A Virtual Machine Framework for Domain Specific Languages

David Fick
dfick@grintek.com

Derrick G. Kourie
dkourie@cs.up.ac.za

Bruce W. Watson
bwatson@cs.up.ac.za

Espresso Research Group
University of Pretoria
South Africa, Pretoria, Gauteng

ABSTRACT

A generic approach to constructing a virtual machine for a DSL in C# is studied. It proposes a generic, object-oriented framework, in which to build the virtual machine, using an abstract instruction class and an abstract environment class. They can be extended to provide a concrete layer whose interface constitutes the set of instructions of a DSL. The framework allows for the generation of a variety of virtual machines each supporting a particular DSL. Comparative performance results in relation to other DSL implementations are also provided.

Keywords

virtual machine, domain-specific language, instruction set, environment, abstract class, generic framework.

1. INTRODUCTION

Domain specific languages (DSLs) have been discussed and used in many contexts. (See, for example, [Arn95] and [Deu98].) In this paper the design and implementation of a VM Framework for DSLs is investigated, using .NET. Two other approaches for constructing a DSL are also briefly examined. For all three approaches, time of execution is examined and timed points are declared.

The Shlaer-Mellor (SM) software construction method has been adopted. A fundamental difference between SM and other methods is the identification of separate subject matters, called *domains*. An SM domain is a separate real, hypothetical, or abstract world inhabited by a distinct set of classes that behave according to rules and policies characteristic of the domain [Shl92a]. The VM Framework is layered on top of an existing domain. As a programming language construct, a domain is simply represented as a namespace. The namespace forms a home for related classes and these classes facilitate the semantics of the DSL.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

.NET Technologies'2005 *conference proceedings,*
ISBN : 2/: 8; 65/23/3
Copyright UNION Agency – Science Press, Plzen, Czech Republic

The VM Framework outlined in this study is an extension to the typical VM, in that it defines a VM with an empty instruction set whose environments and instructions can later be extended.

2. FRAMEWORK DESIGN

The VM Framework provides the basic functionality of a typical VM, including an Intermediate Representation (IR) program loader, a program counter, internal temporary values, and conditions on which to build branching instructions. A proxy object is provided through which to start up and configure an instance of a VM. 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 each discussed in the following subsections, and is shown in figure 1.

The `EVM` Class

The `EVM` class (Extendable Virtual Machine) is the proxy class. Once instantiated, the object represents an instance of a configurable VM, with an empty instruction set and no environment. A specific configuration can then be applied to the VM instance. When an IR program is executed, the VM will invoke the correct `Inst` instance created at load time, defined in the configuration file. The `EVM` class also encapsulates the internal temporary values in the `temps` hash table. 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

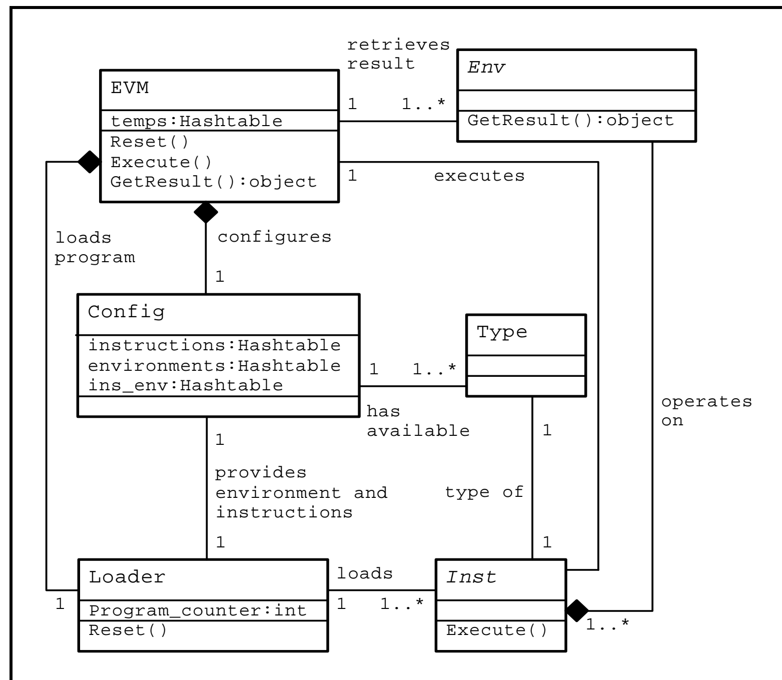
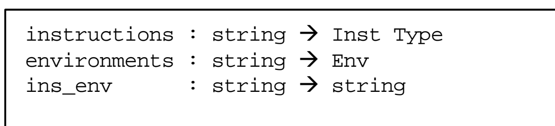


Figure 1. Information model of the VM Framework

loop in its `Execute()` method, that iterates through each instruction stored by the `Loader`. The very next instruction to be executed is first fetched, and then the `Execute()` method of its class is invoked. 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.

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`, and are defined in Figure 2 below.



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

Figure 2. Configuration mappings

The `instructions` mapping maps the string name of an instruction, to an `Inst` type. Derived instances of the `Inst` class are only created upon program loading. The `environments` mapping, maps the string name of an environment, to an instance of `Env`. As indicated below, the `Env` instance will typically encapsulate some Abstract Data Type (ADT) such as a runtime stack. The last mapping, `ins_env`, maps the name of an instruction to the name of the environment that the instruction is to use. The name of the environment is looked up in the `environments` mapping, and the actual instantiated environment is retrieved, and later accessed by the instruction during the execution of the loaded IR program.

The `Loader` Class

The `Loader` class encapsulates a loaded IR program and the program counter. The loaded program is an array of `Inst` instances, for each instruction of the program. The `Loader` class also maps labels to program counter values. The mapping is updated with a program counter entry for each label in the program. When a branch occurs, the index of the next instruction can be retrieved using the mapping. The parser has the string name of the instruction and uses the mapping defined in the `Config` class to retrieve the `Inst` type that is used to create the `Inst` instance. Thus when a program is fully loaded, the array will contain instances of `Inst`, each `Inst` encapsulating its own operands ready for execution, and the program counter is reset to the beginning of the array.

The `Inst` Abstract Class

The abstract `Inst` class encapsulates a reference to a `Env`. This will be the environment updated by the instruction during the execution of the loaded IR program. Note that it is only a reference and other instructions will have a reference to the same `Env` instance. The `Inst` class does not define how the updates are performed, and instead provides an abstract `Execute()` method, that further extensions to the instruction are obligated to override. While there are still instructions to be executed by the loaded program, the `Execute()` method is called for each instruction. When the program counter has run through each instruction instance, the program has completed execution and the result of the execution can be retrieved.

The `Env` Abstract Class

The abstract `Env` class encapsulates some ADT, or even a number of ADT's that form the central data storage mechanism for the language. 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 extensions of `Env` are obligated to override.

3. ENVIRONMENTS

The purpose of the abstract `Env` class is to have an ADT that is updated during runtime, and that is appropriate, or convenient for processing the semantics of the language. 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. 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 bitmap image type. Since framework users will be aware of the data type 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 3. It encapsulated a real-valued stack, and returns the last entry on the stack. If all operations on the stack are consistent, there should be only one remaining value on the stack, which is the result of evaluating the expression.

```
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 3. Example `EnvExp` class

4. INSTRUCTIONS

There are five classes of instructions, each represented by an abstract class extending `Inst`. The user creates their own instruction by extending one of the classes of the abstract instructions provided. Each instruction will take at most one operand. The five classes of instructions are defined in terms of their operand types. Instructions written in the source IR program can be labeled, if they are targeted by any branching instructions. The token and grammar definition for parsing IR program code is shown in Figure 4.

```
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 4. IR Token definitions and grammar

Instructions with No Operands

A Boolean property of this class, `LoadProgramCounter`, in Figure 5, is an

option to recall the last saved program counter. This allows the creation of instructions that return from a branch into a subroutine.

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

Figure 5. The `Inst_OpCode` abstract class

As an example, the `Add` instruction is presented in Figure 6, as used in a simple expression language. The instruction `Add` simply pops the two topmost operands off a stack and pushes the sum back on.

Instructions with a Branch 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 to 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. The second property, `SaveProgramCounter`, dictates whether the program counter should be saved for a corresponding return call into a subroutine. The complete class is shown in Figure 7.

Instructions with a Temporary

Internally, temporaries are implemented with a `Hashtable` that map temporary names (ID's) to object references. They are akin to conventional registers, but a temporary can be treated as any object type as illustrated in Figure 8. Instructions have full access to a temporary. The instruction can modify the temporary by typecasting the object to the required usable type.

Instructions with a String Parameter

Instructions with string parameters are useful in string processing applications such as those that deal with regular expressions. This class, shown in Figure 9, exists to provide the means to parse a string defined in the source IR program and to store it in the variable `str`.

Instructions with a Number Parameter

Similarly to the above string parameter instruction, instructions with number parameters exist to provide the means to parse numbers defined in the source IR program, or to facilitate instructions that provide any intermediate arithmetic calculation. Real or integer numbers can be parsed. However, internally they are treated as `double` values.

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

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

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

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

      r = d1 + d2;

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

Figure 6. The `Add` instruction

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

   protected bool
   BranchCond = false;

   protected bool
   SaveProgramCounter = false;
}
```

Figure 7. `Inst_OpCode_Br`

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

Figure 8. `Inst_OpCode_ID`

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

Figure 9. `Inst_OpCode_Str`

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

Figure 10. `Inst_OpCode_Num`

5. EXTENDING THE FRAMEWORK

The Configuration File

Before an instantiated VM can execute instructions in a loaded IR program, the VM needs to be configured as a specific VM type. This is achieved through a configuration file that is initially loaded. Once the VM has been configured, an IR program

can be loaded and executed. The configuration file specifies the name of the class used as an environment, as well as the names of all instruction classes, both stored as .NET DLL's. The configuration file will give the complete instruction set for a particular VM. Figure 11 gives an example configuration file for an expression language.

The keyword `environment` is followed by an environment class name, and one environment instance will be instantiated for that class. When instructions are instantiated at program load time, the environment that will be used by the instruction is named after the `using` keyword.

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. A ray-tracer language needs to render a scene onto a bitmap, but may also require a means to perform numeric calculations. Thus the `EnvExp` environment of the expression language can be extended with two extra data structures; a scene and a bitmap, giving rise to an `EnvRT` environment suitable for a ray-tracer.

Extending the Instruction Sets

Instructions are extended from one of the five instruction classes mentioned earlier, to a set of concrete instruction classes instantiated at load time. Extending instruction *sets* with environments that are subclasses of each other, makes for a scalable framework in which to design a tailored VM for a DSL. The ray-tracer language serves as an example. The `EnvRT` environment is a subclass of `EnvExp`, so any one of the instructions operating on an `EnvExp`, can also operate on a `EnvRT`, as illustrated in the configuration file for the ray-tracer language, depicted in Figure 12.

```
(* 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 11. Example configuration file to setup the VM for a small expression language

```
(* 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. *)
LookAt    using EnvRT
Specular  using EnvRT
Diffuse   using EnvRT
Reflect   using EnvRT
Translate using EnvRT
Quad      using EnvRT
```

Figure 12. Configuration file to setup a ray-tracer language borrowing some instructions from an expression language

6. 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. The example DSL in this section was built using LG to define the tokens for the lexer, and PG to define the grammar for the parser. Both tools bear a familiar syntax to most commonly used industry tools. The translation of a small, functional expression language can be intuitively understood by the following illustrative example. The program in Figure 13 evaluates the expression

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

in a functional manner. The token and grammar definitions for each of the five instruction types are given in section 5, and the translated IR code for this program is shown in Figure 14 as 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 13. Programmatic representation of the summation expression (1)

The generated IR code performs operations on a runtime stack. This stack is indeed defined as part

of the environment `EnvExp` discussed earlier, and once the IR code has completed execution, the only remaining value on the stack will be the result of the expression.

```
      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 14. Translated IR program of the summation expression (1)

7. COMPARATIVE RESULTS

Comparative performance results were done between three different DSL implementations: an interpreter, a hardcoded VM and the VM Framework. Two time intervals were compared for each implementation; compiling DSL source code to IR (SRC→IR), and executing the IR to observe the semantics (IR→SEM). The total time (SRC→SEM) is also calculated. The measured time is in units of 100ns. Only the total time (SRC→SEM) is relevant for the interpreter. The hardcoded 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, a ray-tracer language, used to define geometric objects to be rendered onto a scene.

From the performance results in Figure 15, it can be seen that using some of the reflection properties of .NET does not necessarily impede the IR program's execution speed, and in this case it is actually shown to perform better than its hardcoded counterpart. Naturally, the interpreter is quickest to deliver observable results, however, it will suffer from a lack of scalability. The hardcoded VM will suffer less from scalability problems, as it is easier to add new instructions as part of the VM core. The VM Framework treats environments, and instructions that access these environments, as separate external libraries, or DLL's, and they do not form part of the VM Framework's core execution unit. Rather, these DLL's are configured together as a set of building blocks to yield a customized VM for a particular DSL. Furthermore, the VM framework easily accommodates a scaling up of the DSL with new constructs as the need arises.

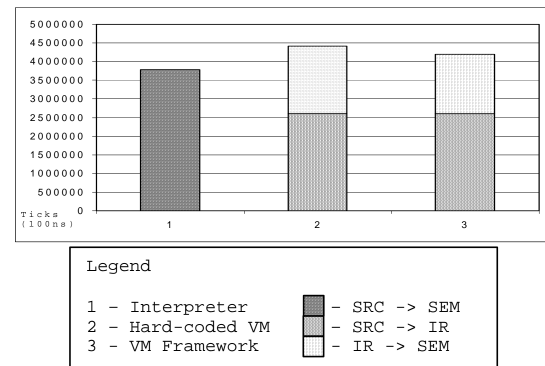


Figure 15. Comparative performance results of three types of DSL implementations

8. CONCLUSION

This paper described a framework that allows rapid development of DSL's, with emphasis on language scalability. The VM Framework also relies on certain reflective constructs of .NET to configure an instantiated VM at runtime, and .NET DLL's are used extensively to aid in scalability 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 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 framework will aid in the development lifecycle of the DSL.

REFERENCES

[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

Deu98] Deursen, A. v., and Klint, P. Little Languages: Little Maintenance? *Journal of Software Maintenance*, volume 10, 1998

[Shl92a] Shlaer, S., and Mellor, S. J. *Object Lifecycles: Modeling the World in States.* Yourdon Press, P. T. R. Prentice Hall, 1992

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