Chapter 6

Byte Code Transformation Framework

This chapter proposes a bytecode transformation framework, combining previously described technologies. The intention is to show MDA’s power to embrace different technologies and to give a non-trivial use case scenario. Most MDA examples are targeted to help understanding what MDA really is and therefore, are very simple. Thus, cross-compilation is chosen to show practical application of MDA.

Virtual machines, in this case JVM and CLR, are chosen as the execution platforms because they are the most prevalent and complete solutions for enterprise computing. The original intention, given the dissemination of JVM, is to enable mapping of various CLR aspects to the JVM world.

XMLVM has an important role in the proposed framework. Thanks to its flexibility its purpose is twofold. First is to represent instances of UML models, carrying all necessary bytecode data, and second is to replace UML, acting as a domain specific modelling language (DSM).

Next few sections will discuss overall architecture of the framework and its important parts. Using the usual MDA approach, first modeling results will be discussed followed by their transformations.
6.1 Framework Architecture

Figure 6.1: Architecture of Byte Code Framework

Model repository

Transformation repository

Transformation tools

XMLVM programs

Architecture of the overall framework is given in Figure 6.1 with its four essential parts:

1. Model repository
2. Transformation repository
3. XMLVM program repository
4. Transformation tools

Model repository. It stores all necessary models for the framework to use. Those are PIM\textsubscript{xmlvm}, PSM\textsubscript{xmlvm-jvm}, PSM\textsubscript{xmlvm-clr}, XMLVM\textsubscript{meta} and JVM\textsubscript{uml}. PIM\textsubscript{xmlvm} is the UML representation of XMLVM’s structure from Figure 5.1. It is present because of the possible future extensions. As it was described, XMLVM can be easily extended to encompass more languages than it supports now. Therefore, PIM\textsubscript{xmlvm} is needed to make this framework flexible since for each new language, transformation definition can be written to derive PSM from PIM\textsubscript{xmlvm} for that particular language.
CHAPTER 6. BYTE CODE TRANSFORMATION FRAMEWORK

Both, \( \text{PSM}_{\text{xmllvm-jvm}} \) and \( \text{PSM}_{\text{xmllvm-clr}} \), represent models of JVM and CLR bytecode languages. They are also created using UML and beside elements from PIM\(_{\text{xmllvm}}\), they contain their respective bytecode specific elements. Since PIM\(_{\text{xmllvm}}\), PSM\(_{\text{xmllvm-jvm}}\) and PSM\(_{\text{xmllvm-clr}}\) are created using UML, they share a common metamodel - UML metamodel. As it will be described later, only PIM\(_{\text{xmllvm}}\) has to be created manually and both PSMs can be derived automatically using appropriate transformations.

\( \text{XMLVMM}_{\text{meta}} \) is a special metamodel, which defines domain specific language (DSL). It is direct descendant of MOF and it resides on level M2, together with UML metamodel. Since it has nothing to do with UML metamodel, except the same level of abstraction, this approach is rather different from what UML community is used to. Instead of using elements of UML metamodel to create models on level M1, elements of XMLVMM\(_{\text{meta}}\) can be used to create those models. While UML is general purpose modeling language and can be used to model everything, DSL can not. As its name says, they are only used for the special domain. In that context, XMLVMM\(_{\text{meta}}\) can only be used to model bytecode programs. Advantages of this approach will be discussed later in this chapter, comparing it with commonly used UML modelling.

This framework is built with a premise to support static transformation of bytecode programs. Static in this case refers to a transformation of a structure of a program which is not in execution. Another scenario, transformation of programs in execution, is also possible. In order to do that, beside structure of a program, information about underlying virtual machine and its states is necessary. Thus, JVM\(_{\text{uml}}\) model is created, which basically represents Java virtual machine.

**Transformation repository.** This repository contains all transformation definitions, necessary to perform changes to models. Exactly three different definitions are made:

1. PIM\(_{\text{xmllvm}}\) to PSM\(_{\text{xmllvm-jvm}}\)
2. PIM\(_{\text{xmllvm}}\) to PSM\(_{\text{xmllvm-clr}}\)
3. PSM\(_{\text{xmllvm-clr}}\) to PSM\(_{\text{xmllvm-jvm}}\)
While both, QVT Relations and QVT Operational languages, are used to define transformation from PSM$_{xmlvm-clr}$ to PSM$_{xmlvm-jvm}$, only QVT Relations is used to define transformations from PIM$_{xmlvm}$ to PSMs. Transformations from PIM$_{xmlvm}$ are defined on UML metamodel level, M2, and are executed on level of UML models, M1. On the other hand, PSM$_{xmlvm-clr}$ to PSM$_{xmlvm-jvm}$ transformation is defined on level M1, where UML models are, and is executed on level M0, where instances of PSMs are.

**XML VM program repository.** This repository is meant to store XML VM programs that are instances of their respective models. In the process of transformation, XML VM programs are M0 fragments that are actually transformed. However, this is not true when XML VM meta DSL is used to define models. As it will be described later, in this case XML VM programs are on level M1. Here it is important to note that this framework does not follow the usual model driven approach, where the code to be executed is generated from models. Although models are created using MOF layers, code for this framework is generated using XML VM’s input compilers.

**Transformation tools.** In order to execute transformation some tool is needed. Basically, tool should act as a compiler or interpreter, translating a given transformation definition into an executable form. Unfortunately, there are not so many tools that support QVT specification. This is mainly due to the fact that QVT is still young. Another problem is that vendors are not providing 100% QVT-conformant implementations but rather QVT-like solutions, that sometimes use very different languages to define transformations. In order to leave theory domain and to make practical use of the concepts described in this thesis two solutions are used: 1) smartQVT [24] and 2) mediniQVT [25].

### 6.2 Model Repository

There are two possibilities for creating models. First one is to use UML. It is general purpose modeling language (GPML) based on well established standards. It can be applied to wide variety of problems across a broad range of domains. This is a common approach, used by majority of the MDA community.
Second one is to define own modeling language for a special domain, which is known as Domain Specific Modeling Language (DSML). In contrast to GPML, DSML can be only used to describe problems in one narrow domain. Examples of such languages are Business Process Modeling Language [26], used to model business processes, and Himalia [27], used to model user interfaces for programs.

This framework covers both approaches. However, only UML solution is used in conjunction with QVT transformations because available tools only support UML as the modeling language.

### 6.2.1 Modeling with UML

In this approach all models are derived from UML metamodel, meaning that PIM$_{xmlvm}$, PSM$_{xmlvm-jvm}$ and PSM$_{xmlvm-clr}$ are its direct descendants. Figure 6.2 shows this hierarchy. By using UML metamodel elements, UML Class, UML Attribute and UML Association, various models can be built.

![Figure 6.2: UML models](image)

As it was mentioned, PIM$_{xmlvm}$ is present as the UML model of XMLVM’s abstract structure from Listing 4.1. It is not necessary but its presence is beneficial from few points. It enables easy realization to PSMs via transformations. In few lines of code, complete structure of PIM$_{xmlvm}$ can be replicated to any PSM and new elements can be added as well. Without PIM$_{xmlvm}$, both PSM$_{xmlvm-jvm}$ and PSM$_{xmlvm-clr}$, would have to be created manually. Since both PSMs are large models, with more than 200 elements...
each, lot of time would be needed to create them manually and not to mention the possibility of errors. Beside it lowers the amount of work and increases the speed of PSMs generation, $\text{PIM}_{\text{xmlvm}}$ makes sure that all derived models follow the original structure of XMLVM programs, contained in the $\text{PIM}_{\text{xmlvm}}$ itself.

Figure 6.3 illustrates $\text{PIM}_{\text{xmlvm}}$. In order to continue naming convention used in XMLVM all elements are placed in $\text{vm}$ package. Since all elements were explained in the section 5.2 there is no need to repeat explanations here again. Maybe the information about element associations should be given since they are not explicitly present in XMLVM. Here a \textit{composition}, which is a form of aggregation, is chosen since it best reflects associations between elements of XMLVM program. Composition means that one element is composed of two or more other elements, and that this element is responsible for their creation and removal. For example, every class of a program is responsible for creation and removal of its fields and methods from memory. Similarly, when a method is removed from stack, its belonging instructions and data are also removed.

Next important model is $\text{PSM}_{\text{xmlvm-clr}}$, realization of $\text{PIM}_{\text{xmlvm}}$ for CLR. Beside common elements from $\text{vm}$ package, it also has elements from $\text{clr}$ and $\text{dfa}$ packages. $\text{clr}$ package contains CIL instructions, such as \textit{stloc}, \textit{ldc}, \textit{ldloc} etc., and other possible members of \texttt{vm::code}. $\text{dfa}$ package has elements related to the data flow analysis described earlier.
Figure 6.4 shows how all these elements are put together to form PSM\textsubscript{xmlvm-cil}. It follows the original PIM\textsubscript{xmlvm} structure with additional elements associated to \texttt{Code} class. One important difference is that a new element, named \texttt{clrelem}, is added. It is unavailable in the original XMLVM programs. It represents a base class for all CIL instructions, labels and variables. As it will be described later, it was necessary to introduce \texttt{clrelem} due to some inefficiencies of available tools.
PSM$_\text{xmlvm-jvm}$ is very similar to PSM$_\text{xmlvm-clr}$ except it has JBC related instructions from jvm package. Since data flow analysis is not necessary for JBC, dfa package is not included in this model. Figure 6.5 shows PSM$_\text{xmlvm-jvm}$ model.
6.2.2 Creating Domain Specific Modeling Language

Creation of DSML demands modeling using elements of the highest level of MOF hierarchy. MOF constructs, such as *MOF Class*, *MOF Association*, *MOF Attribute*, *MOF Data Type*, *MOF Operation*, etc., can be used to define elements of M2 metamodel, which is a metamodel for DSML. Purpose of this metamodel is to formally express key aspects of bytecode programs. This task is not an easy one since it demands creation of abstract and concrete language syntaxes and static semantics [29].

Without any pretension to show complete solution of DSML, this section will discuss simple metamodel, XMLVM$_{meta}$, used to describe how XMLVM can be used as a DSML and thereat, focusing on how DSML metamodel can be created and what are the differences with UML approach.

Figure 6.6 shows metamodel for XMLVM DSML. It resembles PIM$_{xmlvm}$ because they share the same base, which is the abstract structure of XMLVM language from Listing 4.1. Thanks to this fact, XMLVM language can be used for different purposes on more than one level of MOF 4-layer architecture which makes proposed framework very flexible. It can be used as an M0 fragment, carrying real data, or as an M1 model, enabling dynamic bytecode transformation.
CHAPTER 6. BYTE CODE TRANSFORMATION FRAMEWORK

Figure 6.6: XMLVM meta

XMLVMContainer
- name: string
  1
  1..*

XMLVMClass
- name: string
- package: string
- extends: XMLVMClass

0..*

XMLVMField
- name: string
- isPublic: bool
- isProtected: bool
- isPrivate: bool
- type: object

1

XMLVMMethod
- name: string
- isPublic: bool
- isProtected: bool
- isPrivate: bool
- access: int
- locals: int

1

XMLVMSignature
- classType: XMLVMClass
  0..1

XMLVMParameter
- type: object
  0..*

XMLVMBasicBlock
- type: object

1

XMLVM_insn
- index: int
- code: int

1

XMLVMCodeElement

1

XMLVMCode

1

XMLVMElement

1

XMLVMStackElement
- type: object
- pushedBy: int

0..*
Although XMLVM\textsubscript{meta} borrows syntax from UML it would be better if completely new syntax could be defined. In that case \texttt{Class} could be represented by square, \texttt{Method} by circle, \texttt{Field} by diamond and so on.

Figure 6.7: Connection between XMLVM DSML and MOF

Figure 6.7 shows how MOF constructs are actually used to build XMLVM\textsubscript{meta}. For example, MOF Class is the base element for XMLVM\texttt{Class} and XMLVM\texttt{Method}. All attributes they contain are instances of MOF Attribute. Similarly, association between XMLVM\texttt{Class} and XMLVM\texttt{Method} is an instance of MOF Association.

XMLVM DSML can be used to model PIM\texttt{xmlvm}, PSM\texttt{xmlvm-jvm} and PSM\texttt{xmlvm-clr} in the same way as UML can be used. In this case, created model would have elements of XMLVM\textsubscript{meta} and would resemble XMLVM\textsubscript{meta} itself very much. It is more interesting that now programs can be created in a modeling process. Instead of writing code in Java or C\# and then compiling it to JBC, CIL and XMLVM, they can be modeled using XMLVM DSML and then directly compiled to XMLVM language (Figure 6.8 level M1).
Another benefit, gained by introducing XMLVM\textsubscript{meta} on level M2, is possibility to transform bytecode in execution. XMLVM programs, are now fragments of level M1 instead of level M0. In this case, level M0 can be treated as a program in execution (Figure 6.8, level M0). In order to achieve this, additional information about underlying virtual machine is necessary. Because XMLVM\textsubscript{meta} does not have all the necessary elements to describe bytecode in execution, new model of underlying virtual machine must be created and used in conjunction with XMLVM\textsubscript{meta}. Solution will be presented later in this chapter.
6.3 Transformation Repository

QVT Relations and QVT Operational languages are used to define transformations in this repository. While PIM to PSM transformations are defined using QVT Relations only, mapping from CIL to JBC is defined using both languages. The reason for this is to give comparative overview of both languages using non-trivial use case scenario.

Only UML models from model repository are used with transformations. XMLVM DSML is not due to certain issues with available tools. However, this should not be considered as a disadvantage since these transformations would look very similar to those used with UML models. Only the name of variables and model elements would be different, while semantic would be the same.

6.3.1 PIM Mappings

There are two mappings defined on PIM$_{xmlvm}$, each used to generate different PSMs for CLR and JVM bytecodes. Since both mappings are not complex, QVT Relations is used to define them. However, this does not mean that QVT Relations is not powerful enough to describe more complex use cases.

Figure 6.9: PIM transformations in MOF architecture
Figure 6.9 shows how mappings are linked to models in 4-layer architecture. They are defined on the same metamodel which makes them endogenous [20]. The goal of the transformations is refinement of PIM\textsubscript{xmlvm}, including more concrete details and generating PSMs capable to describe their bytecodes.

Unusual is that transformations are defined on M2 level, which is perfect illustration of how MDA flexible is. Beside it allows transformation of 'live' data on level M0, it allows transformation of models as well. This means that models can be automatically manipulated rather than manually. Instead of using tools to manually create or alter their models, designers can write transformations, covering many model elements with few mapping rules. Similarly, transformations can be defined on MOF level and executed on level of meta-models, M2. This would eventually lead to scenario where languages are created by other languages.

Transformation of PIM\textsubscript{xmlvm} to PSM\textsubscript{xmlvm-jvm} and PSM\textsubscript{xmlvm-clr} can be divided in three parts, where each part is done using different relations:

1. Replication of common elements from PIM\textsubscript{xmlvm}
2. Creation of associations between generated elements
3. Creation of bytecode specific elements

Listing 6.1 can introduce first line of QVT Relations code. transformation keyword is used to mark the beginning of transformation definition and to encapsulate all defined relations between model elements. In this case, name is PIM2PSMjvm, meaning that transformation is used to generate PSM\textsubscript{xmlvm-jvm}. In order to save space, only transformation to PSM\textsubscript{xmlvm-jvm} will be examined. Complete transformation to PSM\textsubscript{xmlvm-clr} is in Appendix A.

Listing 6.1: Transformation definition in QVT Relations

```plaintext
1 transformation PIM2PSMjvm(pim : uml, psmjvm : uml) {
2 ...
3 }
```

Next important parts are parameters. uml is the name of UML metamodel used in this transformation, which is persisted in XMI document. pim and psmjvm are variables of uml type, used to declare domains for relations by referencing elements of the UML metamodel.
In order to properly map elements from PIM$_{xmlvm}$ to PSMs, package hierarchy must be maintained. Therefore, it is necessary to first map all packages since they are containers for the rest of model elements.

Listing 6.2: Relating model packages

```java
1 top relation PIMPackage2PSM_xmlvm_jvmPackages {
2 varName : String;
3 checkonly domain pim pim_package : uml::Package {
4 name = varName
5 };
6 enforce domain psmjvm psm_vm_package : uml::Package {
7 name = varName
8 };
9 enforce domain psmjvm psm_jvm_package : uml::Package {
10 name = 'jvm'
11 };
12 where {
13 replicatePIMClasses(pim_package, psm_vm_package);
14 createJVMCodeElements(psm_jvm_package, psm_vm_package);
15 }
16 }
```

Listing 6.2 shows relation, which copies vm package to the generated model, and at the same time, it creates a new one, named jvm. This relation is marked with a keyword top, which means that relation must always hold in order for the whole transformation to be successfully executed. It relates elements from two, earlier defined domains, pim and psmjvm. In this case pim is marked as a checkonly domain, which means that it will be only checked for the existence of elements of type uml::Package. If they are not present in the model, relation will not hold and whole transformation will fail. On the other hand, psmjvm is marked with enforce keyword. This means that elements of type uml::Package will be created in target model if they are not present.

According to code in Listing 6.2, two packages will be created. One is the copy of existing vm package in PIM$_{xmlvm}$, and the other one is newly generated jvm. Variable varName is used here to copy the name of vm package. In the line 4, varName is bounded with the name attribute of the vm package and later, in the line 7, it is copied to the attribute of the newly generated vm package in the domain psmjvm. Transformation tool, which executes this transformation, can decide whether to copy from attribute to a variable or vice versa, based on the transformation direction.

Finally, this relation defines a condition that must be satisfied in order for this relation to hold. In QVT Relations, these additional conditions are
placed in where part of the relation. Usually where clause contains simple OCL expressions that must be evaluated as true but it can be also used to make chain of relations, like it is done here. If a relation contains calls to other relations in its where clause, it will initiate evaluation of the called relations as well. This is similar to making expressions using Boolean algebra, where variables are connected with AND operator. Expression is only true if all variables are true and same is with relations. Chain of relation will be evaluated as true only if all relations hold.

When packages are generated, their content can be generated. The purpose of the relation, given in Listing 6.3, is to replicate all classes from the vm package to the same package in the psmjvm domain. This relation is not marked as top but it must hold in order for the whole transformation to be successful, since it is called from the top relation. It is also defined on uml packages but this time classes within packages are also included.

Listing 6.3: Replicating PIM classes

```java
relation replicatePIMClasses {
  varClassName : String;
  checkonly domain pim pim_package : uml::Package {
    packagedElement = pim_class : uml::Class {
      name = varClassName
    }
  };
  enforce domain psmjvm psm_vm_package : uml::Package {
    packagedElement = psm_class : uml::Class {
      name = varClassName
    }
  };
  where {
    replicatePIMAttributes(pim_class, psm_class, psm_vm_package);
  }
}
```

When replicatePIMClasses relation is is called from where clause of PIM-Package2PSM_xmlvm_jvmPackages relation (Listing 6.3 line 13), parameters of already related packages are passed. This practically means that only those packages that satisfy PIM-Package2PSM_xmlvm_jvmPackages relation, should be examined in replicatePIMClasses relation. Other packages that may exist, are excluded and no classes are generated for them. Variable varClassName ensures that all classes in newly generated PSM_xmlvm_jvm have same name as their corresponding classes in PIM_xmlvm.
Listing 6.4: Replicating attributes

relation replicatePIMAttributes {
  varAttributeName : String;
  checkonly domain pim pim_class : uml::Class {
    ownedAttribute = pim_prop : uml::Property {
      name = varAttributeName
    }
  }
  enforce domain psmjvm psm_class : uml::Class {
    ownedAttribute = psm_prop : uml::Property {
      name = varAttributeName
    }
  }
  primitive domain owning_package : uml::Package;
  where {
    if pim_prop.association.oclIsUndefined()
      then true
    else createAssociation(psm_prop, owning_package, pim_prop, psm_class)
  }
}

Relation in Listing 6.4 copies all belonging attributes of classes, bounded by replicatePIMClasses relation. Similarly like in previous relation, only to those classes that satisfy replicatePIMClasses their attributes will be copied.

This relation uses two features of QVT Relations language not mentioned here before. First one is the use of primitive domain (Listing 6.4, line 13), which is used to pass variables between different relations. It has all qualities of regular domains with the difference that it can not be generated in the target model. In this case, primitive domain is used to pass reference on vm package in the target model. Second feature is the use of OCL constructs to make additional constraints (Listing 6.4, lines 15-18). Here, OCL is used to determine whether class attribute is association or not, by examining its association property (Listing 6.4, lines 15). If association is not undefined, meaning that attribute is an association, relation called createAssociation is examined. Although it is more verbose, since more variables and domains are involved, createAssociation is very similar to the previous relation. Its completion, together with previously described relations, means the success of first two phases.

The last phase, generation of JBC specific elements, is done inside createJVMCodeElements relation, given in Listing 6.5. The complete code, together with previous transformations, is given in Appendix A.
Listing 6.5: Creation of JBC related elements

```java
relation createJVMCodeElements {
  enforce domain psm/jvm psm_jvm_package : uml::Package {
    Package psm_jvm_package = psm_jvm_package;
    Package jvmel = jvmel;
    Package code_class = code_class;
  }
},
  Package load = load;
  Package store = store;
  Package gen1 = gen1;
  Package gen2 = gen2;
} where {
  var CodeClass = getClassName('Code', psm_vm_package);
  bindInstructions2Code(psm_vm_package, varCodeClass);
}
```

Here, first `Jvmelem` is created, which is a base class for all classes that represent JBC instructions (Listing 6.5 lines 3-5). After that, classes that represent JBC instructions are generated, each containing generalization association with `Jvmelem` class (Listing 6.5 lines 8, 14). Finally, in order to associate generated classes from `jvm` package to those in `vm`, `bindInstructions2Code` is called, associating `Jvmelem` class to the `Code` class.

### 6.3.2 PSM\textsubscript{xmlvm-clr} to PSM\textsubscript{xmlvm-jvm}

This is the main mapping in framework, which realizes the most important purpose of the proposed framework. Two implementations, using QVT Relations and QVT Operational languages, are provided in order to show comparable view of both languages.
Figure 6.10 shows how mapping is defined and executed. It is *exogenous* mapping, meaning that it is defined on two different metamodels. On level M1 are models generated from PIM_{xmlvm}, described earlier in this chapter. XMLVM programs are on level M0 and actual mapping is executed directly on them. The goal is to enable code migration from CIL to JBC and to maintain the same level of abstraction.

Three mapping use-cases are implemented using QVT languages. These are mapping of arithmetical operations, value types and delegates. Beside the fact that they show non-trivial QVT mapping implementations, they highlight the most important differences between CIL and JBC. Their original description and solution in XSLT is given in [14].

### 6.3.3 Pre-processing and Post-processing of XMLVM Programs

Before XMLVM programs can be used in MDA tools, a simple pre-processing of underlying XML documents is necessary. The problem is that MDA demands XMI [26] format to be used for fragments on level M0. In order to transform XML, which is used by XMLVM, simple XSLT stylesheet is used to add necessary namespace, xmlns:xmi="http://schema.omg.org/spec/XMI/2.1", and attribute, xmi:version="2.1", which declares that document has XMI format.

Some names, used in XMLVM, are reserved keywords in QVT Operational language and therefore, it is necessary to change them. This is another task, performed by XSL transformation. Names that can not be used in
QVT Operational are class, extends, package and return. In order to avoid overlapping with reserved keywords they are changed as follows: class to x_class, extends to c_extends, package to c_package and return to x_return.

One more syntactical change is performed by this XSLT due to imperfection of the used tools. Child elements of vm:code cannot be serialized properly although they are transformed how they should be. To be more precise, serialization mechanism of used tools is sorting the child elements of the vm:code, which produces semantically different result.

Listing 6.6: Improper serialization of XMLVM

```xml
<vm:code>
  <jvm:iadd/>
  <jvm:iload index="1"/>
  <jvm:iload index="2"/>
  <jvm:istore index="1"/>
  <jvm:istore index="2"/>
  <jvm:istore index="3"/>
  <jvm:ldc type="int" value="19" oid="4"/>
  <jvm:ldc type="int" value="81" oid="6"/>
  <jvm:return/>
  <jvm:var id="0" name="this" type="Numeric"/>
  <jvm:var id="1" type="int"/>
  <jvm:var id="2" type="int"/>
  <jvm:var id="3" type="int"/>
</vm:code>
```

Listing 6.6 shows the result of improper serialization of the program given in Listing 4.5 which is the result of the transformation of the program given in Listing 4.2. As it can be seen, all instruction elements are present but they are sorted alphabetically. This doesn’t make sense since instructions are accessing the stack in order which is not semantically the same as the order implied by the original sequence of CIL instructions. Beside that, this sequence will produce error in more than one place. For example, iadd can not access empty stack (line 2) and neither can istore (line 7).

In order to solve the problem, XMLVM program in the Listing 4.5 must be made immune to described serialization problem. One way to achieve that is to add two new elements to clr and jvm packages. Those are clrelem and jvmelem, where clrelem is a superclass of all possible child elements of vm:code for CIL and jvmelem does the same for JBC. By doing so, serializer can sort elements but that will not affect the semantic of the bytecode since elements are described with generic name clrelem or jvmelem.
Listing 6.7: Pre-processed XMLVM

```xml
<vm:code>
  <clr:clrelem xsi:type="clr:var" id="0" type="int"/>
  <clr:clrelem xsi:type="clr:var" id="1" type="int"/>
  <clr:clrelem xsi:type="clr:var" id="2" type="int"/>
  <clr:clrelem xsi:type="clr:ldc" type="int" value="19"/>
  <clr:clrelem xsi:type="clr:stloc" index="0"/>
  <clr:clrelem xsi:type="clr:ldc" type="int" value="81"/>
  <clr:clrelem xsi:type="clr:stloc" index="1"/>
  <clr:clrelem xsi:type="clr:ldloc" index="0"/>
  <clr:clrelem xsi:type="clr:ldloc" index="1"/>
  <clr:clrelem xsi:type="clr:add"/>
  <clr:clrelem xsi:type="clr:stloc" index="2"/>
  <clr:clrelem xsi:type="clr:return"/>
</vm:code>
```

Listing 6.7 shows XMLVM program from Listing 4.2 after it has been preprocessed. Rather than being elements, actual bytecode instructions are now values of the attribute `xsi:type`, which contains information about derived types of the actual element. Mapping is now defined on `clrelem` and actual underlying type, encoded as a value of `xsi:type` attribute, is determined at the runtime. This means that for each `clrelem` a `jvmelem` will be produced after transformation, containing corresponding JBC instructions in its `xsi:type` attribute. The result is showed in Listing 6.8.

Listing 6.8: Generated XMLVM\textsubscript{jvm} after transformation

```xml
<jvm:code>
  <jvm:jvmelem xsi:type="jvm:var" id="0" name="this" type="Numeric"/>
  <jvm:jvmelem xsi:type="jvm:var" id="1" type="int"/>
  <jvm:jvmelem xsi:type="jvm:var" id="2" type="int"/>
  <jvm:jvmelem xsi:type="jvm:var" id="3" type="int"/>
  <jvm:jvmelem xsi:type="jvm:ldc" type="int" value="19" oid="4"/>
  <jvm:jvmelem xsi:type="jvm:istore" index="1"/>
  <jvm:jvmelem xsi:type="jvm:ldc" type="int" value="81" oid="6"/>
  <jvm:jvmelem xsi:type="jvm:istore" index="2"/>
  <jvm:jvmelem xsi:type="jvm:iaload" index="1"/>
  <jvm:jvmelem xsi:type="jvm:iaload" index="2"/>
  <jvm:jvmelem xsi:type="jvm:add"/>
  <jvm:jvmelem xsi:type="jvm:istore" index="3"/>
  <jvm:jvmelem xsi:type="jvm:return"/>
</jvm:code>
```

Generated XMLVM\textsubscript{jvm} from Listing 6.8 can not be used to produce JBC, since it is not original XMLVM format. It must be returned to its original format before using it with output compilers. To do that another post-processing XSL transformation is used. Basically, this transformation makes reverse changes, deleting XMI namespaces, renaming attributes to their original form and restoring JBC instructions on element positions. Its implementation, together with pre-processing script, is given in Appendix B.
6.3.4 Mapping Class-Structure

The first clear difference between the two used QVT languages is in the declaration of transformation (Listing 6.9). While QVT Relations (QVTR) does not state which model is input and which is output, QVT Operational (QVTO) must use `in` and `out` keywords to make the difference. This is due to the fact that QVTO transformations are unidirectional.

Each QVTO transformation must have one entry point function called `main`, which is very similar to standard programming languages. `main` function can have one or more mapping calls. As it was described earlier, QVTR transformation is consisted of one or more bi-directional relations, but QVTO transformation has mappings that are unidirectional. Mappings are very similar to functions in standard programming languages.

Listing 6.10 shows QVTO mapping and QVTR relation used to map XMLVM container from XMLVM\_clr to XMLVM\_jvm. Same like QVTR, QVTO mapping support direct invocation of another mappings. QVTO uses `->map` construct to connect element with its mapping. In this case it means that all classes inside XMLVM container will be mapped using `class_to_class` mapping.

QVTO does not make distinction between mapping with high priority like QVTR does, using `top` keyword to prioritize relations. The way to be sure that a QVTO mapping will be executed no matter what, is to put it in the `main` function or to call it from a mapping which is called from the `main` function, making similar effect like QVTR chain of relations.
Listing 6.11: QVTO mapping and QVTR relation for classes

```java
// QVTO
mapping vm::Class::class_to_class() : vm::Class {
    name := self.name;
    c_package := self.c_package;
    c_extends := self.c_extends;
    ...
}

// QVTR
top relation class2class {
    varName, varPackage, varExtends : String;
    checkonly domain clr_model clr : vm::xmlvm {
        x_class = class1 : vm::Class {
            name = varName,
            c_package = varPackage,
            c_extends = varExtends
        }
    }
    enforce domain jvm_model java : vm::xmlvm {
        x_class = class2 : vm::Class {
            name = varName,
            c_package = varPackage,
            c_extends = varExtends
        }
    }
    ...
}
```

Listing 6.11 shows part of the previously mentioned QVTO class mapping, together with QVTR relation. It is noticeable that QVTR is much more verbose. For simple mapping, QVTR demands lot of code. One reason is that QVTR must maintain parent-child relation between model elements to enable complex pattern matching. For example, in line 10, xmlvm container is repeated again only to constrain relation of classes that are inside it (line 11). Another reason is that QVTR has less language constructs than QVTO.

Transformation of other class elements, fields and methods is very similar and it will not be described here. For complete definition of both types of transformation see Appendix C.

### 6.3.5 Mapping Arithmetical Operations

Simple arithmetic operations are addition, subtraction, multiplication and division of numbers. CIL and JBC programs that implement these operations are very simple. However, mapping from CIL to JBC is not due to the fact that CIL has un-typed instructions. Because of this data flow analysis, which was described earlier, must be performed.
CHAPTER 6. BYTE CODE TRANSFORMATION FRAMEWORK

Listing 6.12: XMLVM

```xml
<vm:code>
  <clr:var id="0" type="int"/>
  <clr:var id="1" type="int"/>
  <clr:ldc type="int" value="2"/>
  <clr:stloc index="0"/>
  <clr:ldc type="int" value="3"/>
  <clr:stloc index="1"/>
  <clr:ldlloc index="0"/>
  <clr:ldlloc index="1"/>
  <clr:add/>
  <clr:call has-this="false" class-type="System.Console" method="WriteLine">
    <vm:signature>
      <vm:return type="void"/>
      <vm:parameter type="int"/>
    </vm:signature>
  </clr:call>
  <clr:return/>
</vm:code>
```

Listing 6.12 shows simple XMLVM program that can be used as an example for this kind of mapping. Two variables are declared (lines 2, 3) and instantiated (lines 4-7), their values are added (line 10) and the result is written on the output (line 11-16).

Since this program multiplies two integers, all un-typed instructions from CIL must be mapped to corresponding JBC instructions that operate on integer data. Using this logic, stloc, ldloc and add instructions, will be mapped to istore, iload and iadd. This means that some kind of mechanism must be implemented in order to decide whether to map, for example, stloc to istore or to dstore. This can be easily done via OCL expressions since they are supported in both languages.

Listing 6.13: QVTO and QVTR OCL constructs

```xml
\QVTO
  if self.stack_post.elem.last().type = 'int' then
    resultCode.jvmelem := self.map.add_to_iadd()
\QVTR

if clr.elem.oclAsType(clr::Add).stack_post.elem.last().type = 'int' then
  add2iadd(code1, code2)
```

Listing 6.13 shows OCL constructs used to implement mentioned decision mechanism. Both solutions are almost identical. In QVTO, `self` is a runtime reference to `Add` class and it is used to access the type stack in order to determine which data type is on the top (line 3). If it is integer, `add_to_iadd`
mapping will be called using this reference. In QVTR, clrelem base class has to be casted to Add first and then stack can be examined. If it is integer relation add2iadd is examined. Beside simple OCL constructs, QVTO OCL expressions can be extended with features of QVTO language itself allowing for more complex constraints to be built. On the other hand, QVTR support only pure OCL.

Listing 6.14: QVTO mapping and QVTR relation for add instruction

```
\(\text{QVTO}\)
mapping clr::Add::add_to_iadd() : jvm::iadd {}
\(\text{QVTR}\)
relation iadd {
  enforced domain jvm_model code2 : vm::Code {
    jvmelem = jvmelem1 : jvm::iadd {}
  }
}
```

Regarding the transformation of add instruction, QVTO provides solution with less code again (Listing 6.14, line 2). What is interesting is that QVTR relations are allowed to have only one domain, like in this case (Listing 6.14, line 5). Since it is enforced, element Iadd will be created in the target domain without any preconditions in the source domain.

Previous example showed one-to-one mapping between CIL and JBC instructions. More complex scenario would be mapping of CIL instructions that are not supported in JBC at all. This is the case with transformation of instructions that operate on unsigned values or they use overflow checking.

Listing 6.15: QVTO mapping and QVTR relation for add_ovf instruction

```
\(\text{QVTO}\)
mapping clr::Add::add_ovf_to_invokestatic() : jvm::invokestatic {
  class_type := "org.xmlvm.clr.Math";
  method := "add_ovf";
  signature := object vm::Signature {
    x_return := object vm::Return {
      type := "int";
    }
    parameter := object vm::Parameter {
      type := "int";
    }
    parameter := object vm::Parameter {
      type := "int";
    }
  }
}
\(\text{QVTR}\)
relation add_ovf2invokestatic {
  checkonly domain clr_model code1 : vm::Code {
```

59
An example would be mapping of \texttt{add\_ovf} which has no corresponding instruction in JBC. According to [14], problem can be solved by creating special Java class, \texttt{MathLib}, with static method that can throw overflow exceptions. By doing so, \texttt{add\_ovf} can be mapped to an invocation of this static method which mimics behavior of this instruction. Listing 6.15 shows QVTO and QVTR constructs used to map this instruction.

### 6.3.6 Mapping Value Types

Value Types are special kind of types, treated similarly as classes by CLR. Constructor is used to instantiate value types but instead of being placed on heap they are placed on stack. They are automatically removed from memory when method, which allocated them, exits. This can significantly improve memory management in CLR since value type instances 'live' as long as their associated method.

Listing 6.16: Value Types in C#

```csharp
public struct Person {
    public string Name;
    public Person(string name) {
        Name = name;
    }
}
```

In C#, a keyword \texttt{struct} is used to define value types (Listing 6.16 line 1). They are very similar to classes with the difference that they can not inherit behavior from other structures. Since they are placed on stack, assignment of their variables has non-aliasing semantic, meaning that a deep copy will be always performed on value type instances.
Listing 6.17: Value Types in XMLVM<sub>clr</sub>

```
<clr:var varId="0" isValueType="true" type="Person" oid="1" />
<clr:var varId="1" isValueType="true" type="Person" oid="2" />
<clr:ldc type="System.String" value="Bob" oid="4" />
<clr:call has-this="true" class-type="ValueType.Person" method="<init>" oid="5">
  <vm:signature>
    <vm:return type="void"/>
    <vm:parameter type="System.String"/>
  </vm:signature>
</clr:call>
```

Listing 6.17 shows a part of XMLVM<sub>clr</sub> program. Two variables of type `Person` are declared (lines 1, 2). The attribute `isValueType` makes sure they are treated as value types. Instead of using `newobj` to create instance of `Value.Type` in memory, `ldloc` is used to push the address of a memory location where the value type is allocated (line 3). `ldc` places string 'Bob' onto the top of the stack (line 4), which is then used as a parameter in the `call` instruction to invoke the constructor of the value type (lines 5-10). When a value type has been initialized, first variable that points to it, is copied to another variable (lines 11, 12). `ldloc` and `stloc` instructions are used to do this.

Since value types are not supported in JVM, their behavior need to be simulated using heap allocated objects. It is important to retain deep copy semantics of value type instances. While CIL instructions, `stloc` and `ldloc`, can perform both shallow and deep copy, JBC does not have instructions that can perform deep copy. To simulate such behavior in JBC, `stloc` and `ldloc` can be mapped to a call of a method that implements this behavior.

First step is to map value type variables. This is not an easy step since they need to be mapped into an array of instructions that will instantiate heap allocated object. This object will simulate value type.

Listing 6.18: Value type as a heap allocated object

```
<jvm:var varId="1" type="Person"/>
<jvm:new type="Person"/>
<jvm:dup/>
<jvm:invokeSpecial method="init" class-type="Person"/>
<jvm:signature class-type=""/>
<jvm:return type="void"/>
</jvm:signature>
```

Since value types are not supported in JVM, their behavior need to be simulated using heap allocated objects. It is important to retain deep copy semantics of value type instances. While CIL instructions, `stloc` and `ldloc`, can perform both shallow and deep copy, JBC does not have instructions that can perform deep copy. To simulate such behavior in JBC, `stloc` and `ldloc` can be mapped to a call of a method that implements this behavior.

First step is to map value type variables. This is not an easy step since they need to be mapped into an array of instructions that will instantiate heap allocated object. This object will simulate value type.
Listing 6.18 shows instructions needed to create above mentioned object. This portion of the code corresponds to the line 1 in Listing 6.17. In the case of JBC, this variable is reference to the class named Person (line 1). Class Person is initialized using invokespecial instruction, which invokes init constructor (lines 4-8). At the end reference is stored to variable (line 9).

Listing 6.19: QVTO mapping for variables

```
\helper method
helper clr::Var::processVAR(resultCode : Code, selfCode : Code) : Integer {
resultCode.jvmelem += self.map var_to_var(selfCode);
if (self.isValueType = true)
resultCode.jvmelem += object jvm::New {
type := self.type;
};
resultCode.jvmelem += object jvm::Dmp {};
resultCode.jvmelem += object jvm::InvokeSpecial {
class_type := self.type;
method := "init";
signature := object vm::Signature {
x_return := object vm::Return {
type := "void";
};
};
resultCode.jvmelem += object jvm::Astore {
index := self.id + selfCode.varcnt;
}
}
end if;
return 0;
}
\variable mapping
mapping clr::Var::var_to_var(selfCode : Code) : jvm::Var {
id := self.id + selfCode.varcnt;
type := self.type;
}
```

QVTO code in Listing 6.19 performs mapping of variables. A helper method, which returns integer value, is defined on elements of type clr:Var. Parameters are Code elements in source and target domains, marked as selfCode and resultCode respectively. Helper methods are useful to divide mapping logic and to organize it in more functional parts. Here, they are used to divide mapping of regular variables (lines 3, 28-31) from mapping in case they are value type variables (lines 4-26).

QVTO can treat model elements as objects that encapsulate their attributes (lines 7, 10, 11, 20). Keyword object is used to create objects of certain type. In this case those are objects that represent additional instructions,
needed to simulate value type variables. This is useful feature of the language since object oriented programming is most commonly used today, allowing new language learners to adjust easily.

**New**, **Dup** and **Invokespecial** objects are added manually via `+=` operator. This operator is used when element in target model can have 0 or more elements as its child elements. In the code above, `jvmelem` is the name of association between `Code` class, on the one side, and `New`, `Dup` and `Invokestatic` classes, on the other side. Since `Code` can have many instructions, operator `+=` is used to associate new elements with `Code`.

**Listing 6.20**: QVTR relations for variables

```java
relation var2var { 
    varID : Integer; varType : String; 
    checkonly domain clr_model code1 : vm::Code { 
        cirelem = cirelem1 : clr::Var{ 
            id = varID, 
            type = varType 
        } ; 
    } ; 
    enforce domain jvm_model code2 : vm::Code { 
        jvmelem = jvmelem1 : jvm::Var{ 
            id = varID, 
            type = varType 
        } ; 
    } ; 
    where { 
        if cirelem1.isValueType = true 
        then relateVTVariable(cirelem1, code2) 
        else true 
        endif; 
    } 
} 

relation relateVTVariable { 
    primitive domain ovr : clr::Var; 
    enforce domain jvm_model code2 : vm::Code { 
        jvmelem = jvmelem1 : jvm::New{ 
            type = ovr.type, 
        }, 
        jvmelem = jvmelem1 : jvm::Dup{}, 
        jvmelem = jvmelem1 : jvm::Invokestatic{ 
            method = "init", 
            signature := object vm::Signature{ 
                x_return := object vm::Return{ 
                    type := "void"; 
                } 
            } 
        } 
    } ; 
} 
```

63
QVTR code that will perform the same thing as described QVTO, is realized with two relations given in Listing 6.20. For each variable that is found in Code element of a source model, exactly the same variable will be created in the Code element of the target model (lines 9-14). This is done with \texttt{var2var} relation. In case that variable marked as \texttt{clrelem1} (line 4) is value type variable, relation \texttt{relateVTVariable} is called. \texttt{relateVTVariable} appends \texttt{New}, \texttt{Dup} and \texttt{Invokestatic} elements to the Code element in target model.

Next step is initialization of heap allocated object. In order to assign 'Bob' value to a member variable of the object created in Listing 6.18, call instruction can be mapped to an invocation of a method that will simulate behavior of the value type constructor.

Listing 6.21: Mapping call instruction

```xml
<jvm:load index="1" type="Person"/>
<jvm:new type="System.String"/>
<jvm:dup/>
<jvm:ldc type="java.lang.String" value="Bob"/>
<jvm:invokestatic method="init" class-type="System.String">
<vm:signature class-type=""/>
<vm:return type="void"/>
<parameter type="java.lang.String"/>
</vm:signature>
</jvm:invokestatic>
<jvm:invokevirtual method="_CONSTRUCTOR" class-type="Person">
<vm:signature class-type=""/>
<vm:return type="void"/>
<parameter type="System.String"/>
</vm:signature>
</jvm:invokevirtual>
```

Listing 6.21 shows how this can be done. First, reference to the object object created in Listing 6.18 is pushed onto the stack (line 1). After that String object is pushed and initialized with the value 'Bob' (lines 2-10). Finally, in order to assign 'Bob' to a member variable, call is mapped to \texttt{invokevirtual} instruction, which calls \texttt{_CONSTRUCTOR} method. This method initializes member variable with 'Bob' value.

Finally, in order to perform deep copy on instances of \texttt{Person} class, \texttt{stloc} instruction is mapped to an invocation of a method that simulates deep copy in JBC world.
Listing 6.22: Deep copy in JBC

```
<jvm:aload index="1" type="Person"/>
<jvm:aload index="2" type="Person"/>
<jvm:invokestatic method="_COPY" class_type="System.ValueType">
  <vm:signature>
    <vm:return type="void"/>
    <parameter type="System.ValueType"/>
    <parameter type="System.ValueType"/>
  </vm:signature>
</jvm:invokestatic>
```

Listing 6.22 shows JBC sequence of instructions that perform deep copy on `Person` instances. Two `aload` occurrences push variables of type `Person` onto the stack. After that `invokestatic` calls `_COPY` method that copies value of the first variable to the second one. Although `_COPY` method accepts parameter of `System.ValueType`, it is valid to pass parameters of type `Person` since `Person` is subclass of `System.ValueType`.

Listing 6.23: QVTO mapping and QVTR relation for `stloc` mapping

```
\ \ QVTO
resultCode.jvmelem += object jvm::Aload {
  index := self.index + selfCode.varcnt;
  type := el.type;
};
resultCode.jvmelem += object jvm::Invokestatic {
  class_type := "System.ValueType";
  method := "_COPY";
  signature := object vm::Signature {
    x_return := object vm::Return {
      type := "void";
    };
    parameter := object vm::Parameter {
      type := "System.ValueType";
    };
    parameter := object vm::Parameter {
      type := "System.ValueType";
    };
  };
};
\ \ QVTR
primitive domain clr_model clrelem1 : clr::Stloc;
enforce domain jvm_model code2 : vm::Code {
  jvmelem = jvmelem1 : jvm::Aload {
    index = clrelem1.index + 1,
    type = clrelem.stack_pre.elem.last().type,
  };
  jvmelem = jvmelem2 : jvm::Invokestatic {
    type = "System.ValueType",
    method = "_COPY",
    signature = "sig : vm::Signature {
      x.return = ret : vm::Return {
        type = "void";
      };
      parameter = par1 : vm::Parameter {
        type = "System.ValueType";
      },
      parameter = par2 : vm::Parameter {
```

65
Listing 6.23 shows QVTO and QVTR code, necessary to perform mapping of stloc instruction to the appropriate invocation of __COPY method.

6.3.7 Mapping Delegates

As it was explained in the Chapter 3, delegates are special mechanism used to implicitly invoke one or more different methods. Since they are not available in JBC, their functionality must be simulated somehow. One way, described in [14], is to create a helper class that will use reflection to achieve the same behavior as delegates. When such a class is available, CIL instructions, related to delegates, can be mapped to calls of appropriate functions in that class.

Listing 6.24: Print method in C#

```csharp
public class DelegateTest {
    public delegate void Print(String s);
    public void SayHello(String word) {
        Console.WriteLine(word);
    }
    public static void Main() {
        DelegateTest dt = new DelegateTest();
        Print p = new Print(dt.SayHello);
        p("Hellooooo");
    }
}
```

In Listing 6.24 is a simple program that shows how delegates are used in C#. Delegate Print is declared in line 2. It is instantiated and assigned the SayHello method (line 8). In line 9 it is used to implicitly invoke SayHello method.

Listing 6.25: Delegate in XMLVM

```xml
<clr:ldftn class="DelegateTest" method="SayHello" oid="2">
    <vm:signature/>
    <vm:return type="void"/>
    <vm:param type="System.String"/>
</clr:ldftn>
<clr:newobj type="Print" oid="3">
    <vm:signature/>
    <vm:return type="void"/>
    <vm:param type="System.Object"/>
    <vm:param type="int"/>
</vm:signature>
```
Excerpt from XMLVMCLR program, that corresponds to the CIL of the program above, is given in Listing 6.25. `ldftn` places a function pointer onto the stack which has the address of the `SayHello` method as its value. This pointer and reference to the `DelegateTest` class are parameters for the constructor of the `Print` delegate, invoked by the `newobj` instruction (lines 7-13). When `ldc` instruction pushes 'Hellooooo' string onto the stack (line 14), `SayHello` method is called indirectly through the `Invoke` method of the delegate class `Print`. This invocation is made through the `callvirt` instruction (lines 15-20).

First step is to map `ldftn` to a call of a method in the helper library that will mimic behavior of this instruction. Class `DelegateManager` has method called `registerFunctionPtr`, which is used for this purpose. Method’s internal implementation is explained in [14].

Listing 6.26: Array of JBC instructions that simulate `ldftn`

```
<Jvm:ldc type="java.lang.String" value="DelegateTest"/>
<Jvm:ldc type="java.lang.String" value="SayHello"/>
<Jvm:ldc type="java.lang.String" value="void; System.String"/>
<Jvm:invokestatic class="DelegateManager" method="registerFunctionPtr">
  <vm:signature>
    <vm:return type="int"/>
  </vm:signature>
</Jvm:invokestatic>
```

In Listing [6.26] JBC sequence of instruction is showed. Three `ldc` instructions push data, necessary for the `registerFunctionPtr` to simulate `ldftn` instruction (lines 1-3). Those are the name of a class where the method is declared, method name and parameters of the method, encoded using string type. `invokestatic` calls `registerFunctionPtr` with these parameters (lines 4-10).
Listing 6.27: QVTO for `ldftn` instruction

```java
helper clr::Ldftn::processLDFTN(resultCode : Code) : Integer {
    resultCode.jvmelem += object jvm::Ldc {
        type := "java.lang.String";
        value := self.class_type;
    };
    resultCode.jvmelem += object jvm::Ldc {
        type := "java.lang.String";
        value := self.method;
    };
    resultCode.jvmelem += object jvm::Ldc {
        type := "java.lang.String";
        value := self.signature.return.type + self.signature.parameter.type;
    };
    resultCode.jvmelem += object jvm::Invokestatic {
        class_type := "org.xmlvm.clr.DelegateManager";
        method := "registerFunctionPtr";
        signature := object vm::Signature {
            x_return := object vm::Return {
                type := "int";
            };
        parameter += object Parameter {
            type := "java.lang.String";
        };
    parameter += object Parameter {
            type := "java.lang.String";
        };
    parameter += object Parameter {
            type := "java.lang.String";
        };
    };
    return 0;
}
```

Mapping of `ldftn` instruction is realized with helper method in QVTO (Listing 6.27). `processLDFTN` method is defined on elements of type `clr::Ldftn` and as parameter it accepts `Code` element associated with the target domain. Mapping itself is straightforward. Ldc classes are appended as child elements to the `Code` class, together with `Invokestatic` class.

Listing 6.28: QVTR for `ldftn` instruction

```java
relation ldftn2APICall {
    varClassType, varMethod : String;
    checkonly domain clr_model code1 : vm::Code {
        clrelem = clrelem1 : clr::Ldftn {
            class_type = varClassType,
            method = varMethod
        };
    };
    enforce domain jvm_model code2 : vm::Code {
        jvmelem = jvmelem1 : jvm::Ldc {
            type = "java.lang.String",
            value = varClassType
        };
    }
```

68
CHAPTER 6. BYTE CODE TRANSFORMATION FRAMEWORK

Same code, implemented in QVTR, is given in Listing 6.28. Only one relation is used to map `ldftn` to the appropriate sequence of JBC instructions. It is interesting how QVTR implementations, with increasing transformation complexity, are demanding similar amount of code as those written in QVTO. For the simple cases, where it is necessary to perform one-to-one mapping, QVTO transformations are smaller which is not a great advantage since most of the bytecode mappings are complex.

Listing 6.29: Invoking SayHello in XMLVM

```xml
<jvm:aaload type="" index="0"/>
<jvm:new type="System.String"/>
<jvm:dup/>
<jvm:ldc type="java.lang.String" value="Hellooooo"/>
<jvm:invokestatic class="System.String" method="&lt;init&gt;"/>
<jvm:signature>
  <vm:return type="void"/>
  <vm:parameter type="java.lang.String"/>
</jvm:signature>
<jvm:invokevirtual class="Print" method="Invoke"/>
<jvm:signature>
  <vm:return type="void"/>
  <vm:parameter type="System.String"/>
</jvm:signature>
</jvm:invokevirtual>
```
Finally, in order to invoke `SayHello` in JBC, `callvirt` can be mapped to the `invokevirtual` instruction. Listing 6.29 shows how delegate class is instantiated and used to invoke `SayHello` method. `aload` pushes reference onto the stack, which points to the delegate class (line 1). When string ‘Hellooooo’ is placed onto the stack (lines 2-10), `invokevirtual` calls `Invoke` method, which is used to implicitly invoke `SayHello`.

Listing 6.30: QVTO and QVTR for `callvirt` instruction

```java
\// QVTO
mapping clr::Callvirt::callvirt_to_invokevirtual() : jvm::Invokevirtual {
    class_type := self.class_type;
    method := self.method;
    signature := self.signature;
}
\// QVTR
relation callvirt2invokevirtual {
    var ClassType, var Method, var ReturnType : String;
    var HasThis : Boolean;
    checkonly domain clr_model code1 : vm::Code {
        clrelem = clrelem1 : clr::Callvirt {
            class_type = var ClassType,
            method = var Method,
            has_this = var HasThis,
            signature = signature1 : vm::Signature {
                class_type = var ClassType,
                x_return = return1 : vm::Return {
                    type = var ReturnType
                }
            }
        }
    }
    enforce domain jvm_model code2 : vm::Code {
        jvmelem = jvmelem1 : jvm::Invokevirtual {
            class_type = var ClassType,
            method = var Method,
            signature = signature2 : vm::Signature {
                class_type = var ClassType,
                x_return = return2 : vm::Return {
                    type = var ReturnType
                }
            }
        }
    }
}
```

QVTO and QVTR implementations in Listing 6.30 are used for mapping of `callvirt` instruction. Once again, for a simple one-to-one mapping, implementation in QVTR needs much more code than QVTO.
6.3.8 QVTO or QVTR?

QVTO language is very similar to the modern object oriented languages. In order to support object oriented style as much as possible many features are supported. As it was described, it allows model elements to be used as objects and their attributes to be accessed as in Java or C#. Mapping and transformation inheritance is also possible, although not used in this thesis. Classical control structures are present as well. while and foreach loops, if-then-else, etc. are all there to make this environment more comfortable for the programmers.

On the other hand, QVTR is more verbose since it does not support many different language constructs like QVTO does. Although this simplicity might seem as disadvantage it is not. As it can be noticed in the examples from the previous section, the amount of code necessary to perform complex mappings, tends to be the same in both languages. QVTR might be easier to learn for the new users because of this simplicity. The word 'might' stands to emphasize the fact that QVTR is declarative language. Having fewer constructs that are used frequently would indeed shorten the learning curve but only with users that are used to declarative languages.

Comparison of the previous examples written in QVTR leads to a conclusion that there is only one possible way to write relations, meaning that all relations have the same structure. Although rigid structure of relations implies more code, it is beneficial because it can be easily expressed graphically. This would enable creation of a graphical language that could be used like UML to describe relations between models.

Maybe the most important feature of the QVTR is that it enables bi-directional transformations. This is because relations do not define source and target models in the code, unlike QVTO mappings do. Having this feature in QVTR, it is necessary to write only one relation for certain model elements and then to choose later in which direction it will be executed.

Both languages can be used for complex tasks such as bytecode mapping. Perhaps two factors can be used when deciding which one to use: previous programming background and bi-directionality of transformations. In the case of the first, QVTO is the right choice since it offers similar approach
as widely accepted object-oriented programming. If bi-directionality is a necessity, QVTR is the only choice.

6.4 Basic Work Flows

This framework covers two scenarios of bytecode transformation: 1) static and 2) dynamic.

6.4.1 Static Transformation

Static means that only a program structure is transformed. In this case, the program structure is a sequence of bytecode instructions with additional information. An example for this can be transformation of the program given in the Figure ?? to the program given in the Figure ??.

Figure 6.11: Static transformation

Figure 6.11 illustrates how this process is performed within the framework. At first, .Net assembly with all CIL instructions and accompanying data is loaded by XMLVM input compiler. Based on assembly data, XMLVM representation of CIL is produced (step 2). Previously described pre-processing is performed in the step 3 to allow usage of XMLVM\_cl in the available QVT tools. Based on the bytecode models and the transformation definition, tool will perform mapping of model elements (step 4). As a result XMLVM\_jvm is generated in the step 5. Now, in order to use this result with XMLVM
CHAPTER 6. BYTE CODE TRANSFORMATION FRAMEWORK

compilers, it must be returned to the original XML format. Therefore, post-processing is performed in the step 6 to return generated output in its original format. Finally, output compilers in XMLVM can produce JBC (step 7).

6.4.2 Dynamic Transformation

Dynamic transformation is a more complex scenario. In contrast to the previous scenario, where only XML files are transformed, here programs that are in execution are transformed. By supporting dynamic transformation, this framework can be used for strong migration.

Without going into too much details, code migration allows programs to be executed remotely. Whole or a part of a program can be transferred and executed on another computer. Reasons for a code migration are different. Some benefits gained by this process are performance increasing through parallelism, utilization of idle resources, computations close to data and so on. Depending whether a code of a program is migrated or a complete process in memory, difference can be made between weak and strong migration.

Figure 6.12: Dynamic transformation

Figure 6.12 illustrates process of dynamic transformation. First step is to get information about running program. During the execution, programs are

73
tied to different parts of a virtual machine. Therefore, information about the state of a virtual machine is also necessary. To do so a snap-shot of a running virtual machine is taken and persisted (step 1). For example, in CLR this snap-shot would include information related to each executing thread, like instruction pointer, stack and heap. In the next two steps (steps 2 and 3), previously extracted information is transformed using QVT tools. In contrast to the previous scenario, where it was sufficient to have models that describe XMLVM programs, here additional model, describing involved virtual machines, is necessary. Finally, after transformation has been performed, model instances are returned to their native format (step 4).

XMLVM programs provide XML representation of Java class files and .NET assemblies. They do not include information about possible states of a virtual machine, which is necessary to perform dynamic bytecode transformation. Therefore, it is necessary to provide some kind of a model that will describe virtual machines. This model can be merged with PSMxmlvm-clr and PSMxmlvm-jvm to enable dynamic bytecode transformation.

Figure [6.13] shows possible combination of PSMxmlvm-jvm and JVMuml. Each element of JVMuml must be bound to a corresponding element in PSMxmlvm-jvm according to its purpose. For example, each frame has to have method data and therefore, Frame and Method classes are bounded together to symbolize this relationship.
This scenario has not been practically realized for the thesis and it can be part of the future work.