DESIGN AND IMPLEMENTATION OF THE MATILDA DISTRIBUTED UML VIRTUAL MACHINE

Hiroshi Wada and Junichi Suzuki
Department of Computer Science
University of Massachusetts, Boston
Boston, MA 02125-3393
email: {shu, jxs}@cs.umb.edu

Manikya Madhu Babu Eadara
Prolifics
Waltham, MA 02451
email: em@cs.umb.edu

Adam Malinowski
Motorola, Inc.
Tewksbury, MA 01876
email: AdamMalinowski@motorola.com

Katsuya Oba
OGIS International, Inc.
Palo Alto, CA 94301
email: oba@ogis-international.com

ABSTRACT
This paper describes a distributed UML virtual machine, called Matilda, which allows developers to design their applications as UML models and directly execute the models. Matilda accepts a UML model as an input, validates it against the UML metamodel, constructs a Java abstract syntax tree (JAST) according to the input model, and executes Java bytecode generated from the JAST. The architecture of Matilda is designed as a pipeline of plugins, each of which implements a functionality in Matilda such as validating UML models. The pipeline architecture allows Matilda to flexibly configure its structure and behavior by replacing a plugin with another one or changing the order of plugins. Also, Matilda can deploy plugins on multiple network hosts and seamlessly connect them to form a pipeline. This facilitates distributed software development in which developers collaboratively work on UML models at physically dispersed places. This paper describes the design, implementation and performance of Matilda.

KEY WORDS
Software Design, Modeling Languages, Model Driven Software Development

1 Introduction

Modeling is becoming a critical process in software development, and software modeling technologies have matured to the point where they can offer significant leverage in all aspects of software development [1]. For example, the Unified Modeling Language (UML) provides a rich set of modeling notations and semantics, and allows developers to specify and communicate their application designs at a higher level of abstraction [2]. The notion of model-driven development (MDD) aims to 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 technologies [3, 4]. Traditional MDD frameworks allow developers to model their applications with a modeling language such as UML, generate skeleton code in a programming language such as Java, and complete the generated skeleton code by, for example, adding method code (Figure 1). The key issues in this model transformation process are *abstraction gap between modeling layer and programming layer* and *lack of traceability between models and programs*. When programmers complete generated skeleton code to the final (compilable) code, they often suffer from abstraction gap between modeling layer and programming layer 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 broader views of application designs, and they have to repeatedly go up and down abstraction gap to identify where to implement what in skeleton code. 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 by losing synchronization with them. Due to the above two issues, traditional MDD frameworks do not maximize the benefits of modeling application designs at a higher level of abstraction.

This paper describes a new MDD framework, called Matilda, which addresses the above two issues. Matilda is an execution runtime engine (or virtual machine) for software models described in UML. It allows developers to model their applications with UML and directly execute the models through automatically transforming them to executable code (Figure 2). Matilda addresses the issues in the current MDD practice (i.e., abstraction gap and lack of traceability) 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).

Matilda accepts a UML model (UML 2.0 class diagrams and sequence diagrams) as an input, validates it against the UML metamodel and transforms the input

model to an implementation 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 particular implementation and/or deployment technologies. For example, a transformation rule may specialize an input model to a database system, while another rule may specialize it to a remoting middleware. Currently, Matilda transforms an implementation specific model to a Java abstract syntax tree (JAST) and generates Java bytecode from the JAST.

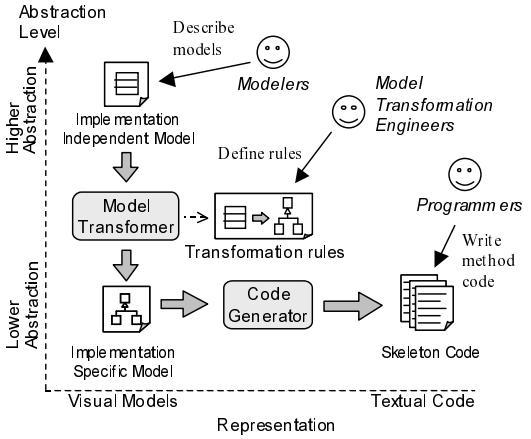


Figure 1. Traditional MDD Process

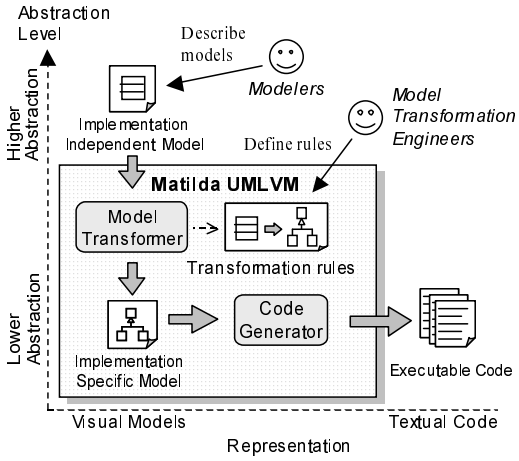


Figure 2. Development Process with Matilda

The architecture of Matilda is designed as a pipeline of plugins. Different plugins implement different functionalities in Matilda, such as visualizing UML models, validating UML models and building a JAST. The pipeline architecture allows Matilda to flexibly configure its structure and behavior by replacing a plugin with another one or changing the order of plugins. Also, Matilda's pipeline can be distributed. Matilda can spread plugins over 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 UML models at physically dispersed places.

2 Design Principles

Matilda is designed based on the following principles.

1. **Avoidance of Round-Tripping.** In order to address the issues described in Section 1 (i.e., abstraction gap and traceability issues), Matilda inherently avoids the round-trips between models and source code by hiding the existence of source code from developers. All design changes are directly made on models instead of source code (see 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 in an ad-hoc manner. It represents UML and Java metamodels as a set of objects (APIs), and aids its plugins to implement their functionalities on metamodel basis.

3. **Modularity and Loose Couping.** 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. Plugins are intended to be reused in a wide range of development projects (i.e., various pipeline configurations).

4. **Configurability.** Matilda is expected to be used in a variety of development projects; from in-house development, distributed open-source development to offshore 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 particular deployment descriptors as well as Java code generation plugin. Therefore, Matilda is designed to make pipelines configurable and extensible. It defines standard interfaces 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 standard plugin interface.

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 it invokes a local or remote plugin in a pipeline.

3 Architecture

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

The pipeline architecture of Matilda is designed based on the Pipes and Filters architectural pattern [5, 6]. This pattern provides a structure of the systems that process a stream of data. The task of a system is divided into several processing steps. These steps are connected through the 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 best suited when a system can naturally decompose its data processing task into independent steps and the requirements for the data processing task are likely to change over time. This pattern increases the reusability of filters, and allows a system to be flexible through exchanges and recombinations of filters [5].

In Matilda, each plugin works as a filter, and implements an individual step to process UML models and JASTs (Figure 3). For example, a model loader plugin accepts a UML model in the format of XML Metadata Interchange (XMI) [7] and transforms it to an in-memory representation. A model validation plugin validates a UML model against the UML metamodel. A JAST generation plugin transforms a UML model to a JAST.

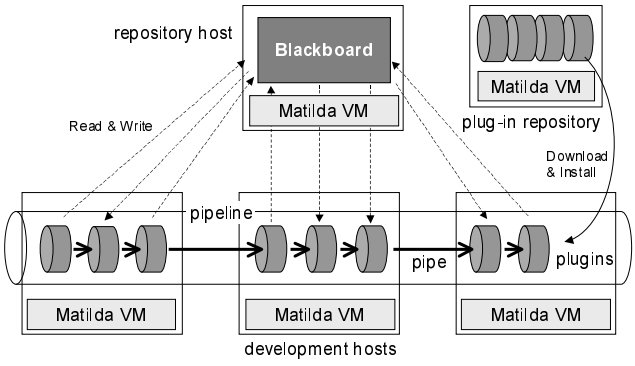


Figure 3. The Architecture of Matilda

A collection of plugins is called a pipeline (Figure 3). Each pipeline instantiates plugins and connects them with pipes based on a configuration file that a VM maintenance engineer defines. (a pipe works as a pointer to a subsequence plugin.) The configuration file specifies plugins used in a pipeline and the execution order of plugins. Plugins are executed in a sequential or a parallel manner.

As described in Section 2, plugins can be transparently distributed on multiple hosts in the network. Each plugin operates on a Matilda VM, which in turn runs atop of a Java VM. Plugins can be dynamically downloaded from a plugin repository and deployed in a pipeline (Figure 3). Pipelines can be configured dynamically at runtime as well as statically before executing plugins.

A common issue of the Pipes and Filters pattern is lack of robust error handling, because there is no global system state information and multiple asynchronous threads of execution exist in a system [5]. In order to overcome this issue, Matilda implements a shared repository, called *blackboard*, based on the Blackboard architectural pattern (Figure 3) [5]. This pattern is organized as a collection of independent processing units that work cooperatively on a common data structure. Each processing unit specializes to process a particular part of the overall task. It fetches data from a blackboard (shared data repository), 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, JASTs and Java bytecode), and makes the data available to subsequent plugins (Figure 3). It also stores the data 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 plugins (Figure 3).

4 Matilda UML Profile

This section describes a UML profile, called the Matilda UML profile, which Matilda uses to interpret UML models and transforms them to 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) [2]. 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 [8]. 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 profile specifies the conventions to build input UML models for Matilda, and defines stereotypes and tagged-values to precisely describe computationally-complete input models[1].

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,

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

and sequence diagrams are used to define the behavior of the application. 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 profile defines two types of stereotypes: (1) stereotypes for application semantics, and (2) stereotypes for Java mapping. Stereotypes for application semantics include three stereotypes shown in Figure 4. 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, and `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 an exactly one class stereotyped with ≪`UMLVMexecutable`≫.

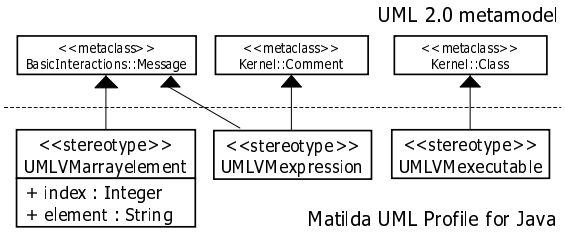


Figure 4. Stereotypes in the Matilda UML Profile

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.

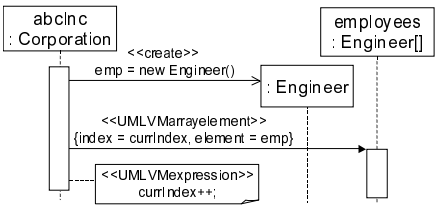


Figure 5. An Example Model using Matilda UML Profile

Figure 5 shows an example sequence diagram defined with the Matilda UML profile. It repeats a sequence that (1) `abcInc` (an instance of `Corporation`) creates a new instance of `Engineer`, and (2) `abcInc` inserts the new `Engineer` instance into the array `engineers` (an array of `Engineer`s). 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 profile also defines a stereotype and five tagged-values to specify the mapping between UML models to Java (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 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.

5 Implementation

Matilda is implemented in Java. Currently, it provides 10 plugins shown in Figure 6. Its current code base contains approximately 18,000 lines of Java code, and has been open for public use since May 2005[3].

Figure 6 shows a typical pipeline configuration in Matilda. Plugins are categorized into two groups: *frontend* and *backend*. Frontend plugins are used to validate UML models, and backend plugins are used to transform

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

[3]http://umlvm.cs.umb.edu/

validated UML models to Java bytecode through JASTs. Plugins can read/write UML model information from/to a blackboard. A pipeline executes plugins in a sequential or parallel manner, and controls the execution of plugins. For example, when a blackboard receives an execution error log form a plugin, a pipeline stops executing plugins (Figure 6).

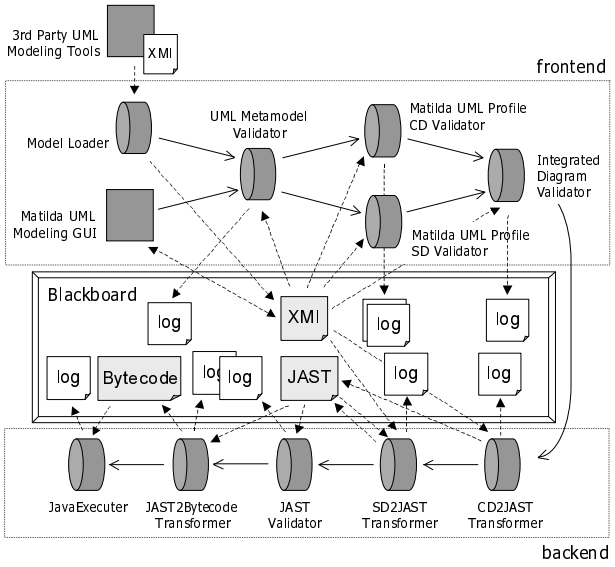
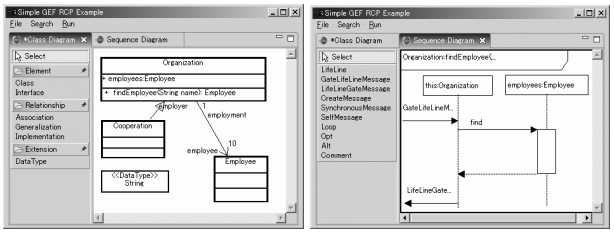


Figure 6. Typical Sequence of plugins

5.1 Plugin Implementation

Matilda accepts a UML model as an input in two ways: using Matilda's modeling GUI or third-party modeling tools. Matilda provides a UML modeling GUI, which allows developers to define UML class diagrams and sequence diagrams (Figure 7(a) and 7(b)). The modeling GUI serializes a UML model into XMI data and writes it to a blackboard (Figure 6). It is implemented with Eclipse Rich Client Platform (RCP), and runs on the Eclipse platform. *ModelLoader* is a plugin used to read XMI data from third-party modeling tools and store it in a blackboard.



(a) Class Diagram (b) Sequence Diagram

Figure 7. Matilda Modeling GUI

Each UML model is validated with four validators: *UML Metamodel Validator*, *Matilda UML Profile Class Diagram (CD) Validator*, *Matilda UML Profile Sequence Diagram (SD) Validator* and *Integrated Diagram Validator*. *UML Metamodel Validator* validates an input model against the UML metamodel using the `UML2Validator` class provided by Eclipse UML2[4]. *Matilda UML Profile CD Validator* and *Matilda UML Profile SD Validator* check if an input model is valid against the Matilda UML profile. The purpose of this validation step is to determine whether the model is ready to transform to a JAST. This step is implemented by extending Eclipse UML2.

The code fragment shows a simplified model validation in *Matilda UML Profile CD Validator*. First, the plugin reads an UML model from a blackboard and executes validation process. It checks whether the model is compliant with the Matilda UML profile. For example, a model element stereotyped with ≪`UMLVMexecutable`≫ should be a class that has a main method (see also Section 4).

```
class MatildaProfileCDValidator implements Plugin {
  void execute() {
    // obtain a reference/proxy of a blackboard
    blackboard = pipeline.getBlackboard();
    // read a UML model from a blackboard
    model = blackboard.read( ModelID );
    // validate the obtained model
    validate( model );
  }
  void validate( Model 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 InvalidModelException();
} } }
```

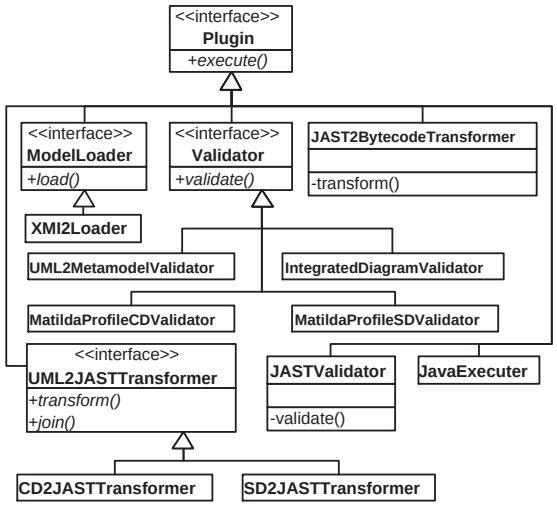


Figure 8. Design of Plugins

Integrated Diagram Validator checks the consistency between a class diagram and sequence diagrams. Its major responsibility is to validate that sequence diagrams are defined corresponding to all methods in each class.

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

After validating with its frontend plugins, Matilda transforms a valid UML model to a JAST with two backend plugins: *CD2JAST Transformer* and *SD2JAST Transformer*. They transform a class diagram and sequence diagrams, respectively, to a JAST based on the data structure in the Eclipse Java Development Tooling (JDT). *CD2JAST Transformer* creates a new JAST based on the type (class and interface), member field and method declarations appeared in a UML model, and then it generates a JAST compilation unit for each type declaration. *SD2JAST Transformer* reads a JAST from a blackboard and updates it with method definitions mapped from each sequence diagram.

JAST Validator validates the generated JAST, and *JAST2Bytecode Transformer* generates Java bytecode (i.e., class files) using Eclipse JDT. Finally, the last plugin in the pipeline, *JavaExecuter*, reads the generated Java bytecode, determines which class is executable, sets up the execution environment (JVM), and executes class files.

Figure 8 shows how each plugin is designed in Matilda. All plugins implements the `Plugin` interface. Matilda provides additional interfaces (`ModelLoader`, `Validator` and `JAST2BytecodeTransformer`) for common functions.

5.2 Pipeline Implementations

Matilda pipeline can contain arbitrary number of plugins, and executes them in a sequential or a paralell manner. The following is a fragment of a plugin configuration file. A pipeline is configured to load and execute plugins as specified in the configuration file. As described in Section 3, a pipeline can dynamically download plugins' bytecode from a plugin repository, and deploy and configure them at at runtime. Each plugin's name and class file are specified by the tags `name` and `class` respectively. If a plugin takes parameters other than data from a blackboard (e.g., name of a model), they are specified by the tag `input`.

```
<plugin>
 <name>Model Loader</name>
 <class>matilda.plugins.frontend.XMI2Loader</class>
 <input>
  <param-name>Model Filename</param-name>
  <param-value>models/model.uml2</param-value>
 </input>
</plugin>
<plugin>
 <name>Class to JAST Transformer</name>
 <class>matilda.plugins.backend.CD2JASTTransforder</class>
</plugin>
<plugin>
 <name>Sequence to JAST Transformer</name>
 <class>matilda.plugins.backend.SD2JASTTransformer</class>
</plugin>
<plugin>
 <name>Java Executer</name>
 <class>matilda.plugins.backend.JavaExecuter</class>
 <input>
   <param-name>arguments</param-name>
   <param-value>34 + 2 - 8 x 10 / 4</param-value>
 </input>
</plugin>
```

In this example, a pipeline loads a UML model that specified with parameters (`param-name` and `param-value`) using a model loader plugin (`XMI2Loader`). After that, the pipeline transforms the loaded UML model (class and sequence diagrams) into a JAST using transformer plugins (`CD2JASTTransformer` and `SD2JASTTransformer`), and executes the JAST using executer plugin (`JavaExecuter`).

As described in Section 3, one of key features of a pipeline is that it can be transparently distributed. To achieve this feature, Matilda employs Java RMI that allows Java objects to transparently communicate with each other in a distributed envitonment. In Matilda, a pipeline and a blackboard are implemented as Java RMI objects, and they can communicate in a distributed manner. Plugins are implemented as regular Java objects, but they are contained in a pipeline and a pipeline allows plugins to communicate with a blackboard. This design hides details of a remoting infrastructure (i.e., Java RMI) from plugins, and plugin developers do not need to know the detail. It makes easy to develop and deploy plugins.

Figure 9 shows a class structure around `Pipeline`. The class `VirtualMachine` contains the class `Pipeline`. `BlackboardImpl` is the implementation of a blackboard, and runs in a different process from `Pipeline`. `BlackboardProxy` hides the detail of remoting infrastructure and allows `Pipeline`s to communicate with `BlackboardImpl` in a distributed manner. `Pipeline` maintains a set of `Plugin`s.

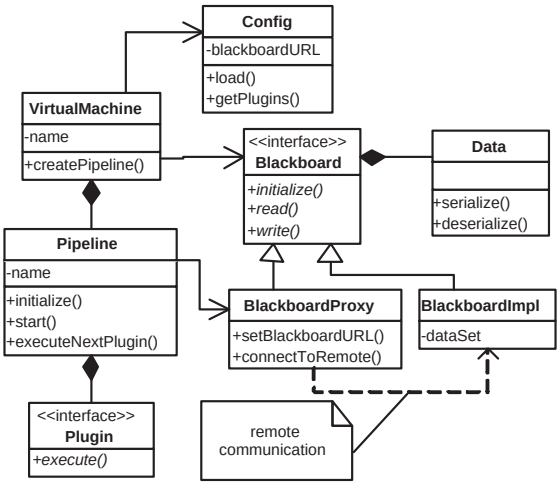


Figure 9. A Class Structure of Pipeline

By leveraging the feature that a pipeline can be transparently distributed, Matilda enables distributed software development as depicted in Figure 10. In this figure, there are two types of developers, one defines a class diagram and the other defines a sequence diagram. Matilda provides a set of plugin sequences for both types of developers. After they define UML models, plugins validate the models and a blackboard stores them. Then, the *Integrated Diagram Validator* checks the consistency between the class diagram and the sequence diagram, and the models are converted into a JAST and executed as described in the backend process of Figure 6.

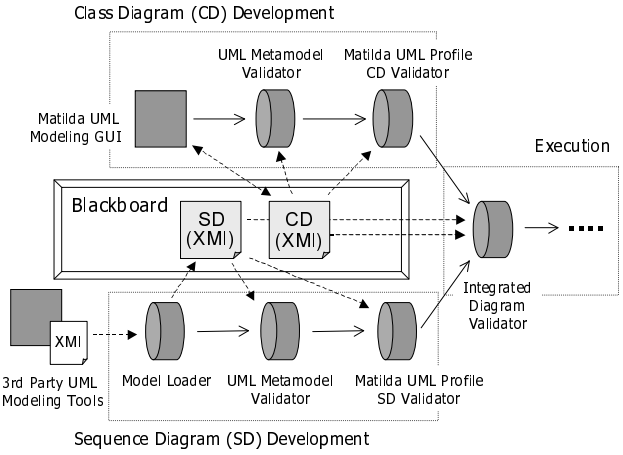


Figure 10. Distributed Software Development with Matilda

6 An Example Application

This section describes a simple example application built with Matilda, and demonstrates how to describe UML models for Matilda. It is a command-line calculator that accepts an arithmetic expression in Reverse Polish Notation and returns a calculation result. It supports summation, difference, division, multiplication and factorial operations.

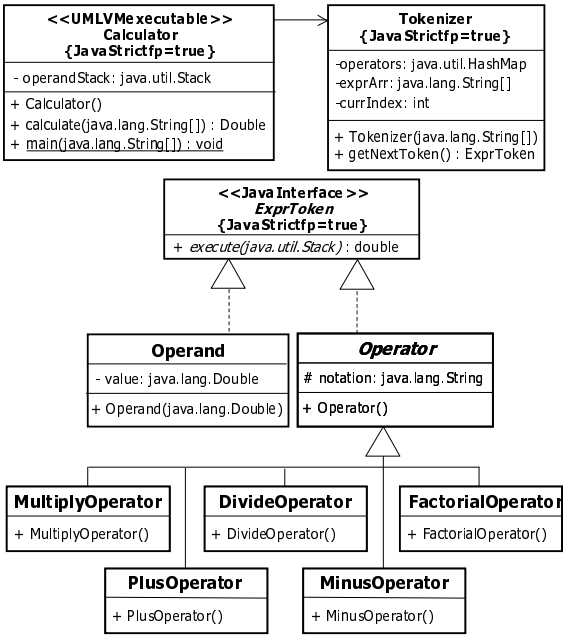


Figure 11. Class Diagram of an Example Application

Figure 11 shows the class diagram for the calculator example application. `Calculator` is the execution entry class, which is stereotyped with ≪`UMLVMexecutable`≫; it has a main method to which an input arithmetic expression is passed. The input expression can be passed with the part of *Java Executor* configuration in a pipeline configuration (see a pipeline configuration example in Section 5.2). 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.

Currently, Matilda requires developers to define a sequence diagram for each operation/method. Figure 12 shows the sequence diagram for `getNextToken()` of `Tokenizer` (see also Figure 11). 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)[5]. `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()`.)

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 12 can reference `exprArr`, `operators` and `currIndex`, which are the data fields of `Tokenizer`.

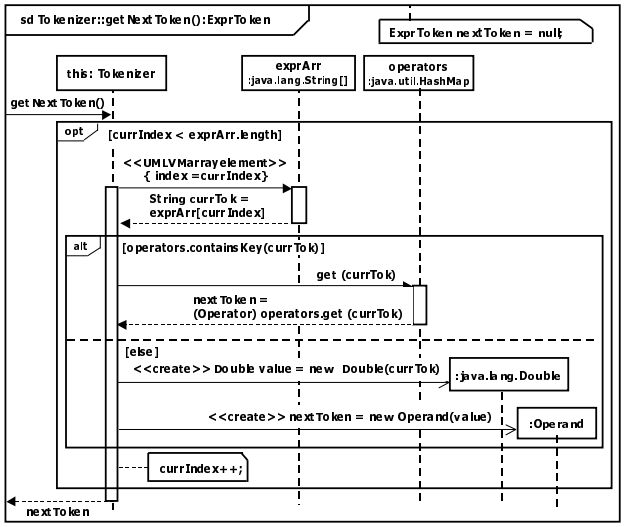


Figure 12. A Sequence Diagram of an Example Application

Matilda uses the `opt`, `alt` and `loop` fragments to specify control flows. Figure 12 uses the `opt` and `alt`

[5] `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`.

frames to define *if* and *if-then* control flows, respectively. Guard conditions for the frames are represented by 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 12, 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 12, the return value of calling `get()` on `operations` is casted to `Operator`[6], 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 12, 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 12), within a reply message (e.g., `Token nextToken`, or within a ≪create≫ message (e.g., `Double value`). The scope of each local variable is limited to the innermost fragment or `sd` frame.

7 Empirical Evaluation

This section empirically evaluates the execution overhead and memory footprint of Matilda. Matilda's pipeline is sequentially configured with eight plugins in order to (1) load an input model with *Model Loader (MV)*, (2) validate the input model with *UML Metamodel Validator (UMV)*, *Matilda UML Profile CD Validator (CDV)* and *Matilda UML Profile SD Validator (SDV)*, (3) transform the validated model to a JAST with *CD2JAST Transformer (CDJ)* and *SD2JAST Transformer (SDJ)*, (4) transform the generated JAST to Java bytecode with *JAST2Bytecode Transformer (JBC)*, and (5) execute the generated Java bytecode with *Java Executer (JE)*. All measurements use 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 13 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)[7] and read/write the input model from/to a blackboard. The proportion of each plugin's overhead to total overhead does not change significantly against varying the number of classes in an input model. The overhead of MV is extremely larger than that of other plugins. It occupies over 60% of total overhead. This result comes from the performance of `UML2Validator` in Eclipse UML2, which Matilda 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 7) so that it validates an input model in background while developers are drawing the model.

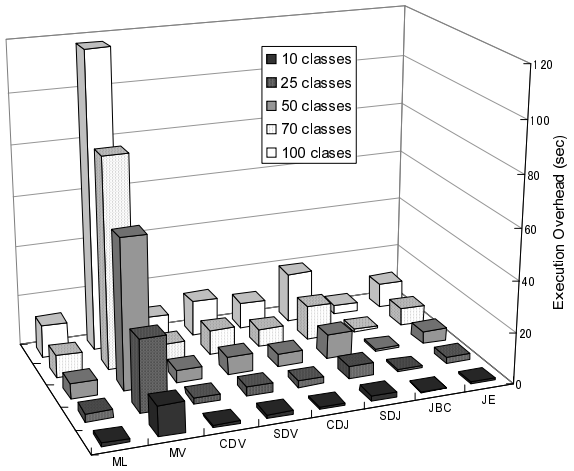


Figure 13. Execution Overhead of Plugins

Table 3. Execution Overhead of the Frontend and Backend

# of classes	Matilda (sec)			javac (sec)
	Frontend	Backend	Total	
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 execution 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 Matilda's performance is comparable with `javac` when the number of classes is less than 25 in an input model. (Matilda's overhead is less than 10 times of the `javac` overhead.)

Figure 14 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 blackboard 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[8], the time to

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

[7] 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.

[8] When a plugin reads XMI data from a blackboard, it compresses the data with the zip encoding to reduce the data transmission overhead be-

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.

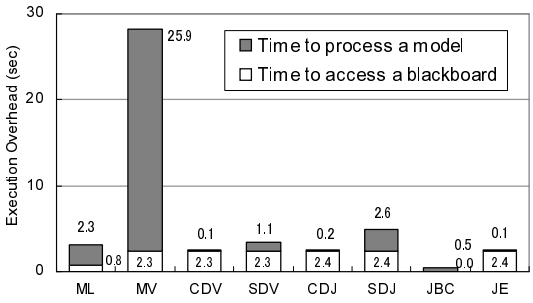


Figure 14. Breakdown of Plugin Execution Overhead

In order to eliminate the blackboard access overhead, Matilda 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, Matilda (backend overhead) is comparable with `javac` when the number of input classes is less than 70. Tables 3 and 4 show that Matilda works efficiently in small to medium scale applications.

Table 4. Execution Overhead of the Frontend and Backend in the Case that Plugins are Deployed in the Same Process

# of classes	Matilda (sec)			javac (sec)
	Frontend	Backend	Total	
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 15 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 disable. 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), Matilda's memory consumption is acceptable in small to medium scale applications. MV consumes memory space most because it loads an input model and the definition of UML metamodel and Matilda UML profile, and validates the model against the UML metamodel definitions.

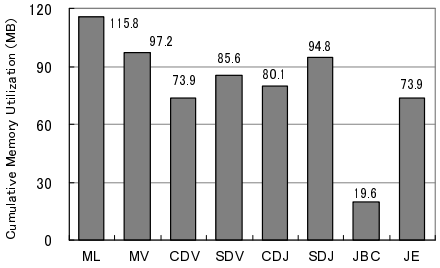


Figure 15. Memory Consumption of Plugins

8 Related Work

There are several work to investigate UML virtual machines. [9] 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. [9] 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.

ASM virtual machine [10] and USE (UML-based Specification Environment) [11] address the issue of validating models. They support Object Constraint Language (OCL) [2] 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 [12, 13, 14]. Action languages implement the UML action semantics, defined as a part of the UML specification [2]. 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., [15, 16]). This means that developers need to learn action language syntax every time they use different xUML tools. Also, there is no interop-

tween 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.

erability 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[9] 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.

As discussed in Section 1, the current common practice in MDD is to model application designs with UML and transform them to skeleton source code (e.g., OptimalJ[10] , Rose XDE[11], Together[12], UMLX [17], KMF [18], J3 [19], [20] and [21]). Unlike them, Matilda focuses on direct execution of UML models so that no manual programming is necessary (see Figures 1 and 2).

9 Conclusion

This paper describes and empirically evaluates a new MDD framework, called Matilda, which accepts UML models and directly executes them through transforming them to executable code. 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 UML models at physically dispersed places. Empirical measurement results show that Matilda works efficiently with a small memory consumption in small to medium scale applications.

10 Acknowledgment

The work by Hiroshi Wada and Junichi Suzuki are supported in part by OGIS International, Inc. The authors would like to thank Chengjing Hu, Anu Lall, Murtaza Qureshi, Gina Skaff and Kathiresan Solaiappan for their contributions to implement Matilda.

References

[1] B. Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, September/October 2003.

[2] Object Management Group. UML2.0 Super Structure Specification, October 2004.

[3] G. Booch, A Brown, S Iyengar, J. Rumbaugh, and B. Selic. An MDA Manifesto. In *The MDA journal: Model Driven Architecture Straight from the Masters*, chapter 11. Meghan-Kiffer Press, December 2004.

[4] S. Sendall and W. Kozaczynki. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, September/October 2003.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, August 1996.

[6] A. Vermeulen, G. Beged-Dov, and P. Thompson. The Pipeline Design Pattern. *OOPSLA Workshop on Design Patterns for Concurrent Parallel and Distributed Object-Oriented Systems*, October 1995.

[7] Object Management Group. MOF 2.0 XML Metadata Interchange, 2004.

[8] L. Fuentes and A. Vallecillo. An Introduction to UML Profiles. *The European journal for the Informatics Professional*, April 2004.

[9] D. Riehle, S. Fraleigh, D. Bucka-Lassen, and N. Omorogbe. The Architecture of a UML Virtual Machine. *ACM Int'l Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2001.

[10] W. Shen, K. Compton, and J. Huggins. A Method of Implementing UML Virtual Machines With Some Constraints Based on Abstract State Machines. *IEEE Asia-Pacific Software Engineering Conference*, 2003.

[11] M. Gogolla, J. Bohling, and M. Richters. Validation of UML and OCL Models by Automatic Snapshot Generation. *Int'l Conference on Unified Modeling Language*, 2003.

[12] S. Mellor and M. Balcer. *Executable UML: A Foundation for Model Driven Architecture*. Addison-Wesley, 2002.

[13] C. Raistrick, P. Francis, and J. Wright. *Model Driven Architecture with Executable UML*. Cambridge University Press, March 2004.

[14] M. Balcer. An Executable UML Virtual Machine. *the 4th OMG Workshop On UML for Enterprise Applications: Delivering the Promise of MDA*, June 2003.

[15] Project Technology. BridgePoint Tutorial, 2000.

[16] Kennedy Carter Ltd. The UML Action Specification Language Reference Guide, November 2004.

[17] E. Willink. UMLX: A Graphical Transformation Language for MDA. *ACM Int'l Conf. on Object-Oriented Programming, Systems, Languages and Applications*, Nov. 2002.

[18] O. Patrascoiu. Mapping EDOC to Web Services using YATL. *Int'l Conference on Enterprise Distributed Object Computing*, September 2004.

[19] J. White, D. Schmidt, and A. Gokhale. Simplifying Autonomic Enterprise Java Bean Applications. *ACM/IEEE Int'l Conference on Model Driven Engineering Languages and Systems*, October 2005.

[20] W. Harrison, C. Barton, and M. Raghavachari. Mapping UML designs to Java. *the 15th ACM Int'l Conference on Object-Oriented Programming, Systems, Languages and Applications*, October 2000.

[21] Z. Ahmed and C. Umrysh. *Developing Enterprise Java Applications with J2EE and UML*. Addison-Wesley, 2002.

[9] www.openarchitectureware.org/

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

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

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