, D., Fraleigh, S., Bucka-Lassen, D., and Omorogbe, N. The Architecture of a UML Virtual Machine. Proceedings of the 2001 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2001). ACM Press, 2001

[Set97] Sethi, R. Programming Languages: Concepts and Constructs, Second Edition. Addison-Wesley, 1997

[Shl92] Shlaer, S., and Mellor, S. J. Object Lifecycles: Modeling the World in States. Yourdon Press, P. T. R. Prentice Hall, 1992

[Shl96] Shlaer, S., and Mellor, S. J. The Shlaer-Mellor Method. Project Technology, Inc., 1996. http://www.projtech.com

[Smi05] Smith, J. E., and Nair, R. Virtual Machines: Versatile Platforms for Systems and Processes. Morgan Kaufmann, 2005

[Ull94] Ullman, J. D. Elements of ML Programming. Prentice Hall, 1994

[Wat00a] Watt, A. 3D Computer Graphics. Addison-Wesley, pp.342-369, 2000

[Wex81] Wexelblat, R. L. History of Programming Languages. Academic Press. New York, 1981

[Whi98] Whitten, J. L., and Bentley, L. D. Systems Analysis and Design Methods, Fourth Edition. McGraw-Hill, 1998

...oooOooo...