The *ExTrEm Compiling*[1] Pattern Language

The Construction of Compiler Back-Ends by Stepwise Transformation of Virtual Machines

J.García-Martín M.Sutil-Martín

Universidad Politécnica de Madrid
Campus de Montegancedo, s/n Boadilla del Monte - 28660- Madrid
Phone: 34-1-3367455 Fax: 34-1-3367412
juliog@fi.upm.es, msutil@arrakis.es

Abstract

A standard technique in compiler construction[2] consists in structuring the compilation of source languages to target code by intermediate levels[21][22][23]. Such an approach is practicable because compilation phases between intermediate levels become simpler and increase the flexibility regarding different source and target codes. Furthermore, recently rediscovered virtual machine technology can be effectively used as an intermediate low-level architecture suitable for supporting serious implementations of a wide variety of programming languages. Additionally, the virtual machines provide several desirable features such as portability, code optimizations, and native machine code generation. Now, we consider this task of constructing intermediate-level machine architectures and compilers generating code for these architectures in the context of design patterns.

This paper presents the ExTrEm-Compiling pattern language, a pattern-based approach to construct compiler back-ends by staging transformation of virtual machines. There are six patterns in this ExTrEm-Compiling pattern language. The patterns and their combination supply compiler designers with a highly customizable three-fold architecture: Firstly, with a architectural skeleton of patterns that captures the essential features underlying virtual machines. In this architecture constant and varying parts are clearly identified and de-coupled. Secondly, with a product-line architecture for code generation. The translation rules defining a compilation phase are the rules driving code generation at that level. The translation-rule construction and management are provided by a translation architecture. Finally, ExTrEm-Compiling provides designers with a generic infrastructure to test the validity and consistency of decisions taken during compiler construction.

The results reveal the suitability of design patterns to tackle compiler construction. Firstly, the number of variation elements and customization points found in the patterns of the language are significantly large. This result suggests that our proposal provides a significant higher-degree of flexibility and reuse than traditional approaches. Moreove, the use of ExTrEm-Compiling in the redefinition of a previous work [2][3][16] has shown conclusive proofs of the benefits provided by this pattern-based solution in even the ‘roughish’context of formal specifications [16]. Remarkably, the ability of ExTrEm-Compiling to be highly adaptable and pluggable advocates its use, as well, beyond the context of compilation techniques. In this sense, some links with other models on software design are commented. The ExTrEm-Compiling pattern language is not only a contribution for a more abstract and systematic way of constructing compiler back-ends, but also an attermpt for improving the understanding of compilation processes. Therefore, a methodology for compiler construction based on ExTrEm-Compiling must be an objective in further research. Other future work related to this proposal is outlined in the paper too.

[1] An acronym of Excutor-Translator-Emulator.

[2] Our investigation is concerned with those implementation aspects traditionally belonging to the back-end of a compiler, that is, the mapping of complex source features to the primitive operations of target machines. This sets the focus on code generation and optimization techniques. Other topics such as parsing, type checking, and static program analisys are not considered.

1. Introduction

Compilation has been traditionally viewed as the translation of one programming language into another language which is understood by a CPU or to some ´primitive´ language (typically assembler). Today, the gap between modern high-level programming languages and existing hardware can be so large that it is hard to see how the source language relates to the assembler language or the hardware. To face this situation, we can adopt a broader view of compilation as a piecemeal transformation of a language.

A standard technique in compiler construction is to structure the compilation of a source language to target code by intermediate levels. Such an approach is more practicable because compilation between intermediate levels become simpler, as well as it increases the flexibility regarding differente source and target codes. In order to find suitable intermediate levels we first develop a design method based on which we then derive a compiler. Hence, a compiler which compiles a higher level language *SL* (source language) into a machine language *TL* (target language) uses a sequence of intermediate languages (longer if the levels of the languages differ too much), as follows:

$$\boxed{SL = IL_0,\ \dots,\ IL_n = TL}$$

Instead of compiling *SL*-programs directly into *TL*-programs, SL-programs are compiled into IL_1-programs which are then compiled into IL_2-programs, etc. The levels of the intermediate languages (IL_i, IL_{i+1}) do not differ too much.

Nowadays, one of the buzzwords in compiler construction is JVM (Java Virtual Machine). As is well-known, the use of virtual machine technology has been largely significant in the success of Java. However, though the big impact at this moment, the use of virtual machines is far from being a new issue. Moreover, long before the boom of Java, virtual machines had been effectively used as intermediate architectures for supporting serious implementations of a wide variety of programming languages, including imperative, declarative [9][10] [13][25] (e.g., functional and logic[3]) and object-oriented programming languages [5] [24]. Among others, virtual machines provide several desirable features such as portability, code optimizations, and native machine code generation.

In a previous work [3], we described the task of compiling as a stepwise-translation of virtual machines. Although focused on compiler specification for logic programming languages, the work established a common abstract model for sequential execution on virtual machines. So that, a virtual machines is defined by the following elements:

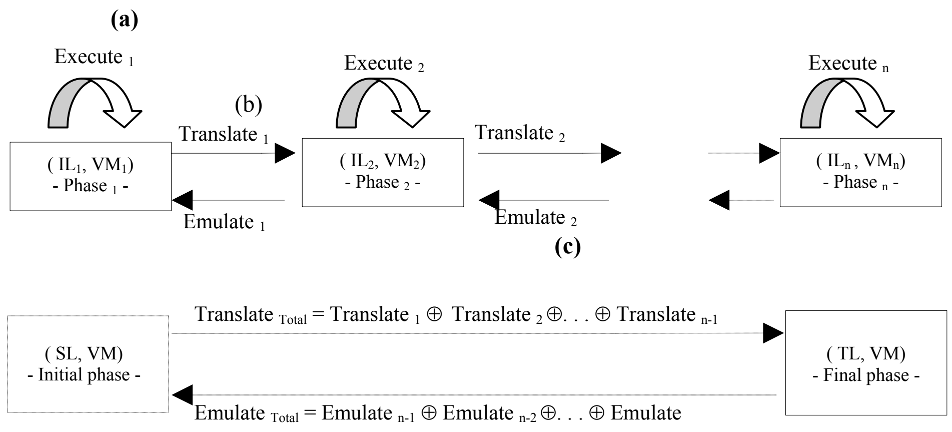
- the *data area*, which defines the *configuration* of the machine;
- the *instruction set* and a *semantic function* for each of its elements (defining the changes on the configuration after executing one instruction);
- the *transition function* between an initial and a final configuration which is guided by the semantic function of the instruction being currently executed; and
- the *translation function* which compiles a program into machine code.

Such a simple definition can be organized as an abstract architecture suitable for analysis and experimentation. Unlike concrete architectures can varies widely depending, in part, on the language being described and, also, on the representation of source programs as data. Consequently, any

[3] Prolog [10] , SML [9] or Haskell [25]are good examples of high-performance programming languages implemented by virtual machines.

approach with the aim of capturing the essential features underlying virtual machines has to find a balance between two forces: generality and diversity.

Our approach relies on structuring compilation in several phases as a piecemeal transformation. Furthermore, we consider the task of constructing intermediate-level machine architectures and compilers generating code for these architectures. As shown in Figure 1, we propose a conceptual graph to define intermediate levels as pairs (Intermediate Level, Virtual Machine) and their relationships (Compile and Decompile). Besides, we provide a method upon which we then derive compilers. The method is supported by a set of patterns integrated in a pattern language.


(a)
Execute 1
Execute 2
Execute n
(b)
Translate 1
Translate 2
(IL1, VM1) - Phase 1 -
(IL2, VM2) - Phase 2 -
(ILn , VMn) - Phase n -
Emulate 1
Emulate 2
(c)
Translate Total = Translate 1 ⊕ Translate 2 ⊕. . . ⊕ Translate n-1
(SL, VM) - Initial phase -
(TL, VM) - Final phase -
Emulate Total = Emulate n-1 ⊕ Emulate n-2 ⊕. . . ⊕ Emulate

(a) Any action on a intermediate-stage (i.e., execution, debugging, visualization, etc.)
(b) Any forward transformation (i.e., translation, specification, code generation,etc .)
(c) Any backward transformation (i.e., VM rebuilt, correctness proof, emulation, etc .)
(d) ⊕ any kind of composition between intermediate phases.
(e) ***Execute*** $_{\textbf{Initial}}$ *≡ Execute* $_1$ *≡ Execute* $_2$ *≡ . . . ≡ Execute* $_n$ *≡* ***Execute*** $_{\textbf{Final}}$

Figure 1. Compiler Construction by stepwise refinement of virtual machines (conceptual model)

At this point, we are ripe for presenting the main goal in this work: the EXTREM-COMPILING pattern language. The development of EXTREM-COMPILING involves the definition of six design patterns and the three pattern architectures (by the moment). These architectures are orthogonally integrated in the language.

2. Motivation

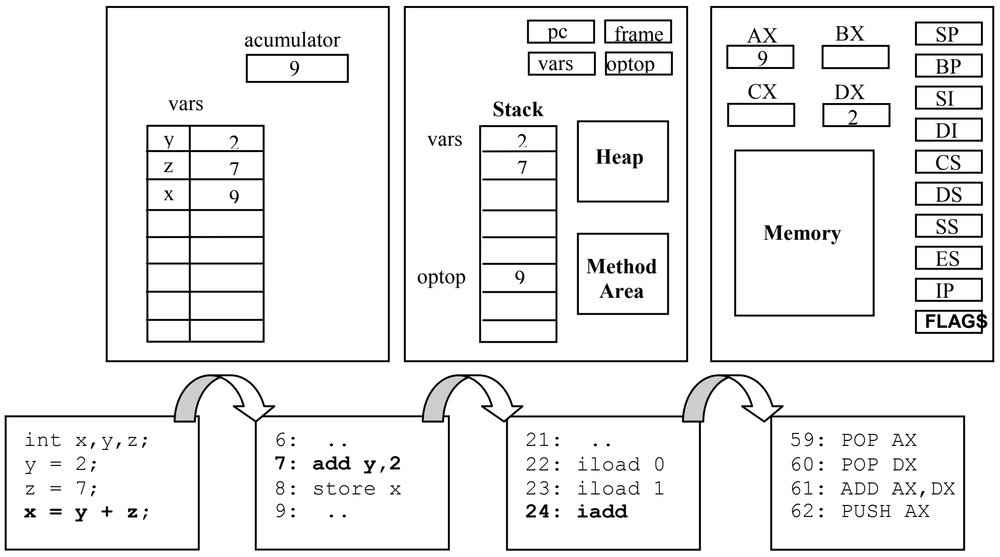
In order to illustrate the nature of the problem addressed by this work, we describe now several common scenarios around language processing and compiler construction. The section is organized by two examples. Firstly, we describe in a simple way the process of compiling Java programs. Secondly, we sketch the outcomes of a previous work, which served like antecedent for the present proposal. More concretely, the example describes some relevant parts of a formalization for a Prolog compiler. Though the second example may include specific questions on logic programing, however, the intention is in no way requiring a full understanding of the example.

Example 1 . *The Java Virtual Machine*

The Java Virtual Machine (JVM) [24] is an abstract software-based machine which can operate over different microprocessor machines (i.e., hardware independent). Designers of the 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 scenes bridge allows the software developers to *"Write Once, Run AnyWhere"* [1] because the JVM must behave the same regardless of the underlying microprocessor following the standard specifications of the JVM.

As said before, the gap between modern high-level programming languages (Java) and existing hardware (Intel processor) makes it necessary to introduce intermediate languages running on virtual machines (JVM). In Figure 2, we focuse the question by depicting, in a snapshot, a common scenario for the process of compiling a Java program. Let's consider the following two phases:

1. Firstly, the process requires the design of an intermediate-level machine JVM (**J**ava **V**irtual **M**achine). Then, by means of a concrete interpretation of the Java operational semantics, to construct a Java compiler to translates Java code into JVM code.
2. Next, the JVM architecture has to be mapped into a concrete hardware machine (e.g., Intel machine) and JVM instructions has to be interpreted in terms of hardware instructions.



```
int x,y,z;
y = 2;
z = 7;
x = y + z;
```

```
6:  ..
7: add y,2
8: store x
9:  ..
```

```
21:  ..
22: iload 0
23: iload 1
24: iadd
```

```
59: POP AX
60: POP DX
61: ADD AX,DX
62: PUSH AX
```

Figure 2. An hypothetical scenario for the compilation of Java

The portability of Java is due to the definition of a virtual machine (the JVM) and a compact instruction set which may be mapped onto a real hardware machine (phase 1). On its side, an object code translator, appended to the Java compiler, will generate fine-grained object instructions for the host computer (phase 2), by micro-coding the JVM instructions. At this point, the gap between JVM and Intel architectures could be so large that it could be hard to define a good translation function. Not for nothing, most of the optimizations achieved by (Java) executions come from the translation to machine code. As a consequence, the work of compiling is almost carried out in one-hit, as a whole, making the task not modular and hardly reusable. Furthermore, useful high-level optimizations are not used because of the internal complexity of the compiler.

Example 2 . *The Abstract Prolog Compiler*

In the last years, the importance of logic programming languages has increased. For logic languages we understand not only Prolog but also several languages that uses logical components[4] (deductive inference as operational semantics, unification, backtracking, etc.). Long before the boom of Java, many works on declarative programming had already focused on designing virtual machines as an alternative to common implementation techniques. In this sense, the success of Prolog can be largely attributed to the work of D.H.D. Warren and his abstract/virtual machine (the Warren Abstract Machine or WAM). The WAM, as was described in [10], resembles an intricate puzzle, whose many pieces tightly together in a miraculous way. Certainly, the contribution of Warren points out the possibility of compiling Prolog and getting efficient code. Most of current Prolog systems are based on the resulting machine.

a) The Prolog Computational Model

Given a Prolog program, its computation can be seen as systematic search of a space of possible solutions to initially given query or goal. The set of computation states (resolution states or configurations) is often viewed as carrying a tree structure, with initial state at the root, and son relation representing alternative (single) resolution steps.

```
? flatten([[a,b,c, [d,e,f]], []], L), write(L).
flatten([], []).
flatten([[] | T], L) :- flatten(T, L).
flatten([[X|Y] | T], L) :-
                flatten([X|Y], L1),
                  flatten(T, L2),
                append(L1, L2, L).
flatten([H | T], [H | L]) :- flatten(T, L).
append([], L,L).
append([H | T], L, [H | TL]) :- append(T, L, TL).
```

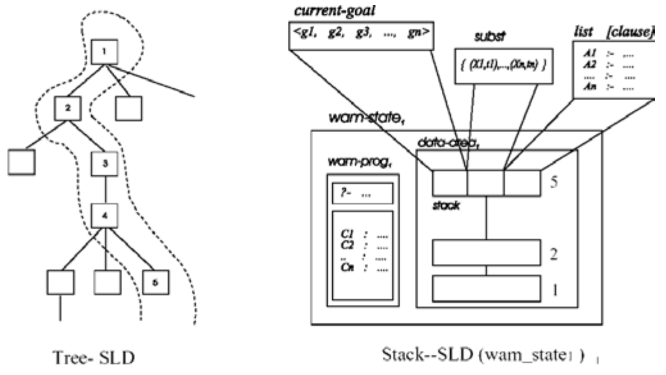


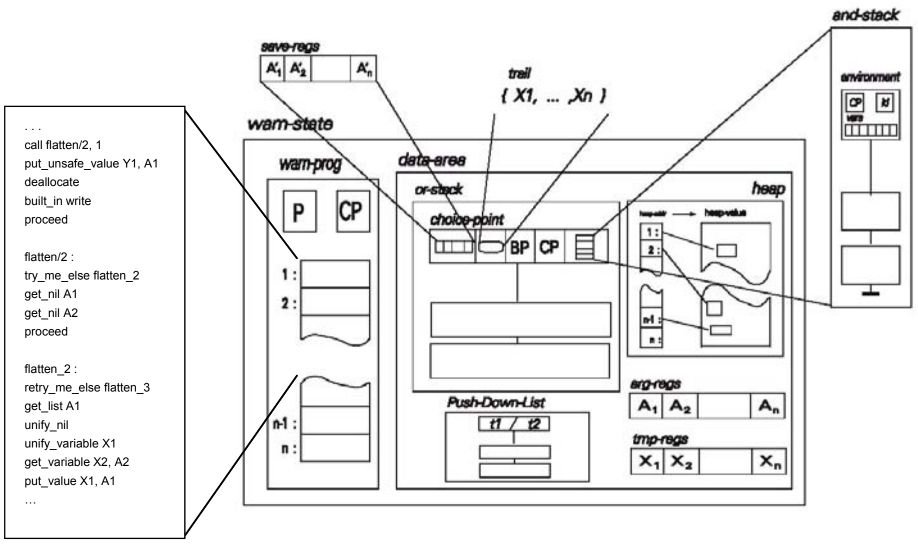
Figure 3. Prolog computational model

[4] See [18] for a more detailed study.

Then, we represent Prolog computation states as a set of tree nodes. Each node has to carry all information relevant at the desired abstraction level for the computation state it represents. This information consists in the sequence of goals still to be executed, the substitution computed so far, and possibly the sequence of alternative resolvent states still to be tried. The classification of Prolog tree nodes suggests a straigforward stack representation: disregarding the abandoned nodes (since they, once abandoned, play no further role in the computation), we may view the path of active nodes as a stack. In Figure 3, it is shown an example of a Prolog program and both SLD-tree and SLD-stack computation models.

b) The WAM Computational Model

Prolog computational model shows an apparent simplicity. In contrast, the WAM is characterized by a space-efficient representation for the significant elements of Prolog programs (procedures, clauses, predicate application, terms and logical variables). The resultant machine resembles an intricate puzzle of machine components (such as stacks, environments, registers, etc.), as well as a set of very specialized machine instructions. In Figure 4, we can see a WAM program example, and it is sketched the structure of its data area.


```
. . .
call flatten/2, 1
put_unsafe_value Y1, A1
deallocate
built_in write
proceed

flatten/2 :
try_me_else flatten_2
get_nil A1
get_nil A2
proceed

flatten_2 :
retry_me_else flatten_3
get_list A1
unify_nil
unify_variable X1
get_variable X2, A2
put_value X1, A1
…
```

Figure 4. The WAM computation model

Following Figure 4 , the following elements constitutes the WAM:

- *frame stack (and-stack*): frames store local procedure variables.
- *choice point stack (or-stack):* a choice point frame encapsulates a machine state. It stores the argument registers and machine registers, the current frame, and pointers to the trail stack and heap.

- *trail stack*: stores pointers to locations that store logical variables that have been bound.
- *heap*: the heap stores terms and logical variables. It is managed in a stack-like fashion.
- *push down list*: This is a stack that supports the recursive calls of a general unification procedure.
- *argument registers*: arguments are passed in registers

c) From Prolog to the WAM : A compiler specification by stepwise refinement

The Abstract Prolog Compiler was an attempt to approximate the reader to a good understandable view of compilation of Prolog [3] . Even though there were several previous descriptions of the WAM at that moment (see [26][30] and, specially, [28]) the explanations (for instance [29][31], and, best of all [27]) did not provide enough insights to understand the process. To overcome the gap, in Abstract Prolog Compiler we presented an abstract view of the WAM and Prolog compilation to the WAM. For an abstract view of the WAM is meant a description of the WAM focused in how: a) it implements SLD-resolution with backtracking and b) the main elements of Prolog (unification and backtracking) can be compiled. Implementation details and optimizations are delayed as most as possible. Abstract Prolog Compiler allow us to achieve several conclusions:

1. We provided a full abstract description of Prolog operational semantics by providing abstract descriptions for chained virtual machines.
2. The stepwise refinement allowed us to cover the compilation process without exposing implementation details. Furthermore. criteria based on language constructs allow the pieces of the puzzle (the WAM) were easily discovered.
3. We define a methodology for compiler specification based on stepwise refinement of virtual machines.
4. Correctness proof for Prolog compiler was achieved by compositionality, as a result of composing the intermediate correctness proofs.
5. Similar ideas were successfully applied to other logic systems with a high degree of reuse.
6. Stepwise refinement produced a product-line architecture (of virtual machines) as a chain of compilation alternatives.
7. Each compilation step defined a repository of reusable compilation components. Furthermore, A highly reusable family of related instruction-sets (sorted by granularity) were obtained along the process.

3. The Patterns

Pattern 1 : Program-Loader

Context

An assembler program is commonly stored in a text file, one instruction per line. However, the program execution needs the assembler code be represented in an internal representation.

Problem

How to translate the textual representation of the program to an internal representation?

Forces

- The textual representations for a program maybe is not unique.
- The textual representations of a program is the sequential textual representation of its instructions.
- The textual representation maybe is corrupted or full of mistakes.
- There is a biyective relationship between the textual representation of one instruction and the instruction itself.

Structure

The structure of the PROGRAM LOADER pattern is shown on Figure 5.

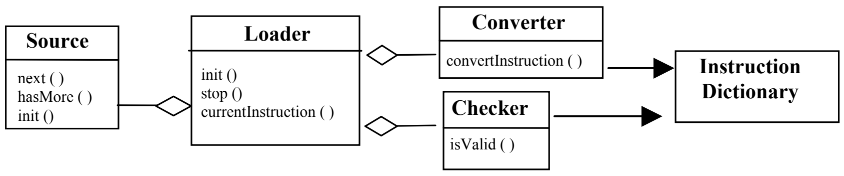


Figure 5. The PROGRAM-LOADER pattern (structure)

Participants

- **Source** : It declares a common interface for a step-by-step traversal on source programs.
- **Converter** : It maps a related group of instructions (one or more) from one specific format into another (i.e., text format into object format).
- **Checker** : It tests whether an instruction satisfies or not a given conversion property (i.e., it's in a given instruction dictionary).
- **Loader** : It coordinates the relationships among the other participants of the pattern.
- **InstructionDictionary** : It abstracts the instruction set supported by a virtual machine by registering pairs (mnemonic, instruction), and supply functionality for translating assembly instructions (see the INSTRUCTION SET pattern).

Collaborations

Collaborations among the pattern participants are depicted on Figure 6.

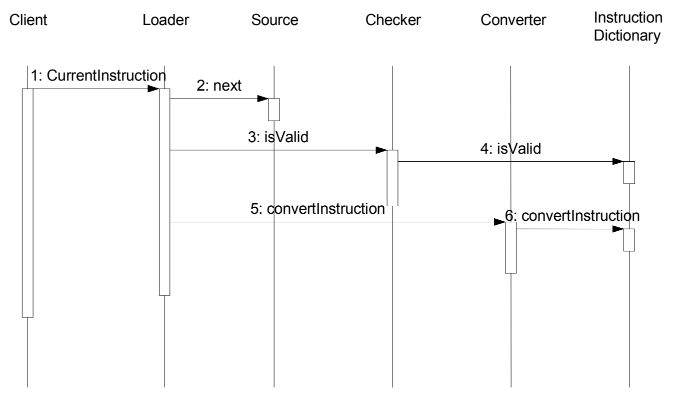


Figure 6. The PROGRAM-LOADER pattern (collaborations)

Implementation

Consider the following implementation issues:

1. *Source as an ITERATOR*. It allows us to abstract the format of the source program (e.g., a text file, a stream, an array of instructions, etc.), as well as the program's structure (e.g., one instruction each line, etc).
2. *Checker and Converter can be implemented as FUNCTION OBJECTS* [7]. They provide the LOADER pattern with a way to integrate behavioral parameterization.

Sample Code

1. The following sample code gives some hints about the implementation of PROGRAM-LOADER.

```
class Loader
{
// ...
public Instruction CurrentInstruction ()
{
    Object inst= null;
    boolean valid = false;
    while (!valid && !stop())
    {
      inst = source.next();
      valid = check.isValid(inst);
    }
    return converter.convertInstruction(inst);
  }
//...

} // class “Loader"
```

2. Next, is presented a sample use of PROGRAM-LOADER taking the instructions from a text file.

```
class Example
{
  public  static void main (String args[]) throws IOException
  {
    JavaInstructionSet set = new JavaInstructionSet();
    CheckerInst check = new CheckerInst(set);
    ConverterInst converter = new ConverterInst(listset);
    SourceStream source= new SourceStream (args[0]);
    Loader loader = new Loader(source, check,converter);
    loader.Init();

    while (!loader.Stop())
    {
      AbstractInstruction inst = loader.CurrentInstruction();
      // Process an instruction....
    }
  }
}
```

Related Patterns

- PROGRAM LOADER is an instance of the MEDIATOR pattern[6] that integrates the other participants by behavior parameterization (in the same way as FUNCTION OBJECT [7]).
- Source and Loader can be seen as an ITERATOR [6] for program representation and traversal. Checker and Converter are closely related to the TRANSLATION RULE pattern.
- Checker and Converter can be abstracted in a Loader Factory, following the ABSTRACT FACTORY pattern [6]).
- How to make instructions is described in the INSTRUCTION SET pattern.

Pattern 2 : The VIRTUAL MACHINE[5]

Context

Virtual machines provide a compilation technology that has been effectively used as an intermediate or low-level architecture suitable for supporting implementations of a wide variety of programming languages, including imperative, object-oriented, functional or logic programming languages. They facilitate portability, code optimizations, and native machine code generation. Their simple structure makes them suitable for analysis and experimentation. The structure of abstract machines varies widely, depending in part on the language being described and also on the representation of source programs as data.

Problem

Define a common design template of virtual machine so that it captures essential features addressed by any virtual machine.

[5] In an earlier version, we called to this pattern ABSTRACT MACHINE [4].

Forces

- A virtual machine is defined by its elements, it is, the program, the data area and the instruction set.
- Elements in a virtual machine do not necessarily provide the same functionality or structure.
- Programs executed by virtual machines are stored in files (as textual representations)
- To execute a program, the virtual machine needs to construct an internal representation of the program. Therefore, loading the program from its textual representation is needed.
- The state of a virtual machine is defined by the values of its parts.
- During the execution of a program, the virtual machine can achieve three different states: initial, executing and final.
- Before the execution of a program the virtual machine remains in initial state.
- After the execution of a program, the virtual machine remains in final state.
- To execute a program consist on executing the instructions of the program, in a step-by-step process.
- The execution of an instruction may change the data area or the program or both. How are these changes must be described by programmers.

Structure

The structure of the VIRTUAL MACHINE pattern is shown on Figure 7.

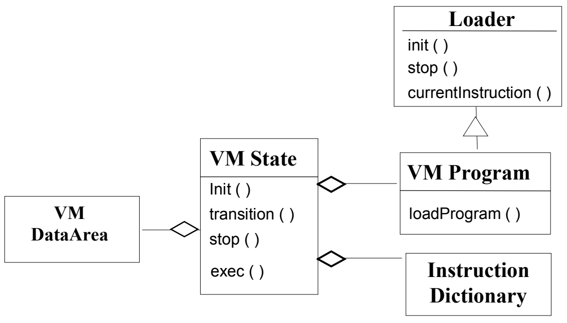


Figure 7. The VIRTUAL MACHINE pattern (structure)

Participants

At the highest level of abstraction, an abstract machine is defined by two parts: (i) the *static* part, components related to the state, (ii) and the *dynamic* part, components associated with the machine's behavior. The VIRTUAL MACHINE pattern locates each of these components into different participants.

The *state* of an abstract machine consists of the following three components:

- **VM_DataArea**: It defines an abstract data configuration of the machine.
- **VM_Program**: It defines an abstract configuration of the machine assembler program. It is defined as a concrete instance of the Loader class adding the operation LoadProgram, which is responsible of constructing the assembler program.

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 of describing the different states that the abstract machine achieves during the execution of a given program. These operations are described below:

- **VM_State**: It coordinates the interactions between the VM_DataArea and the VM_Program (such as a MEDIATOR pattern [6]). It determines the machine's state. There are three different stages for the state: *initial, executing* and *final.*
- **InstructionDictionary**: It abstracts the instruction set supported by a virtual machine by registering pairs (mnemonic, instruction), and supply functionality for tranlating assembly instructions (see the INSTRUCTION-SET pattern).

Collaborations

Three different scenarios describe the three different states that a virtual machine may reach during its execution: *initialization, transition* and *ending*. Figures 8, 9 and 10 depict these scenarios.

Scenario 1: Initialization

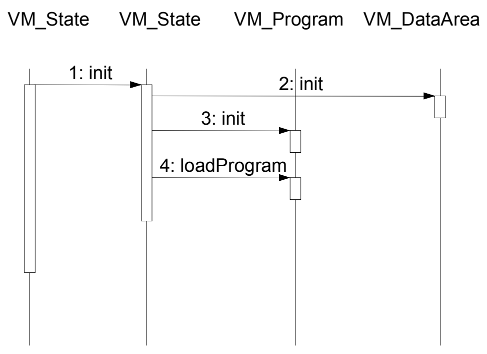


Figure 8. The VIRTUAL MACHINE pattern (collaborations I)

At this step, the virtual machine is initialized. As a result, both the VM-DataArea and VM-Program are initialized. The initialization of the VM-Program is carried out by the *init* operation, and sets its components to the initial values (i.e., program counters, array of instructions, etc…). Next, it executes the *loadProgram* operation, which is in charge of loading 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, executing the init operation (i.e., the data registers or control registers, etc…) initializes the components that define the VM-DataArea.

<u>Scenario 2</u>*: Transition*

The *transition* operation performs the machine execution. To do this, the operation *transition* requests from the program the instruction to execute that is pointed by the program counter (*currentInst* operation). The execution of the machine consists of executing instructions until the ending condition (instruction-fetching loop). Each machine's instruction is responsible of defining its own semantics. So, each instruction provides the *SI* operation, which execution modifies the machine's state (i.e., the program and/or the data area). Therefore, depending on the instruction will modify the data area, the program or both. The order in which these modifications are made depends on the instruction semantics.

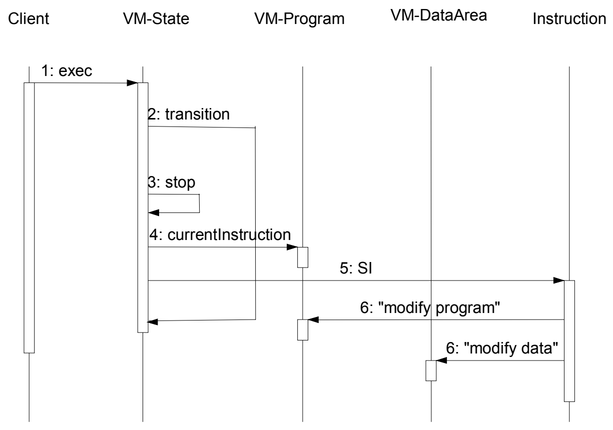


Figure 9. The Virtual Machine pattern (collaborations II)

<u>Scenario 3</u>*: Ending*

The ending stage for the machine's execution is achieved when one of its two components (or both) achieve the ending condition (e.g., it executes a concrete stop instruction, there is a data area overflow, etc.). The Stop operation on the State asks 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.

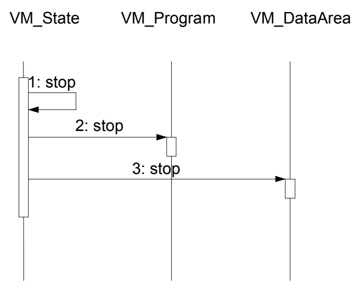


Figure 10. The VIRTUAL MACHINE pattern (collaborations III)

Implementation

Consider the following implementation issues:

1. Concrete VM_DataArea might be a complex object composition. Follow the same implementation issues as in for the COMPOSITE pattern [6]
2. Defining VM_DataArea and Instruction as interfaces or abstract classes. 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 DataArea
{
    public void init ();
    public boolean stop ();
}
```

3. The Instruction provides methods to determine information about instructions (i.e., *name*, *and numArguments*), a method defining the semantic interpretation (*SI*) and the method *process* that compiles the arguments of the assembler instruction. The VM_State is an argument of *SI* operation and it is only used during this instruction.
4. Omitting the VM_State class. It is similar to the case of the MEDIATOR pattern [6].
5. Primitive operations. Some operations defined in VM_DataArea, Instruction are primitives. 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 VM_State must never be overridden.

Sample Code

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

1. The implementation of the VM_State.

```
import VM_DataArea;
import VM_Program;
import VM_Instruction

public class VM_State
{
     private VM_DataArea da;
     private VM_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);
       }
     }
     public void exec (FileInputStream targetCode)
     {
         init (targetCode);
         transition ();

     }
}
```

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

```
class M_Client
{

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

3. 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). 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
{
     public I ()
     {}
     static public String name ()
     {
         return “I”;
     }
     static public int numArguments ()
     {
      return 1;
     }
     public String toString ()
     {
         return name() + “ “ + numVars;
     }
     public void SI (VM_State state)
     {
         M_DataArea M_da = (M_DataArea) state.da;
         M_Program  M_prog = (M_Program) state.prog;

         // Implementation of I semantics
         ...

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

Related Patterns

- VIRTUAL MACHINE is an instance of the MEDIATOR pattern [6] that integrates the other participants by structural parameterization.
- A VM State Factory could parameterize the VM State (i.e., as an ABSTRACT FACTORY pattern [6]).
- VM Program is related to the ITERATOR pattern (see PROGRAM LOADER).
- The *Init, Transition* and *Stop* operations are examples of TEMPLATE METHODs [6].
- VM DataArea is candidate to be a COMPOSITE pattern [6].
- How to locate and execute instructions is described in INSTRUCTION SET.

Pattern 3: The INSTRUCTION SET

Context

The set of instruction types comprises the instruction set of the virtual machine and is provided by the user. The virtual machine has to facilitate the encapsulation of the instruction set, so they can be accessed in a uniform manner.

Problem

The virtual machine executes instructions, which access and update the state of the virtual machine. Means for storing, locating and executing instructions are needed. This includes the division of responsibility between the program loader (see PROGRAM-LOADER pattern), the instruction-fetching loop (see VIRTUAL-MACHINE pattern) and the instructions themselves.

Forces

- The internal representation of a program maybe is not unique.
- In order to facilitate the managing and processing of instructions, a sequential internal representation is needed.
- The internal representation of a program is a sorted internal representation of its instructions.

Structure

The structure of the INSTRUCTION-SET pattern is shown on Figure 11.

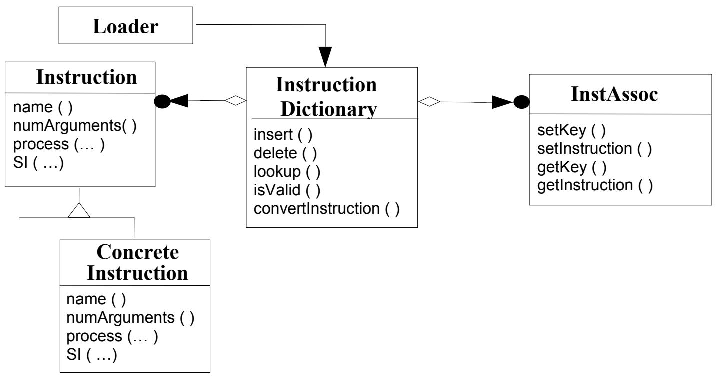


Figure 11. The INSTRUCTION-SET pattern (structure)

Participants

- **Instruction**: It defines an abstract interface for a machine instruction. The *SI* operation (i.e., the Semantic Interpretation) determines how the configuration of the machine evolves when the instruction is executed. The *SI* operation must be defined for each instruction defined in the instruction dictionary.
- **Concrete Instruction:** It is a concrete instance of Instruction for a concrete virtual machine.
- **InstructionDictionary**: It abstracts the instruction set supported by registering pairs (mnemonic, instruction), and supply functionality for tranlating assembly instructions.

- **InstAssoc**: It defines a biyective relation between a key-value representing an instruction (i.e., a mnemonic) and the corresponding Instruction.

Implementation

Consider the following implementation issues:

1. Defining Instruction as an interface or an abstract class. 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 [])
}
```

2. The Instruction provides methods to determine information about instructions (i.e., *name*, *and numArguments*), a method defining the semantic interpretation (*SI*) and the method *process* that compiles the arguments of the assembler instruction. The VM_State (see VIRTUAL-MACHINE pattern) is an argument of *SI* operation and it is only used during this instruction.
3. Defining the Instruction Dictionary as a hash-table of InstAssoc elements.

Sample Code

See related sections in the VIRTUAL MACHINE and PROGRAM LOADER patterns

Related Patterns

- The *SI* operation in the Instruction class is a kind of hidden STRATEGY pattern [6].
- PROGRAM LOADER and VIRTUAL MACHINE are related patterns in the pattern language.

Pattern 4: The TRANSLATION RULE

Context

You have two intermediate languages IL_i *and* IL_{i+1}. Now, you want to describe the translation of IL_i-programs to IL_{i+1}-programs. Each intermediate language defines a different instruction dictionary.

Problem

How to define a correct and non-ambiguous translation system between two different programming languages?

Forces

- A translation rule translates a source instruction (or a sequence of instructions) into a target instruction (or a sequence of instructions).
- Given source instruction (or sequence of instructions) there are many possible translation rules to define.
- A translation rule is injective relation.

Structure

The structure of the TRANSLATION RULE pattern is shown on Figure 12.

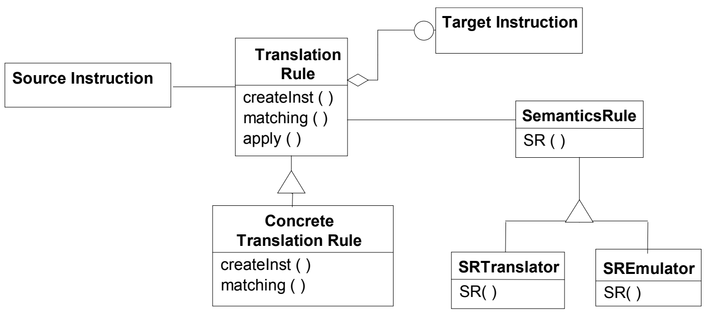


Figure 12. The TRANSLATION RULE pattern (structure)

Participants

- **Source Instruction**: It defines the program instruction that we want to translate from.
- **Target Instruction**: It defines the program instruction or instructions that we translate to.
- **Translation Rule**: It provides an abstract interface to define a translation rule between two languages. It includes three operations:
  - *matching*: Given an origin Instruction, it decides if the rule can be applied or not.
  - *createInstructions*: It is responsible of creating the resulting target Instructions.
  - *apply*: it applies a concrete semantic operation to the rule.
- **SemanticsRule**: It provides a translation rule with the semantics for each target instruction obtained. The operation *SR* defines the concrete meaning of the semantics (compilation, emulation, etc.).
- **SRTranslator**: It is the concrete instance of SemanticsRule for the case of translating to a target textual representation (see TRANSLATOR pattern).
- **SREmulator**: It is the concrete instance of SemanticsRule for the case of translating to target instructions (i.e., internal representation) executable by a virtual machine (see EMULATOR pattern).

Collaborations

Once a translation rule matches an origin Instruction, the concrete semantic associated to the rule is executed. It is carried out by invoking the operation apply, whose meaning is twofold:

- Firstly, constructing the target Instruction (i.e., the operation *createInstruction*).
- Secondly, applying the semantics rule (i.e., the operation *SR*).

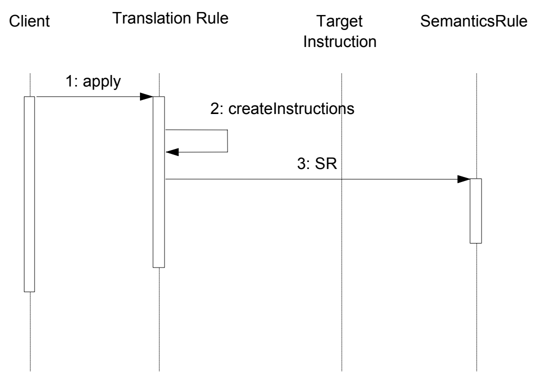


Figure 13. The TRANSLATION RULE pattern (collaborations)

Implementation

Consider the following implementation issues:

1. The operation *createInstructions* creates the target Instructions. We may use whether an ABSTRACT FACTORY or by subclassing.

2. The operation *matching* must be understood in a wider sense. Thus, it could be corresponding to the *instanceof* operator of Java, the functional programming pattern-matching mechanism, etc.

Sample Code

Taking the example presented in the Motivation section (see Figure 2), we illustrate how to translate JVM instructions into Intel instructions. As shown below, the Intel instructions are stored in a text file.

```
class CompilationRule_Demo
{
       public static void main (String args[])
       {
         Instruction inst[] = new  Instruction[3];

         RuleSet rset = new RuleSetJVMtoIntel();
```

```
            SemanticsRule sr = new TranslatorSemantics();
            // ....

            // JVM instruccions are created
            inst[0] = new Iload(0); // Cargar operando 1
            inst[1] = new Iload(1); // Cargar operando 2
            inst[2] = new Iadd(); // Sumar operandos 1 y 2

            for (int i = 0; i < inst.length; i++)
            {
              CompileRule cr = rset.currentRule(inst[i]);
              cr.apply(inst[i],target, sr); // target is the output file
            }

          }
    }
```

Related Patterns

- The *SR* operation in the SemanticsRule is a kind of hidden STRATEGY pattern [6].
- The *apply* operation is an example of TEMPLATE METHODS [6].
- INSTRUCTION SET and TRANSLATOR are related in the pattern language.

Pattern 5: The TRANSLATOR

Context

You have achieved a compilation stage described by the pair (IL_i, VM_i). Now, you want your compilation process makes explicit some new features and optimizations in a new pair (IL_{i+1}, VM_{i+1}). Each intermediate language defines a different instruction dictionary.

Problem

A compilation process based on the stepwise refinement of virtual machines involves the translation of a source program (executable by the origin machine) into a target program (executable by the target machine).

Forces

- For each compilation step users must provide two elements: the new target machine's data area, and the set of translation rules to apply to the source programs.
- The translation rule set collects and manages a set of translation rules.
- In order to facilitate translation-rule management, a well-suited representation for the translation rule set is needed.
- Given a request of traslating, translation-rule set must comply with the following constraints:
    - it must be exclusive, i.e., only one translation rule can be applied each time (no overlapping).

- it must be exhaustive, i.e., there exist ever a translation rule applicable.

Structure

The structure of the TRANSLATOR pattern is shown on Figure 14.

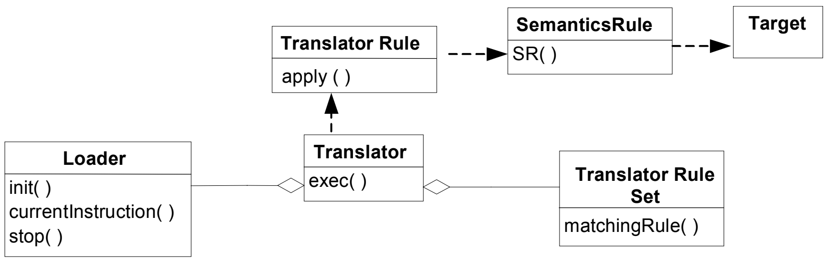


Figure 14. The TRANSLATOR pattern (structure)

Participants

- **Loader**: It provides object format Instructions from the source program (see PROGRAM LOADER pattern).
- **TranslatorRuleSet**: It defines the set of compilation rules used during the translation. Moreover, it provides the mechanism to select the right translation rule (see TRANSLATOR pattern).
- **Translator**: It acts as a MEDIATOR among the other participants of the pattern. The method *exec* coordinates the actions executed by components Loader and CompilationRuleSet, in order to obtain the target program.

Collaborations

Collaborations among TRANSLATOR' s participants are shown on Figure 15.

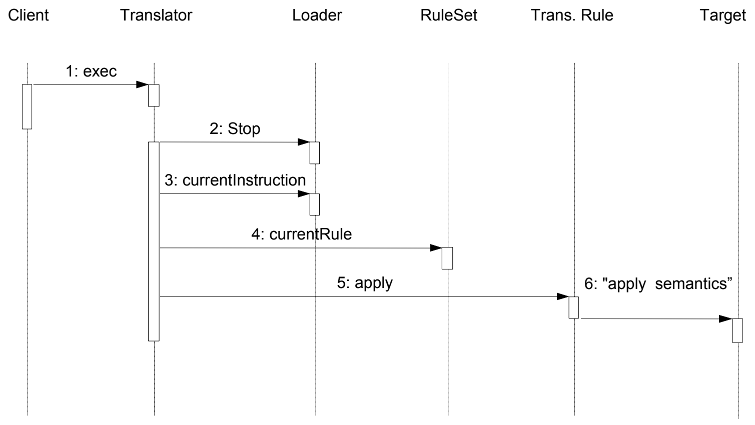


Figure 15. The TRANSLATOR pattern (collaborations)

The operation *exec* carries out the translation of a source program in a loop. The body of this loop defines the following sequence of steps:

1. Test if Loader has more origin instructions.
2. If so, then, get the next origin instruction by calling the operation *currentInstruction.*
3. Choose from the TranslationRule set the rule that matches the *current Instruction*.
4. Apply the semantics of the above rule (i.e., *SR* in a SRTranslator) for each of the targets Instructions created. As a consequence, it is created a target program (Target).

Implementation

Consider the following implementation issues:

1. The implementation of the method *exec* as a TEMPLATE METHOD.
2. *SemanticRule* is a STRATEGY pattern. It allows us to get different behaviors in the translation process. Thus, it may keep the target code in a text file (SRTranslator) or may execute the target Instructions on a virtual machine (SREmulator). See the EMULATOR pattern for the last case.

Sample Code

1. Next, it is shown the implementation of operation *exec* in the class Translator.

```
public void exec (SemanticsRule semantics, Object target)
{
    Instruction inst;
    CompileRule cr;

    while (!loader.stop())
    {

      inst = loader.currentInstruction();

      cr = rset.currentRule(inst,target);
      cr.apply(inst, target,semantics);
    }
}
```

2. The translation from a JVM program into an Intel program is presented below:

```
class PruebaTranslator
{

  public  static void main (String args[]) throws IOException
  {

    // Creation
    Loader loader;
    JVMInstructionSet set = new JVMInstructionSet();
    CheckerInst check = new CheckerInst(set);
    ConverterInst converter = new ConverterInst(set);
    SourceStream source;
    //....

    source = new SourceStream (args[0]);

    loader = new Loader(source, check,converter);
```

```
            loader.init();

            OutputStream target = OutputStream(args[1]);

            // Creation of CompilationRuleSet
            CompilationRuleSet rset = new RuleSetJVM2Intel();

            // Creation of Traslator
            Translator translator = new Translator (loader, rset);

            // Execution of Translator
            translator.exec (new SRTranslator(),target);

          }
        }
```

Related Patterns

- TRANSLATOR is an instance of the MEDIATOR pattern [6] that integrates the other participants by behavioral parameterization (in the same way as FUNCTION OBJECT [7]).
- The e*xec* operation is an example of TEMPLATE METHODs [6].
- As is used in this pattern, SemanticsRule is a clear example of STRATEGY pattern [6].
- TRANSLATION resides in the core of the pattern language and relates with the rest of patterns except INSTRUCTION SET.

Pattern 6: The EMULATOR

Context

Let's suppose we have carried out a compilation step, it is:

- We desire to add one or more optimizations in the current compilation stage (defined by an intermediate language IL_i and virtual machine VM_i.).
- we have defined a new intermediate language IL_{i+1}
- we have obtained a virtual machine VM_{i+1} from a previous one (by stepwise refinement), and
- We have defined a translation process for the source language and the target language.

Problem

How to know (or test) if the compilation step is correct (i.e., it has the effect that we expected!).

Forces

- None conjectures about how source machine and target machine are.
- During the test, the reference machine is the source machine.

Solution

Set out the execution of source code as the execution of compiled code running on the target machine, and then, rebuild the source machine's state from the target machine's state (as a kind of de-compile process). The ideas underlying this solution are detailed now:

- The execution of source code is carried out instruction by instruction
- The origin instruction to execute on the source machine is translated dynamically into target instructions. Then, these instructions are executed on the target machine, changing the machine's state.
- Rebuild the origin machine's state by using the target machine's one.

Structure

The structure of the EMULATOR pattern is shown on Figure 16.

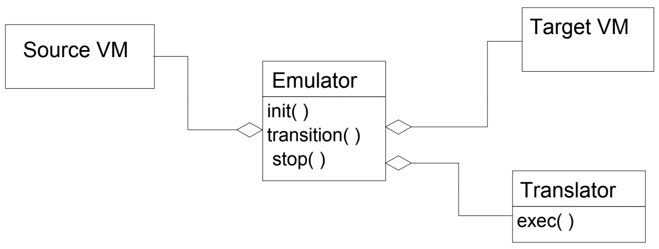


Figure 16. The EMULATOR pattern (structure)

Participants

- **Source VM**: it is the source virtual machine.
- **Target VM**: it is the target virtual machine.
- **Translator**: It is responsible of translating the source code (from Source VM) and executing the compiled code on Target VM. In this case, the Source VM's program is the Loader associated with Translator. On the other hand, it uses the SREmulator as SemanticsRule.
- **Emulator**: It acts as a MEDIATOR of the other participants of the pattern.

Collaborations

Three different scenarios describe the three different phases during the emulation: *initialization, transition* and *ending*.

<u>Scenario 1</u>: Initialization

Source VM is initialized completely (i.e., state initializing and program loading). Unlike only state initializing in the case of Target VM (see the Virtual Machine pattern).

<u>Scenario 2</u>: Transition

The Emulator starts up the Translator using a concrete emulation semantics (see the TRANSLATION RULE pattern), and it passes the execution control to it (see the TRANSLATOR pattern).

<u>Scenario 3</u>: Ending

The ending stage for the emulator is achieved when one of two (or both) machines achieve their ending condition (see the VIRTUAL MACHINE pattern).

Sample Code

The following sample code shows a Java implementation of the EMULATOR pattern for the example shown in the Figure 2 (i.e., JVM to Intel compiler).

1. The implementation of the Emulator class.

```
class Emulator
{
  public VM_State sourceVM, targetVM;
  public Translator translator;

  public Emulator (VM_State sourceVB, VM_State targetVM, RuleSet
ruleset)
  {
    this.sourceVM = sourceVM;
    this.targetVM = targetVM;
    this.translator = new Translator (sourceVM.Getprog(), ruleset) ;
  }


  public void init (InputStream target)
  {
    machine1.init (target);
    machine2.getda().init();
    machine2.getprog().init();
  }


  public  boolean stop ()
  {
    return machine1.stop ();
  }


  public void transition ()
  {
    SemanticsRule sr = new SREmulator();
    translator.exec(sr,this);
  }
 }
```

2. Concrete JVMtoIntel rules are obtained by subclassing CompileRule

```
class IaddRule extends CompileRule
```

```
{
  public  boolean matching (Instruction inst, Object target)
  { ... }
  public void createInstructions (Instruction inst, Object target)
  {
    instructions = new IntelIntruction [4];
    IntelDataArea ida = .. // Intel machine's data area

    Address add1 = ida.register[AX].getAddress(); //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
  }
}
```

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

```
class Add extends IntelInstruction
{
  public void SI (VM_State 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
  }
}
```

4. The JVMtoIntel rule set derives from the RuleSet class.

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

  public RuleSetArray ()
  {
    rule = new Rule[n_rule];
    rule[0] = new IAddRule ();
    //....
  }
}
```

5. Below is sketched a client for the JVM2Intel Compiler:

```
public class JVM2IntelCompilerClient
{
  public static void main(String args[]) throws IOException
  {
    VM_State sourceVM, targetVM
    Emulator m;
    FileInputStream fichero = new FileInputStream (args[0]);
```

```
            sourceVM = new VM_State (new JVMFactory());
            targetVM = new VM_State (new IntelFactory());

            m = new Emulator (sourceVM, targetVM, new JVMtoIntel());

            m.init (fichero);
            m.transition ();
        }
    }
```

Related Patterns

- EMULATOR is an instance of the MEDIATOR pattern [6] and integrates the other participants by both behavioral and structural parameterization.
- The *init, transition and stop* operations are an example of TEMPLATE METHODS [6].
- EMULATOR can be seen as a kind of virtual machine, where the data area is the composition of Source VM and Target VM, and the program is the translation of Source program 'on-the-fly'.
- VIRTUAL MACHINE and TRANSLATOR are related patterns in the language.

4. The Pattern Language Overview

Here, we have seen a collection of patterns that must to help developers, at least apparently, to obtain abstract compiler back-ends based on virtual machine technology.

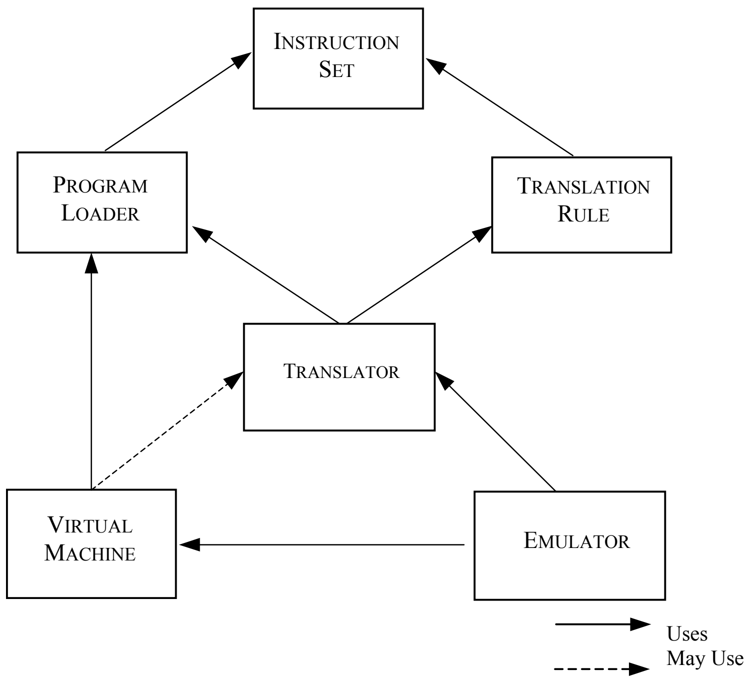


Figure 17. The *roadmap* of the EXTREM-COMPILING pattern language

However, an important task is still ahead, it is, to fix the relationships among the patterns and, afterwards, to analyze the generative and emergent effect of a pattern language. The figure 17 shows a diagram with the EXTREM-COMPILING roadmap. The diagram describes the use-relations among the patterns in the language. Relations are considered to be 'routing' constrains to obtain feasible paths. If any path makes sense in the context of compilation processes, it is called a *route* and, likely, we are in the face of having provide a pattern architecture.

a) Patterns Classification and Description

A short description of the patterns is presented in a table, below:

Pattern Name	Description
PROGRAM LOADER	*An assembler program is commonly stored in a text file, one instruction each line. However, the program execution needs the assembler code be represented in an internal object-oriented representation. Therefore, a mechanism to translate the textual representation of the program to an object representation is needed. How to realize this translation?*
VIRTUAL MACHINE	*How to define a common design template of virtual machine so that it captures the essential features addressed by virtual machines?*
INSTRUCTION SET	*A virtual machine executes instructions, which access and update the state of the virtual machine. How to store, locate and execute machine instructions?*
TRANSLATION RULE	*How to define a correct and non-ambiguous translation system between two different programming languages?*
TRANSLATOR	*Compilation based on the stepwise refinement of virtual machines involves the translation of a source program (executable by the origin machine) into a target program (executable by the target machine). How to realize this translation?*
EMULATOR	*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!)?*

b) Route Classification and Description: 3 Compilation Architectures

A short description of the pattern architectures is described below:

Architecture Name	Pattern Combination	Description
EXECUTION^{ARCH}	PROGRAM LOADER VIRTUAL MACHINE INSTRUCTION SET	*A common skeleton to construct virtual machines and execute machine programs.*
TRANSLATION^{ARCH}	PROGRAM LOADER TRANSLATION RULE TRANSLATOR INSTRUCTION SET	*A product-line architecture supporting program translation by the refinement of intermediate virtual machines..*
EMULATION^{ARCH}	PROGRAM LOADER TRANSLATION RULE TRANSLATOR EMULATOR INSTRUCTION SET	*An high-level architecture for testing program execution on virtual machines.*

c) How to Use these Patterns

Consider the following *routes* on the EXTREM-COMPILING pattern language:

- Route 1 (*Execution)*.
PROGRAM LOADER ⇒ INSTRUCTION SET ⇒ VIRTUAL MACHINE
USE: Whenever you want to execute a IL_i-program running on a VM_i.

- Route 2 *(Translation)*.
PROGRAM LOADER ⇒ INSTRUCTION SET ⇒ TRANSLATOR RULE ⇒ TRANSLATOR
USE: Whenever you want to translate a IL_i-program into a IL_{i+1}-program.

- Route 3 (*Translation First, Then Execution*).
PROGRAM LOADER ⇒ INSTRUCTION SET ⇒TRANSLATOR RULE ⇒ TRANSLATOR ⇒
PROGRAM LOADER ⇒ INSTRUCTION SET ⇒VIRTUAL MACHINE
USE: Whenever you want to execute a IL_i-program running on a VM_j $(j > i)$

- Route 4 *(Emulation)*.
PROGRAM LOADER* ⇒ INSTRUCTION SET ⇒ TRANSLATOR RULE ⇒ TRANSLATOR ⇒ EMULATOR
USE: Whenever you want to execute a IL_i-program running on a VM_j $(j < i)$

5. Conclusions

We want virtual machine technology to fill an important position in compiler construction of modern languages, similarly as declarative languages did in the past.

To this aim, in this paper we have presented the EXTREM-COMPILING pattern language, a collection of 6 patterns and 3 architectures to help developers to obtain abstract compiler back-ends based on virtual machine technology. The constant and varying parts of patterns have been clearly identified and de-coupled. Besides, we have also identified, as patterns, significant elements involved in the process of translating virtual machines.

We succeeded in applying EXTREM-COMPILING in the redefinition of a previous work [2][3][16]. The possibility of comparing two different object oriented versions has been crucial for evaluating the impact of the pattern one. The resulting framework has provided us with a formal description repository to board new developments in the same field: logic programs [16].

The EXTREM-COMPILING pattern language is still too naïve. It is reasonable to believe, then, that some of the patterns could be split, mined or refined for obtaining a more accurate set of patterns. On the other hand, it seems feasible the construction of language-dependent compilation frameworks that could be used in the context of a concrete paradigm.

As a final remark, the pattern language is not only a contribution for constructing compiler back-ends, but it should serve to improve the understanding of the process. In this sense, we make a plea for a '*extreme compiling*' initiative based on EXTREM-COMPILING.

6. Future Work

The present work is one part of a major project called *AC/DC-e*[32]. The main goal in *AC/DC-e*[6] is the definition and construction of a software framework to develop e-commerce applications. The novelty of the proposal is trying to adapt several concepts of declarative paradigms into the context of real application development.

The project is divided in several sub-projects, covering both theoretical and practical aspects, such as conceptual models, semantics and tools. More precisely, the *AC/DC-e* project outlines the following objectives:

- ❑ *Xurry* language: The Markup Declarative Language Xurry. It is a markup language describing the functional-logic language Curry [13].
- ❑ Xurry Virtual Machine (*Xurry-VM*): A Markup Virtual Machine for Xurry. An enhanced instance of VIRTUAL MACHINE in the present work.
- ❑ Xurry Java Framework for Xurry (*Xurry-JF*). A Java Framework for Xurry. It includes a repository of pre-defined Declarative Java Components and other built-ins. Every correct Xurry program can be translated into Xurry-JF program as a set of components.
- ❑ Xurry Compiler (*Xurry-C*): An enhanced instance of the EXTREM-COMPILING pattern language presented in the present work.

[6] *AC/DC-e* : Towards an Application of Declarative Concepts in Electronic Commerce [32].

***Xurry*: A Markup Declarative Language**

Let's consider the following projects as the background for Xurry.

- **The *Curry* Project**. Declarative paradigms (functional and logic) are good candidates to improve the development process of real world software. However, the attempts of integrating common mechanisms required by real applications into pure declarative languages have failed by now. *Curry* [13] is a universal programming language aiming at the amalgamation of the most important declarative programming paradigms, namely functional programming and logic programming. Curry combines in a seamless way features from functional programming (nested expressions, lazy evaluation, higher-order functions), logic programming (logical variables, partial data structures built-in search), and concurrent programming (concurrent evaluation of constraints with synchronization on logical variables). Moreover, Curry provides additional features in comparison to the pure languages (compared to functional programming: search, computing with partial information; compared to logic programming: more efficient evaluation due to the deterministic evaluation of functions). Moreover, it also amalgamates the most important operational principles developed in the area of integrated functional logic languages.
- **The RuleML Initiative**. Rule markup languages [12] allow to express business rules as modular, stand-alone units in a declarative way, and to publish them and interchange them between different systems and tools, will play an important role for facilitating business-to-customer (B2C) and business-to-business (B2B) interactions over the Web.

***Xurry-VM*: A Markup Virtual Machine for Xurry**

Let's consider the following projects as the background for *Xurry-VM*.

- **The Adaptive Object-Model Approach**. The Adaptive Object-Model approach is a proposal focused on represents classes, attributes, relationships, and behavior as *metadata*. As a consequence, Adaptive systems are based on instances rather than classes. Changes in metadata (object model) reflect changes in the system's behavior. The system stores its *Object-Model* in a repository (a database or XML files) and interprets it.
- **A UML Virtual Machine**. The work presents a virtual machine for UML [11] that interprets UML models without any intermediate code-generation step. The paper shows how to embed UML in a metalevel architecture so that a key property of model-based systems, the causal connection between models and model instances, is guaranteed. With this architecture, changes to a model have immediate effects on its execution, providing users with rapid feedback about the model's structure and behavior.
- **The Virtual Virtual Machine Project**. The Virtual Virtual Machine [14] is a multi-language, hardware independent execution platform. The VVM offers both a programming and an execution environment allowing, to adapt the execution environment (language + system) to a given domain, to be extensible by changing on the fly the execution environment (adding new functionality, algorithms, or upgrading the hardware), to promote interoperability between applications.

***Xurry-JF* : A Java Framework for Xurry**

Let's consider the following projects as the background for *Xurry-JF*.

- **The Service Provider Framework**. The Service Provider domain pattern [15] is the result of trying to unearth commonalties and mine patterns across several industrial projects. The result of this effort was what was named the Java Business Frameworks (JBF). Its foundation lies on the meta-domain pattern called Service Provider. Service Provider contains many smaller patterns.
- **The SGPD Project**: A Document Management System for WWW-Publishing.

Xurry-Compiler: A Markup Pattern Language for Compiler Construction

Let's consider the following projects as the background for *Xurry-Curry*.

- The ExTrEm-Compiling pattern language**.**
- Some previous works on this topic or similar [4][16][17].

References

[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 Algorithmic 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] T. Kuehne. A Functional Pattern System for Object-Oriented Design. (PhD) University of KaisersLutern. URL: http://www.agce.informatik.uni-kl.de/~kuehne .

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

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

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

[11] D. Riehle, S. Fraleigh, D. Bucka-Lassen, N. Omorogbe. "The Architecture of a UML Virtual Machine". Proceedings of the 2001 Conference on Object-Oriented Program Systems, Languages and Applications (OOPSLA '01), October 2001.

[12] RuleML Initiative. http://www.dfki.de/ruleml.

[13] Curry Project: http://www.informatik.uni-kiel.de/~curry/.

[14] Folliot, Bertil; Piumarta, Ian; Ricardi Fabio. A Dynamically Configurable, Multi-language Execution Platform. Proceedings of the ACM SIGOPS Workshop, Sintra, Portugal, Sept. 1998.

[15] Ali Arsanjani. "Service Provider: A Domain Pattern and Its Business Framework Implementation," presented to PloP '99.

[16] García J. & Sutil M. A Pattern-based Formal Definition of the Abstract Prolog Compiler. Technical Report FI-UPM TR2202. July 2002.
[17] García J. & Sutil M. Virtual Machines & Abstract Compilers. 6th. Annual Conference on the Pattern Languages of Programs (EUROPLOP'00), Irse, Germany, July 2000.
[18] *Virtual Library on Logic Programming:* http://www.comlab.ox.ac.uk/archive/logic-prog.html
[19] *Virtual Library on Functional Programming* http://www.cs.nott.ac.uk/~gmh/faq.html
[20] *Virtual Library on Logic Functional Programming* http://www.informatik.uni-kiel.de/~mh/FLP/
[21] 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
[22] Hannan, J. *Operatinal Semantics-Directed Compilers and Machine Architectures,* ACM Transactions on Programming Languages and Systems, Vol. 16, No. 4, July, 1994
[23] Bosch, J. *Parser Delegation - An Object-Oriented Approach to Parsing.* In Proceedings of TOOLS Europe'95, 1995.
[24] Lindholm, T. Yellin, F. *The JavaTM Virtual Machine Specification. S*un Microsystems, Inc.
[25] Haskell Project: http://www.haskell.org/.
[26] Russinoff, M.D. *A VeriEd Prolog Compiler for the Warren Abstract Machine*, Journal of Logic Programming 13, 367-412. 1992.
[27] Ait-Kaci, H. *Warren's Abstract Machine. A Tutorial Reconstruction*. MIT Press 1991.
[28] E. Böerger, D. Rosenzweig: *The WAM -- Definition and Compiler Correctness*, Technical Report TR 14/92, Dipartamento di Informatica, Universita di Pisa, Italy, 1992.
[29] D. Maier, D.S. Warren: *Computing with Logic: Logic Programming with PROLOG*, Ed. Benjamin Cummings, 1988
[30] P. Kursawe: *How to Invent a PROLOG Machine*, New Generation Comp., 5, 1989.
[31] J. Gabriel, T. Lindholm, E.L. Lusk, R.A. Overbeck: *A Tutorial for the WAM for Computational Logic*, ANL-64-84, Argonne Nat. Lab., 1985
[32] AC/DC-e Project : *A*plicación de *C*onceptos *D*eclarativos al Comercio *E*lectrónico (Spanish Project TIC2001-2798). http://lml.ls.fi.upm.es/ACDC
[33] The Publishing-21 Project: A Document Management System for WWW-Publishing. http://www.worldnet21.es.