Virtual Machines and Abstract Compilers

Towards a Compiler Pattern Language

Julio García-Martín Miguel Sutil-Martín
Universidad Politécnica de Madrid[1].

Abstract

Because of the increasing gap between modern high-level programming languages and existing hardware, it has often become necessary to introduce intermediate languages and to build virtual machines on top of the hardware. This paper describes the VIRTUAL MACHINE and the ABSTRACT COMPILER patterns, a proposal that captures the essential features underlying the compilation processes based on staging transformation of virtual machines. Having as a support the VIRTUAL MACHINE pattern, we set out the task of compiling as a stepwise-refinement process guided by the ABSTRACT COMPILER pattern, so that the piecemeal acquisition of high-performance properties is posed in terms of relationships between intermediate virtual machines. Each phase during the compilation makes explicit some new features that are added to the global compilation process. The equivalence between one machine and the next is preserved though out the process.

1. Introduction

As is well known, the Java Virtual Machine (JVM) is an abstract software-based machine that can operate over different microprocessor machines (i.e., hardware independent). Designers of a JVM must comply with the specification of the JVM and make the necessary bridge from the JVM virtual scene into concrete operating systems and microprocessors. This behind-the-scene bridge allows the software developers to *"Write Once, Run AnyWhere"* [1] because the JVM must behave the same regardless of the underlying microprocessor according to standard specifications of JVM [17].

Though a big success up to the moment, the usage of virtual machines is far from being a new issue. Long before the boom of Java, virtual machines had been effectively used as intermediate or low-level architectures suitable for supporting serious implementations of a wide variety of programming languages, including imperative, declarative (i.e., functional and logic[2]) and object-oriented programming languages [5]. The virtual machines provide several desirable features such as portability, code optimizations, and native machine code generation. On the other hand, their simple structure makes them suitable for analysis and experimentation. Unfortunately, the structure of architectures called virtual machines varies widely, depending in part on the language being described and the representation of source programs as data.

The gap between modern high-level programming languages and existing hardware makes necessary to introduce intermediate languages running on virtual machines. However, the gap could be often so large that it is either hard to see how the source language relates to the intermediate language or, alternatively, how the intermediate language relates to the hardware. The Figure 1 centers the question showing a snapshot of the common compilation/execution scenario for a Java program.

---

[1] LSIIS Department, Facultad de informática, Campus de Montegancedo, Boadilla del Monte, 28660 Madrid, Spain. Email: juliog@fi.upm.es
[2] Prolog [9] or SML [8] provide good examples of high-performance programming languages implemented by virtual machines.

1. Firstly, the process requires the design of an intermediate-level machine (JVM) from the Java operational semantics, and a Java compiler to translates Java code into JVM code,
2. Next, the JVM architecture must be mapped into a concrete hardware machine (e.g., Intel machine) and JVM instructions interpreted in terms of hardware instructions.

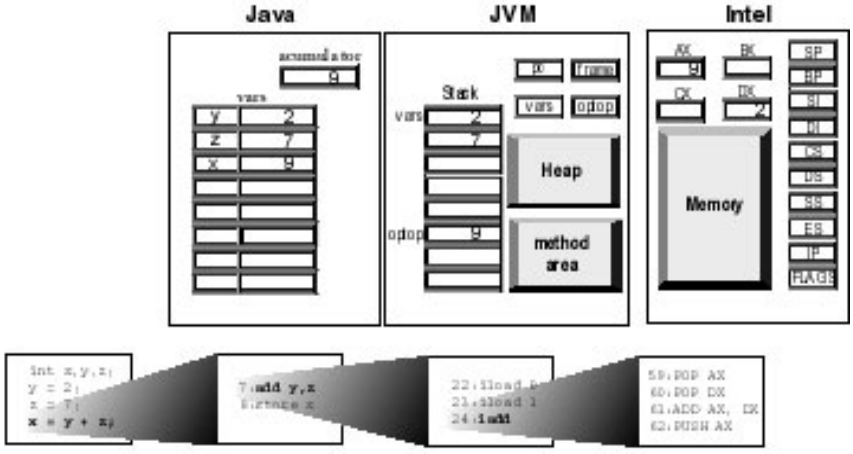

Figure 1. Compilation/Execution roadmap for a Java program

The main objective of the paper is to state a pattern language for the design of compiler back-ends[3] for programming languages, based on taking an operational semantics for a source language and defining an intermediate-level target language and a compiler that translates the source language to this target language. Thus, compiling can be planned as a stepwise-refinement process, so that the piecemeal acquisition of high-performance properties is posed in terms of relationships of intermediate virtual machines. Each compilation step makes explicit some new features and changes that are added to the global compilation process. The process contributes to getting a more abstract and systematic way of constructing compilers. Furthermore, it improves the understanding of the process (compilation) and simplifies the task of refining and reusing previous designs. It is likely that a (prototype) compiler can be extracted more or less automatically as a side effect of the design of the virtual machine.

2. The Patterns Overview

This paper presents the VIRTUAL MACHINE and the ABSTRACT COMPILER patterns, a germ proposal to set up a pattern language helps developers to obtain abstract compilers for programming

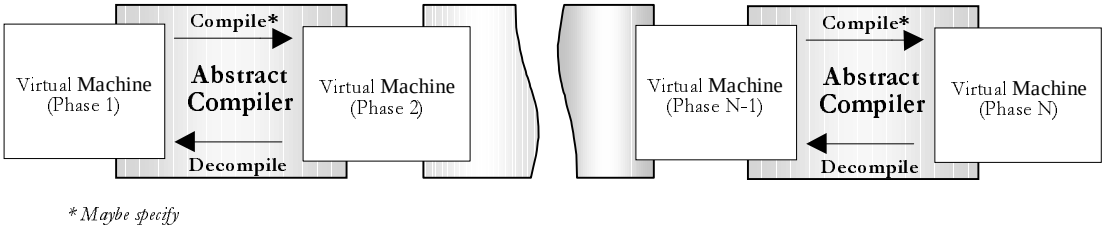

Figure 2. The VIRTUAL MACHINE & ABSTRACT COMPILER patterns interaction

languages, based on the virtual machine technology. The Figure 2 depicts the relationships between the two patterns.

[3] Thus, this work does not tackle the problem of converting program text into a token stream, lexical analysis, parsing, etc.

A short description of the patterns is presented now:

Name	Description
Virtual Machine	How to define a common design to capture the essential features addressed by virtual machines?
Abstract Compiler	Suppose we have described a compilation step, such that: 1. We have obtained a virtual machine from a previous one (by stepwise refinement), and 2. We have defined a translation process from the origin language onto the target one. How to know (or test) if the compilation step is correct (i.e., it has the effect that we expected!)?

3. The Virtual Machine Pattern

Intent

Define a common template to design virtual machines. The pattern captures the essential features underlying a virtual machine as well as encapsulates them in separated loosely-coupled components. Furthermore, the proposal encapsulates how components of a virtual machine interact.

Also known as

Abstract Machine, Abstract State Machine.

Applicability

Use the VIRTUAL-MACHINE pattern in any of the following situations:

- *When we want to specify a virtual machine.* The pattern provides a common skeleton to write high-level specifications, allowing the programmer to pay more attention to specifying machine's components instead of the whole machine. The specifications can be formal [3] or informal [17].
- *When we want to compile a language using the virtual machine based technology.* Virtual machines provided a well-suited framework to describe compiling processes by stepwise-refinement of intermediate languages (i.e., development "*by prototyping*"). In this process, relationships between intermediate languages can be expressed in terms of the relationships of related virtual machines (see the ABSTRACT COMPILER pattern).
- *When we want to give a precise description for a compiling process by using the virtual machine technology.* As a consequence of combining the two situations previously commented (see the ABSTRACT COMPILER pattern).
- *When we want to test different instruction semantics in a virtual machine.* It might be possible to define different semantics for the same machine instruction.
- *When we want to test different instruction sets in a virtual machine.* It might be possible to define different instruction sets for the same virtual machine definition
- *When we want to incorporate visualization and debugging facilities to our virtual machine.* The pattern's

participants are highly de-coupled. Therefore, the incorporation of new facilities will not introduce obtrusive effects in the pattern.

Structure

The structure of the VIRTUAL-MACHINE pattern is shown on Figure 3.

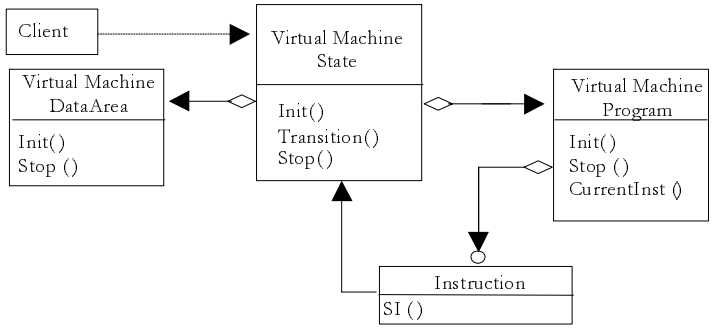

Figure 3. The VIRTUAL MACHINE pattern (structure).

Participants

A virtual machine may be defined as the union of two parts: (i) the *static* part, consisting of the components related to the state, (ii) the *dynamic* part, which is associated with the machine's behavior. The VIRTUAL MACHINE pattern encapsulates how static and dynamic components locate each of these components into different participants.

The *state* of a virtual machine consists of the following components:

1. *DataArea*: It declares an abstract interface for any data area configuration of the machine. It declares two abstract operations:
   - The *Init* operation, to determine the initial configuration for the data area,
   - The *Stop* operation, to determine if the machine has achieved the final stage. If this final stage does not depend on any data area configuration, then Stop returns *true*.
2. *Concrete DataArea*: It defines a concrete data area object. The concrete data area may be a simple or a complex object structure (COMPOSITE). It implements the DataArea interface.
3. *Program*: It declares a common interface for any assembler program. It includes, as an attribute, a collection of Instructions. Besides, it declares four abstract operations:
   - The *Init* operation, to determine the initial configuration for assembler program,
   - The *Stop* operation, to determine if the machine has achieved its final stage. If this final stage does not depend on any program configuration, then the Stop returns *true*.
   - The *LoadProg* operation is in charge of constructing the assembler instructions by reading the target code from an input stream and translating textual representations into machine's instruction.
   - The *CurrentInst* operation returns the instruction to be executed by the virtual machine.

4. *Concrete Program*: It defines a concrete program object, it is, a concrete instruction set and the program counters needed. It has to implement the Program dealing with the definition of the operations Init, Stop and CurrentInst.

On the other hand, the *behavior* is provided by some operations that operate over the static components. These operations are part of the state definition and they will be responsible the different states the virtual machine achieves during the execution of a given program. These operations are described below:

1. *State*: It coordinates the interactions between the DataArea and the Program (such as a MEDIATOR pattern [4]). It determines the machine's state. There are three different stages for the *State*: *initial, executing* and *final*.
    - The *Init* operation determines the initial state of the machine, just at the beginning of execution.
    - The *Stop* operation determines if the machine has achieved its final state.
    - The *Transition* operation performs the program execution. So, the instruction pointed by the program counter is executed. Transition starts the machine execution and continuous until the final state has been achieved.
2. *Instruction*: It defines an abstract interface for a machine's instruction.
    - The *SI* operation (semantic function) determines how the configuration of the machine evolves after the execution of an instruction. The semantic function must be defined for each instruction in the instruction set.
3. *Concrete-Instruction*: It defines a concrete instruction for a concrete virtual machine. Concrete instructions are related to concrete data areas and concrete programs.

Collaborations

Three different scenarios model the three different states a virtual machine may reach during its execution: *initialization*, *transition* and *ending*. Figures 4, 5 and 6 sketches these scenarios.

Scenario 1: Initialization

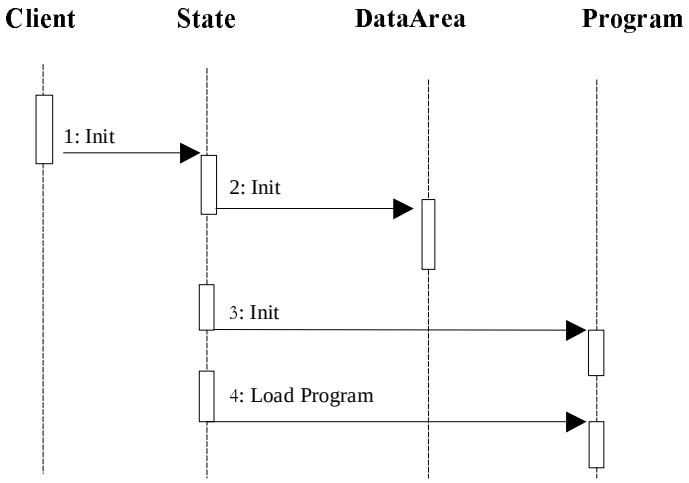

Figure 4. VIRTUAL-MACHINE pattern (collaborations I)

At this step, the virtual machine is initialized. As a result, both the DataArea and Program and are

initialized. The initialization of the Program is carried out by the Init operation, and involves setting components with the initial values (i.e., program counters, array of instructions, etc.). Next, the LoadProgram operation is executed, which loads the assembler program (i.e., to translate the text representation of assembler instructions from the inputStream into instruction objects of the program). On the other hand, components in the DataArea are initialized by the Init operation (i.e., the data registers or control registers, etc.).

Scenario 2: *Transition*

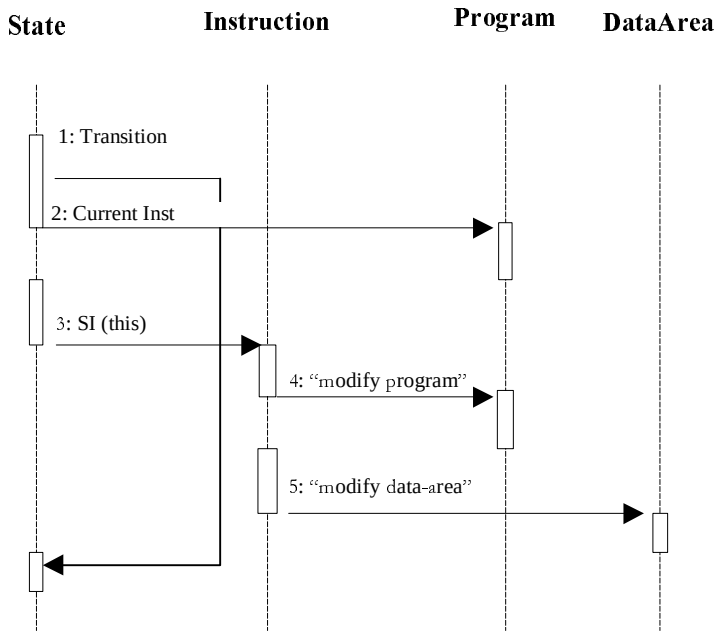

Figure 5. VIRTUAL-MACHINE pattern (collaborations II)

As said before, the Transition operation performs the machine execution. To do this, the operation Transition requests from the program the instruction to execute that is pointed to by the program counter (`CurrentInst` operation). The execution of the machine consists of executing instructions until the ending condition. Each machine's instruction is responsible of defining its own semantics. So, each instruction provides the `SI` operation, whose execution modifies the machine's state (i.e., the program and/or the data area). Therefore, depending on the instruction the data area, the program (or both) will be modified. Each instruction semantic determines the order these modifications are done.

Scenario 3: Ending

The ending stage for the machine's execution is achieved when one of its two components (or both) reach the ending condition (e.g., it is executed a concrete stop instruction, there is a data area overflow, etc.). The Stop operation on the State asks to the corresponding Stop operations on the DataArea and the Program, and combines their result. If the ending condition does not depend on the DataArea (or the Program), then its Stop operation must return true as the result.

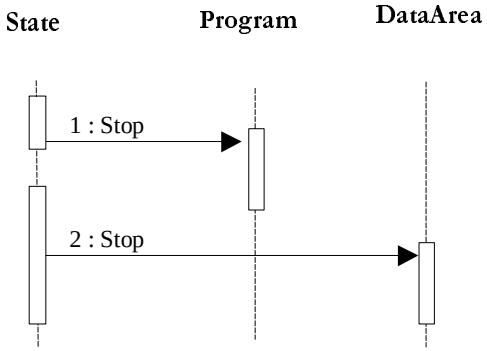

Figure 6. VIRTUAL-MACHINE pattern (collaborations III)

Consequences

The VIRTUAL MACHINE pattern presents the following advantages:

- *Provide a general framework to develop virtual machines and abstract compilers.* The pattern allows the high-level design of virtual machines, not imposing constraints about the sort of program languages to compile to. Besides, the success of virtual machines also lies in their ability to obtain a good performance by execution a highly specialized code. (See also consequences of ABSTRACT COMPILER pattern).
- *Test different semantics for the same instruction, without changing the data area and the program.*
- *Promote a methodology for a most systematic development of virtual machines and abstract compilers.* By separating data area, program and instruction semantics we introduce some design constraints that force the user to follow a most systematic approach. (See also consequences ABSTRACT COMPILER pattern).
- *Offer a higher-level degree of encapsulation a re-use of virtual machine based developments.* Semantics for virtual machine instructions are encapsulated as classes.
- *Reuse previous virtual machine designs in new applications.*

The Virtual Machine presents the following disadvantages:

- *Yield to obtain inefficient implementations.*
- *Provoke a communication overhead between State and Instruction.*

Implementation

Consider the following implementation issues:

1. *Concrete DataArea might be a complex object composition.* Follow the same implementation issues as in for the COMPOSITE pattern [4]
2. *Defining the DataArea, the Program* and *Instruction as interfaces.* For example, following the Java conventions:

```
public interface Instruction
{
     public void SI (State state);
     public String Name ();
     public int NumArguments ();
     public String ToString ();
     public void Process (String args [])
```

```
}


public interface Program
{
     public Instruction CurrentInst ();
     public void Init ();
     public boolean Stop ();
     public void LoadProgram (Stream assemblerCode);
}

public interface DataArea
{
     public void Init ();
     public boolean Stop ();
}
```

The Instruction provides methods to determine information about instructions (i.e., Name, NumArguments), a method defining the instruction semantics (SI) and the method Process that compiles the arguments of the assembler instruction. The State is an argument of *SI* operation and it is only used during this instruction.

The Program defines the common interface for any assembler program in the virtual machine. It defines an operation CurrentInst to access the instruction is currently being executed. The LoadProgram method is in charge of building the instructions of the program from the assembler code file.

3. *DataArea, Program and Instruction as template parameters*. In a C++ like syntax, as follows:

```
class Instruction
{ .. }

template
class Program <class Instruction>
{ .. }

class DataArea
     { .. }

template
class State < class DataArea,
class Program <Instruction>>
{ .. }
                                                    }
```

4. *Omitting the State class*. It is similar to the case of the MEDIATOR pattern [4].
5. *Primitive operations*. Some operations defined in DataArea, Program and Instruction are primitive. Then, they must be overridden. For example, they could be declared as pure virtual (in C++ conventions) or as part of an interface (Java conventions). The operations *Init, Transition* and *Stop* in the State must be never overridden.

Sample Code

The following sample code shows Java implementations for some parts of the VIRTUAL-MACHINE pattern.

1. Firstly, we define the interfaces corresponding to DataArea, Program and Instruction (see the Implementation section)
2. An alternative implementation of the Program may be done as an abstract class. In this case, some extra functionality is provided (e.g., a program counter P and the set of instructions).

```
import Instruction;
import InstructionSet;
```

```
public abstract class Program
{
    public int P;
    private Vector program; // A container of AbstractInstruction
    private InstructionSet instSet;

    public void Init ()
    {
           P = <<init program address>>;
           Program = new Vector ();
    }
    public void LoadProgram (Stream assemblerCode)
    {
           << load instructions from the assembler code >>
    }
    public Instruction CurrentInst ()
    {
           return program.Get (P);
    }
    public void Next ()
    {
           P++;
    }
    abstract public boolean Stop ();
}
```

3. The implementation of the State.

```
import DataArea;
import Program;
import Instruction

public class State
{
    private DataArea da;
    private Program prog;

    // ...
    public void Init (Stream s)
    {
           da.Init ();
           prog.Init ();
           prog.LoadProgram (s);
    }
    public  boolean Stop ()
    {
           return prog.Stop () && da.Stop();
    }
    public void Transition ()
    {
           while (!Stop ())
           {
              prog.CurrentInst().SI (this);
           }
    }
}
```

4. The creation of a concrete State (say an *M* machine) is done by following the next sequence of actions.

```
    import VirtualMachine;
    import WAM_Factory;
    import WAM_InstructionSet;

    class M_Client
    {

           static public void main (String args []) throws IOException
           {
              FileInputStream targetCode = new FileInputStream (arg [0]);
```

```
            State machine = new State ();
            machine.Init (targetCode);
            machine.Transition ();
        }
}
```

5. Finally, we describe some examples of instructions in the M-machine's instruction set. Let's consider how each of the following instructions implements its semantics (how the M-machine evolves) by modifying the M_State (i.e., the M_DataArea and/or the M_Program).

   a) The following Java code is associated with the instruction *I* of the *M* machine. The execution of this instruction (its semantics) provokes some changes in the M_DataArea and the M_Program.

```
public class I implements Instruction
{
        I ()
        {}
        public String Name ()
        {
            return "I";
        }
        public int NumArguments ()
        {
            return 1;
        }
        public String ToString ()
        {
            return Name() + " " + numVars;
        }
        public void SI (State state)
        {
            M_DataArea M_da = (M_DataArea) state.DataArea();
            M_Program  M_prog = (M_Program) state.Program();

            // Implementation of I semantics
             ...

        }
        public void Process (String args[] )
        {
           // Proces the text representation of the instruction
           // arguments into an int value.
              numVars = Integer.valueOf (args [0]).intValue();
        }
}
```

   b) Let us consider now, a new version of the above definiton of the *I* instruction, in order to support visualizing facilities. For this reason, we redefine the SI operation as follows:

```
public class ViewI extends Call
{
        public void SI (State state)
        {
           super.SI (state);  // Execute the Call semantics
           state.notify ();   // Notify the change and redraw
        }

}
```

4. The Abstract Compiler Pattern

Intent

Provide a common framework to design abstract compilers by stepwise refinement of virtual machines. The task of compiling is planned as a piecemeal acquisition of high-performance properties poses in terms of relationships between intermediate virtual machines and translation rules.

Also known as

Compiler Generator, Program Translator.

Applicability

Use the ABSTRACT COMPILER pattern in any of the following situations:

- *When we want to specify an abstract compiler.* The pattern provides a common formalism/schema (the virtual machine) and a development technique (staged refinement) to write high-level specifications. The specification may be obtained in several steps (i.e., several intermediate-level machines) by defining the inter-machine compilation rules. The specification can be formal or informal. The whole compilation process is obtained by composing the compilation rules between intermediate languages (i.e., virtual machines).
- *When we want to proof the compiler correctness.* Given a stage of the refinement process, the equivalence between two consecutive virtual machines provides a partial proof of compiler correctness. Then, the complete proof can be obtained by composition.
- *When we want to compile a programming language by using the virtual machine technology.* The relationship between an interpreter for a language and a compiler/executor pair for the same language can be given informally in terms of separating computations of the interpreter:: one performing computations only on program structures (the compiler) followed by one performing computations primarily on runtime structures (the executor).
- *When we want to test different alternatives to compile a language.* Given a programming language, it is possible to define different virtual machines to implement/interpret its semantics.
- *When we want to construct emulators/tracers for the execution of our programs.* To emulate the execution of a program forces us to incorporate visualization and debugging facilities to the compiling process. As shown above, the VIRTUAL MACHINE is able to include some of these facilities. On the other hand, since the ABSTRACT COMPILER pattern behaves as a MEDIATOR between the two virtual machines involved in a compilation phase, somehow it should be possible to connect both visualizing and debugging mechanisms. In particular, it would be highly useful the visualization could help us to get a visual confirmation/validation about the compiler correctness.

Structure

The structure of the COMPILER pattern is shown on Figure 7.

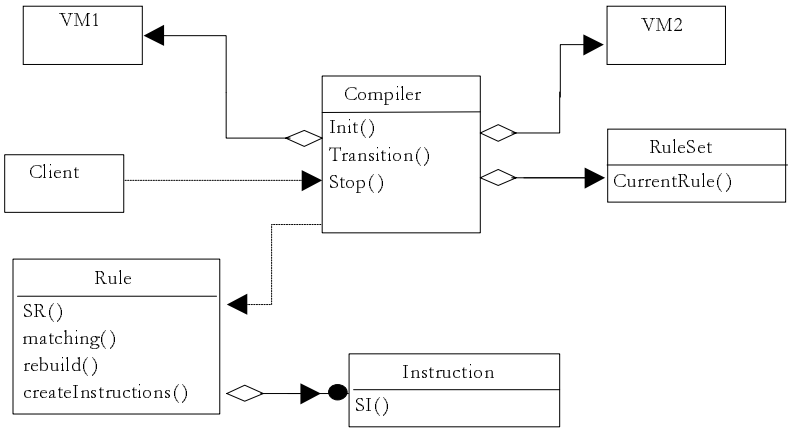

Figure 7. The COMPILER pattern (structure).

Participants

The ABSTRACT COMPILER pattern consists of the following components:

1. *VM1 (JVM)*: It represents the source virtual machine.
2. *VM2 (Intel)*: It represents the target virtual machine.
3. *Rule Set (JVM2Intel Rule Set)*: It represents the set of compilation/translation rules from a VM1 instruction block into a VM2 instruction block. Given a VM1 instruction block, the CurrentRule operation fetches the current rule to apply.
4. *Rule (Iadd Rule)*: It declares a common interface to represent any compilation rule. A compilation rule is defined as a pair (<Block_VM1>, <Block_VM2>), where
   - <Block_VM1> represents a block of VM1 instructions,
   - <Block_VM2> represents a block of VM2 instructions.

   The operation createInstructions constructs the Block_VM2 from Block_VM1. On its part, the operation matching determines if the current state of the VM1 program matches with the rule. The rebuild operation is responsible of reconstructing the VM1's data area from VM2's one. Finally, the SR operation provides the semantics for the rule by gluing previous operations. As a final remark, the operations createInstruction and rebuild must be concretized each time.
5. *Instruction*: It declares a common interface to represent a virtual machine's instruction. It is fully defined in [4].
6. *Compiler*: It coordinates the interactions between VM1 and VM2 and describes a rule-by-rule execution. It behaves as a MEDIATOR pattern [6]. The lifecycle of the Compiler component covers three different phases (*initial*, *transition* and *final)* that are related to the following operations:
   - The *Init* operation determines the initial state of compiling process by configuring the initial states of the VM1 and VM2.

- The *Stop* operation, to determine if VM1 has achieved its final state, in which case the compiler execution stops.
- The *Transition* operation performs the compilation of the machine. It is defined as a while-loop control structure; at every round of the loop the compilation rule pointed by the rule counter (the CurrentRule) is executed. Transition starts at the initial machine's configuration and continuous until the final state is achieved.

Collaborations

As described above, three different scenarios describe the Compiler execution: *initialization*, *transition* and *ending*.

Scenario 1: *Initialization*

The initialization of Compiler is done by initilizing both VM1 and VM2.

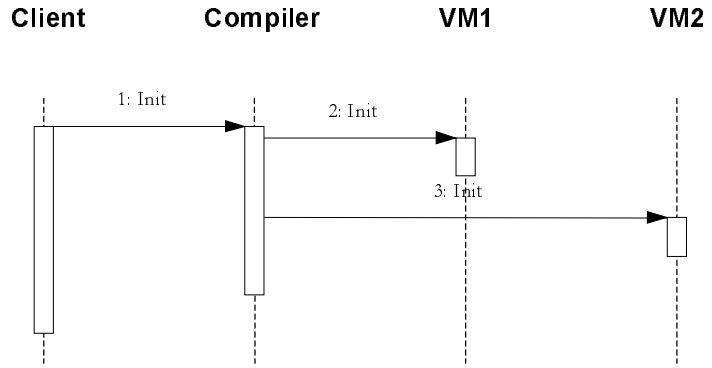

Figure 8. The COMPILER pattern (Collaborations I).

Scenario 2: *Transition*

It performs the compiler execution in several phases:

1. Firstly, given the current VM1 program and Rule Set, translation selects the compilation rule to execute.
2. Then, it interpretes the semantics associated to the rule on the target machine (VM2).

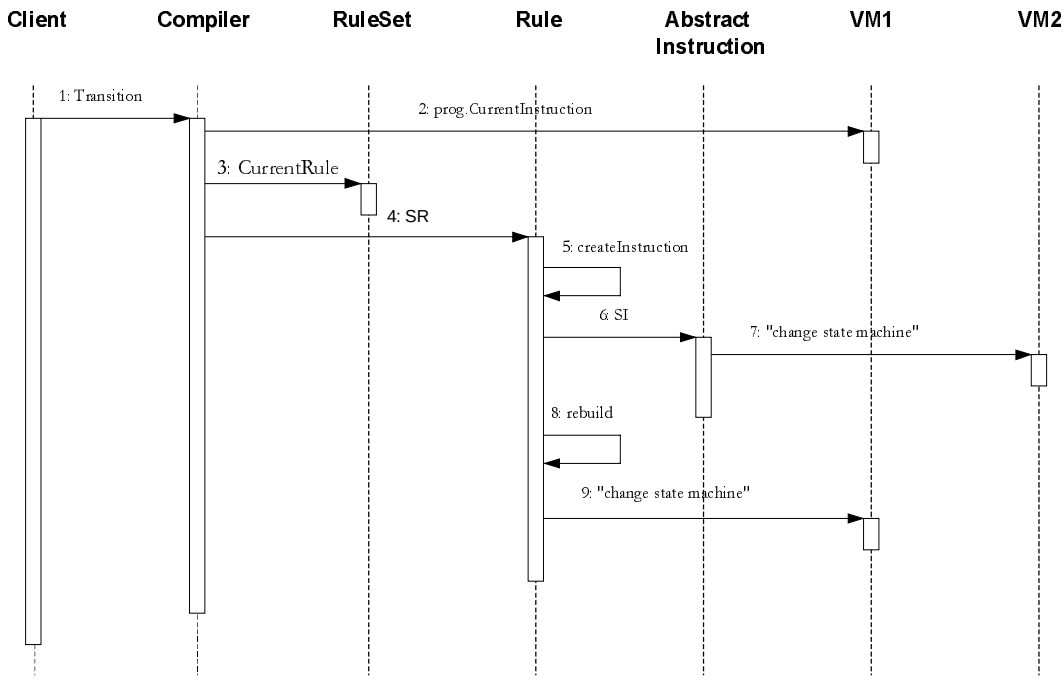

Figure 9. The COMPILER pattern (Collaborations II).

Scenario 3: Ending

Compiler's execution ends when VM1 reaches its ending condition

Consequences

The ABSTRACT COMPILER pattern presents the following advantages:

- *Provide a framework to develop compilers by stepwise refinement of virtual machines.*
- *De-couple the relationships among the Abstract Compiler pattern participants.* The Compiler Factory isolates Compiler from concrete virtual machines (source and/or target) and specific rule sets. On its part, the Rule Set behaves as an Abstract Factory [6]for the rule sets managed by the compilers.
- *Provide a framework to test different compilation strategies.* The pattern makes exchanging compilation-rule set easy, allowing the involved virtual machines to maintain independence of any concrete rule set. The change of a concrete rule set re-configures the compiler.
- *Provide a framework to test different compilation rules (compiling strategies) for the same virtual machines.* Because the compilation is encapsulated by the compilation rules (the SE operation), we can redefine them without affecting to the virtual machines.
- *Offer a higher degree on the re-use of abstract compilers.* In most of the cases, the resulting compiler, in whole or in part, can be highly re-used in new developments.
- *Promote a methodology for the systematic development of abstract compilers.* By separating the participants of compilation in components (i.e., source virtual machine, target virtual machine, rule set and rule semantics) we introduce some design constraints that force the user to follow a more systematic approach.

The ABSTRACT COMPILER presents the following disadvantages:

- *Inefficient implementations.*
- *Communication overhead between Compiler and Virtual Machines (VM1 and VM2).*

Implementation

Consider the following implementation issues:

- *Creating concrete VM1, VM2* and *Rule Set*. For a better representation of components related in the pattern, we may consider to use a Compiler Factory. Follow similar hints that those given by the ABSTRACT FACTORY pattern [6].
- *Concrete Rules might involve dealing with blocks of concrete instructions*. Follow the same implementation issues as in for the COMPOSITE pattern [6].
- *Defining VM1, VM2 and Rule Set as interfaces*. In the first case, follow the VIRTUAL MACHINE pattern [4].
- *Primitive operations*. Some operations defined in Rule component are primitive. Therefore, they must be overridden.

Sample Code

The following sample code shows Java implementations of the ABSTRACT COMPILER pattern applied to the example shown in the Figure 1 (i.e., JVM to Intel compiler).

1. The implementation of the Compiler Factory:

```
import VirtualMachine;
import RuleSet;

public interface CompilerFactory
{
    public VirtualMachine    sourceVM ();
    public VirtualMachine    targetVM ();
    public RuleSet           ruleSet ();
}
```

2. The implementation of the Compiler class.

```
class Compiler
{
       public VirtualMachine sourceVM, targetVM;
       public RuleSet ruleset;

   public Compiler (CompilerFactory factory)
   {  << initialization of VMs and rule set>> }

   public void Init (InputStream tar) {
        sourceVM.Init (tar);
        targetVM.Init ();
   }
   public  boolean Stop () {
        return sourceVM.Stop ();
   }
   public void Transition ()
   {
     AbstractInstruction inst;
     Rule rule;
     while (!Stop ())
     {
        inst = sourceVM.Getprog().CurrentInst();
        rule = ruleset.CurrentRule(inst, this);
        rule.SR (inst, this);

     }
```

```
  } // Transition
}
```

3. The abstract class Rule provides a generic interface to define the translation of source machine instructions to target machine's one.

```
public abstract class Rule
{
    public Instruction instructions [];

    public void SR (Instruction inst, Compiler compiler)
    {
           createInstructions(inst, compiler);

           for (int i = 0; i < instructions.length; i++)
           {
                  instructions[i].SI (compiler.target);
           }
           rebuild (compiler);
    }

    public abstract void rebuild (Compiler compiler );
    public abstract boolean matching (Instruction inst, Compiler compiler);
    public abstract void createInstructions(Instruction inst, Compiler compiler);
}
```

4. The translation-rules corresponding to JVM-2-Intel compiler can be organized around the abstract class JVMtoIntelRule, as follows:

```
abstract class JVMtoIntelRule extends Rule
{
       public  void rebuild (Compiler compiler)
       {
              rebuildDa(compiler);
              rebuildProg(compiler);
       }
}
```

5. Concrete JVMtoIntel rules are obtained by subclassing:

```
class IaddRule extends JVMtoIntelRule
{
       public  boolean matching (Instruction inst, Compiler compiler)
       { ... }
       public void createInstructions (Instruction inst, Compiler compiler)
       {
              instructions = new IntelIntruction [4];
              IntelDataArea ida = (IntelDataArea) compiler.targetVM.da;

              Address add1 = ida.register[AX].getAddress(); //register acumulator
              Address add2 = ida.register[DX].getAddress(); //register data

              instructions[0] = new Pop (add1); // get operand 1 of stack
              instructions[1] = new Pop (add2); // get operand 2 of stack
              instructions[2] = new Add (add1, add2); // operand 1 += operand 2
              instructions[3] = new Push (add1); // put result into stack
       }
}
```

6. We describe an example of the Intel instruction *Add*.

```
class Add extends IntelInstruction
{
       public void SI (VirtualMachine machine)
       {
              IntelDataArea ida  = (IntelDataArea) machine.da;
              IntelProgram iprog = (IntelProgram)  machine.prog;

              Term op1 = ida.getValue (arg[0]);    // Get values from
              Term op2 = ida.getValue (arg[1]);    // arg[0] y arg[1]
```

```
                op1.add(op2);                          // op1 = op1 + op2
                ida.putValue (op1, arg[0]);            // Put result in address arg[0]
                iprog.Inc();                           // Inc pc
        }
}
```

7. The JVMtoIntel rule set derives from the RuleSet interface.

```
public class JVMtoIntel implements RuleSet
{
        Rule rule[];
        static int n_rule = 30;

        public RuleSetArray ()
        {
                rule = new Rule[n_rule];
                rule[0] = new IAddRule ();
                //....
        }
        public Rule CurrentRule (Instruction inst,Compiler compiler)
        {
                for (int i = 0; !rule[i].matching(inst, compiler); i++);
                return rule[i];
        }
}
```

8. Below, it is sketched a client for the JVM2Intel Compiler:

```
class JVM2IntelCompilerClient
{
        public static void main(String args[]) throws IOException
        {
                JVM2IntelFactory factory = new JVM2IntelFactory();
                FileInputStream JVM_code = ...

                compiler = new Compiler (factory);
                compiler.Init (JVM_code);
                compiler.Transition ();
        }
}
```

5. Known Uses

The VIRTUAL MACHINE and ABSTRACT COMPILER patterns have been used to formalize and implement an Abstract Compiler for the Prolog language [2]. As a result of this work, it has been obtained a multi-phase compilation process based on the virtual machine technology. Furthermore, we have found the VIRTUAL-MACHINE pattern is well suited to easily include visualization facilities and debugging mechanisms and facilities in a non-intrusive way [3].

Several commercial products focused on machine emulation, such as Virtual PC [10] and SoftWindows [11], are able to execute programs compiled for MS-Windows over a completely different architectures, such as MacOS. Thus, assembler programs thought to run on Intel processors can be executed (i.e., by means of emulation) on a Motorola processor. This emulation may be seen as three-phase process: 1) the run-time translation from Intel code to Motorola code, 2) the execution of Motorola code obtained from the translation and 3) the reconstruction of Intel machine state from the new Motorola state. Therefore, this kind of emulation software can be modeled as an ABSTRACT COMPILER pattern, taken as VIRTUAL MACHINES the Intel and Motorola hardware.

With no doubts, declarative programming has meant for a long time the best example of an extensive use of the virtual machine technology. As said in the introduction section, Prolog [9] or SML [8] are classic examples of high-performance programming languages using virtual machines. However, the list of declarative languages supported by the virtual machine's approach has been dramatically increased during the last years [12, 13].

As a final remark, the technique of compiler construction by staging transformation is in the vein of the ideas presente here. Staging transformation were introduced in [14], as a general approach to separating stages or phases of a computation based on the availability of data, with an inmediate application to the development of compilers from interpreters. This approach consider the task of automatically construction intermediate-level machine architectures and compilers generating code for them, given operational semantics for source languages [15].

6. Related Patterns

The work presented in [7] explores the definition of a pattern language for building virtual machines. Unlike, our proposal focuses on providing a higher-level design framework and not so much on describing guidelines to obtain low-level implementation of virtual machine architectures.

The VIRTUAL MACHINE and ABSTRACT COMPILER patterns combine the following patterns:

- The ABSTRACT COMPILER can be seen as a kind of VIRTUAL MACHINE, which state is the join of VM1 and VM2 states and which instruction semantics are the semantics of the translation rules.
- Compiler Factory and Instruction Set are ABSTRACT FACTORY patterns [6],
- The *SI* operation (in the Instruction class) and the *SR* operation (in the Rule class) are variations of the STRATEGY patterns [6], and
- Finally, the *Init*, *Transition* and *Stop* operations in the VIRTUAL MACHINE state and ABSTRACT COMPILER state are clear examples of TEMPLATE METHOD patterns [6].

7. Conclusions and Future Work

Virtual machines provide an important stage in the efficient implementation of a kind of languages. However, the construction of such machines and the implementation of high-level languages in them have previously received little attention as (or under) a common framework. In our opinion, the most important contribution of this paper is having outlined the framework to design abstract compilers by step-wise refinement of virtual machines. This way, the constant parts from the varying parts of a virtual machine have been clearly identified and de-coupled, as well as components involved in the process of compiling high-level programs. As a result, the translation/compilation has became a task of constructing new lower-level machine architectures, and then, applying some translation rules to generate code for these architectures. The emulation of a source program can be planned as the execution of the compiled code (obtained dynamically) on a lower-level machine, and therefore, the compilation can be structured as a process provided with methodological guidelines that help to get a most systematic development of abstract compilers.

The Virtual Machine and Abstract Compiler patterns are still too complex. Thus, it is reasonable to split them in a more refined set of patterns. Thus, the Abstract Compiler could be divided in two different patterns that set up clearly differences between the translation and emulation processes. We think it is also necessary to pursue a deeper study around translation rules and the functionality they should provide. Actually, we feel translation rules could be expressed as well-formatted rewriting rules operating on the machine's state. Also, we haven't yet faced how to tackle the instruction loading and to structure the refinement of data areas. Finally, a future version of a Compiler pattern language should include new components that help us to cover the design of compiler front-ends. To this purpose, the work presented in [16] seems to be a promising starting point.

Bibliography

[1] Arnold & Gosling. *The Java Programming Language*. Addison-Wesley, Reading, Massachusetts, 1996.

[2] García J. & Moreno J.J. *Visualization as Debugging : Understanding/Debugging the WAM*, Automated and Algoritmic Debugging (AADEBUG'93), Lecture Notes in Computer Science (LNCS 749), Springer-Verlag, 1993.

[3] García J. & Moreno J.J. *A Formal Definition of an Abstract Prolog Compiler*, AMAST'93. Workshops in Computer Science, Lecture Notes in Artificial Intelligence (LNAI), Springer-Verlag, 1993.

[4] García J. & Sutil M. *The Abstract Machine: A Pattern for Designing Abstract Machines*. 6th. Annual Conference on the Pattern Languages of Programs (PLOP'99), Monticello, Illinois, August 1999.

[5] Goldberg A.J. & Robson D.: *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA, 1983

[6] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns. Elements of reusable object-oriented software*. Addison-Wesley, Reading, MA, 1995.

[7] Jacobsen, E.E. & Nowack, P. *A Pattern Language for Building Virtual Machines. 2th European* Conf. Pattern Languages of Programming, Irse (Germany) , July 1996.

[8] Reade, C, *Elements of Functional Programming*, Addison-Wesley, 1989.

[9] Warren, D. H.D, *An Abstract Prolog Instruction Set*. Tec. Note 309, SRI International, Menlo Park, California, October 1983.

[10] *Virtual PC 1.0* Mac-User Magazine, October 1997.

http://macuser.zdnet.com/mu_1097/reviews/virtual.html

[11] *SotfWindows 95 for Power Macintosh,* Mac-User August 1996. http://macuser.zdnet.com/mu_0896/reviews/review09.html

[12] *WWW Virtual Library on Logic Programming*

http://www.comlab.ox.ac.uk/archive/logic-prog.html

[13] *WWW Virtual Library on Functional Programming*

http://www.engr.uconn.edu/~jeffm/FuncProg/Papers/funcprog.html

[14] Jorring, U. Scherlis, W. *Compilers and Staging Transformation.* In 3th ACM Symposium on Principles of Programming Languges, 1989, pp. 281-292.

http://macuser.zdnet.com/mu_1097/reviews/virtual.html

[15] Hannan, J. *Operatinal Semantics-Directed Compilers and Machine Architectures,* ACM Transactions on Programming Languages and Systems, Vol. 16, No. 4, July, 1994

[16] Bosch, J. *Parser Delegation - An Object-Oriented Approach to Parsing.* In Proceedings of TOOLS Europe'95, 1995.

[17] Lindholm, T. Yellin, F. *The JavaTM Virtual Machine Specification*. Sun Microsystems, Inc.