Matilda: A Generic and Customizable Framework for Direct Model Execution in Model-Driven Software Development

Hiroshi Wada[1], Junichi Suzuki[1], Adam Malinowski[2] and Katsuya Oba[3]

[1] Department of Computer Science
University of Massachusetts, Boston
Boston, MA 02125-3393
email: {shu, jxs}@cs.umb.edu

[2] Faculty of Arts and Sciences
Harvard University
Cambridge, MA 02138
amalinowski@cgr.harvard.edu

[3] OGIS International, Inc.
San Mateo, CA 94404
email: oba@ogis-international.com

Abstract

Traditional Model Driven Development (MDD) frameworks have three critical issues: (1) abstraction gap between modeling and programming layers, (2) a lack of traceability between models and programs, and (3) a lack of customizability to support various combinations of modeling technologies and implementation/deployment technologies. In order to address these issues, this chapter proposes a new MDD framework, called Matilda, which is a framework to build execution runtime engines (or virtual machines) for software models. It directly executes models defined with certain modeling technologies such as UML and BPMN by automatically transforming them to executable code. Matilda is designed based on the Pipes and Filters architectural pattern, which allows for configuring its structure and behavior flexibly by replacing one plugin with another one or changing the order of plugins. Also, plugins can be deployed on multiple network hosts and seamlessly connect them to form a pipeline. This facilitates distributed software development in which developers collaboratively work at physically dispersed places. This chapter overviews Matilda's architectural design, describes the implementations of Matilda-based virtual machines, and evaluates their performance.

Introduction

Software modeling has advanced to the point where it can offer significant leverage to manage complexity and improve productivity in software development. A driving force in this advance is a series of mature modeling technologies. For example, the Unified Modeling Language (UML) provides a wide range of modeling notations and semantics used in various types of applications (*UML Super Structure Specification 2.1.2*, 2007). The Business Process Modeling Notation (BPMN) provides a set of well-defined notations and semantics for business process modeling (*Business Process Modeling Notation (BPMN) 1.0*, 2004). UML and BPMN allow developers to specify and communicate their application designs at a high level of abstraction. Using these modeling technologies, the notion of model-driven development (MDD) aims to graphically build application design models and transform them into running applications.

A key process in MDD is automated (or semi-automated) transformation of implementation-independent models to lower-level models (or application code) specific to particular implementation/deployment technologies such as programming languages, databases, middleware and business process engines (Booch, Brown, Iyengar, Rumbaugh, & Selic, 2004; Sendall & Kozaczynki, 2003). Traditional MDD frameworks allow developers to model their applications with modeling languages such as UML and BPMN, generate skeleton code in a programming language such as Java, and manually complete the generated skeleton code by, for example, adding method code (Figure 1). There exist three critical research issues in traditional MDD frameworks: (1) abstraction gap between modeling and programming layers, (2) a lack of traceability between models and programs, and (3) a lack of customizability to support various combinations of modeling technologies and implementation technologies.

The first issue is that, when programmers complete generated skeleton code to the final (compilable) code, they often suffer from abstraction gap between modeling and programming layers because the granularity of skeleton code is usually much finer than that of models. Skeleton code tends to be complicated to read and maintain. Thus, it is hard for programmers to obtain a broad view of an application design, and they have to repeatedly go up and down abstraction gap to identify where to implement what in skeleton code.

The second issue is that models tend to lose synchronization with programs through development process. For example, when programmers find bugs or design/implementation alternatives in the final (compilable) code, they often change the code directly rather than models. As a result, the program code becomes untraceable from models. Due to the above two issues, traditional MDD frameworks do not maximize the benefits of modeling application designs at a higher level of abstraction than programming layer.

The third issue in traditional MDD framework is that they often lack generality to support a variety of choices in modeling technologies and lack customizability to tailor model transformation and code generation according to the implementation/deployment technologies used in application development. This degrades reusability and longevity of application models; it is often hard to evolve application models for a long time by, for example, introducing new modeling technologies and changing implementation/deployment technologies.

This chapter describes and evaluates a new MDD framework, called Matilda, which addresses the three issues described above. Matilda is a generic framework to build execution runtime engines (or virtual machines) for various types of software models. Each virtual machine (VM) accepts and directory executes models defined with certain modeling technologies such as UML and BPMN through automatically transforming them to executable code (Figure 2). Matilda addresses the abstraction and synchronization issues by hiding the existence of source code from developers. Using Matilda, developers analyze, design, test, deploy and execute their applications consistently at the modeling layer, rather than shifting between the modeling and programming layers (Figure 2). Each

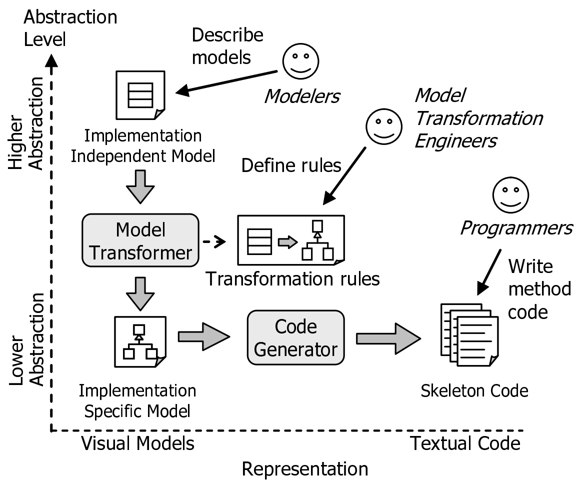


Figure 1. Traditional MDD Process

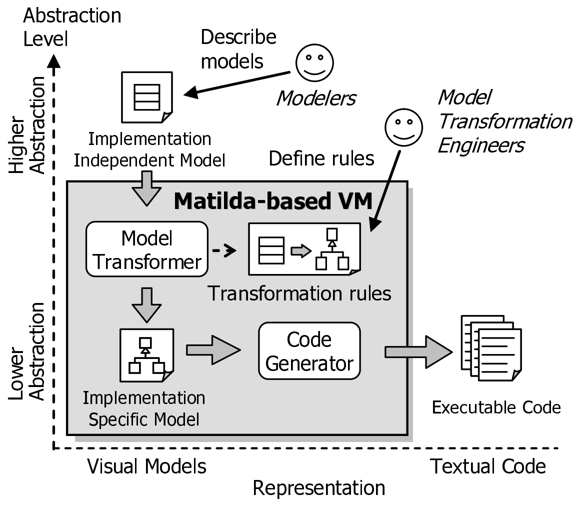


Figure 2. Development Process with Matilda

Matilda-based VM accepts a model as an input, validates it against its metamodel, and transforms the input model to an implementation/deployment specific model by applying a given transformation rule (Figure 2). Matilda allows developers (model transformation engineers in Figure 2) to define arbitrary transformation rules, each of which specifies how to specialize an input model to a particular implementation/deployment technology. For example, a transformation rule may specialize an input model to a database, while another rule may specialize it to a remoting middleware.

Matilda addresses the customizability issue with its architectural design using a pipeline of plugins. Different plugins implement different functionalities in Matilda, such as visualizing, validating and transforming models. The pipeline architecture allows Matilda to flexibly customize its structure and behavior by replacing one plugin with another or changing the order of plugins. Also, Matilda's pipeline can be distributed over the network. Matilda can spread plugins to multiple network hosts and seamlessly connect them to form a pipeline. This enables distributed software development in which developers can collaboratively build, integrate and execute models at physically dispersed places.

Currently, Matilda provides two different VMs: UML-VM and SOA-VM. UML-VM accepts UML 2.0 class and sequence diagrams, validates them against the UML metamodel, transforms them to a JAST (Java Abstract Syntax Tree), generates Java bytecode and runs the generated code on a Java VM (JVM). SOA-VM accepts BPMN and UML sequence diagrams, validate them against the BPMN and UML metamodels, transforms a BPMN diagram to a UML class diagram, transforms UML diagrams to JAST, generates Java bytecode and a deployment descriptor in Business Process Execution Language (BPEL) (*Web Services Business Process Execution Language*, 2003), and runs the generated code on a BPEL engine.

This chapter overviews the design and implementation of Matilda, describes how UML-VM and SOA-VM are built on Matilda, evaluates their performance, and concludes with some discussion on related and future work.

Design Principles in Matilda

Matilda is designed based on the following principles.

1. **Avoidance of Round-Tripping.** In order to address the abstraction and synchromization issues, Matilda inherently avoids the round-trips between models and source code by hiding the existence of source code from developers. All bug fixes and design changes are directly made on models instead of source code (Figure 2).

2. **Metamodel-Driven.** Matilda performs all of its functionalities in a metamodel-driven manner. For example, UML model validation is performed against the UML metamodel, and JAST generation is performed with a metamodel of Java program elements. By following metamodels consistently, plugins in Matilda avoid to perform their functionalities ambiguously. Matilda represents UML, BPMN and Java metamodels as a set of objects (APIs), and aids its plugins to implement their functionalities on a metamodel basis.

3. **Modularity and Loose Coupling.** Matilda is designed to maximize the reusability of plugins by making them modular and loosely coupled. Matilda decomposes its functionalities into independent processing units and implements them as plugins. The functionality of each plugin does not depend on other plugins.

4. **Configurability.** Matilda is intended to be used in a variety of development projects; from in-house development, distributed open-source development to off-shore development. Different projects use different sets of plugins in different orders. For example, a project may require a plugin for generating Java bytecode, and another project may require a plugin for generating BPEL deployment descriptors as well as a Java bytecode generation plugin. Therefore, Matilda is designed to make pipelines configurable and extensible. It defines common APIs for pipelines and plugins so that each developer can choose plugins and configure a pipeline of the plugins. Matilda also allows developers to implement new plugins with its plugin API.

5. **Transparent Distribution.** Matilda supports distributed execution of plugins for distributed software development. Different plugins can run on different hosts in the network. For example, the plugins for model visualization and validation can run at a place, and the plugins for code generation can run at a different remote place. Plugins can be transparently distributed; each of them does not have to know whether others reside on the same host.

The Architecture of Matilda

There are four roles of users who involve in development process with Matilda. A *modeler*, or application developer, builds application design models (M1 models) and load them to Matilda (Figure 2). A *metamodel engineer* builds and/or registers metamodels (M2 models) including the UML metamodel, UML profiles and BPMN metamodel. A *plugin engineer* develops and registers plugins. A *transformation engineer* is a special type of plugin engineer, who defines transformation rules and implements them as plugins (Figure 2). A *VM maintenance engineer* is responsible for customizing a pipeline for a given VM.

The pipeline architecture of Matilda is designed based on the Pipes and Filters architectural pattern (Buschmann et al., 1996; Vermeulen, Beged-Dov, & Thompson, 1995). This pattern defines a system architecture to process data streams. The task of a system is divided into several processing steps. These steps are connected along with a data flow in the system; an output data in a step becomes an input to a subsequent step. Each processing step is implemented as a *filter*, and filters are connected with *pipes*. The Pipes and Filters pattern is well applicable when a system can naturally decompose its data processing task into independent steps and the task is likely to change over time. This pattern increases the reusability of filters, and allows a system to be flexible for exchanges and recombinations of filters (Buschmann et al., 1996).

In Matilda, each plugin works as a filter and implements an individual step in an application development (Figure 3). For example, a model loader plugin accepts a UML model in the format of XML Metadata Interchange (XMI) (*MOF 2.0 XMI Mapping Specification*, 2007), a model validation plugin validates a UML model against the UML metamodel, and a JAST generation plugin transforms a validated UML model to a JAST. A pipeline contains one or more plugins on each network host, and multiple pipelines form a distributed, composite pipeline over multiple hosts (Figure 3). Each pipeline downloads required plugins form a plugin repository and connects them based on a configuration file that a VM maintenance engineer defines. The configuration file specifies plugins used in a pipeline and their execution order. Plugins can be executed sequentially or in parallel. Each plugin operates on the Matilda runtime, which operates on a JVM.

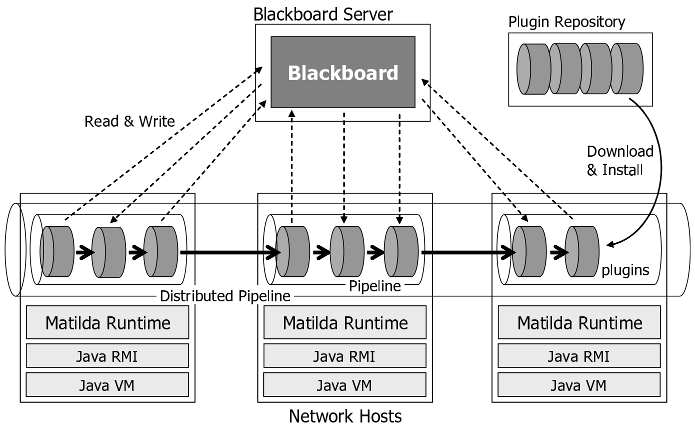


Figure 3. The Behavioral Architecture of Matilda

The Pipes and Filters pattern often lacks robust error handling because multiple asynchronous threads of execution does not share the global system state (Buschmann et al., 1996). In order to overcome this issue, Matilda implements a shared repository, called *blackboard*, based on the

Blackboard architectural pattern (Figure 3) (Buschmann et al., 1996). This pattern is organized as a collection of independent processing units that work cooperatively on common data structures. Each processing unit specializes to process a particular part of the overall task. It fetches data from a blackboard, and stores a result of its data processing to the blackboard.

In Matilda, a blackboard stores data that each plugin generates (e.g., UML models and JASTs), and makes the data available to subsequent plugins (Figure 3). It also stores a processing log in each plugin (e.g., successful completion, errors, warnings and time stamp) in order to trace the processing status in a pipeline. In Matilda, data flow between a blackboard and plugins, and processing control flows between pipelines (Figure 3).

Figure 4 shows the structural architecture of Matilda. Matilda consists of its kernel and plugins. The kernel is responsible for low-level house-keeping functionalities required to operate plugins. Matilda-based VMs are defined and deployed on Matilda by choosing particular plugins in particular configurations. Currently, Matilda's codebase contains 18,294 lines of Java code, which implements Matilda, UML-VM and SOA-VM.

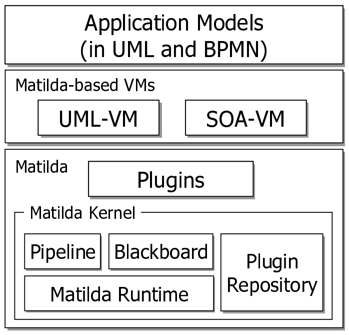


Figure 4. The Structural Architecture of Matilda

Matilda Kernel

Matilda's kernel consists of Matilda runtime, pipeline, blackboard and plugin repository. Figure 5 shows the class structure of Matilda's kernel. The class `Runtime` instantiates and configures `Pipeline` according to a given pipeline configuration file. The class `Pipeline` contains a set of plugins. They are defined as subclasses of the abstract class `Plugin`, which provides the methods to access a blackboard (`readFromBB()` and `writeOnBB()`). A `Pipeline` executes plugins by calling their `execute()` methods; each plugin implements its own functionality in `execute()`. The interface `Blackboard` defines the methods to read/write data from/on a blackboard.

The Matilda runtime (`Runtime`) configures a `Pipeline` by loading a pipeline configuration file with `Config::LoadFile()`. `Config` and `PluginConfig` maintain the configuration at runtime. Listing 1 shows an example pipeline configuration file that defines a sequential execution of four plugins. Each plugin's name and class file are specified with the `plugin` and `class` tags, respectively. For example, a plugin called `Model Loader` is implemented by the class `matilda.plugins.frontend.ModelLoader`. The `parameter` tag defines a set of parameters passed to a plugin. The `name` attribute specifies the name of a parameter. The parameters in the `pipeline` tag are passed to all plugins, while the parameters in the `plugin` tag are passed only to a plugin that is designated by the `plugin` tag. For example, the parameter `uml2resource` can be referred by all plugins; however, the parameter `modelpath` can be referred only by `Model Loader`. Once a `Runtime`

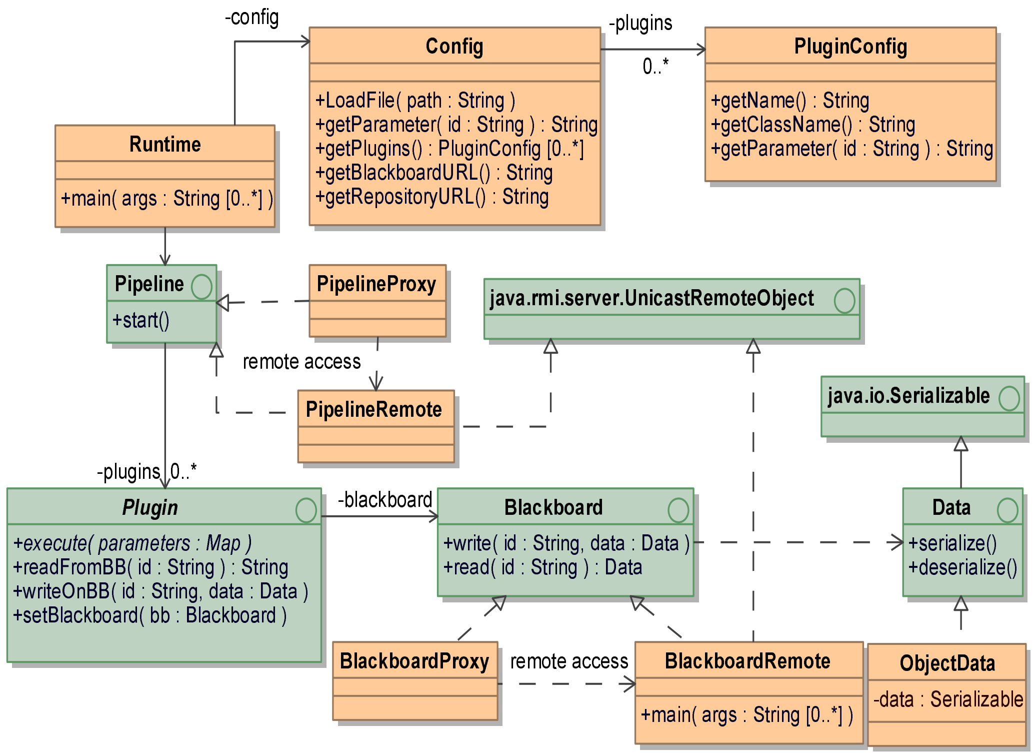

-config
Config
+LoadFile(path : String)
+getParameter(id : String) : String
+getPlugins() : PluginConfig [0..*]
+getBlackboardURL() : String
+getRepositoryURL() : String
-plugins
0..*
PluginConfig
+getName() : String
+getClassName() : String
+getParameter(id : String) : String
Runtime
+main(args : String [0..*])
Pipeline
+start()
PipelineProxy
java.rmi.server.UnicastRemoteObject
remote access
PipelineRemote
java.io.Serializable
-plugins 0..*
Plugin
+execute(parameters : Map)
+readFromBB(id : String) : String
+writeOnBB(id : String, data : Data)
+setBlackboard(bb : Blackboard)
-blackboard
Blackboard
+write(id : String, data : Data)
+read(id : String) : Data
Data
+serialize()
+deserialize()
BlackboardProxy
remote access
BlackboardRemote
+main(args : String [0..*])
ObjectData
-data : Serializable

Figure 5. A Class Structure of Matilda's Kernel

configures a `Pipeline`, the `Pipeline` downloads required plugin class files from a plugin repository and execute them.

Listing 1 A Pipeline Configuration File

```
<pipeline>
 <parameter
   name="uml2resource">jar:file:org.eclipse.uml2.uml.resources_2.0.2.jar!/</parameter>

 <plugin name="Model Loader">
  <class>matilda.plugins.frontend.ModelLoader</class>
  <parameter name="modelpath">models/model.uml2</parameter>
 </plugin>

 <plugin name="Class to JAST Transformer">
  <class>matilda.plugins.backend.CD2JASTTransforder</class>
 </plugin>

 <plugin name="Sequence to JAST Transformer">
  <class>matilda.plugins.backend.SD2JASTTransformer</class>
 </plugin>

 <plugin name="Java Executor">
  <class>matilda.plugins.backend.JavaExecutor</class>
  <parameter name="args">-h</parameter>
 </plugin>
</pipeline>
```

In Matilda, `Pipeline` and `Blackboard` are defined as interfaces with Java Remote Method

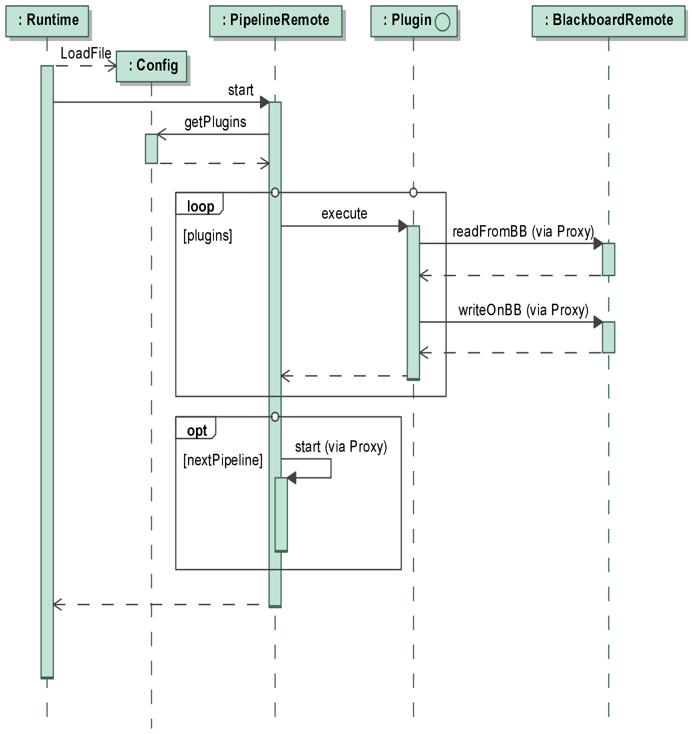


Figure 6. Interactions among Pipeline, Plugins and Blackboard

Invocation (RMI) so that plugins can be distributed over the network. The two interfaces define remotely-accessible methods, and its implementation classes (`PipelineRemote` and `BlackboardRemote`) implement the interface methods. `PipelineProxy` and `BlackboardProxy` hide remoting details in Java RMI and provide location transparency for callers/clients for `Pipeline` and `Blackboard`, respectively. (These proxy classes are generated by Java RMI.)

Plugins are implemented as regular Java classes and contained in a pipeline. Then, the pipeline allows plugins to communicate with a blackboard. This design hides remoting details from plugins, and plugin developers do not need to know them. It makes it easy to develop and deploy plugins.

Figure 6 shows how kernel components interacts with each other. A pipeline executes its contained plugins one by one by calling their `execute()` methods. When it calls `execute()` on a `Plugin`, it passes a set of configuration parameters as the method's argument. According to the parameters, the `Plugin` downloads necessary data from a blackboard, process the data and write processed data to the blackboard. When all plugins are executed in a pipeline, the pipeline calls `start()` on another pipeline running on another network host if multiple pipelines are connected over the network.

Matilda UML Virtual Machine (UML-VM)

This section describes a Matilda-based VM for UML models, called UML-VM. UML-VM accepts UML 2.0 class and sequence diagrams, validates them against the UML metamodel, transforms them to a JAST (Java Abstract Syntax Tree), generates Java bytecode and runs the generated code as a command-line application on a Java VM (JVM). UML-VM is built with a set of plugins as well as the Matilda UML VM profile.

Matilda UML VM Profile

The Matilda UML VM profile is a UML profile that provides modeling conventions to build input UML models and run them as Java bytecode. A UML profile is an extension to the standard UML metamodel. The UML metamodel specifies the syntax (or notation) and semantics of every standard (default) model element (e.g., class, interface and association) (*UML Super Structure Specification 2.1.2*, 2007). In addition to standard model elements, UML provides extension mechanisms (e.g., stereotypes and tagged-values) to specialize the standard model elements to precisely describe domain or application specific concepts (Fuentes & Vallecillo, 2004). A stereotype is applied to a standard model element, and specializes its semantics to a particular domain or application. Each stereotyped model element can have data fields, called tagged-values, specific to the stereotype. Each tagged-value consists of a name and value. A particular set of stereotypes and tagged-values is called a UML profile. The Matilda UML VM profile defines a set of stereotypes and tagged-values to precisely describe computationally-complete[1] UML models for Matilda.

In Matilda, a UML input model is defined as a set of UML 2.0 class diagrams and sequence diagrams. Class diagrams are used to define the structure of an application, and sequence diagrams are used to define its behavior. Each sequence diagram specifies the body of a method (operation). The model elements in a class diagram are mapped to structural elements in a Java program, such as Java types, generalization (inheritance) relationships, data fields and method declarations. The model elements in a sequence diagram are mapped to behavioral elements in a Java program, such as object instantiations, value assignments, method calls and control flows.

The Matilda UML VM profile defines two types of stereotypes: (1) stereotypes for application semantics and (2) stereotypes for Java mapping. Figure 7 shows the stereotypes for application semantics. A message stereotyped with ≪`UMLVMarrayelement`≫ represents an array access (i.e., data retrieval or insertion on an array). Its tagged-value `index` specifies the array index where data retrieval or insertion is performed (Table 1). The tagged-value `element` specifies a data element to be inserted to an array (Table 1). A message or comment stereotyped with ≪`UMLVMexpression`≫ has Java expressions or statements. A class stereotyped with ≪`UMLVMexecutable`≫ indicates an entry point at which a model execution starts. The class must contain a main method (`public static void main (String[])`). Each application has only one class stereotyped with ≪`UMLVMexecutable`≫.

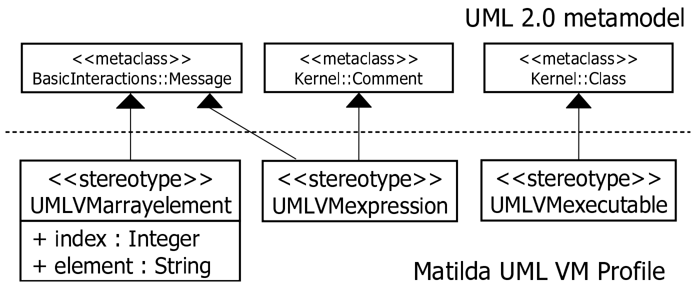

UML 2.0 metamodel
<<metaclass>> BasicInteractions::Message
<<metaclass>> Kernel::Comment
<<metaclass>> Kernel::Class
<<stereotype>> UMLVMarrayelement
+ index : Integer
+ element : String
<<stereotype>> UMLVMexpression
<<stereotype>> UMLVMexecutable
Matilda UML VM Profile

Figure 7. Stereotypes for Application Semantics

[1]"Computationally complete" means sufficiently expressive so that Matilda can interpret and execute models.

Table 1: Tagged-values of ≪`UMLVMarrayelement`≫

Name	**Type**	**Description**
`index`	Integer	Index of an array element to be accessed (retrieved or inserted). Must be between 0 and (array size – 1).
`element`	String	If `null` is assigned, array access is data retrieval. Otherwise, it is data insertion. Represents a variable that contains an element to be inserted.

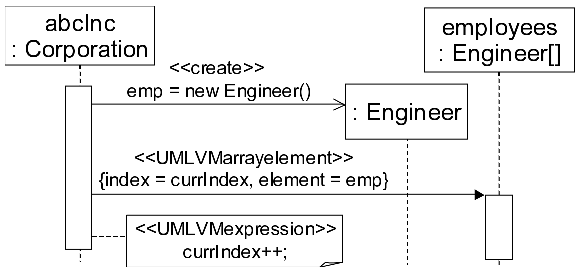


Figure 8. An Example Sequence Diagram using the Matilda UML VM profile

Figure 8 shows an example sequence diagram defined with the Matilda UML VM profile. `abcInc` (an instance of `Corporation`) creates a new instance of `Engineer`, and inserts the new `Engineer` instance into the array `engineers` (an array of `Engineers`). A message stereotyped with ≪`create`≫ indicates that the message instantiates a class[2]. For data insertion on the array `engineers`, a message stereotyped with ≪`UMLVMarrayelement`≫ specifies that `abcInc` inserts an `Engineer` instance (contained in the variable `emp`) to the array at the index of `currIndex`. At the end, `abcInc` increments `currIndex` by using a comment stereotyped with ≪`UMLVMexpression`≫.

The Matilda UML VM profile also defines a stereotype and five tagged-values to specify a mapping between UML models and Java programs (Table 2). A class stereotyped with ≪`JavaInterface`≫ represents a Java interface. `JavaStrictfp` indicates whether a Java class is FP-strict. If it is true, all float and double values in the class are used in the IEEE standard float/double size during floating point calculation. `JavaStatic` indicates whether a class/interface is static in Java. `JavaDimensions` specifies the number of array dimensions declared by corresponding field or parameter in Java. `JavaFinal` indicates whether a parameter is final in Java.

Table 2: Tagged-values in the Matilda UML VM profile

Name	**Type**	**Applied To**	**Description**
`JavaStrictfp`	Boolean	Class	Indicates a class is FP-strict.
`JavaStatic`	Boolean	Class or Interface	Indicates a class/interface is static.
`JavaDimensions`	Integer	Property or Parameter	Indicates the number of array dimensions.
`JavaFinal`	Boolean	Parameter	Indicates a parameter is final.

[2]The stereotype ≪`create`≫ is one of the standard stereotypes defined in the UML 2.0 specification. The UML notation of a message is an arrow in a sequence diagram.

Plugins in Matilda UML Virtual Machine

UML-VM consists of 10 plugins: *model loader*, *UML metamodel validator*, *Matilda UML VM profile class diagram (CD) validator*, *Matilda UML VM profile sequence diagram (SD) validator*, *Integrated diagram validator*, *CD2JAST transformer*, *SD2JAST transformer*, *JAST validator*, *JAST2Bytecode transformer* and *Java executor*. Plugins are categorized into two groups: *frontend* and *backend*. Frontend plugins are used to validate UML models, and backend plugins are used to transform validated UML models to Java bytecode through JASTs. Figure 9 shows the class structure of plugins in UML-VM. All plugins implement the `Plugin` interface. UML-VM provides extra interfaces (`ModelLoader`, `Validator` and `Transformer`) to indicate common functionalities in plugins.

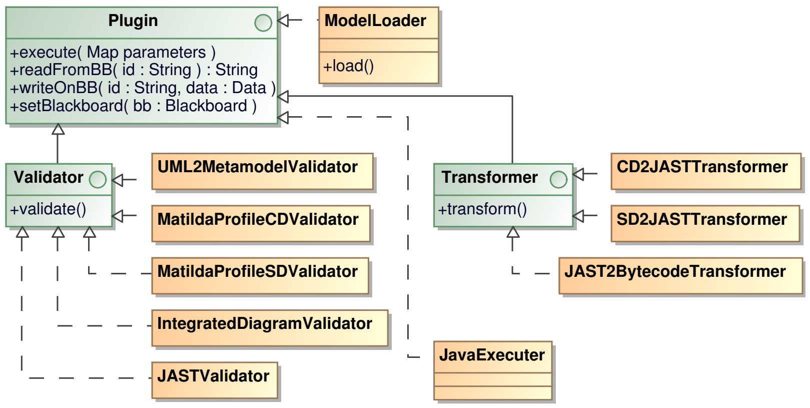

Plugin
+execute(Map parameters)
+readFromBB(id : String) : String
+writeOnBB(id : String, data : Data)
+setBlackboard(bb : Blackboard)
ModelLoader
+load()
Validator
+validate()
UML2MetamodelValidator
MatildaProfileCDValidator
MatildaProfileSDValidator
IntegratedDiagramValidator
JASTValidator
Transformer
+transform()
CD2JASTTransformer
SD2JASTTransformer
JAST2BytecodeTransformer
JavaExecuter

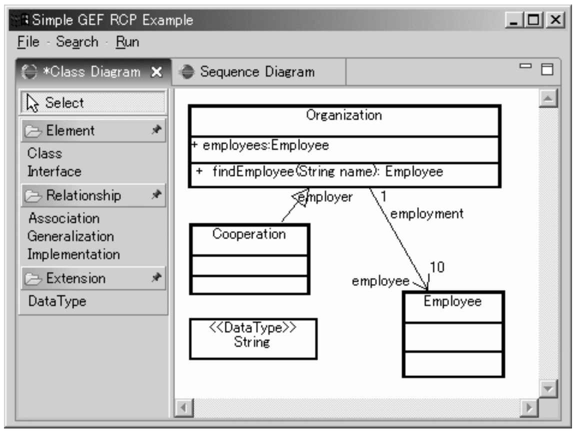
Figure 9. Class Structure of Plugins in Matilda UML VM

UML-VM accepts a UML model as an input in two ways: using UML-VM's modeling GUI or third-party modeling tools. UML-VM provides a modeling GUI, which allows developers to define UML class diagrams and sequence diagrams (Figures 10(a) and 10(b)). The modeling GUI serializes a UML model into XMI data and writes it to a blackboard (Figure 11). It is implemented with the Eclipse Rich Client Platform (RCP), and runs on the Eclipse platform. A *model loader* is a plugin used to read XMI data from third-party modeling tools and store the data in a blackboard (Figure 11).

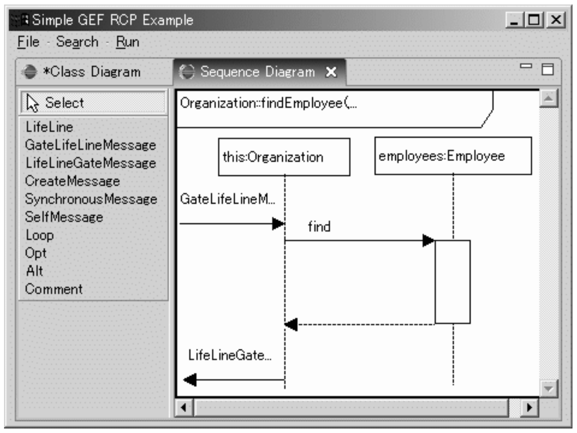
Each input UML model is validated with four validators: *UML metamodel validator*, *Matilda UML VM profile CD validator*, *Matilda UML VM profile SD validator* and *Integrated diagram validator*. A *UML metamodel validator* validates an input UML model against the UML metamodel using the `UML2Validator` class provided by Eclipse UML2[3]. A *Matilda UML VM profile CD/SD validator* validates an input model against the Matilda UML VM profile. These validation steps are intended to determine whether an input model is ready to be transformed to a JAST. An *integrated diagram validator* examines the consistency between a class diagram and sequence diagrams. Its major responsibility is to validate that sequence diagrams are defined for all methods of each class.

Listing 2 shows a code fragment of *Matilda UML VM profile CD validator*. The plugin reads an UML model from a blackboard and executes its validation process. It examines whether the model is compliant with the Matilda UML VM profile. For example, it validates that a model element stereotyped with ≪`UMLVMexecutable`≫ is a class that has a main method.

[3]http://www.eclipse.org/uml2



(a) Class Diagram



(b) Sequence Diagram

Figure 10. Matilda Modeling GUI

Once frontend plugins complete validating an input UML model, UML-VM transforms the model to a JAST with two backend plugins: *CD2JAST transformer* and *SD2JAST transformer*. They transform a class diagram and sequence diagrams to a JAST, respectively, using the data structures in the Eclipse Java Development Tooling (JDT). A *CD2JAST transformer* creates a new JAST based on the types (class and interface), data fields and method declarations in a UML model, and then it generates a JAST compilation unit for each type declaration. A *SD2JAST transformer* reads a JAST from a blackboard and updates it with method definitions mapped from each sequence diagram. A *JAST validator* validates the generated JAST, and a *JAST2Bytecode transformer* generates Java bytecode (i.e., class files) using Eclipse JDT. Finally, a *Java executor*, reads the generated class files, sets up a JVM, and executes the class files.

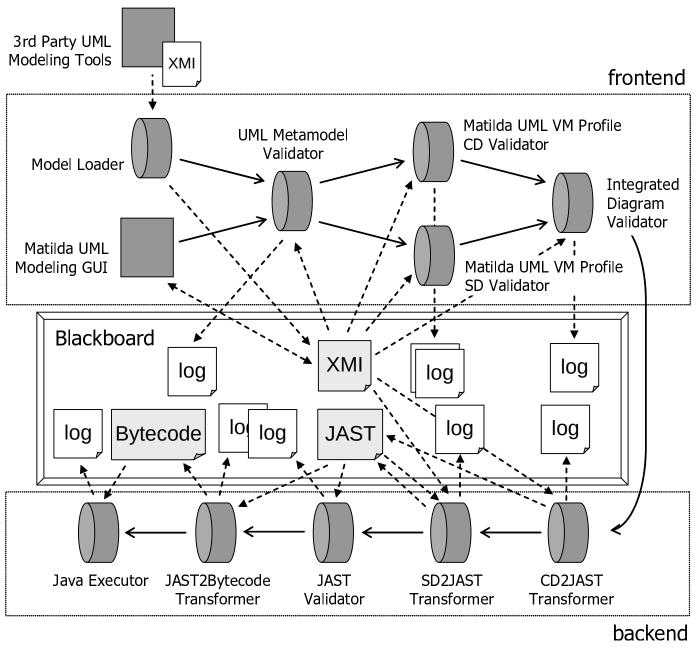


Figure 11. A Typical Pipeline Configuration for Matilda UML Virtual Machine

Listing 2 A Code Fragment of UML Profile CD Validator

```
class MatildaProfileCDValidator implements Plugin {
  void execute(Map<String, String> parameters)
    throws PluginException {
    // read a UML model from a blackboard
    UMLData data = (UMLData)this.readFromBB(parameters.get("uml"));
    UMLModel model = data.getModel();
    // validate the obtained model
    validate( model );
  }

  void validate( UMLModel model ){
    foreach (element in model){
      // check if each model element is stereotyped
      // with <<UMLVMexecutable>>
      if( element.stereotyped( "UMLVMexecutable" ) ){
        // checks whether
        //  - the element is a class
        //  - the class has a main method
        //  - the main method conforms a predefined signature (public void main(...))
    } }
    // if validation fails, an exception is thrown.
    if( valid != true ){
      throw new PluginException("An input UML model is invalid");
} } }
```

Figure 11 shows a typical pipeline configuration for UML-VM. In this configuration, a pipeline executes plugins sequentially and controls their execution. For example, when a blackboard receives an execution error log from a plugin, a pipeline stops executing plugins. Figure 12 shows another pipeline configuration customized for distributed software development. In this configuration, class diagrams and sequence diagrams are intended to be developed at physically dispersed places. Once they are validated, an *integrated diagram validator* examines their consistency. Then, the validated models are transformed to Java bytecode via JAST as shown in Figure 11.

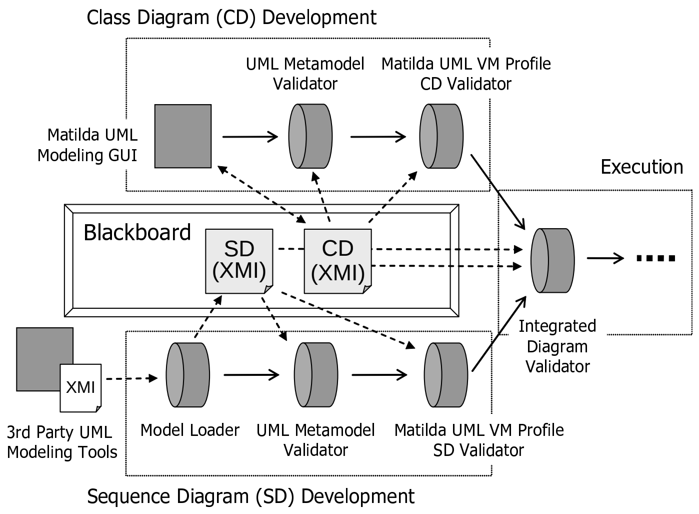

Class Diagram (CD) Development
UML Metamodel Validator
Matilda UML VM Profile CD Validator
Matilda UML Modeling GUI
Execution
Blackboard
SD (XMI)
CD (XMI)
Integrated Diagram Validator
XMI
3rd Party UML Modeling Tools
Model Loader
UML Metamodel Validator
Matilda UML VM Profile SD Validator
Sequence Diagram (SD) Development

Figure 12. A Pipeline Configuration for Distributed Software Development with Matilda UML Virtual Machine

An Example Application for Matilda UML Virtual Machine

This section shows an example application built with UML-VM. The example application is a command-line calculator that accepts an arithmetic expression in the Reverse Polish Notation and returns a calculation result. It supports summation, difference, division, multiplication and factorial operations. Figure 13 shows the class diagram for the calculator application. `Calculator` is the execution entry class, which is stereotyped with ≪`UMLVMexecutable`≫; it has the main method to which an input arithmetic expression is passed. An input arithmetic expression can be passed as a part of the application's pipeline configuration. Except local variables, all variables and methods are defined in the class diagram. (Local variables are defined in sequence diagrams.) UML attributes and associations are mapped to Java data fields. UML operations are mapped to Java method declarations that have empty bodies.

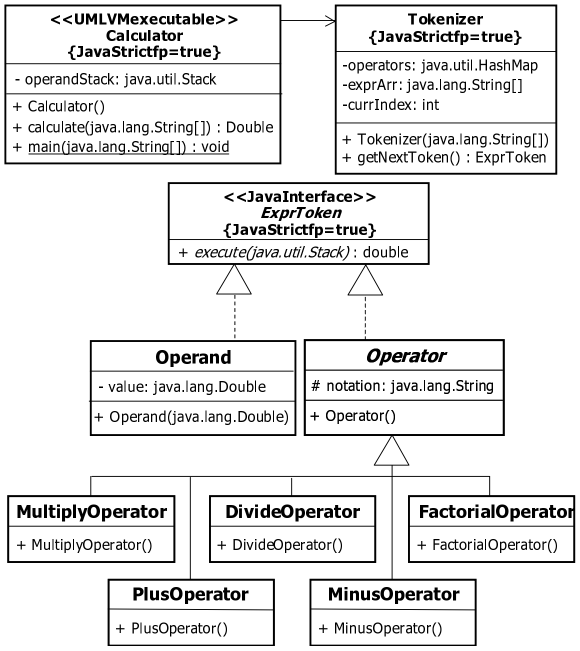


Figure 13. Class Diagram of an Example Calculator Application

Currently, UML-VM requires developers to define a sequence diagram for each operation/method. Figure 14 shows the sequence diagram for `getNextToken()` of `Tokenizer` (see also Figure 13). Each sequence diagram is described with the `sd` frame. The upper left corner of each `sd` frame indicates the method signature that the frame (sequence diagram) models. `getNextToken()` is used to obtain tokens of an input arithmetic expression one by one. The tokens are stored in `exprArr` (an array of string data)[4]. `Tokenizer` keeps track of the index of the next token to be obtained, using `currIndex`, and `getNextToken()` returns an instance of `Operator` or `Operand` depending on the type of the token being obtained. The entry and exit points to/from a sequence diagram is represented by an arrow (message) from/to the left most edge of the diagram. The arrow labeled with `getNextToken()` shows the entry point, and the arrow labeled with `nextToken` shows the exit point. (`nextToken` contains a value returned to a caller of `getNextToken()`.)

[4]`Calculator` is designed to pass an input arithmetic expression to `Tokenizer` via its constructor. In the constructor, `Tokenizer` tokenizes the passed expression and stores tokens in `exprArr`.

The object `this` and its lifeline represent the execution flow of a method (or sequence diagram). Each sequence diagram can reference the data fields and methods declared in the class of `this`. For example, the diagram in Figure 14 can reference `exprArr`, `operators` and `currIndex`, which are the data fields of `Tokenizer`.

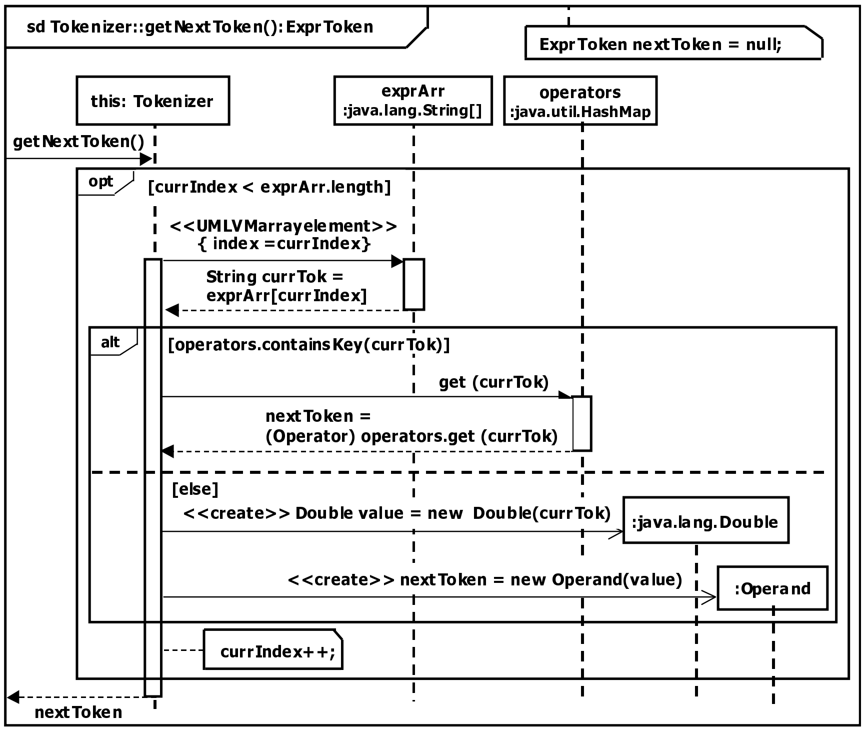

sd Tokenizer::getNextToken(): ExprToken
ExprToken nextToken = null;
this: Tokenizer
exprArr :java.lang.String[]
operators :java.util.HashMap
getNextToken()
opt
[currIndex < exprArr.length]
<<UMLVMarrayelement>> { index =currIndex}
String currTok = exprArr[currIndex]
alt
[operators.containsKey(currTok)]
get (currTok)
nextToken = (Operator) operators.get (currTok)
[else]
<<create>> Double value = new Double(currTok)
:java.lang.Double
<<create>> nextToken = new Operand(value)
:Operand
currIndex++;
nextToken

Figure 14. A Sequence Diagram of an Example Calculator Application

Matilda uses the `opt`, `alt` and `loop` fragments to specify control flows. Figure 14 uses the `opt` and `alt` frames to define *if* and *if-then* control flows, respectively. Guard conditions for the frames are represented with the expressions between `[` and `]`.

The messages (arrows) between the objects in a sequence diagram are either synchronous, reply or ≪create≫ messages. A synchronous message indicates a method call and parameters associated with the call. For example, in Figure 14, calling `get()` on the instance `operations` of `HashMap` is expressed with a synchronous message. A reply message represents the return from a method call, and indicates the assignment of a return value to a variable. In Figure 14, the return value of calling `get()` on `operations` is casted to `Operator`[5], and the casted value is assigned to `nextToken`. A ≪create≫ message represents an instantiation of a class. It points a class being instantiated, passes parameters to the class's constructor, and specifies the assignment of a newly created instance to a variable. In Figure 14, an instance of `Double` is created, and the instance is assigned to `variable`.

Local variables are defined as the notes attached to `sd` frames or fragments (e.g., `ExprToken nextToken` in Figure 14), within a reply message (e.g., `Token nextToken`, or within a ≪create≫

[5]`operations` maintains pairs of a string and object representing an operator (e.g., a pair of "+" and an instance of `PlusOperator`)

message (e.g., `Double valueST`)). The scope of each local variable is limited to the innermost fragment or `sd` frame.

Empirical Evaluation of Matilda UML Virtual Machine

This section empirically evaluates the execution overhead and memory footprint of UML-VM. Its pipeline is sequentially configured with eight plugins in order to (1) load an input UML model with a *model loader (ML)*, (2) validate the input model with a *UML Metamodel Validator (MV)*, a *Matilda UML VM profile CD validator (CDV)* and a *Matilda UML VM profile SD validator (SDV)*, (3) transform the validated model to a JAST with a *CD2JAST transformer (CDJ)* and a *SD2JAST transformer (SDJ)*, (4) transform the generated JAST to Java bytecode with a *JAST2Bytecode transformer (JBC)*, and (5) execute the generated bytecode with a *Java executor (JE)*. All measurements used a Sun J2SE 5.0.4 VM running on a Windows 2000 PC with an AMD Sempron 3.0 Ghz CPU and 512 MB memory space. Plugins are executed on the same process in the PC, and a blackboard run on a different process on the same PC.

Figure 15 shows the overhead to execute each plugin. The overhead includes the time for each plugin to process an input model, which contains varying numbers of classes (from 1 to 100 classes)[6] and read/write the input model from/to a blackboard. The proportion of each plugin's overhead to total overhead does not change significantly by varying the number of classes in an input model. The overhead of MV is extremely larger than those of other plugins. It occupies over 60% of total overhead. This result comes from the performance of `UML2Validator` in Eclipse UML2, which UML-VM uses to validate input UML models. The execution of MV can be omitted to improve the total overhead by extending the Matilda modeling GUI (Figure 10) so that it validates an input model in background while developers draw the model.

Table 3: Execution Overhead of UML-VM's Frontend and Backend

	Matilda (sec)			
# of classes	**Frontend**	**Backend**	**Total**	**javac (sec)**
10	15.2	4.4	18.4	1.0
25	37.3	11.0	45.2	1.2
50	76.7	21.6	92.1	1.4
70	108.2	30.4	129.9	1.5
100	153.9	45.9	187.2	1.7

Table 3 shows the overhead to execute frontend plugins (ML, MV, CDV and SDV) and backend plugins (ML, CDJ, SDJ and JBC) as well as the overhead of `javac` to compile Java code equivalent to input models. By comparing the backend overhead and `javac` overhead, because `javac` does not validate UML model elements, Table 3 shows that UML VM's performance is comparable with `javac` when the number of classes is less than 25 in an input model. (UML-VM's overhead is less than 10 times of the `javac` overhead.)

Figure 16 shows the breakdown of plugin execution overhead. Each plugin's overhead is divided to the time to process an input model containing 25 classes and the time to access a black-

[6]Each class has a method that contains message sequences corresponding to 100 lines of code (LOC) in Java. This LOC is obtained from the average per-class LOC (101.2) in major development environments such as J2SE 5.0 standard library, JBoss 4.0.4, Mule ESB 1.2, ArgoUML 0.20 and Teamwork 3.0.

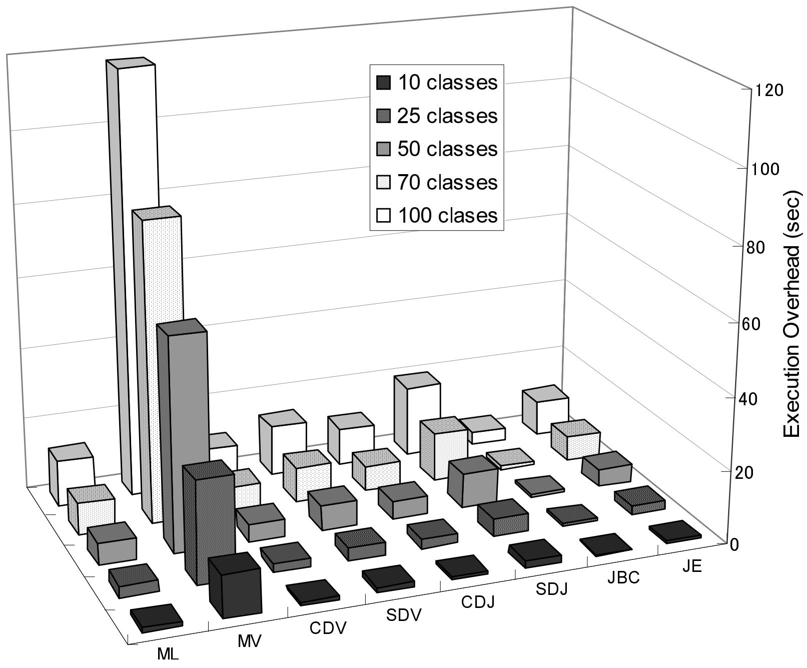
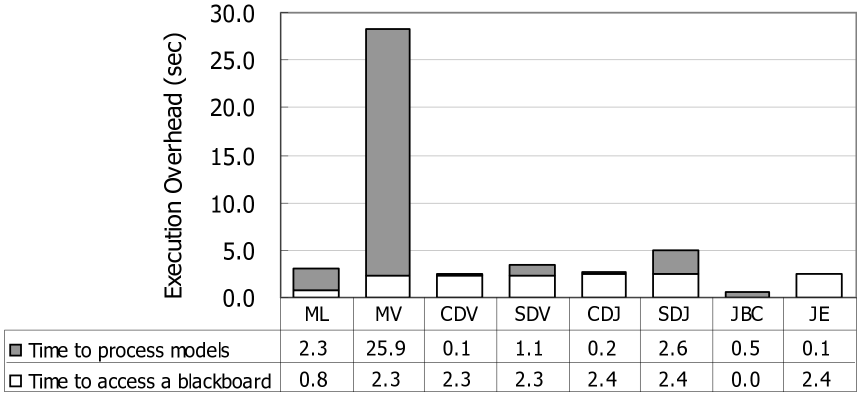


Figure 15. Execution Overhead of Plugins

board to write/read the model. Every plugin is efficient enough to process an input model except MV. Since it is relatively heavyweight to transform XMI data to an in-memory model representation[7], the time to access a blackboard is much longer than the time to process an input model (except the case of MV). For example, in CDV, blackboard access takes 23 times longer than processing an input model. Note that JBC reads a JAST from a blackboard; however, the blackboard access overhead is very small (less than 0.1 second) because JBC simply transforms the JAST to Java bytecode rather than transforming it to an in-memory model representation.



	ML	MV	CDV	SDV	CDJ	SDJ	JBC	JE
▨ Time to process models	2.3	25.9	0.1	1.1	0.2	2.6	0.5	0.1
□ Time to access a blackboard	0.8	2.3	2.3	2.3	2.4	2.4	0.0	2.4

Figure 16. Breakdown of UML VM Plugin Execution Overhead

[7]When a plugin reads XMI data from a blackboard, it compresses the data with the zip encoding to reduce the data transmission overhead between the plugin and blackboard. For example, the XMI data containing 100 classes is compressed from 15.7 MB to 1.0 MB. This significantly reduces the data transmission overhead between plugins and a blackboard. However, it is still a heavyweight process to transform XMI data to an in-memory model representation.

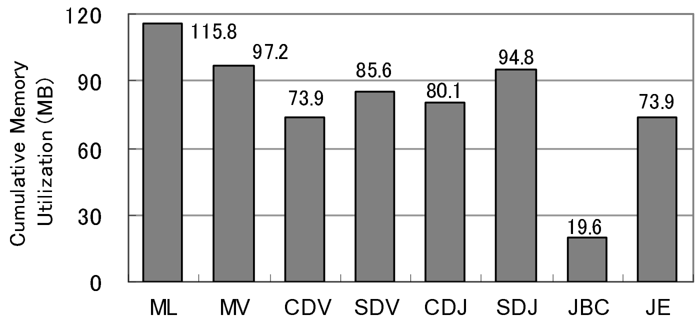


Figure 17. Memory Consumption of UML VM Plugins

In order to eliminate the blackboard access overhead, UML-VM can deploy multiple plugins in a single process so that they can pass an in-memory model representation between them. Table 4 shows a variation of Table 3; it measures the frontend and backend overhead when all the eight plugins run in the same process. As shown in this table, UML-VM's backend overhead is comparable with `javac`'s overhead when the number of input classes is less than 70. Tables 3 and 4 show that UML-VM works efficiently in small to medium scale applications.

Table 4: Execution Overhead of the Frontend and Backend

	Matilda (sec)			
# of classes	**Frontend**	**Backend**	**Total**	**javac (sec)**
10	12.2	2.4	13.7	1.0
25	29.7	5.7	33.1	1.2
50	61.8	11.2	68.2	1.4
70	87.3	15.6	96.2	1.5
100	123.8	24.7	138.8	1.7

Figure 17 shows the cumulative memory consumption of each plugin to execute an input model containing 70 classes. In this measurement, Java VM's garbage collection is disabled. Therefore, the memory consumption includes the footprint of each plugin and the amount of data the plugin generates. Compared with the size of XMI data each plugin reads from a blackboard (11 MB in the case of 70 classes in an input model), UML-VM's memory consumption is acceptable in small to medium scale applications. ML consumes memory space most because it loads an input model and the UML metamodel and Matilda UML VM profile, and validates the model against the UML metamodel definitions.

Matilda SOA Virtual Machine (SOA-VM)

This section describes another Matilda-based VM: SOA-VM. SOA is an emerging style of software architectures to build, integrate and maintain distributed applications (Bichler & Lin, 2006; Papazoglou & Heuvel, 2007). In SOA, each application is often designed with a set of *services* and a *business process*. Each service encapsulates the function of an application component, and each business process defines how services interact to accomplish a certain business goal. SOA-VM allows developers to model their service oriented applications in BPMN and UML and and directly execute the models.

Plugins in Matilda SOA Virtual Machine

SOA-VM (1) accepts a BPMN model that defines a business process (i.e., control and data flows among services) and UML sequence diagrams that define behaviors of services, (2) transforms the input models to Java code implementing services and a BPEL script, (3) deploys the generated Java code as XML web services on an application server, and (4) deploys the generated BPEL script on a BPEL engine to establish a workflow between the XML web services.

SOA-VM consists of 7 plugins: *model loader*, *BPMN2CD transformer*, *CD2Java transformer*, *SD2Java transformer*, *Java integrator*, *BPMN2BPEL transfomer* and *deployer*. Figure 18 shows a typical pipeline configuration. First, a *model loader* loads a BPMN model and UML sequence diagrams into a blackboard.

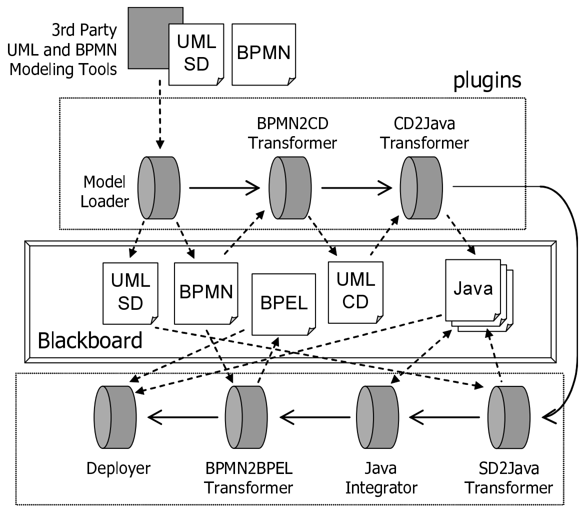


Figure 18. A Typical Pipeline Configuration for Matilda SOA VM

A *BPMN2CD transformer* extracts the structural aspect of an input BPMN model and generates a UML class diagram that defines the structure of services. A BPMN model generally consists of *pools*, *tasks* and *sequence/message flows*. A pool, represented by a rectangle, denotes a participant in a business process; for example, `Client` and `Server` in Figure 19. A task, represented as a rounded-corner rectangle, denotes a task performed by a participant; for example, `Send Query` in `Client`. A sequence flow, represented as a solid arrow, denotes the order of tasks performed by a participant. For example, `Client` executes `Send Query` and `Show Results` in order. A message flow is represented as a dashed arrow between two participants. A participant starts its process from a message start event (a circle with an envelope icon), when it receives a message from other participants. Also, a participant returns a message to a caller when its process ends with a message end event (a bold circle with an envelope icon). Sequence/message flows also define parameters that they carry. (Their graphical representations are not available in BPMN.) In Figure 19, `Send Query` returns a `String` value that is delivered to `Server`. `Show Results` takes a `String` value as a parameter. `Process Query` takes and returns a `String` value. A *BPMN2CD transformer* transforms a BPMN pool to a UML class and transforms a BPMN task to a UML method. Listing 3 shows a code fragment of a *BPMN2BPEL transformer*. This transformer transforms a BPMN model in Figure 19 to a UML diagram in Figure 20. Method parameters in a UML class are generated according to the parameter definitions in a BPMN model.

Listing 3 A Code Fragment of BPMN to Class Diagram Transformer

```
class BPMN2CD implements Plugin {
  void execute(Map<String, String> parameters)
    throws PluginException {
    // read a BPMN model from a blackboard
    BPMNData bpmndat = (BPMNData)this.readFromBB(parameters.get("bpmn"));
    BPMNModel bpmnmodel = bpmndat.getModel();

    // transform to BPMN
    UMLModel umlmodel = toUML(bpmnmodel);
    UMLData umldat = new UMLData(umlmodel);

    // write a UML model on a blackboard
    this.writeOnBB(parameters.get("umlclass"), umldat);
  }

  UMLModel toUML(BPMNModel bpmnmodel){
    UMLModel umlmodel = ... // create a new UML model
    // check each model element in a BPMN model
    foreach (element in bpmnmodel){
      if( element instanceof Pool ){ // if an element is Pool
        Class c = ...              // create a new UML class
        umlmodel.add(c);           // add a class to a UML model
        foreach (ce in element.ownedElements){  // check all nested elements
          if(ce instanceof Task){    // if a Pool contains a Task
            Method m = ...           // create a corresponding UML method
            c.add(m);                // add a method to a UML class
          }
        }
      }
    }
    return umlmodel;
} }
```

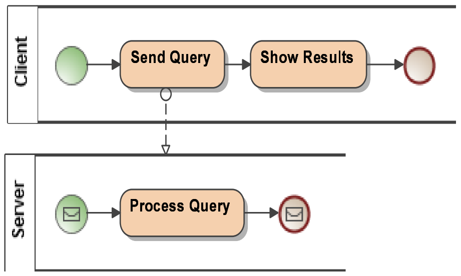


Figure 19. An Example BPMN Model

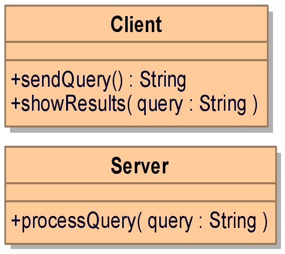

Client
+sendQuery() : String
+showResults(query : String)
Server
+processQuery(query : String)

Figure 20. A Generated UML Class Diagram

A *CD2Java transformer* transforms a generated UML class diagram into Java code. Methods are empty in the generated Java code because a BPMN model does not define behaviors of tasks. A *SD2Java transformer* transforms input UML sequence diagrams, which are defined with the Matilda UML VM profile, into Java code (method implementations). A *Java integrator* integrates Java code generated by a *CD2Java transformer* and a *SD2Java transformer* in order to complete Java implementations of services.

A *BPMN2BPEL Transformer* transforms a BPMN model into a BPEL script. After a BPEL script is generated, a *deployer* deploys the BPEL script and services on an application server. SOA-

VM currently uses Apache Axis[8] and Apache Orchestration Director Engine (ODE)[9] to operate XML Web services and a BPEL script, respectively. Listing 4 shows a code fragment of a *deployer*. First, it obtains a BPEL script from a blackboard and writes it as a file to be deployed on Apache ODE. Then, it obtains Java classes (i.e., services) from a blackboard and deploys them on Apache Axis. Finally, it downloads WSDL interfaces generated for deployed services, and copies the WSDL interfaces to a directory where a BPEL script is deployed. The BPEL script requires the WSDL interfaces to access services.

Listing 4 A Code Fragment of BPMN to Class Diagram Transformer

```
class Deployer implements Plugin {
  void execute(Map<String, String> parameters)
    throws PluginException {
    // obtain a directory to deploy a BPEL script.
    // For example, /tomcat/webapps/ode/WEB-INF/processes/SOAVM/
    String bpelPath = this.readFromBB(parameters.get("bpelpath"));

    // obtain a BPEL script from a blackboard and write it in bpelPath
    String bpelPath = this.readFromBB(parameters.get("bpel"));
    FileWriter writer = ...

    // obtain a directory to deploy services.
    // For example, /tomcat/webapps/axis/services/
    String deployPath = this.readFromBB(parameters.get("deploypath"));

    // obtain Java classes from a blackboard
    String[] classes = (String[])this.readFromBB(parameters.get("java"));
    // deploy each services
    foreach(class in classes){
      FileWriter j = ... // write a Java class in deployPath
      String wsdl = ...  // read a WSDL file corresponding to a Java class
      FileWriter w = ... // write a WSDL file in bpelPath
    }
} }
```

Figure 21 shows a pipeline configuration customized for distributed software development with SOA-VM. In this configuration, BPMN and UML sequence diagrams are intended to be developed at physically dispersed places. A network host accepts a BPMN model and transforms it into a BPEL script, and another network host accepts UML sequence diagrams and generates Java classes. Then, a deployer deploys generated Java classes and a BPEL script.

An Example Application for Matilda SOA Virtual Machine

This section shows an example application build with SOA-VM: an electronic travel arrangement application. Figure 22 shows a international travel arrangement process in BPMN. This input BPMN model contains four participants: `Travel Agent`, `Airline Reservation`, `Hotel Reservation` and `Currency Converter`. Once a travel agent receives an itinerary from a customer with the `Receive Itinerary` task, the travel agent calls `Airline Reservation` and `Hotel Reservation` in parallel to search airline tickets and hotel rooms according to the itinerary. Each itinerary contains the departure date, return dates, travel destination and currency that a customer uses. `Airline Reservation` and `Hotel Reservation` start their processes with `Message` events.

[8]ws.apache.org/axis/
[9]ode.apache.org

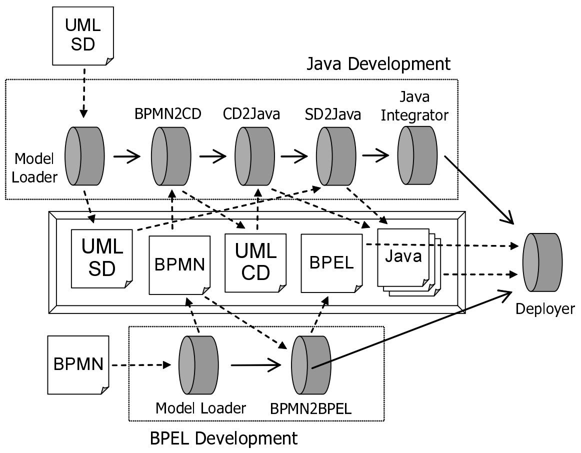


Figure 21. A Pipeline Configuration for Distributed Software Development with Matilda SOA Virtual Machine

The `Search` task searches airline tickets or hotel rooms. The `Convert Currency` task converts airfare and room charge to the currency a customer uses by sending a message to `Currency Converter`. Once `Currency Converter` receives a message, it executes `Convert` and returns a result to a caller. `Travel Agent` returns search results of airline tickets and hotel rooms to a customer by executing `Send Results`.

Figure 23 shows a fragment of a UML class diagram that a *BPMN2CD Transformer* generates from a BPMN model in Figure 22.

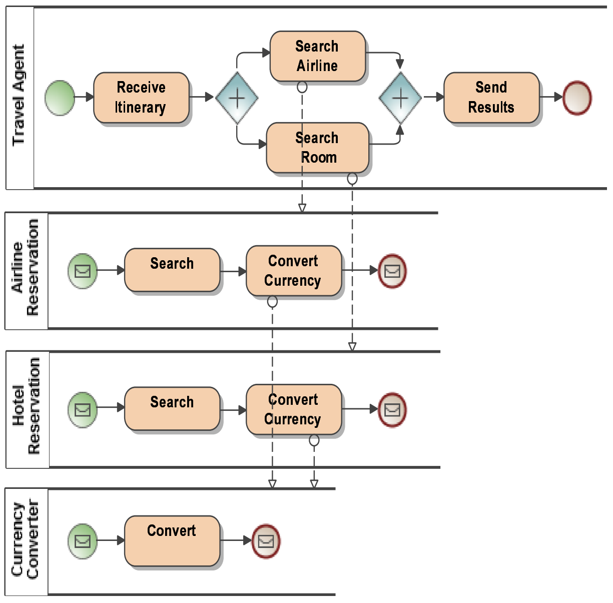


Figure 22. A Travel Arrangement Process in BPMN

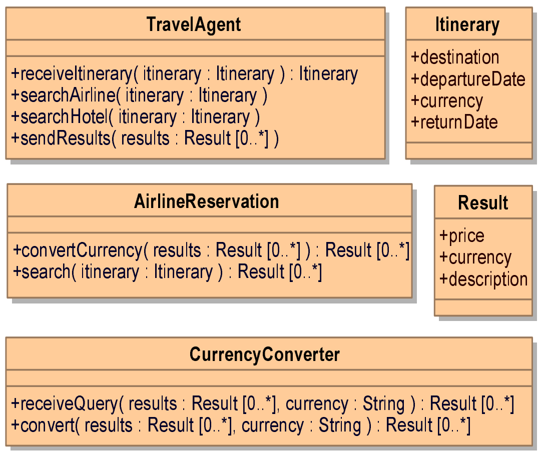

TravelAgent
+receiveItinerary(itinerary : Itinerary) : Itinerary
+searchAirline(itinerary : Itinerary)
+searchHotel(itinerary : Itinerary)
+sendResults(results : Result [0..*])
Itinerary
+destination
+departureDate
+currency
+returnDate
AirlineReservation
+convertCurrency(results : Result [0..*]) : Result [0..*]
+search(itinerary : Itinerary) : Result [0..*]
Result
+price
+currency
+description
CurrencyConverter
+receiveQuery(results : Result [0..*], currency : String) : Result [0..*]
+convert(results : Result [0..*], currency : String) : Result [0..*]

Figure 23. Generated UML Classes

Figure 24 shows one of input UML sequence diagrams, which defines the behavior of the `Convert` task in `Currency Converter`. This task takes two parameters, a set of search results (`results`) and the currency that a customer uses (`currency`), and converts the currency that each of `results` uses (from USD to EUR, or from EUR to USD).

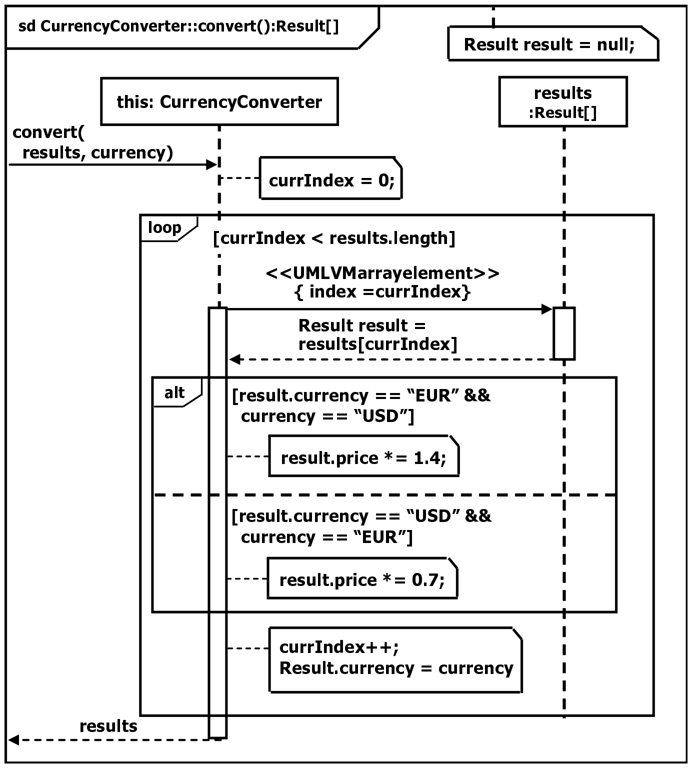

sd CurrencyConverter::convert():Result[]
Result result = null;
this: CurrencyConverter
results :Result[]
convert(results, currency)
currIndex = 0;
loop
[currIndex < results.length]
<<UMLVMarrayelement>> { index =currIndex}
Result result = results[currIndex]
alt
[result.currency == "EUR" && currency == "USD"]
result.price *= 1.4;
[result.currency == "USD" && currency == "EUR"]
result.price *= 0.7;
currIndex++; Result.currency = currency
results

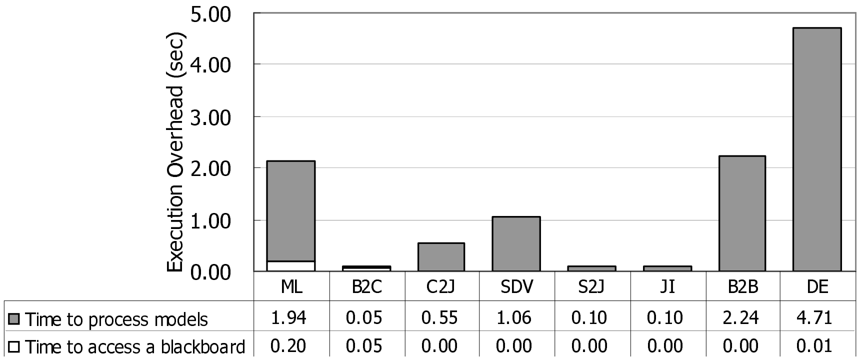
Figure 24. An Input Sequence Diagrams

Empirical Evaluation of Matilda SOA Virtual Machine

This section empirically evaluates the execution overhead and memory footprint of SOA-VM. Matilda's pipeline is sequentially configured with seven plugins in order to (1) load input models with a *model loader (ML)*, (2) transform a BPMN model to a UML class diagram with a *BPMN2CD transformer (B2C)*, (3) transform a UML class diagram to Java code with a *CD2Java transformer (C2J)*, (4) validate UML sequence diagrams with a *Matilda UML VM profile SD validator (SDV)*, (5) transform UML sequence diagrams to Java code with a *SD2Java transformer (S2J)*, (6) integrate Java code with a *Java integrator (JI)*, (7) transform a BPMN model to a BPEL script with a *BPMN2BPEL transformer (B2B)*, and (8) deploy Java code and a BPEL script on Apache Axis and Apache ODE with a *deployer (DE)*.

Figure 25 shows the breakdown of plugin execution overhead of each plugin when SOA-VM processes a BPMN model in Figure 22. Plugins and a blackboard are deployed on the same host. Each plugin's overhead is divided to the time to process a model and the time to access a blackboard to write/read the model. As Figure 25 illustrates, every plugin runs efficient enough; the total overhead is 9.9 second. Similar to UML-VM, a *model loader* has a relatively large overhead to transform XMI data to in-memory model representations. Also, a *deployer*'s overhead is the largest because it involves in downloading and uploading Java classes and WSDL interfaces.

Since SOA-VM uses Apache Axis and Apache ODE to deploy applications, it requires a host where both software installed beforehand. Since plugins in Matilda-based virtual machine can be



	ML	B2C	C2J	SDV	S2J	JI	B2B	DE
■ Time to process models	1.94	0.05	0.55	1.06	0.10	0.10	2.24	4.71
□ Time to access a blackboard	0.20	0.05	0.00	0.00	0.00	0.00	0.00	0.01

Figure 25. Breakdown of SOA VM Plugin Execution Overhead

distributed over a network, SOA-VM can use a *deployer* on a remote host with Apache Axis and Apache ODE via a distributed pipeline. It allows multiple SOA-VMs to share a host for deploying the final applications and frees application developers to prepare a host themselves. Depending on requirements, a configuration of a distributed pipeline can be altered. For example, when a machine where application developers work is not powerful enough, all plugins except a *model loader* are deployed on a remote host and all processes are performed on a remote host.

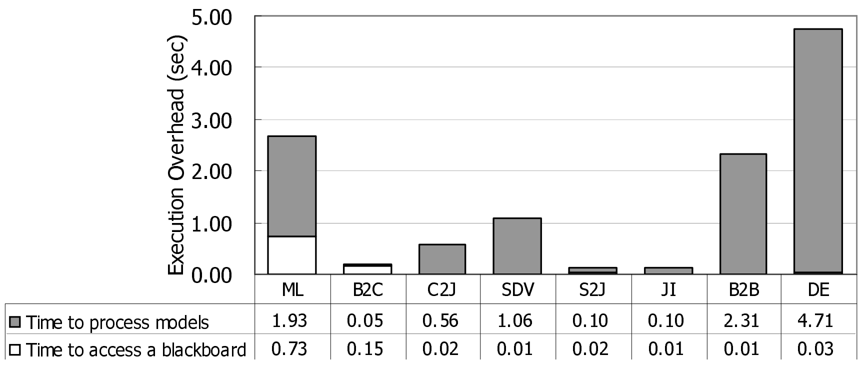
Table 5 shows the overhead when plugins and a blackboard are deployed on different hosts. Figure 26 shows the breakdown of plugin execution overhead when plugins and a blackboard are distributed over a network. Plugins exchange data via a blackboard deployed on a remote host that connected via 802.11g wireless router. Since an input model (i.e., a BPMN model and sequence diagrams) are same as in the previous measurement study, the overhead to process models are almost same. Although the overhead to access to a blackboard increases 250% when a blackboard is deployed on a remote host, the total overhead increases only 7 % since the overhead to access to a remote blackboard is quite small compare to compared to the overhead to process models. It shows that Matilda works efficiently in distributed environment as well as in a non-distributed environment.

Table 5: Overhead of SOA-VM

	Process Models (ms)	**Access to a Blackboard (ms)**	**Total (ms)**
SOA-VM on a single host	10.78	0.28	11.02
SOA-VM on distributed hosts	10.82	0.98	11.80

Figure 27 shows the cumulative memory consumption of each plugin. (Java VM's Garbage collection is disabled.) As well as in UML-VM, a *model loader* consumes memory space most because it loads a BPMN model and UML sequence diagrams, however SOA-VM's memory consumption is small enough and acceptable.

Table 6 shows a set of artifacts consisting of the final deployed application. As the table shows, even a simple service-oriented application requires application developers to learn various technologies and tools. SOA-VM, however, hides the details of these implementation technologies and tools. This way, SOA-VM removes the existence of a programming layer and allows application developers to work on only a modeling layer.



	ML	B2C	C2J	SDV	S2J	JI	B2B	DE
■ Time to process models	1.93	0.05	0.56	1.06	0.10	0.10	2.31	4.71
□ Time to access a blackboard	0.73	0.15	0.02	0.01	0.02	0.01	0.01	0.03

Figure 26. Breakdown of SOA-VM Plugin Execution Overhead (Distributed Pipeline)

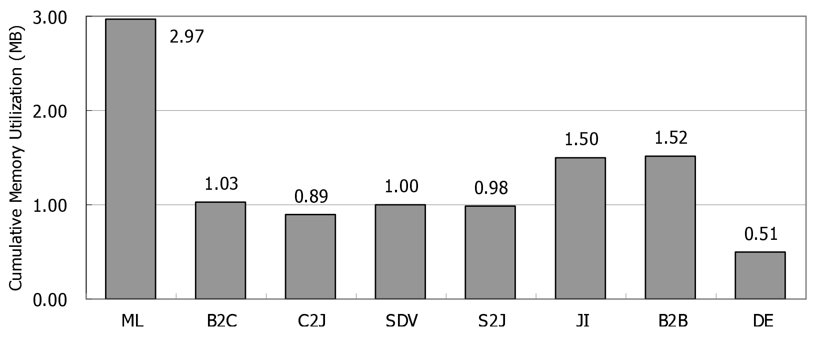


Figure 27. Memory Consumption of SOA-VM Plugins

Related Work

This chapter describes a set of extensions to the authors' prior work (Wada, Babu, Malinowski, Suzuki, & Oba, 2006). One extension is to investigate a new application of Matilda, i.e., SOA-VM, which (Wada et al., 2006) does not consider.

There are several work to investigate UML virtual machines. (Riehle, Fraleigh, Bucka-Lassen, & Omorogbe, 2001) addresses the issues of validating models and generating executable code. It maintains causal connections among four meta layers in UML (M0 to M3 layers), and uses the connections to validate models and propagate changes between models. For example, the connections can be used to validate the consistency between M1 and M2 models and reflect changes in an M2 model to M1 models. Although Matilda implements model validation, it does not explicitly maintains causal connections among different meta layers. (Riehle et al., 2001) does not support behavioral modeling, and it is not clear how to transform models to executable code. Matilda supports behavior modeling, and provides workable plugins to generate executable code.

Table 6: Artifacts consisting of an Application

Artifacts	**LOC**
6 Java Classes	145
4 WSDL Interfaces	94
A BPEL Script	60

ASM virtual machine (Shen, Compton, & Huggins, 2003) and USE (UML-based Specification Environment) (Gogolla, Bohling, & Richters, 2003) address the issue of validating models. They support Object Constraint Language (OCL) (*UML Super Structure Specification 2.1.2*, 2007) to validate the consistency and integrity of models. Matilda is similar to ASM VM and USE in that it also supports model validation; however, the model validation logic in Matilda is hard coded in model validator plugins rather than using OCL. Matilda currently puts a higher priority on model execution through operating plugins in distributed environments. Matilda validates the consistency and integrity of class diagrams and sequence diagrams, while ASM VM and USE checks those of class diagrams only. They do not focus on model execution.

Similar to Matilda, executable UML (xUML) focuses on directly executing models. In xUML, developers use class diagrams for structural modeling, and statechart diagrams and textural action languages for behavioral modeling (Mellor & Balcer, 2002; Raistrick, Francis, & Wright, 2004; Balcer, 2003). Action languages implement the UML action semantics, defined as a part of the UML specification (*UML Super Structure Specification 2.1.2*, 2007). However, the UML action semantics does not provide the standard language syntax; therefore, different action languages have different syntax with different (proprietary) extensions (e.g., BridgePoint[10] and iUML[11]). This means that developers need to learn action language syntax every time they use different xUML tools. Also, there is no interoperability of models between different xUML tools because different xUML tools assume different subsets of the UML metamodel. Thus, an xUML tool cannot correctly interpret a model that is defined with other xUML tools. On the other hand, Matilda uses the UML metamodel and its standard extensions (profiles) for both structural and behavioral modeling. (Matilda does not require developers to use non-standard mechanisms to build and execute models.) It is more open for future extensions and integration with third party tools such as code generators and optimizers. Furthermore, Matilda inherently supports the distributed execution of plugins. No xUML tools do not address this issue.

openArchitectureWare[12] is similar to Matilda in that it provides a set of plugins (e.g., model loader, validators and transformers) and allows developers to form a sequence of plugins using its workflow language. However, unlike Matilda, it does not support executing models and deploying plugins in a distributed manner.

The current common practice in MDD is to model application designs with modeling languages and transform them to skeleton source code. For example, OptimalJ[13], Rose XDE[14], Together[15], UMLX (Willink, 2002), KMF (Patrascoiu, 2004) and J3 (White, Schmidt, & Gokhale, 2005) support UML. Visual Paradigm[16], Intalio[17] and eClarus[18] support BPMN. (Gardner, 2003; Chowdhary et al., 2006) leverages UML for modeling business processes. Unlike them, Matilda focuses on direct execution of UML models so that no manual programming is necessary (see Figures 1 and 2).

Several research and products facilitate the simulation of business processes such as TIBCO

[10]www.mentor.com/products/sm/uml_suite/

[11]www.kc.com

[12]www.openarchitectureware.org/

[13]www.compuware.com/products/optimalj/

[14]www.ibm.com/software/awdtools/developer/rosexde/

[15]www.borland.com/together/architect/

[16]www.visual-paradigm.com/

[17]bpms.intalio.com/

[18]www.eclarus.com/

Business Studio[19], activeBPEL[20] and (Jansen-Vullers & Netjes, 2006). Matilda SOA-VM is similar to them in that it also supports direct model execution; however, SOA-VM generates artifacts required for the final applications (e.g., WSDL interfaces and Java classes) and actually deploys on an application server. Therefore, SOA-VM verifies not only business process models but also whole systems including artifacts and runtime environment. Verification through SOA-VM is more comprehensive and it reduces the cost and time for verifying and testing the final applications compared to business process simulators.

Conclusion

This chapter describes and empirically evaluates a new MDD framework, Matilda, which is a framework to build virtual machines for software models. Matilda allows developers to analyze, design and execute their applications consistently at the modeling layer by hiding the existence of programming layer. It also enables distributed software development in which developers can collaboratively build, integrate and transform software models at physically dispersed places. Empirical measurement results show that Matilda-based virtual machines work efficiently with small memory consumption in small to medium scale applications.

Several extensions to Matilda are planned as future work. One of them is to implement and evaluate additional plugins. They include a model debugging plugin. It is being designed to intercept transformation errors of input models and runtime errors of generated Java bytecode, and identify the sources of the errors in textual and graphical manners. For example, the error sources will be graphically displayed on input BPMN/UML diagrams so that developers can intuitively understand and fix the errors. The design of the model debugging plugin will follow the Model-level Testing and Debugging Specification (*Model-level Testing and Debugging Specification, Revised Submission*, 2006), which Object Management Group standardizes.

An extended set of empirical measurements is also planned to provide additional performance implications of Matilda. Matilda will be evaluated with larger scale of applications on larger size of network environments (e.g., PlanetLab (Bavier et al., 2004)) to identify the impacts of application/network size on Matilda's performance.

References

Balcer, M. (2003, June). An executable uml virtual machine. In *Omg workshop on uml for enterprise applications: Delivering the promise of mda.*

Bavier, A., Bowman, M., Chun, B., Culler, D., Karlin, S., Muir, S., et al. (2004, March). Operating system support for planetary-scale services. In *Symposium on network system design and implementation.*

Bichler, M., & Lin, K. (2006, June). Service-oriented computing. *IEEE Computer*, *39*(6).

Booch, G., Brown, A., Iyengar, S., Rumbaugh, J., & Selic, B. (2004, December). An mda manifesto. In *The mda journal: Model driven architecture straight from the masters* (chap. 11). Meghan-Kiffer Press.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., Sommerlad, P., et al. (1996). *Pattern-oriented software architecture, volume 1: A system of patterns*. John Wiley & Sons.

Business process modeling notation (bpmn) 1.0. (2004). Business Process Modeling Initiative.

Chowdhary, P., Bhaskaran, K., Caswell, N. S., Chang, H., Chao, T., Chen, S. K., et al. (2006). Model driven development for business performance management. *BM SYSTEMS JOURNAL*, *45*(3), 587–605.

Fuentes, L., & Vallecillo, A. (2004, April). An introduction to uml profiles. *The European journal for the Informatics Professional*, *5*(2).

[19]www.tibco.com/devnet/business_studio/

[20]www.activevos.com

Gardner, T. (2003, July). Uml modeling of automated business processes with a mapping to bpel4ws. In *Ecoop workshop on object orientation and web services.*

Gogolla, M., Bohling, J., & Richters, M. (2003). Validation of uml and ocl models by automatic snapshot generation. In *International conference on unified modeling language.*

Jansen-Vullers, M., & Netjes, M. (2006, Oct). Business process simulation - a tool survey. In *Workshop and tutorial on practical use of coloured petri nets and the cpn tools.*

Mellor, S., & Balcer, M. (2002). *Executable uml: A foundation for model driven architecture*. Addison-Wesley.

Model-level testing and debugging specification, revised submission. (2006). Object Management Group. (OMG document number: realtime/2006-01-05)

Mof 2.0 xmi mapping specification. (2007). Object Management Group.

Papazoglou, M., & Heuvel, W. (2007, July). Service oriented architectures: Approaches, technologies and research issues. *The International Journal on Very Large Data Bases*, *16*(3), 389–415.

Patrascoiu, O. (2004, September). Mapping edoc to web services using yatl. In *International conference on enterprise distributed object computing.*

Raistrick, C., Francis, P., & Wright, J. (2004). *Model driven architecture with executable uml*. Cambridge University Press.

Riehle, D., Fraleigh, S., Bucka-Lassen, D., & Omorogbe, N. (2001). The architecture of a uml virtual machine. In *Acm international conference on object-oriented programming, systems, languages and applications.*

Sendall, S., & Kozaczynki, W. (2003, September/October). Model transformation: The heart and soul of model-driven software development. *IEEE Software*, *20*(5).

Shen, W., Compton, K., & Huggins, J. (2003). A method of implementing uml virtual machines with some constraints based on abstract state machines. In *Ieee asia-pacific software engineering conference.*

Uml super structure specification 2.1.2. (2007). Object Management Group.

Vermeulen, A., Beged-Dov, G., & Thompson, P. (1995, October). The pipeline design pattern. In *Acm sigplan conference on object-oriented programming, systems, languages, and applications, workshop on design patterns for concurrent parallel and distributed object-oriented systems.*

Wada, H., Babu, E. M. M., Malinowski, A., Suzuki, J., & Oba, K. (2006, November). Design and implementation of the matilda distributed uml virtual machine. In *Iasted international conference on software engineering and applications.*

Web services business process execution language. (2003). OASIS.

White, J., Schmidt, D., & Gokhale, A. (2005, October). Simplifying autonomic enterprise java bean applications. In *Acm/ieee international conference on model driven engineering languages and systems.*

Willink, E. (2002, November). Umlx: A graphical transformation language for mda. In *Acm international conference on object-oriented programming, systems, languages and applications.*