The VPL Language

Author: Itay Itzhaky

Advisor: Mr. Ehud Lamm, Course Administrator

Submission Date: 01/10/2005

Written as part of the Programming Languages workshop course 20504,
the Open University, Computer Science Department, 2005

Foreword

This relatively short paper describes in a nutshell what VPL is all about. It explains the essentials of the VPL programming language, its major properties, its design philosophy, and its key features. The final paragraph of the paper describes the VPL Platform that is the implementation which I've built for the VPL language under the framework of the project.

This paper doesn't pretend to be the formal language manual, neither a tutorial for learning the language. Rather it focuses on the important and more unique features (mainly implicit parallelism) and on reasoning and clarifying important design choices that I've made. I won't get into great details about features that are standard and exist in almost every imperative language in such way or another. I also won't demonstrate the exact syntax of the language or how to write code in it. For concrete code samples you might want to take a look at the VPL standard library source files or the sample applications that come along with the VPL Platform Installation CD.

I assume that the reader of this paper has a good knowledge on the topic of programming languages and that he is familiar with at least one modern object-oriented imperative language (C#, Java, etc…). This should be enough to understand every bit of code in VPL.

This paper refers to VPL version 1.0. This version is not complete since it was built for the purpose of the course project only. I could not afford the time needed to build every single feature that I wanted the language to include. Whenever I'll discuss a feature or design choice that I would like to extend or just do differently in future versions I will add a "future extension" note that explains how this feature may look in future versions.

I would like to take this opportunity to thank Ehud Lamm, My advisor who accompanied and instructed me through the whole long-term process of designing and building my first programming language, VPL.

Table of Contents

Introduction 1
The Design Philosophy..1
The Evaluation Model..2
The Type System...2
System Services 4
The Standard Library 5
Packages...5
Preprocessor Directives 6
Comments 6
Namespaces...6
Modules...7
Methods...8
Variables..9
Flow Control 10
Expressions 10
Vector Comprehensions 13
Implicit Parallelism...15
The VPL Platform 17
Bibliography 19
Appendix A: Language EBNF Grammar 20
Appendix B: Operator Expressions Table 24
Appendix C: Type Conversions Table 26
Appendix D: System Services Table 27
Appendix E: Framework Classes Diagrams 28

Introduction

VPL (an acronym for Vector Processing Language) is a modern, high-level, general-purpose, imperative, block-structured language that is designed from scratch with built-in support for implicit parallelism capabilities in mind.

VPL is aimed mainly toward the building of computation-based applications which perform intensive processing operations over vectors.

The primary goal of VPL is to allow for an effective implicit parallelism [3] that is fully automatic and managed solely by the runtime system without any intervention from the programmer side.

The Design Philosophy

A fundamental principle in VPL design was to keep side effects [1] under strict control. This principle had to be achieved even in the price of applying stricter constraints and further limiting the language or otherwise a built-in support for an effective implicit parallelism would have been impossible. The only problematic feature that contradicts this principle is the System Services mechanism (e.g. IO operations). This feature was included in this version however as a temporary solution and it might very well be replaced in future versions with a more flexible and sophisticated mechanism such as Haskell monads [6] that allows a more subtle differentiation between pure and impure functions [1] .

VPL design aspires to achieve full referential transparency [1] and to avoid side effects as much as possible. It does so however without applying too strict limitations like forbidding multiple assignments for variables all together as in pure functional programming languages.

VPL design is inspired and influenced by two well known and important programming languages: the first is C# who contributed mainly to the imperative constructs that the language supports, and to the overall structure and syntax of the language. The second language is Haskell [5] who donated the list comprehensions

construct (in a slightly modified form) which plays an important role in strengthen the expressiveness of vectors so they can be used to resolve even a wider set of problems.

The Evaluation Model

The evaluation model for VPL is eager evaluation [1]. This choice has made a lot of sense for several reasons: first VPL is an imperative language; hence quite often high percentage of the code in it consists of flow control statements that need the expressions values in order to be elaborated. Second, a certain level of side effects is allowed what would have further complicated the implementation of a lazy evaluation mechanism [1]. And Last but not least reason has to do with the implicit parallelism optimization. In large we can say that in lazy evaluation the evaluation of expressions is delayed until the very moment where their values are "really" needed. Thus the values of vector components will be computed one by one as they are being used for real and not altogether when the vector value is first constructed. This behavior constitutes a real problem for the implicit parallelism optimization that is based on current computation of the vector components as you'll see in the paragraph on implicit parallelism.

In future versions however I'd like to add a support for lazy evaluation in some way or another so it will be possible to create infinite vectors using vectors comprehensions (just like infinite lists in Haskell [5]).

The Type System

VPL is a strongly dynamically typed language [1] i.e. variables are not associated with a specific type and no type checking is performed at compile time. Variables however still have to be declared explicitly before they are used in order to avoid errors that may result from misspelling of their names. The runtime system is responsible for checking that operators are applied only to operands of compatible types.

The VPL Type System includes five built-in types: ***Scalar, Char, String, Boolean***, and ***Vector***. The first four are primitive types and the last one, vector, is a constructed type.

Vectors are the primary and only built in data structure in VPL. They are an index based sets of zero or more values. The values are referred to as the vector components and the size of the vector is defined as the number of its components. Vectors must have a finite size. They are pretty much like regular arrays, the only differences between them are that in vectors each component can be of a different type and that vectors don’t have dimensions they can be considered in that aspect as a single dimensional arrays. Multi-dimensional like vectors can be simulated by making vectors that their components are also vectors. Indices for vectors are one based. Vectors values can be created using one of the following dedicate expressions: vector-comprehension expressions, integer-range expressions, and vector-literal expressions. These expressions will be discussed in details in the paragraph about expressions.

All the five built-in types including the vector type are value types i.e. an assignment of value of their type to a variable creates a **copy** of the assigned value. I particularly emphasized the vector type since in most of the languages vectors (arrays) are reference types. Making the vector type a reference type would have been essentially a wise choice in the sense that it would prevent inefficiencies that may result from redundant copy of vector values which are almost always much bigger in size rather than a simple reference. Reference types however causes an effect known as aliasing [1] which might lead to side effects and break the fundamental design principle of achieving referential transparency. Therefore making any type in VPL including the vector type a reference type is not an option at all for references collide with the implicit parallelism optimization feature.

In VPL there are two sets of values, expressed values and denoted values. The expressed values correspond to the built-in types:

- ***Scalar Value*** – Real Number (e.g. 333,-22.55,0.444)
- ***Char Value*** – ASCII character (e.g. 'r' '}' '0')
- ***String Value*** – Text (e.g. "Hello World")

- ***Boolean Value*** – true/false
- ***Vector Value*** – Set of Values (e.g. {1,2,3} , {'r',"ggg",3,{1,9}})

The denoted values are references to the locations in memory where the expressed values reside.

VPL supports implicit and explicit conversion of types through cast expressions. Implicit and explicit conversion operations cannot fail, however they might result in loss of data. As a rule of thumb I minimized the use of implicit casts and avoided any cast the might be confusing or may lead to unexpected results. For the complete list of supported built-in conversions see appendix C.

System Services

System services are a mechanism that enables VPL programs to interact with the "outside world" (the operating system, the machine, the user). A system service is simply an external function (it has neither declaration nor definition in VPL) that can be invoked using a dedicated expression called a system-call expression. System services are intended for performing tasks such as: IO, Threading, Resources management, etc… VPL specification says nothing about which system services should be provided. This is owing to the fact that the required system services depend greatly on the specific target operating system which a VPL implementation is written for and might vary dramatically from one implementation to another. For the complete list of supported system services for this version of the VPL Platform implementation see appendix D.

The System services are thread-safe that is, they can be used inside parallelizable constructs such as vector comprehensions when the implicit parallelism optimization is turned on and the system won't become unstable and surely it won't crush. Nevertheless when being used this way they may lead to indeterministic results. For a concrete example see the paragraph on implicit parallelism.

The Standard Library

The standard library consists of a set of fundamental, commonly used types which provide methods that some of them wrap and abstract the functionality of underlying system services (e.g. the method ***Console.Write*** which prints to console) and others just provide a common re-usable functionality (e.g. the method ***Vector.Quicksort*** which sorts a vector). The standard library allows for better re-usability of common code and even more important it enables VPL programs to be portable, that is, independent of the specific operating system on which they are executed. In order for a VPL program to be portable it should make no direct invocation of system services at all but only use them indirectly through the standard library "layer". For example instead of invoking directly the STD_WRITE system service in order to write something to the console a portable program should use the method ***Write*** in the type ***System.IO.Console*** which does the same thing but is independent of the runtime system.

In the current version, the standard library is yet to be standardized and it contains only few types and even fewer methods.

Packages

Packages are the building blocks of VPL programs. They encapsulate code, resources, and possibly references to other packages. They form the fundamental units of deployment and scoping and they are the only elements that runtime system recognizes and knows how to execute directly. When compiling a VPL source code the output is a package file.

For the current version the only type of package supported is an application package (the equivalent of executable files). In future versions however a support for additional types might be added (e.g. library packages, service packages, etc...).

Application packages reside in a single file (with .vpp extension). They have an additional dedicated property called the entry point, which tells the runtime system where to start executing their code from.

Preprocessor Directives

VPL supports only a single preprocessor directive, the ***include*** directive, which tells the preprocessor to paste the content of the specified file into the location in code where the ***include*** directive appears. It is declared using the @ preprocessor directive symbol character followed by the ***include*** preprocessor keyword and a file-path. Preprocessor directives may appear anywhere in the source code.

Comments

VPL supports two forms of comments: single-line comments, and delimited comments. Single-line comments starts with the characters "/~" and extends to the end of the line. Delimited comments start with the "/*" characters and end with the "*/" characters. Comments may appear anywhere in the source code.

Namespaces

Namespaces enable the organization of the code inside a package and provide the programmer with the ability to control scope of types and methods.

Namespaces are declared through the use of the ***namespace*** keyword followed by the namespace name and braces which delimits the namespace body. Their declaration can span over multiple code files and multiple namespaces can reside in a single file. In compile time however a spanned namespace is unified into one big namespace with all the members defined in it.

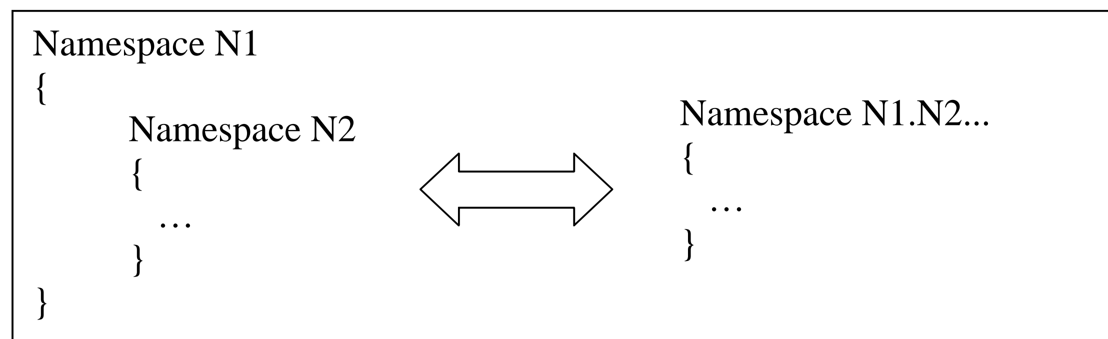
Namespaces may contain as direct members other namespaces or types. Even if no namespace is explicitly declared a default one is always created. This namespace is called the global namespace and it contains as direct members all the namespaces and types that are not members of any other namespace.

The scope of one namespace can be imported into the scope of another by using the ***import*** directive in the latter one. Import directives must appear first in the namespace

body, prior to any other member declaration. When identifiers are resolved in compile time, the identifiers of the namespace direct members take precedence over those of the imported ones.

If an identifier in an outer scope is hidden by an identifier in an inner scope and an access to it is needed the ***global*** keyword can be used to force the compiler to start looking for the identifier from the outermost namespace (e.g. ***global.N1.N2...***)

A syntactic sugar allows the declaration of a namespace hierarchy in a flat form using identifiers and periods instead of in a nested form



Modules

Modules are like classes (OOP terminology) except to the extent that they cannot inherit, be inherited, or be instantiated. The members of a module are implicitly considered to be non-instance members (static members, C# terminology).

Modules are declared using the ***module*** keyword followed by the module name and braces which delimits the module body.

Access modifiers can be specified to module members. They define the level of accessibility of the member. The two supported access modifiers are:

- ***public*** – access to the member is unrestricted
- ***private*** – access is restricted to the containing module only

If an access modifier is not explicitly specified the private modifier is assumed to be the default one.

Modules are the only kind of types supported in this version of VPL. Likewise the only supported members are methods. Field members are not permitted for they are just like global variables in that they are shared between several methods and can easily lead to side effects and break the principle of referential transparency that is crucial for making an effective implicit parallelism. In future versions however I plan to add a support for additional types in order to make VPL a fully-compatible OOP language. Regarding field members I will add a support for them also by making a more subtle differentiation between pure and impure methods.

Methods

Methods are functions that are members of a type. All the functions in VPL are methods. Global functions are not allowed for they somewhat contradict the OOP design pattern.

Methods are declared using the ***method*** keyword followed by the method name, parentheses that enclose the formal parameters list and braces that enclose the method body. Methods can optionally return a single value using a return statement. VPL supports method overloading based on the number of formal parameters

Formal parameters just like local variables are also type-less. They can be specified with a direction modifier that defines how the value of their corresponding actual parameter should be passed (e.g. by value, by reference, etc...). The only direction modifier supported in the current version is ***in;*** it has the same effect of passing a parameter by value in the sense that the method cannot change the value assigned to the corresponding actual parameter and cause a side effect.
This behavior however is achieved not by copying the actual value but by making ***in*** parameters read-only variables. This way we can pass parameters by reference and yet be sure that the values are not changed. Since vector values dominate the VPL language and almost always they are much bigger in size than a simple reference it

becomes more worthwhile to pass parameters by reference and copy the references rather than the value.

One special method is the entry method. This method is where the execution of the application begins. An application must have exactly one entry method. An entry method is declared by adding the ***entry*** keyword prior to the ***method*** keyword in the method declaration. Just like normal methods an entry method can also have formal parameters and return a value. The values for its formal parameters are supplied through the command-line arguments and its return value constitutes the application exit value.

Variables

VPL supports declaration of variables with multiple assignments. The declaration scope for a variable follows the lexical-binding scoping rules. Global variables are not supported for they allow side effects that break the rule of referential transparency.

Variables are declared using the ***var*** keyword followed by the variable name and optionally "**<-**" characters followed by an initial value which will be assigned to the variable when it is first bounded. A syntactic sugar enables multiple variables to be declared using a single ***var*** keyword and commas (var v1<-i1, v2<- i2 …). Variable declaration statements may appear anywhere inside the body of a method.

VPL provides an additional scoping level using blocks. Blocks are declared using the ***block*** keyword followed by braces which delimits the block code. Blocks allow for more flexible control of the variables scope inside a method.

Variables or more precisely their names are bounded to storage locations once on their creation and they cannot be re-bounded later to a different location. Values however can be assigned to variables anytime using the assignment statement which takes the form of a ***left value*** followed by **<-** operator and ***right value***.

Flow Control

VPL borrows all its flow control statements from the C# language. Below is given the complete list of supported statements:

- ***While*** – execute a statement or block of statements until the specified Boolean expression evaluates to false
- ***If-Else*** – select a statement or block of statements for execution based on the value of the specified Boolean expression
- ***Break*** – terminates execution of the enclosing while
- ***Continue*** – skip execution of the current iteration of the enclosing while.
- ***Return*** – terminates execution of the active method and returns control to the calling method. If a value is specified for the return statement this value is returned to the calling method. If the active method is the entry method the application execution is over.

For more descriptive information about those statements see C# formal specification document [7].

Expressions

VPL supports the following types of expressions:

- **Literal expression** – a literal expression is a char, a string, a number, the keywords true/false, or a vector literal. A vector literal is constructed of an expressions list - that represents the vector components – enclosed in braces. Literal expressions evaluate directly to the data value which they represent.
  Examples: ***"Hello", 33.54,'t', {10,'f', {100+50, 6/3}}***

- **Operator expression** – an operator expression is a code fragment that is constructed of an operator and operands. The operator along with the number of operands and their data types determines unambiguously the operation to be performed on the operands values. An operator expression evaluates to the data value that is returned by its underlying operation. The precedence and associativity rules for operator expressions in VPL follow those of the C# language [7]. VPL supports unary and binary expressions. Binary expressions

are written in an intrinsic fashion. Applying an operator to operands of invalid types will result in a runtime error. For the complete list of operators and their associated operations see appendix B. Examples: ***20* + *(60/-4), "Hey"* + *"!!!"***

- **Name expression** – a name expression consists of an identifier that refers to a type, a namespace, a method group, a formal parameter or a variable.
  A name expression evaluates to a value of the same type like the element to which it refers to. If the referred element does not exist a compile error will be raised. VPL allows forward references [2] for every element but variables. Using a name expression that refers to an element that is not associated with a data value (e.g. a namespace element), where a data value is expected (e.g. an operand in an operator expression) will also result in a compile error. Examples***: MyMethod, m_name2_ , t1***

- **Member-Access expression** – a member-access expression consist of a container expression that refers to a namespace or type and a member identifier that refers to the name of a member in the container. A member-access expression evaluates to the same type like the member to which it refers to. If the container expression doesn't refer to a namespace or type or if the member identifier refers to a member that doesn't exist a compile error will be thrown. Examples: ***MyNamespace.MyType.MyMethod***

- **Function-call expression** – a function-call expression consists of an expression which refers to a method group and a list of actual parameters expressions which their values will be passed to the method on invocation. The method group along with the number of actual parameters determines unambiguously the exact method to be invoked. A function-call expression evaluates to the value return by the method, if the method returned no value and the function-call expression is used as a data value (e.g. as an operand in an operator expression) a runtime error will be thrown. Examples: ***f(3,in "hi")***

- **Cast expression** – a cast expression consists of a target-type expression that refers to the type which we'd like to convert the value to and a source expression which its value is to be converted. A cast expression evaluates to the data value returned by the conversion operation (this data value of course is of the same type like the target type). If no direct conversion exists between the value type of the source expression to the target type a runtime error will be thrown. For the complete list of supported conversions see appendix C.
  Examples: ***char!(48) = '0' , scalar!('a') = 97***

- **Instance-of expressions** – an instance-of expression consist of a target-type expression and a source value expression. It evaluates to a ***true*** value if the source expression value type is the same like the target type otherwise it evaluates to ***false*** Examples: ***string?(555) = false, scalar?(true) = false***

- **Indexer expression** – an indexer expression consists of a collection-object expression and an index expression. The collection-object expression must evaluate to a data value of a type that supports indexing. In the current version the only type that support indexing is the vector type however in future versions other types will be supported as well (mainly the string type). The index expression may evaluate to a positive-integer scalar value or a vector value of only positive-integer scalar components. If the index expression evaluates to the former value the indexer expression evaluates to a variable (reference) type. If it evaluates to the latter value the indexer expression evaluates to a vector value which contains the components specified by the set of indices. If an index is not a positive integer scalar or if it is out of the vector's range a runtime error will be thrown. Examples***: {7,5,3}[1] = 7, {7,5,3}[{2,3}] = {5,3}***

- **Integer-Range expression** – an integer-range expression consist of a lower-bound expression and a higher-bound expression. It evaluates to a vector value whose components are the integer values in the range specified by the lower and upper bounds expressions which must both evaluate to a scalar values.

If the lower and higher bound values are not integers they are rounded and their rounded value is considered. If the lower bound is greater than the higher bound an empty vector is returned. Examples: ***{1.2..3.5} ={1,2,3}, {3..1} = {}***

- **Conditional expression** – a conditional expression consists of a condition expression, a true expression, and a false expression. It evaluates to the value of the true expression if the condition expression evaluates to true otherwise it evaluates to the value of the false expression. Examples: ***(10 > 2 -> 4 | 3) = 4***

- **System-call expression** – a system call expression invokes a system service and evaluates to the data value returned by it (or to nothing if no value is returned). It consists of an identifier that represents the name of the target system service and a list of actual parameters that will be passed for the system service on invocation. If the system service doesn't exist a runtime error will be thrown (this rule cannot be checked for statically by the compiler since the compiler and the runtime systems may be of different versions).
For the complete list of system services and their description see appendix D.
Examples***: $STD_WRITE("Hello World!!!");***

- **Vector-Comprehension expression –** vector comprehensions are discussed in details in the following paragraph.

VPL allows only the function/system call expressions to be executed directly i.e. to be used like statements. Other expressions must appear inside a container statement such as the assignment statement.

Vector Comprehensions

The vector comprehensions construct is an important construct in VPL. It plays an important role in easing the manipulation of vectors and it strengthens their expressiveness so they'll be suitable for solving even a wider range of problems.

Vector comprehensions are the equivalent of list comprehensions in Haskell [5]; the only differences between the two are the type of the underlying data structure used to store the results (vectors in VPL, lists in Haskell) and few syntactic differences.

Vector comprehensions guarantee a well-defined order for the generated components Under any circumstances including when the parallelism optimization is turned on.

The operational semantics of vector comprehensions can be modeled using the Haskell's high order functions: ***foldr, map*** and ***filter*** [4]. To start with, consider the general form of vector comprehensions: ***{body|q1, q2,…,qn}*** where body is an expression and q1..qn (n>0) are qualifiers. A qualifier can be of two kinds: a guard expression, or a generator variable. The general form can be transformed into the following simplified form: ***{body(x)|x<-V,p(x)}*** which has yet the same level of expressiveness power. In the simplified form there is exactly one generator variable ***x*** and exactly one guard predicate function ***p*** with ***x*** as it's only bounded variable [1]. The body is also a function with x as its only bounded variable. The universal quantifier effect achieved by multiple generators and multiple guards in the general form can be achieved in the simplified form by nesting comprehension constructs. For example:

{ x+y | x <- {1..3}, y<- {1..x} , x > y } ⇔ { {x + y| y<-{1..x}, x > y }| x <- {1..3}}

Now after we've seen that the simplified and general forms are equivalent in power we can model the simplified form and generalize the results to the general form as well. The transformation phases and the final model are given below:

{body(x) |x<-V, p(x)} ⇔ (map body {x| x<-V, p(x)}) ⇔ (map body (filter p V))

The model above can be transformed further to an expression that depends solely on the ***foldr*** operator by modeling the ***map*** and ***filter*** functions using ***foldr*** [4].

The model depicted above matches the operational semantic for the sequential execution of vectors comprehensions. It doesn't match however that of the parallel execution. In the next paragraph I'll explain why in spite of the differences the semantics are basically equivalent.

Vector comprehension expressions can be optimized by rearranging their qualifiers so the range of values needed to be inspected will be minimized. This optimization however is not trivial at all and it entails several complications.

Implicit Parallelism

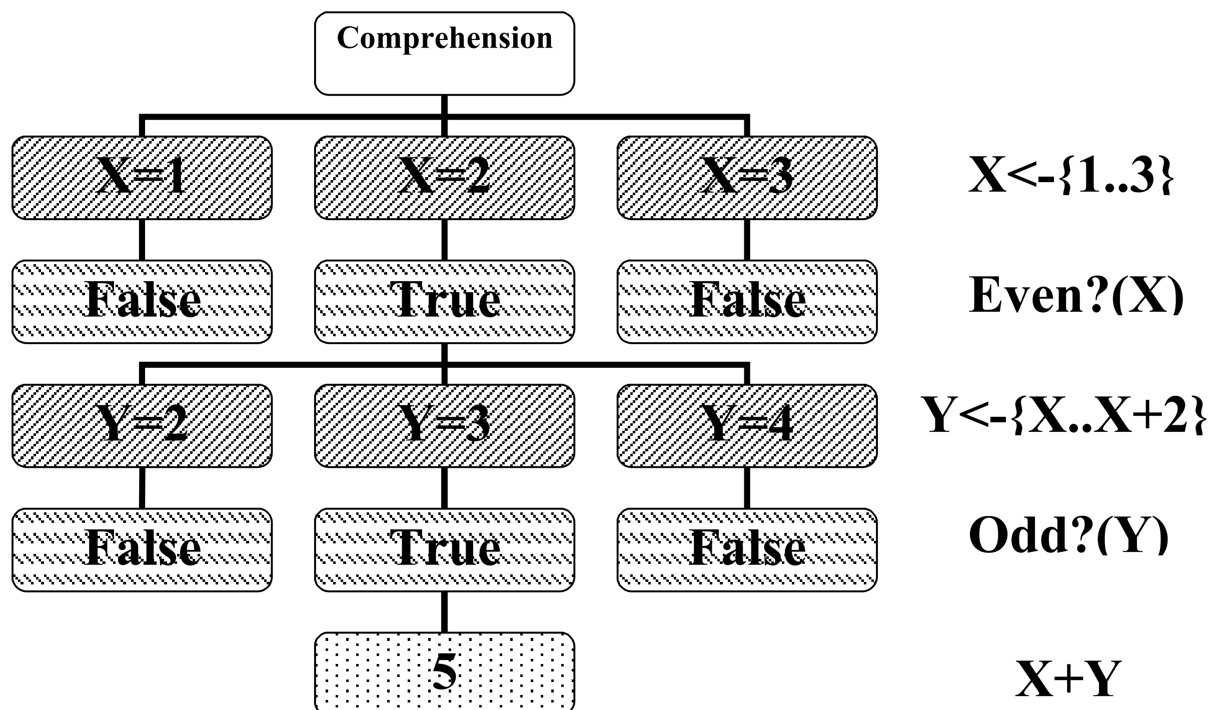
Implicit parallelism [3] is a key feature in VPL; it has had a major influence on the entire design and on the restrictions that finally had to be applied to the language. If taken to the extreme then VPL language is all about applying implicit parallelism techniques which usually allow for positive results mainly in pure functional languages where there are no side effects at all to a language that is essentially an imperative language with a certain level of side effects.

Implicit parallelism allows programmers to write their programs without any concern about how to exploit opportunities for parallelism. Exploitation of parallelism is done instead fully automatically by the compiler or the runtime system. The biggest win with implicit parallelism is that it relief the programmer from the tedious and error-prone manual parallelization process. A major drawback of implicit parallelism however is that it does not always achieve optimal performance; moreover sometimes it can even result in worse performance because it doesn't take into consideration the nature of the problem and the code that it tries to parallelize which might very well be innately a sequential code.

Parallelism in VPL is based on the data-parallelism paradigm [3]. In the current version the only construct that is auto-parallelized is the vector comprehension construct. In future version however additional constructs may be added as well (e.g. vector operators, loop constructs, etc…).

The auto-parallelization of vector comprehensions is done by computing concurrently independent expressions involved in their computation. A vector comprehension involves the computation of three kinds of expressions: the body expression, guard expressions, and generator's values-range expressions. In order to better understand how these expressions may be computed concurrently one can depict an evaluation tree where each level of the tree but the root level significances an expression that has

to be computed starting from the first qualifier to the last qualifier and ending with the body expression as the final level. The inner nodes may represent either a choice of value for a generator variable from its values range or a guard expression that has evaluated to true. The leaf nodes may represent either guard expressions that has evaluated to false or the final values of the generated components that are the results of evaluating the body expression in the different contexts. As an instance consider the following expression: {***x + y / x <- {1..3}, even?(x) , y <-{x..x+2}, odd?(y)}***. The corresponding evaluation tree for this expression is:



As you may see the computation of the value of each node is dependent solely on the values of its parent nodes and it's completely independent of its sibling nodes and their child nodes. Therefore we can compute (expand) concurrently the different nodes of the tree without fearing of any data dependency between the concurrent computations. After the tree is fully expanded i.e. all the concurrent computations are finished the final values of the components can be easily gathered in the right chronological order by visiting the tree leafs from left to right.

Note that although the semantics of the parallel execution of comprehensions doesn't precisely match those of the sequential execution they are basically equivalent. The only difference is essentially that instead of handling the values in the values-range of

the generator variable altogether that is, first filtering the inapplicable values using the ***filter*** function and then applying the body function for the filtered values altogether using the ***map*** function, each value is checked for applicability individually and if it is applicable it is processed independently from the others. This difference doesn't have any influence on the final result but only on the order in which the expressions are evaluated.

The VPL Platform

The VPL Platform is the implementation that I've built for the VPL language under the framework of the project. I've developed it in the C# 2.0 language for the Microsoft .Net Framework 2.0 using Visual Studio .Net 2005 Beta 2 IDE. I've used Microsoft .Net naming convention for naming types, methods, variables, namespaces, etc...(For more information about this convention see http://msdn.microsoft.com/library /default.asp?url=/library/en-us/cpgenref/html/cpconnamingguidelines.asp). I've built the VPL Platform from scratch without using any third party component or any foreign code

The VPL Platform consists of four major components: The Framework Library, The Compiler Program, The Emulator Program, and The Standard Library code modules.

The Framework library is the core of the whole system. It is a .Net library assembly (.dll) that provides the classes and methods required to compile, build, load and run VPL programs. The classes in the Framework are divided into four namespaces: the ***Intersoft.Vpl*** root namespace contains both the common classes such as the ***VplException*** class and the Code-DOM classes such as the ***VplExpression*** and the ***VplStatement*** classes. The ***Intesoft.Vpl.Compilation*** namespace contains the classes related to the compilation and building of VPL programs, such as the ***Parser*** and the ***ContextHandler*** classes. The ***Intersoft.Vpl.Execution*** namespace contains the classes required for executing VPL programs such as the ***ActivationRecord*** and the ***ExecutionEngine*** classes. The last namespace ***Intersoft.Vpl.Execution.Parallelism*** contains the classes that enable parallel execution of VPL programs such as the ***VplParallelInterpreter*** and the ***ParallelExecutionEngine*** classes. For a complete view of the framework classes and how they relate to each other see the diagrams in

appendix E. For further information about the specific role of each class and the technical details of its implementation see the comments in the source code. Note that some classes have the name-prefix ***Vpl*** while others don't. This is a matter of a convention; classes that are public must have this prefix in order to avoid confusion when used from other assemblies. Internal classes don't have this prefix for they are only used internally in the Framework assembly as helper classes.

The Compiler program is a .Net Application assembly (.exe) that provides a shell-based user interface for the compilation services offered by the framework. The input for the compiler is one or more code files. If multiple code files are specified they are compiled as if they were a single file who's the concatenation of all of them in the order in which they were specified. The output of the compiler is either the compile errors found during compilation or if no errors were found a package file that contains the compiled code in a compact binary format that is generated using the .Net serialization binary formatter.

The Emulator program is a .Net Application assembly (.exe) that provides a shell-based user interface for the execution services (both the sequential and parallel) offered by the framework. The input for the emulator program is a compiled package file and optionally a parameter that specifies whether to use the implicit parallelism optimization or not and if to use it how many worker threads to use for parallelization. The output of the emulator program is the exit value of the program if there is any and statistics information about execution. The statistic information for the sequential execution includes two measurements: the total physical time and the total workout which is the number of elements (expressions, statements) that has been processed during execution. The statistic information for the parallel execution includes two additional measurements: the first is the average and median workout of the worker threads and the second is the average and median idle times of the worker threads which can serve as estimation for the total time in which the worker threads sat idle.

The Standard Library code modules contain the code for the standard library types and methods. They can serve as a good demonstration for the language capabilities and for how different algorithms can be implemented in it (e.g. the ***Quicksort*** and the ***Mergesort*** methods of the ***System.Data.Vector*** module type).

For more detailed information on how to use and operate the compiler and emulator programs please refer to the readme file that comes along with the installation CD of the VPL platform.

Bibliography

1. D.P. Friedman, M. Wand, C.T. Haynes *Essentials of Programming Languages*, 2nd ed., M.I.T. 2001

2. Aho Alfred V., Sethi Ravi and Ullman Jeffrey D. *Compilers Principles, Techniques and Tools,* Addison Wesley, 1986

3. William Gropp, Ken Kennedy, Linda Torczon, Andy White, *The Sourcebook of Parallel Computing* , M. Kaufmann, 2002

4. O. Kammar, *The Foldr Operator*, The Open University, Programming Languages Workshop Project. 2004.

5. I. Hazan, *Introduction to Haskell*, The Open University, Programming Languages Workshop Project. 2002.

6. R. Bar-Yanai, *Monadic Programming in Haskell*, The Open University, Programming Languages Workshop Project. 2003.

7. Microsoft Corp, *The C# Language Specification*, Microsoft Press, 2002

Appendix A: Language EBNF Grammar

Lexical Grammar (Regular Expressions):

IDENTIFIER ::= [[a-z] [A-Z]] [[a-z] [A-Z] .]*

STRING ::= " .* "

COMMENT ::= [(/~ .* NEW_LINE) (/* .* */)]

NUMBER ::= [1-9]* ?(. [1-9]*)

CHAR ::= '.'

OPERATOR ::= [+-%|*/#!;?.,&~<>()[]{}$=]

KEYWORD ::= [(application) (namespace)(module)(var)(method)(return)(block)
(entry)(using)(if)(else)(true)(false)(import)(in)(global)(private)
(vector)(string)(scalar)(char)(boolean)(while)(public)]

Syntactic Grammar:

Global Namespace ::= <Namespace Body>

Namespace ::= "namespace" IDENTIFIER "{" <Namespace Body> "}"

Namespace Body ::= <Import Directive>* [<Type>,<Namespace>]*

Import Directive ::= "import" IDENTIFIER ("." IDENTIFIER)*

Type ::= <Module>

Appendix A: Language EBNF Grammar

Syntactic Grammar:

Module ::= "module" IDENTIFIER "{" <Member>* "}"

Member ::= ?["public","private"] <Method>

Method ::= "method" IDENTIFIER "("?(<Parameter>,("," <Parameter>)*) ")" "{" <Statement>* "}"

Parameter ::= ?("in") IDENTIFIER

Statement ::= <Return Statement>| <Variable Statement>| <Evaluation Statement>|<Block Statement> | <If Else Statement>|
<Assignment Statement> | <While Statement> | <Continue Statement>| <Break Statement>

Variable Statement ::= "var" IDENTIFIER ";"

Return Statement ::= "return" <Expression> ";"

Evaluation Statement ::= <Expression> ";"

Block Statement ::= "block" "{" <Statement>* "}"

If Else Statement ::= "if" "(" <Expression> ")" "{" <Statement>* "}" "[" "else" "{" <Statement>* "}"] "}"

Assignment Statement ::= <Expression> <- <Expression> ";"

While Statement ::= "while "(" <Expression> ")" "{" <Statement>* "}"

Continue Statement ::= "continue" ";"

Break Statement ::= "break" ";"

Expression ::= <Primary Expression>

Appendix A: Language EBNF Grammar

Primary Expression ::= <Logical And Expression> ("=>" <Logical And Expression>)*

Logical And Expression ::= <Logical Or Expression> ("and" <Logical Or Expression>)*

Logical Or Expression ::= <Equality Expression> ("or" <Equality Expression>)*

Eequality Expression ::= <Relational Expression> (["=","!="] <Relational Expression>)*

Relational Expression ::= <Additive Expression> (["">","<"] <Additive Expression>)*

Additive Expression ::= <Multiplicative Expression> (["+","-","+!"] <Multiplicative Expression>) *

Multiplicative Expression ::= <Power Expression> (["*","/","%","//"] <Power Expression>)*

Power Expression ::= <Prefix Expression> ("^" <Prefix Expression>)*

Prefix Expression ::= [(["scalar","string","char","boolean","vector"] ["?","!"]),"#","not","-","+"] <Postfix Expression>

Postfix Expression ::= <Atomic Expression> ("." IDENTIFIER, "[" <Expression> "]", "(" ?(<Expression> (, <Expression>)*) ")")*

Atomic Expression ::= <System Call Expression> | <Parenthesized Expression> | <Name Expression> | <Literal Expression>

System Call Expression ::= "$" IDENTIFIER "(" ?(<Expression> (, <Expression>)*) ")"

Parenthesized Expression :: "(" <Expression> ")"

Name Expression ::= IDENTIFIER

Literal Expression ::= <Vector Expression> | <String Literal Expression> | <Char Literal Expression> | <Scalar Literal Expression>

String Literal Expression ::= STRING

Scalar Literal Expression ::= NUMBER

Char Literal Expression ::= CHAR

Appendix A: Language EBNF Grammar

Vector Expression ::= "{" [?(<Expression> (, <Expression>)*) ,<Expression>..<Expression>, <Comprehension Expression>] "}"

Comprehension Expression ::= <Expression> "|" <Qualifier Expression> (, <Qualifier Expression>)*

Qualifier Expression ::= <Expression> ? ("<-" <Expression>)

Appendix B: Operator Expressions Table

Op	Left Operand	Right Operand	Result	Description	Example	
Conditional Expressions						
&&	Boolean	Boolean	Boolean	Conditional and	true && false	false
||	Boolean	Boolean	Boolean	Conditional or	true || false	true
!		Boolean	Boolean	Conditional not	!false	true
Arithmetical Expressions						
+		Scalar	Scalar	Positive value	+33	33
+	Scalar	Scalar	Scalar	Arithmetical addition	5+3	8
+	Vector	Vector	Vector	Concatenate vectors	{2,3}+{4,5}	{2,3,4,5}
+	String	String	String	Concatenate strings	"Hey" + "!!"	"Hey!!"
-	Scalar	Scalar	Scalar	Arithmetical subtraction	9 – 5	4
-		Scalar	Scalar	Negation	-33	-33
*	Scalar	Scalar	Scalar	Arithmetical multiplication	6 * 4	24
*	Positive Integer Scalar	Vector	Vector	Concatenates the vector with itself x times	2 * {1,2}	{1,2,1,2}
*	Vector	Positive Integer Scalar	Vector	"	{1,2} * 2	{1,2,1,2}
/	Scalar	Non Zero Scalar	Scalar	Arithmetical division	9/2	4.5
/	Vector	Integer Scalar	Vector	Divides the vector into sub vectors of size x	{1,2,3,4,5}/2	{{1,2},{3,4},{5}}
%	Scalar	Non Zero Scalar	Scalar	Arithmetical modulo	9%2	1
%	Vector	Positive Integer Scalar	Vector	Returns the remainder in a vector division	{1,2,3,4,5}%2	{5}
//	Scalar	Scalar	Scalar	Arithmetical division without remainder	9//2	4
//	Vector	Scalar	Vector	Vector division without remainder	{1,2,3,45}//2	{{1,2},{3,4}}
Relational Expressions						
>	Scalar	Scalar	Boolean	Arithmetical greater than	10>2	true
>	String	String	Boolean	Alphabetical greater than	"dog">"cat"	true
>	Vector	Vector	Boolean	Contains	{4,1,2,3}>{1,2}	True
=	Scalar	Scalar	Boolean	Equal to	3=9	False
=	String	String	Boolean	Equal to	"hello"="bye"	False
=	Vector	Vector	Boolean	Equal to	{1,2,3}={1,2,3}	True

Appendix B: Operator Expressions Table

Op	Left Operand	Right Operand	Result	Description	Example	
Relational Expressions						
<	Scalar	Scalar	Boolean	Arithmetical smaller than	10<2	False
<	String	String	Boolean	Alphabetical smaller than	"dog"<"cat"	False
<	Vector	Vector	Boolean	Contained in	{4,1,2,3}<{1,2}	false
<=	Scalar	Scalar	Boolean	Arithmetical smaller than or equal to	3<=2	false
<=	String	String	Boolean	Alphabetical smaller than or equal to	"hello"<="hello"	true
<=	Vector	Vector	Boolean	Contained in or equal to	{1,2,3}<={1,2,3}	true
!=	Scalar	Scalar	Boolean	Not equal to	5!=4	true
!=	String	String	Boolean	Not equal to	"dog"!="cat"	true
!=	Vector	Vector	Boolean	Not equal to	{1,2,3}!={2,4}	true
>=	Scalar	Scalar	Boolean	Arithmetical greater than or equal to	9>=9	true
>=	String	String	Boolean	Alphabetical greater than or equal to	"dog">="dog	true
>=	Vector	Vector	Boolean	Contains or equal to	{1,2,3}<={1,2,3}	true
Other Expressions						
\	Any Value	Vector	Boolean	Member of	5\{1,2,3,5}	True
#		Vector	Scalar	Size of vector	#{1,5,10}	3

Appendix C: Type Conversions Table

Explicit Type Conversions:

Source	Target	Result Description
String	Vector	A Vector of the string characters
Scalar	Char	A char represented by the scalar ASCII
Scalar	Boolean	True if scalar is not zero otherwise false

Implicit Type Conversions:

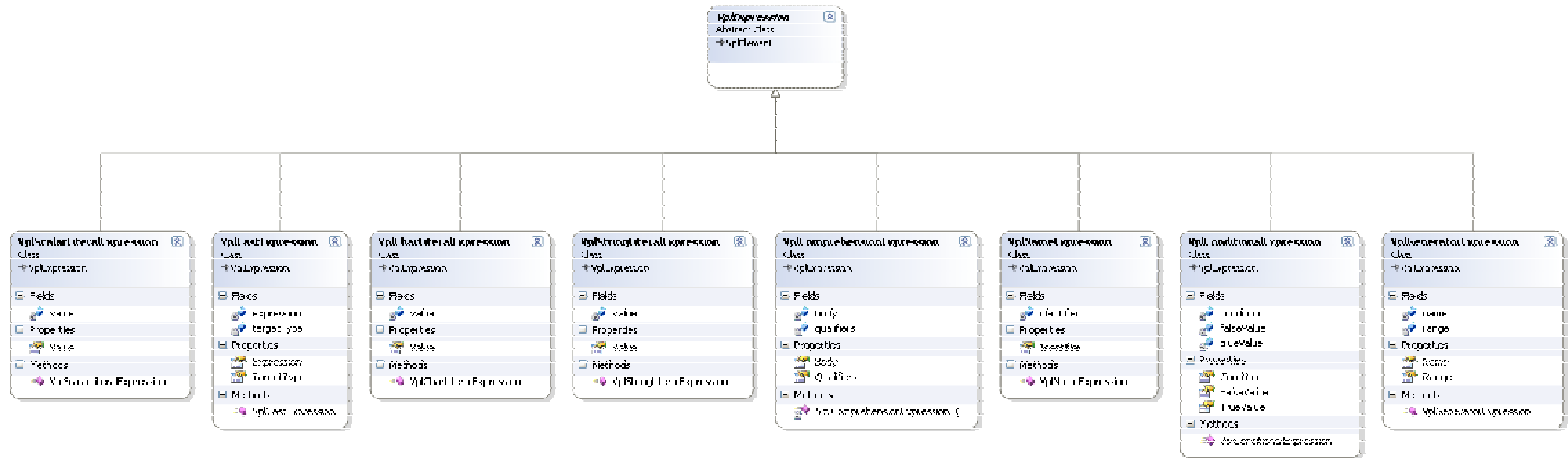
Source	Target	Result Description
Char	Scalar	The ASCII Code Number
Char	String	A String with one char
Boolean	Scalar	If true then 1 otherwise 0

Appendix D: System Services Table

Service Name	Parameters	Result	Description
STD_WRITE	Object:data	Nothing	Writes the string representation of the ***object*** to the standard output (Console)
STD_READ		Char	Reads a character from the standard input (Keyboard)
ERROR	text:string	Nothing	Throws a runtime error with the given ***text*** as the error message and aborts execution
ASSERT	cond:boolean	Nothing	If ***cond*** is true do nothing otherwise throws an assert error and prints the call stack
EXIT	Object:data	Nothing	Terminates the execution of the application and returns ***object*** as the exit value

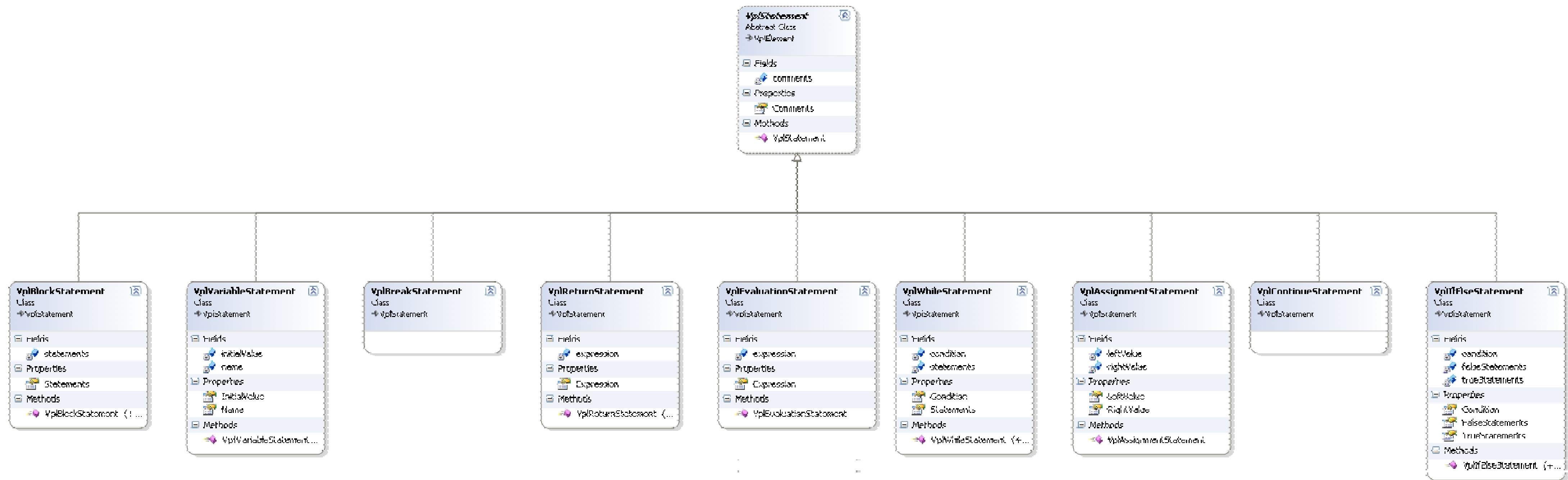
Appendix E: Framework Classes Diagrams

Appendix E: Framework Classes Diagrams



Appendix E: Framework Classes Diagrams

Appendix E: Framework Classes Diagrams



Appendix E: Framework Classes Diagrams

Appendix E: Framework Classes Diagrams