Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

Validation of ModelicaML Models

by

Goutham Gatla

LIU-IDA/LITH-EX-A—12/061

2012-11-22
Final Thesis

Validation of ModelicaML Models

by

Goutham Gatla

LIU-IDA/LITH-EX-A—12/061

2012-11-22

Supervisors:

Olena Rogovchenko
Wladimir Schamai

Examiner:

Peter Fritzson
Abstract

In the world of modeling, Model Validation plays a crucial role. A model editor is not said to be complete without Validation. ModelicaML is a Modeling Language extended from a subset of UML and SysML, developed under OpenModelica Project. It is defined to provide time-discrete and time-continuous models. Papyrus Model Editor is extended to support for ModelicaML using ModelicaML Eclipse plug-in. This plug-in comes with Modelica Code Generator.

Previously, ModelicaML plug-in had a prototype of validation which provided only Batch-mode validation. The validation is used to be done by the Modelica compiler after the code generation phase. Each time the user tried to validate the model; first Modelica code is generated and then validated. This type of validation misses certain validation rules to validate due to the conversion from the ModelicaML model to Modelica code.

The goal of this thesis is to implement Model Validation done at model editor level with both Batch and Live mode validation. This can be done by developing an Eclipse plug-in which does the Model Validation. This plug-in uses the EMF Validation framework for implementing the constraints and validation on ModelicaML models.
Acknowledgment

I would like to thank my examiner Peter Fritzson and my supervisors. I also would like to thank my parents and friends for their continuous support.
## Contents

Abstract ......................................................................................................................... v
Acknowledgment ........................................................................................................ v

1 Introduction ............................................................................................................... 1
  1.1 Problem Statement .......................................................................................... 1
  1.2 Goals ................................................................................................................. 1
  1.3 Thesis Outline ................................................................................................... 1

2 Background .............................................................................................................. 3
  2.1 Model Validations ............................................................................................ 3
  2.2 Modelica and OpenModelica ........................................................................... 3
  2.3 Modelica Modeling Language (ModelicaML) ................................................... 3
  2.4 Papyrus ............................................................................................................. 4
  2.5 Eclipse .............................................................................................................. 6
  2.6 Eclipse Plug-in Development ........................................................................... 6

3 Frameworks ............................................................................................................ 10
  3.1 Eclipse Modeling Framework (EMF) ................................................................. 10
    3.1.1 XML MetaData Interchange (XMI) Serialization ....................................... 11
    3.1.2 Java Annotations ....................................................................................... 12
    3.1.3 Notification and Adapters ......................................................................... 13
    3.1.4 Object Persistence ...................................................................................... 14
  3.2 EMF Validation Framework .............................................................................. 15
    3.2.1 The Validation Service ............................................................................ 17
    3.2.2 Constraint Contribution ........................................................................... 20
    3.2.3 Validation Results Reporting ................................................................... 23

4 Implementation ...................................................................................................... 24
  4.1 Plug-in Design and Contributions ................................................................... 24
  4.2 Validation Constraints ...................................................................................... 26
  4.3 Validation Handlers ......................................................................................... 27
  4.4 Client Context .................................................................................................. 28
  4.5 Reporting Validation Results ......................................................................... 28

5 Testing ..................................................................................................................... 31

6 Conclusion and Future work ................................................................................ 32

Bibliography ............................................................................................................... 33

Appendix A ................................................................................................................. 34
  List of Constraints .................................................................................................. 34

Appendix B ................................................................................................................. 40
  User Guide ............................................................................................................. 40
    Installation ......................................................................................................... 40
    Plug-in Customizations ...................................................................................... 40
    Validating ModelicaML models ....................................................................... 41
Illustration Index

Illustration 1: Papyrus Architecture Overview, taken from [11].................................4
Illustration 2: Work flow of ModelicaML model development, taken from [14]...........5
Illustration 3: Eclipse Plug-in Architecture, taken from [13]......................................7
Illustration 4: Workflow of plug-in development, taken from [12].............................7
Illustration 5: EMF unifies Java, UML and XML, taken from [9]..............................10
Illustration 6: Simplified Ecore model, taken from [9]...........................................11
Illustration 7: ResourceSet, taken from [9]..............................................................15
Illustration 8: Validation Service, taken from [10]..................................................17
Illustration 9: Notification Filter, taken from [10]....................................................19
Illustration 10: Constraint Descriptor, taken from [10]............................................20
Illustration 11: Java Constraint descriptor, taken from [10].....................................21
Illustration 12: Validation Status, taken from [10]..................................................22
Illustration 13: Constraint Properties......................................................................27
Illustration 14: ModelicaML Validation Markers view............................................29
Chapter 1

1 Introduction

1.1 Problem Statement

This project is about building an eclipse plug-in which does validation on ModelicaML models. The elements which violates the validation rules are either reported using dialog prompts or marked in a view. Previously, the validation in ModelicaML is just a prototype and used to work at code generation phase of the ModelicaML plug-in. It is supported in the Batch-mode only and ignores to validate certain rules/constraints on the ModelicaML models. e.g., empty names and name clashes on the model element are not checked.

1.2 Goals

A validation Eclipse plug-in is needed to be build with set of constraints/rules, which does both the batch and live mode validation directly on the ModelicaML models, and the results of validation are reported by generating marker and markers with severity warning or errors are populated them on custom validation problems view.

1.3 Thesis Outline

This section provides an outline of the thesis organization and a quick review of this report.

- Chapter 2
  This chapter explains about the background of the thesis. It covers brief introduction about model validation, Modelica, OpenModelica, ModelicaML, Papyrus, Eclipse and Eclipse Plug-in development.

- Chapter 3
  This chapter explains about the frameworks used in the implementation. It covers Eclipse Modeling Framework and EMF Validation Framework.
• **Chapter 4**
  This chapter explains the thesis implementation. Explains about the plug-in design and its contributions to Eclipse IDE

• **Chapter 5**
  This chapter discuss about the thesis conclusion and future work.
Chapter 2

2 Background

2.1 Model Validations

Model validation is an important segment of the Model development. It provides the well-uniformness and creditability of the model by ensuring that the information provided by the modeled system is accurate and satisfies all the goals of its intended usage.

2.2 Modelica and OpenModelica

Modelica is an object oriented; model based programming language which is used to model the real time objects efficiently with mathematical equations. Modelica is developed by Modelica Association and it is an open source. Modelica is used for multi domain modeling and comes with Modelica Standard Library, which contains models and functions that can support for multi domain modeling [1]. OpenModelica contains set of open source tools including Modelica Compiler which can be used to simulate Modelica models and it is developed by OpenModelica Consortium [2].

2.3 Modelica Modeling Language (ModelicaML)

ModelicaML is an UML profile which provides the efficient and effective usage of the Modelica. It defines time continuous and time discrete based models. ModelicaML is part of OpenModelica Consortium and developed in collaboration with Linköping University and EADS Innovation Works. ModelicaML can be accessible through Papyrus Eclipse plug-in.

ModelicaML is based on the constructs of UML and SysML. The plug-in contains the Modelica Code Generator which generates Modelica code from ModelicaML Models. Papyrus Model Editor is extended to support ModelicaML by an Eclipse plug-in. This plug-in customizes the tools provided by the Papyrus like Model Explorer, Palette etc for ModelicaML [3].
2.4 Papyrus

Model Developing Tools (MDT) is the eclipse based project, provides development of tools which are used to implement Industry standard meta-models and models from these meta-models.

Papyrus is sub-project of MDT project which provides model editor integrated into Eclipse IDE for editing any kind of model which is based upon EMF and supports UML, SysML and MARTE. It binds the EMF based modeling languages to the both GMF based and not GMF based editors and also with other MBD and MDSD tools. It also supports for DSL by providing support for UML profiles [4][5], [6].

Papyrus also supports

- UML and SysML profiling.
- Model Search.
- Model Validation.
- Customizable palates, properties views and model explorer.
- Code Generation.

Illustration 1: Papyrus Architecture Overview, taken from [11]
The implementation of ModelicaML\(^1\) is done by developing an Eclipse plug-in which extends the Papyrus. This plug-in will integrate ModelicaML perspective which is a customized Papyrus perspective along with code generator into the Papyrus. The figure below illustrates the block diagram of ModelicaML with Papyrus.

Illustration 2: Work flow of ModelicaML model development, taken from\([14]\)

The above figure illustrates that the graphical notation of the ModelicaML models developed using the ModelicaML plug-in that can be used to generate the efficient Modelica code using the code generator within the plug-in, Supported by the Acceleo. The generated Modelica code is simulated within the Eclipse IDE using another Modelica Development Toolkit (MDT) eclipse plug-in.

### 2.5 Eclipse

Eclipse is an open source platform indulge in development of efficient Integrated Developing

\(^1\) www.openmodelica.org/modelicaml
Environments (IDEs), Frameworks and Application Programming Interfaces (APIs) which supports the entire software developing life-cycle (SDLC) for better and easy software development. Eclipse is based on Plug-in Architecture which open doors to the multiple developers to develop the tool and technologies particular to their area of experience and easily integrated into the core development environment [7].

Eclipse is not just an IDE for Java. There are more than 200 project developments are going under the roof of Eclipse community which are categorized as follows [8]:

→ Enterprise Development
→ Embedded and Device Development
→ Rich Client Platform
→ Rich Internet Applications
→ Application Frameworks
→ Application Life-cycle Management (ALM)
→ Service Oriented Architecture (SOA)

For more information about Eclipse projects, tutorials and white papers are available from the Eclipse Official website ².

### 2.6 Eclipse Plug-in Development

Now we know that Eclipse is developed using plug-in technology so, let us discuss more about the plug-in. In general, Plug-in is a smallest software component which ranges support from integration of Programming Language or a complete tool to contribution of a small menu. Specifically, Eclipse plug-in is a “self-contained” and “self-describing” JAR (Java ARchive) file. It contains two descriptor files MANIFEST.mf and plugin.xml along with source code and other resources. The descriptors provides the meta-information about itself (i.e., the plug-in) and the source code and resources are self-contained and used while plug-in execution.

---

² www.eclipse.org
In order to start developing an Eclipse plug-in, we need an Eclipse IDE with Plug-in Development Environment (PDE) and Java Developing Tooling (JDT). Usually, standard Eclipse SDK comes with these tools installed. The PDE provides the plug-in editor which is an editor for plugin.xml and MANIFEST.mf files which simplifies the configuring plug-in. The figure below describes the general work-flow of the plug-in development.

Illustration 3: Eclipse Plug-in Architecture, taken from [13]

Illustration 4: Workflow of plug-in development, taken from [12]
Plugin.xml descriptor file is the most important and used to configure the plug-in. This file contains the meta-data of the plug-in. The data provided by the plugin.xml is subdivided into different sections. Each section is discussed as follows:

- **Dependencies:**

  This section lists all the plug-ins whose functionalities are interested by the current plug-in which are added into the classpath. This also allows adding the individual exported packages of other plug-ins into the classpath instead of whole plug-in. The dependencies of the plug-in can be specified under the dependencies tab of the plug-in editor.

- **Runtime:**

  This category describes what packages of the current plug-in can be exposed to the environment. Each plug-in has its own classpath and the libraries which are needed to be included into the classpath are also defined in this section. The exported packages which we discussed under dependencies are the packages which are declared under the runtime. We can also restrict the exported packages to be internal and it is not considered as API.

- **Extensions:**

  Extensions of a plug-in are like Electrical plugs from which services provided by other plug-in are utilized. The extensions to the services are only done to the packages or plug-ins which are listed under the dependencies.

- **Extension Points:**

  Extension points are like sockets which broadcasts the services provided by this plug-in to others so that other can bind to this plug-in using the Extension. For example, Eclipse SDK provides the extension point org.eclipse.ui.editors by which other plug-ins can contribute their own editors. Similarly, Eclipse provides many other extension points for customization and extensible Eclipse IDE.

The plug-in under the development can be tested or debugged any time with the help of self-hosting feature provided by the PDE. With this feature, a runtime instance of the eclipse with the current development plug-in can be launched by which exporting or deploying the plug-in for testing can be avoided. Debugging the plug-in is as similar as debugging the standard Java Application source code.
The more in-detail tutorial of plug-in development is available from IBM's DeveloperWorks' website.

Chapter 3

3 Frameworks

3.1 Eclipse Modeling Framework (EMF)

EMF is a modeling framework and code generation facility which bridges between the modeling and programming. A data model can be defined and implementation source code can be generated from the model. A model can be defined either by using Java interfaces or UML or XML Schema. A model defined in one form can generate other forms along with implementation code by code generator, i.e., if a model is defined using Java interfaces then using EMF we can generate same model in UML, XML Schema. Eclipse with Java Development Toolkit is only required to start development with EMF.

![Illustration 5: EMF unifies Java, UML and XML, taken from [9]]( Illustration 5: EMF unifies Java, UML and XML, taken from [9])

In order to define a model, we need another model by which is used to built, that is a meta-model. In EMF, Ecore is the meta-model which builds the platform independent models. Ecore is based on Essential Meta Object Facility (EMOF) which is one variant of OMG's Meta Object Facility (MOF) which is also used to define the Unified Modeling Language (UML) and difference between Ecore and UML is, Ecore only holds the essential information to define the model whereas UML also holds the behavior of the model. Ecore consists of hierarchy of set of elements or classes by which models are represented. A simplified version of Ecore(meta-model) contains EClass, EAttribute, EReference and EDataType to represent a model.
As we discussed earlier, an Ecore model can be defined using UML or XML Schema or Java Interfaces. In order to define a model using UML, Omondo's EclipseUML tool which is build upon EMF can be used to edit Ecore model directly and can store it directly as Ecore model or Rational Rose can be used to build the Ecore models and store it into the Rational Rose model file (*.mdl) later import feature provided by the EMF can be used to extract the Ecore model. If a model is defined using XML Schema, an Ecore model is constructed based on schema where as if a model is defined using Java Interfaces, Introspection is done on the interfaces and an Ecore model is constructed.

When it comes to Java interfaces to define the model, @model annotation is used to represent the model and its model elements for which implementation code should be generated. Java Introspection is used to retrieve the default information like type of a model attributes and non default information can be given along with annotation. Its not required to include attribute setter methods in the java interfaces, instead these setter methods are generated based on the available getter methods.

### 3.1.1 XML MetaData Interchange (XMI) Serialization

Along with UML, XML schema and Java Interfaces, we can also define the model using XML MetaData Interchange (XMI) which can be default form to persist the model. XMI is used as a standard meta-data serialization. An example serialized XMI taken from book [9] is given below.

```xml
<?xml version="1.0" encoding="ASCII"?>
```
3.1.2 **Java Annotations**

We already discussed that a model can also be defined using the Java Interfaces. The given below Java Interface example taken from [EMF textbook] describes the PurchaseOrder model. The annotation `@model` describes the EMF generator that what things are needed to be included into the generated implementation code. The annotation for the method `getItems` will describe about the reference to another model Item. `containment='true'` will state that the PurchaseOrder will maintain a container for the model Item. From the example below, we can identify that there are no setter methods defined along with getter methods. EMF uses Java introspection to extract the information about the model based on the annotation on getter methods, setter methods are not really required and the code generator will generates them into the interface along with the implementation code [9].
public interface PurchaseOrder {
    String getShipTO();
    String getBillTo();
    List getItems();
}

3.1.3 Notification and Adapters

One of the most important features provided by the EMF Framework is Notifications. Notifications will notify the modification done to the model with the help of notification listeners. Adapter are the Notification Listeners that are attached to the any EObject whose change notification are sent to interest objects. They contains object status and also can able to extend the functionalities of the attached objects. The model interfaces extends an EObject interface which is almost similar to Object class in Java. This EObject interface extends Notifier interface which generates the Notifications. The notifyChange() method of Adapter class is called whenever there is a change in object state. The best way to add Adapters to the EObject is by using AdapterFactory. Adapter Factories will try to adapt if an Adapter of the extension type is already attached to the EObject, else it will create a new Adapter and attach to it. EMF framework provides another adapter class, EcontentAdapter. If an EContentAdapter object is attached to a root EObject, then it will notify the changes of any object or resource within the containment hierarchy, instead of adding adapters to the objects or resources of whole hierarchy [9].
3.1.4 **Object Persistence**

Persisting the Model objects and referencing to the persisted objects are the powerful feature provided by the EMF Framework which can be easily used to embed the model data within the application. Earlier, we already discussed about the XMI serialization for persisting the object.

In EMF framework, a model can be persist in any form but explicitly need to be implement the serialization whereas, it is very easy if a model is defined using XML Schema. Implementing the serialization code will automatically make the object of a model to refer to the objects of another model. The two reflect API methods, `eContainer` and `eResource` can be used for persistence. The method `eContainer` will return the container reference of that object. Let us again follow the example with code snippet taken from the [9].

```java
PurchaseOrder aPurchaseOrder =
    POFactory.eINSTANCE.createPurchaseOrder();
aPurchaseOrder.setBillTo("123 Maple Street");

Item anItem = POFactory.eINSTANCE.createItem();
anItem.setProductName("Apples");
anItem.setQuantity(12);
anItem.setPrice(0, 50);

aPurchaseOrder.getItems().add(anItem);
anItem.eContainer(); // will return the object, aPurchaseOrder
aPurchaseOrder.eContainer(); // will return null
```

From the above example, the PurchaseOrder class holds a container of class Item. So, `anItem.eContainer()` will return the container reference, `aPurchaseOrder` and `aPurchaseOrder.eContainer()` will return null because there are no container reference of PurchaseOrder object. Similarly, the method `eResource` will return the reference of Resource objects. Since, there are no resource available at both the objects, the method call will return null.

The objects can be persist by adding them to the Resources. A Resource can be anything like physical storage. In order to accomplish persisting the model, the root object of the model can be added to the resource and a resource can be saved using the method `save`. 
A ResourceSet is a factory of resources which are accessed together and each resource is strongly referred by the other resources within the set.

![Illustration 7: ResourceSet, taken from [9]](image)

### 3.2 EMF Validation Framework

EMF Validation Framework is an independent component of Eclipse Modeling Framework (EMF) which provides the capabilities to validate the EMF Models in both Live and Batch mode. Validation Constraints can be written either in Java or OCL, and all constraints must bind to particular application context. Validation Framework also provides an ability to customize the model traversal algorithms for Model validations [10].

Features provided by the EMF Validation Framework are [10]:

- **Constraint Definition**

  EMF Validation Framework provides an API to define the well-formed semantic rules / constraints. In general, constraints are categorized into Batch Validation Constraints and Live Validation Constraints. Static Constraints must be register under the extension point org.eclipse.emf.validation.constraintProviders in plugin.xml and Dynamic Constraints which are declared in model are registered by declaring class which implements IModelConstraintProvider interface. All Constraints must provided with auxiliary constraint meta-data like ID, Name, Language, Severity, Constraint Class, Validation Mode, Message, etc and also organized into hierarchy of categories.
• **Customizable Model Traversal**

EMF Validation Framework provides an extensible API for customizable model traversal algorithms. Custom model traversal must be register under the extension point `org.eclipse.emf.validation.traversal`. Model Traversing is needed by Batch mode validation, where validation is done on all its owned elements starting from its root model element selected by user. Custom model traversing is also needed for some special models which does not have containment relationship as regular models have. By default, Validation Framework iterates the model hierarchy using reflection API `eAllContents()`.

• **Extensible Constraint Parsers**

EMF Validation Framework provides support for extensible parsers for languages in which Constraints are define. By default, Java and Object Constraint Language(OCL) are supported by this framework. Constraint parsers must define under the extension point `org.eclipse.emf.validation.constraintParsers`. Validation Framework provides interfaces `IParameterizedConstraintParser` and `IConstraintParser` which must be implemented by the constraint parser class which structures the constraints by creating the objects of `IModelConstraint`.

• **Constraint-Context bindings**

Constraint Binding allows to create a Client Context which binds the Constraint or categories of Constraints to the Objects on which validation should be done. Client context must be register under the extension point `org.eclipse.emf.validation.constraintBindings`. Context binding supports two `enablement` and `selector` types of client contexts to declare.

• **Validation Listeners**

Framework provides support to define Listeners which are triggered at end of validation. Validation Listeners must be define under the extension point `org.eclipse.emf.validation.validationListeners`. For every occurrence of validation, validation service will invoke the validation listeners. Validation Listeners can also be added at runtime using the method `addValidationListener()` of validation service.
3.2.1 The Validation Service

*ModelValidationService* is the validation service class provided by the Validation Framework which addresses the EMF object validations by retrieving validators from the *IValidator* interface. EMF Validation Framework provides two types of validation modes defined by the *EvaluationMode*, Batch and Live mode validation.

The static validation uses the Batch-mode validation to validate on selected set of elements invoked manually by the user. The Live-mode validation is done at runtime and invoked by the change notification of the elements. The changes in model elements are carried in form of transactions using the EMF Model Transaction API. The live mode validations are invoked exactly before the transactions are about to commit.

From the below code snippet, *ModelValidationService* class creates a new validation object type-casted to *IBatchValidator* Interface. Using this validation object *validate* method is called passing set of object

![Illustration 8: Validation Service, taken from [10]](image_url)

The static validation uses the Batch-mode validation to validate on selected set of elements invoked manually by the user. The Live-mode validation is done at runtime and invoked by the change notification of the elements. The changes in model elements are carried in form of transactions using the EMF Model Transaction API. The live mode validations are invoked exactly before the transactions are about to commit.

From the below code snippet, *ModelValidationService* class creates a new validation object type-casted to *IBatchValidator* Interface. Using this validation object *validate* method is called passing set of object
as parameter on which validation should be done. All the Live-mode validation constraints can be
included into the Batch-mode validation using the method `setIncludeLiveConstraints` of validator
object and passing `true` as parameter will enables the feature [10].

```java
List objects = myResource.getContents(); // objects to validate

// create a validator
IBatchValidator validator =
(IBatchValidator)ModelValidationService.getInstance().newValidator(Evaluation
Mode.BATCH);

validator.setIncludeLiveConstraints(true);

// use it!
IStatus results = validator.validate(objects);
if (!results.isOK()) {
    ErrorDialog.openError(null, "Validation", "Validation Failed", results);
}
```

The below code snippet shows how a Live-mode validator object is created using the`ModelValidationService`
class and type-casted to `ILiveValidator` interface. The input to the live-mode validator is set of `Notifications`
instead of model objects as in Batch-mode. These `notifications` contains
objects with changed states along with new and old values. If the new change of the object caused
invalidation, the transactions which discussed earlier are roll-backed [10].

```java
List notifications = transaction.getChanges(); // changes in some
hypothetical transaction scope

// create a validator
ILiveValidator validator = (ILiveValidator)
ModelValidationService.getInstance()
    .newValidator(EvaluationMode.LIVE);

// validate only changes to containment references
validator.setNotificationFilter(new FilteredCollection.Filter() {
    public boolean accept(Object element) {
        Object feature = ((Notification) element).getFeature();

        return feature instanceof EReference && ((EReference)
            feature).isContainment();
    }
}
The Notifications can be filtered and Live validation can only be applied on particular filtered Notification set. The Live-mode validation comes with the filter. By default, Live validation filters and accept the Notifications of the objects attach with Resource. The above code snippet and the class diagram below will give a better view of filters.

![Diagram](image)

*Illustration 9: Notification Filter, taken from [10]*

Similar to the Notification filtering, Constraints of both Live and Batch-mode validation can to filter to validate using IConstraintFilter provided by the validation framework. The sample lines of code below filters the constraints based on severity [10].
List objects = myResource.getContents(); // objects to validate

IVValidator validator =
ModelValidationService.getInstance().newValidator(EvaluationMode.BATCH);

// validate only error severity and worse
validator.addConstraintFilter(new IConstraintFilter() {
    public boolean accept(IConstraintDescriptor constraint, EObject target) {
        ConstraintSeverity sev = constraint.getSeverity();
        return sev.toIStatusSeverity() >= IStatus.ERROR.
    }
});

IStatus results = validator.validate(objects);
if (!results.isOK()) {
    ErrorDialog.openError(null, "Validation", "Validation Failed", results);
}

### 3.2.2 Constraint Contribution

Constraints in EMF Validation Framework are the instances of the `IModelConstraint` interface contains methods `validate()` which is used to check the well-formedness of the model elements and
getDescriptor() which retrieve the constraint meta-data using the interface IConstraintDescriptor. Java, OCL or other constraint based languages can be used to implement the constraints. If a constraint is implemented in Java, that constraint class must be subclass of AbstractModelConstraint abstract class containing abstract method validate() with parameter IValidationContext object and return IStatus [10].

The IValidationContext object contains

- target model element which undergoes validation.
- The eventType, if it is NULL then it is batch mode validation else, it is live mode validation.
- In case of live mode validation, the feature and its value is provided.
- The currentConstraintId, provides the current constraint ID.
- The getCurrentConstraintData() and putCurrentConstraintData() methods allows to store and retrieve data which can be useful for validating multiple model elements.
- The skipCurrentConstraintFor() method call will skips the validation of current constraint.

Illustration 11: Java Constraint descriptor, taken from [10]
The IStatus object returned by the validate() method contains the validation results of constraints which are actually objects of IConstraintStatus interface. The ConstraintStatus class implements factory methods used to create a multi-status object.

IValidationContext object contains more methods addResults() and addResult() will add problem elements to the result which related to constraint violation. It also provides methods, createFailureStatus(), which creates the constraint violation status with added result locus set using methods addResults() and addResult(). createSuccessStatus() method will create a success constraint result if constraint evaluation is success [10].

The example constraint class given below

public class LibraryNameIsUnique extends AbstractModelConstraint {
    public IStatus validate(IValidationContext ctx) {
        // the constraint is declared to target only Library, so we can safely cast
        Library target = (Library) ctx.getTarget(); // object to validate
    }
}
// does this library have a unique name?
Set<Library> libs = findLibrariesWithName(target.getName());
if (libs.size() > 1) {
    // report this problem against all like-named libraries
    ctx.addResults(libs);
    // don't need to validate these other libraries
    libs.remove(target);
    ctx.skipCurrentConstraintFor(libs);
    // arguments are slotted into the message pattern
    return ctx.createFailureStatus(new Object[] {
        target, libs});
}
return ctx.createSuccessStatus();

3.2.3 Validation Results Reporting

EMF Validation Framework provides a validation listener, LiveValidationListener class which reports the live validation results either in a dialog or in a console. In order to use this listener, client context of the application must register under the extension point org.eclipse.emf.validation.ui.UIRegisteredClientContext.

EMF validation framework also provides MarkerUtil utility class which creates appropriate markers in the problems reporting view. This class contains method like createMarkers() or updateMarkers() which will take the IStatus object as parameter and create markers. Batch mode validation results can be reported using this utility class. By default, the marker type reference is org.eclipse.emf.validation.problem and users can have custom marker types [10].
Chapter 4

4 Implementation

This chapter describes the various stages of implementation of a Validation plug-in. First, we discuss about the eclipse plug-in design and the contributions which incorporates the EMF Validation Framework into the plug-in. Then we discuss about adding the validation constraints, validation handlers and later about client context. We will also discuss about the reporting the validation results at the end.

4.1 Plug-in Design and Contributions

Validation feature can be added into the ModelicaML plug-in by developing an another Eclipse plug-in. This validation plug-in supports both the Batch and Live validation. The Batch validation should invoked by the user manually, whereas, Live validation is invoked automatically at runtime on occurrence of events like addition, modification and deletion on model elements.

The validation plug-in should contribute a button and a customized view. The button (validate button) invoke the Batch validation and the view is used to populate the validation results. This plug-in also uses the EMF Validation Framework to create validation handlers and constraints which validate the ModelicaML models.

From chapter 2, A plug-in can extended or contribute using the Extensions section in plugin.xml. A button can be contributed into this plug-in using the extension points

- `org.eclipse.ui.menus`.
  
  A menu of type button and its properties can be register under this extension point. This button should be associated with a command so that whenever this button is pressed, a command will execute. In order to place the button on a main tool-bar, `LocationURI` of the menu should be pointed to `toolbar:org.eclipse.ui.main.toolbar`.

- `org.eclipse.ui.commands`.
  
  The commands represents the specific semantic behavior which are associated with menus and
initiated whenever a menu is pressed. These commands are registered under this extension point.

- **org.eclipse.ui.handlers.**

  The Java classes (handlers) which are associated with the commands and should be invoked when a command is under execution. These handlers are registered under this extension point.

A view can be contributed by using the extension point **org.eclipse.ui.views.** A custom view and its properties like ID, name, class and category are register under here. The class should point to the Java class which holds the functionality of the view.

EMF Validation Framework can be integrated using the extension points as discussed in chapter 3. This plug-in extends only to the extension points, **org.eclipse.emf.validation.constraintProviders, org.eclipse.emf.validation.constraintBindings** provided by the Validation Framework.

Part of plugin.xml descriptor file shows a contribution of validation button.

```xml
<extension
    point="org.eclipse.ui.menus">
  <menuContribution
    allPopups="false"
    locationURI="toolbar:org.eclipse.ui.main.toolbar">
    <toolbar
      id="org.openmodelica.modelicaml.toolbar.validation"
      label="ModelicaML Validation">
      <command
        commandId="org.openmodelica.modelicaml.validation.entiremodel"
        icon="icons/validate.gif"
        label="Validate ModelicaML Model"
        style="push"
        tooltip="Validate ModelicaML Model">
        <visibleWhen>
          <with
            variable="activeEditorId">
            <equals
              value="org.eclipse.papyrus.infra.core.papyrusEditor"/>
          </equals>
          </with>
        </visibleWhen>
      </command>
      <visibleWhen
        checkEnabled="false">
        <with
          variable="activeWorkbenchWindow.activePerspective">
          <equals
            value="org.openmodelica.modelicaml.perspective"/>
        </equals>
      </visibleWhen>
    </toolbar>
  </menuContribution>
</extension>
```
4.2 Validation Constraints

The constraints or validation rules can be contributed into the plug-in under the extension point `org.eclipse.emf.validation.constraintProviders` provided by the EMF validation framework. Within this plug-in, constraints are categories into Live-mode validation constraints and Batch-mode validation constraints. The constraints of Live-mode validation are registered under the category `org.openmodelica.modelicaml.validation.live` (Live-mode) and the constraints of Batch-mode validation under `org.openmodelica.modelicaml.validation.batch` (Batch-mode). The validation needs the `namespaceURI` of the meta-model of the ModelicaML models. Since, the ModelicaML is an UML profile, URI points to the UML meta-model packages. At the time of writing this report, the meta-model packages used by the ModelicaML validation are

- http://www.eclipse.org/uml2/2.0.0/UML
- http://www.eclipse.org/uml2/2.1.0/UML
- http://www.eclipse.org/uml2/3.0.0/UML
- http://www.eclipse.org/uml2/4.0.0/UML

Each constraint must be registered with properties like ID, name, language, severity and implementation class along with constraint message. In ModelicaML model validation, all constraints are implemented in Java language and each constraint is allocated with a number starting from 01 prefixed with 'C' like 'C01'. The ID of each constraint will follow the format like `org.openmodelica.modelicaml.validation.C03` where the last part of the ID i.e., C03 will be different for each constraint. Similarly, the naming convention of the constraints and their Java class implementation will follow as constraint number followed by an underscore and by the actual name of the constraint.
The constraint implementation Java class contains the constraint number, description, severity and validation mode in form of comments. The output messages are well formatted within the constraint implementation class itself.

The constraint message section under constraint description in plugin.xml file must contain the constraint number followed by a semicolon and object 0 (which is the formatted message returned within constraint implementation class) which will look like 'C03: {0}'. The list of all implemented constraints are provided under the appendix section of this report.

### 4.3 Validation Handlers

ModelicaML validation plug-in contains implementation of only one handler for Batch-mode validation. The Live-mode validation handler is already implemented in Papyrus and utilized by the ModelicaML validation plug-in. The Batch-mode validation handler is invoked by handler of *validate* button located on main toolbar. The button handler will create a validator object of *IBatchValidator* interface using the *ModelValidationService* and invokes the validation. The Batch-mode validation also validates Live-mode constraints along with Batch-mode constraints by calling the method *setIncludeLiveConstraints*. This validation plug-in uses the default tree traversal provided by the framework and the object passed to the validator is the object selected by the user or by default, root of the project. The element selected by the user is an UML object and we need it to be converted to EMF using the method *adaptSelectedElement* method provided by the *ModelicaMLServices* class in

Illustration 13: Constraint Properties
4.4 **Client Context**

A client context with ID `org.openmodelica.modelicaml.validation.ClientContext` is registered under the extension point `org.eclipse.emf.validation.constraintBindings` and the two constraint categories are binded to this context along with a selector class.

The selector class should implement an interface `IClientSelector` with method `selects`. The ModelicaML validation plug-in should only validate the objects which are the instances of UML Element.

The Client Context code:

```java
public class ValidationDelegateClientSelector implements IClientSelector {

    @Override
    public boolean selects(Object object) {
        return object instanceof Element;
    }
}
```

4.5 **Reporting Validation Results**

The Live-mode validation results of Severity warning and errors are displayed as dialog prompts and the erroneous validation results are automatically roll- backed to the previous consistent state. The Batch-mode validation results of Severity warning and errors are displayed as markers in the custom view which is similar to the general problems view of the Eclipse. A custom marker type 'ModelicaML Validation' is created using the extension point `org.eclipse.core.resources.markers` and registered with ID `org.openmodelica.modelicaml.marker.validation.problem`.

ModelicaML Validation Marker generation:

```java
MarkerUtil.updateMarkers(status, Constants.MARKERTYPE_VALIDATION_PROBLEM,
    new IMarkerConfigurator() {
```
```java
@Override
public void appendMarkerConfiguration(IMarker marker,
        IConstraintStatus status)
    throws CoreException {
    if (status.getTarget() instanceof NamedElement)
    {
        marker.setAttribute(IMarker.LOCATION,
        ((NamedElement) status.getTarget()).getName());
    }
    else if (status.getTarget() instanceof Element &&
    ((Element) status.getTarget()).getOwner() instanceof NamedElement ){
        marker.setAttribute(IMarker.LOCATION,
        ((NamedElement)((Element) status.getTarget()).getOwner()).getName());
    } else{
        marker.setAttribute(IMarker.LOCATION, status.getTarget().toString());
    }
}
```

The view can be customized to look similar to the problems view using the extension point

![Illustration 14: ModelicaML Validation Markers view](image-url)
The severity, description, location and creation time fields can be added in to the view under this extension point and also the 'ModelicaML Validation' marker type can binded so that only markers of this type will be displayed under this view.

The markers can be easily generated using Status object returned at every occurrence of validation and using the class `MarkerUtil` class provided by the Validation Framework. The marker contents are modified for proper object location and for custom marker type.
Chapter 5

5 Testing

The quality of the constraint is much important to ensure the requirements of a constraint within its scope. The evaluation of validation constraints can be done on the test models. Each constraint will have two different test-models, one should validate and other should invalidate the constraint.

In order to test the constraints on the test-models, the plug-in should be build and installed in the Eclipse environment. The self-hosting feature of the Plug-in Development Environment (PDE) made it easy to test and debug the plug-in without installing them on to current working IDE.

There are currently, 57 constraints are implemented and each have both working and not-working test-models. All test-models are available inside the folder “TestModels” and committed along with the code into SVN repository. These test-models are also used to test the time taken by the validator to validate all the test-models of all the constraints.
Chapter 6

6 Conclusion and Future work

In the process of model development, model validation is an important step which assesses the authenticity of the model and ensures the accuracy of the model within the system. In this thesis, I implemented validation on ModelicaML models with the help of EMF validation framework incorporated into an Eclipse plug-in with the help of provided extension points which does both the Live mode and Batch mode validations. During the progress of this thesis, a total of 57 validation rules on ModelicaML models are implemented into the plug-in using validation framework. A custom view is also added into the plug-in for reporting the Batch-mode validation results. This thesis accomplish the major requirements of the validation plug-in and it is ready-to-use.

In the case of future work, the time taken by validator to validate the ModelicaML models can be optimized with the help EMF In-query framework\(^4\), which does query evaluation in incremental fashion and it is under-development at the time of thesis implementation. And the Live mode validation results can be reported as problem markers instead of dialog pop-up which is currently not possible to accomplish with the current version of Papyrus.

---

\(^4\) [http://viatra.inf.mit.bme.hu/incquery](http://viatra.inf.mit.bme.hu/incquery)
Bibliography


Appendix A

List of Constraints

C1: UML Packages that have the same owner must not have same names.
Severity: Error

C2: UML Classes (that is not UML Behavior, except UML State Machine) that have the same owner must not have same names.
Severity: error

C3: Any UML Class (that is not UML Behavior) must have a ModelicaML stereotype applied.
Severity: warning
Rational: Such elements will be ignored by the code generator.

C4: Any UML Property that is owned by a UML Class with a ModelicaML stereotype applied must have a ModelicaML Stereotype applied as follows:
• If the type of the UML Property is of UML PrimitiveType then the stereotype must be <<Variable>>
• Else if the type of the UML Property has the stereotype <<Requirement>> then the stereotype must be <<RequirementInstance>>
• Else if the type of the UML Property has the stereotype <<CalculationModel>> then the stereotype must be <<CalculatedProperty>>
• Else if the type of the UML Property is not an instance of UML PrimitiveType then the stereotype must be <<Component>>
Severity: error
Rational: Such elements will be ignored by the code generator.

C5: Any UML OpaqueBehavior that is owned by a UML Class with a ModelicaML stereotype applied must have either the stereotype <<Equation>> or <<Algorithm>> applied.
Severity: error
Rational: Such elements will be ignored by the code generator.

C6: Any UML StateMachine that is owned by a UML Class with a ModelicaML stereotype applied must have either the stereotype <<ModelicaMLStateMachine>> applied.
Severity: error

C7: A UML Class with a ModelicaML stereotype applied can only contain elements that are instances of:
• UML Class
• UML Enumeration
• UML PrimitiveType
• UML Property
• UML Port
• UML Parameter
• UML StateMachine
• UML Activity
• UML OpaqueBehavior
• UML FunctionBehavior
• UML Comment
• UML Connector
• UML Constraint
• UML Generalization
• UML Dependency

Severity: warning
Rational: Other elements will be ignored by the code generator.

C8: A UML Class with a ModelicaML stereotype <<ModelicaPackage>> applied can only contain elements that are instances of:
• UML Class with a ModelicaML stereotype applied
• UML Enumeration
• UML PrimitiveType
• UML Property with a ModelicaML stereotype <<Variable>> and variability set to “constant”
• UML FunctionBehavior with ModelicaML stereotype applied
• UML Comment
• UML Generalization with ModelicaML stereotype applied
• UML Dependency

Severity: warning
Rational: Other elements will be ignored by the code generator.

C9: There must be no loops in UML StateMachine sub-statemachine references.
Severity: error

C10: A UML Parameter that is owned by a UML FunctionBehavior with the stereotype ModelicaML <<Function>> applied must have its causality set to “input” or “output”.
Severity: error
C11: A UML FunctionBehavior with the ModelicaML stereotype <<Function>> applied must have at least one UML Parameter with causality “input” and exactly one UML Parameter with the causality “output”.
Severity: error

C12: Any Property of a UML Class with the ModelicaML stereotype <<Block>> applied must have causality “input” or “output”.
Severity: error

C13: Any UML Connector that is owned by a UML Class with the ModelicaML stereotype applied must have the stereotype “Connection” applied.
Severity: error

C14: Any UML Connector with ModelicaML stereotype applied can only connect UML Ports that have the ModelicaML stereotype “ConnectionPort” applied.
Severity: error

C15: UML Class with ModelicaML stereotype applied can only have one UML Dependency with the stereotype <<RedeclareRelation>>.
Severity: error

C16: UML Class with ModelicaML stereotype applied can only have one UML Dependency with the stereotype <<ConstrainedByRelation>>.
Severity: error

C17: UML FunctionBehavior with ModelicaML stereotype applied can only have one UML Dependency with the stereotype <<PartialDerivativeRelation>> which must point to a UML Function Behavior with the ModelicaML stereotype “Function” applied
Severity: error

C18: Any UML Generalization that is owned by a UML Class with a ModelicaML stereotype applied must have the stereotype <<TypeExtendsRelation>> if it points to a UML Primitive Type instance or <<ExtendsRelation>> otherwise.
Severity: error

C19: The inheritance rules of Modelica classes as described P.Fritzson book and the Modelica spec.
Severity: error

State Machines:

C20: All UML Transitions form a UML PseudoState of kind “entryPoint” must lead to the targets that are contained in the same UML Region.
Severity: error
C21: UML PseudoState with kind “exitPoints” can have at most one incoming transition.
Severity: error

C22: UML PseudoState with kind “entryPoint” cannot have incoming transitions.
Severity: error

C23: UML PseudoState with kind “exitPoint” cannot have outgoing transitions.
Severity: error

C24: UML PseudoState with kind “deepHistory” are not supported.
Severity: error

C25: UML StateMachine A ModelicaML StateMachine can contain at most one UML PseudoState of kind “initial”.
Severity: error

C26: A UML PseudoState of kind “initial” can have at most one outgoing transition.
Severity: error

C27: Outgoing transition of A UML PseudoState of kind “initial” must not have a guard.
Severity: error

C28: UML StateMachine A ModelicaML StateMachine can contain at most one UML PseudoState of kind “shallowHistory”.
Severity: error

C29: UML StateMachine A ModelicaML StateMachine must have either one UML PseudoState of kind “initial” or one UML PseudoState of kind “shallowHistory”.
Severity: error

C30: The outgoing transition from the UML PseudoState of kind “initial” or “shallowHistory” node must point to a state in that is in the same UML Region.
Severity: error

C31: A UML PseudoState of kind “choice” or “junction” must have at most one incoming and at least one outgoing transitions.
Severity: error

C32: A UML PseudoState of kind “choice” or “junction” should have one else branch.
Severity: warning
Rational: This is not required by the language but is a modeling guideline.

C33: A UML Transition to UML PseudoState of kind “fork” must have UML State as source which must be in the same UML Region.
Severity: error
C34: A UML Transition to UML PseudoState of kind “fork” must have at least two outgoing UML Transition and at most one incoming UML Transition.
Severity: error

C35: UML Transition from a UML PseudoState of kind “fork” must lead to UML States that are in the same UML State which is composite. All target states must be in orthogonal UML Regions.
Severity: error

C36: UML::Transition from a UML::PseudoState of kind 'fork' must have target as state or final state. Severity: error

C37: UML::PseudoState of kind 'Join' must have at-least two incoming and exactly one outgoing transition.
Severity: error

C38: UML::Transition to a UML::PseudoState of kind 'join' must be from states that are all in different region of a composite state.
Severity: error

C39: UML::Transitions should not cross the region borders inside the same composite state.
Severity: error

C40: ConnectionPointReference may only refer one entry point.
Severity: error

C41: ConnectionPointReference may only refer one exit point.
Severity: error

C42: ConnectionPointReference referencing an entry point may not have outgoing transition.
Severity: error

C43: ConnectionPointReference referencing an exit point may not have incoming transition.
Severity: error

C44: ConnectionPointReference referencing an exit point must have an outgoing transition.
Severity: error

C45: Final State may not have outgoing transitions.
Severity: error

C46: Terminate State may not have outgoing transitions.
Severity: error

C47: A ModelicaML Activity must have exactly one initial node.
Severity: error

C48: A ModelicaML Activity must have exactly one Outgoing Transition from initial node.
Severity: error

C49: A ModelicaML Activity must own only the following elements:
   - OpaqueAction
   - CallBehaviorAction
   - ControlFlow
DecisionNode

InitialNode

MergeNode

Severity: error

C50: In ModelicaML Activity, A DecisionNode must have 'If' or 'When' Stereotype applied.
Severity: error

C51: In ModelicaML Activity, A MergeNode can have only one outgoing ControlFlow.
Severity: error

C52: In ModelicaML Activity, A outgoing ControlFlow of MergeNode cannot have a Guard.
Severity: error

C53: In ModelicaML Activity, A DecisionNode can only have one incoming ControlFlow.
Severity: error

C54: A UML::Dependency with the stereotype <<UseToVerify>> or <<DoNotUseToVerify>> should point from an element with stereotype <<VerificationScenario>> to an element with <<Requirement>>.
Severity: error

C55: A UML::Property with the stereotype <<Mediator>> should not have more that one UML::Dependency with stereotype <<ProvidesValuefor>> or <<RequiresValueFrom>> pointing to the same UML::Element.
Explanation: Dependencies between Mediator and clients or providers should be unique
Severity: error

C56: Each UML::Dependency must have exactly 1 source and 1 target set.
Severity: error

C57: Each UML::Generalization must have exactly 1 source and 1 target set.
Severity: error
Appendix B

User Guide

This guide describes how to use ModelicaML validation plug-in.

Installation

This validation plug-in is deployed with ModelicaML. A latest ModelicaML plug-in is needed to be installed in order to use the validation.

The guide for installing and configuring ModelicaML is available at ModelicaML homepage.⁵

Plug-in Customizations

There are two customization of ModelicaML validation plug-in for effective usage.

1. Adding the ModelicaML Validation view. In Eclipse, all views are located from the main menu Window > Show View > Other … opens a dialog window as below.

---

⁵ www.openmodelica.org/modelicaml
ModelicaML Validation view can be located under the ModelicaML folder. Or view can be search through the search box provided.

2. There are many other plug-in along with ModelicaML registers the different categories of constraints which may not be useful. We can disable them at Model Validation preference page. The preference window is located by main menu Window > Preferences.

The constraints can be disabled at Constraints page located under the Model Validation. The categories of ModelicaML validations (are ModelicaML Batch and ModelicaML Live.) must be enable in order to work.

Validating ModelicaML models

The Batch validation can be invoked by using the validate button located on the main toolbar of the Eclipse. And we already discussed that the live validation will invoke automatically when state of
model element changes.

The validation results of Batch-mode are populated in the ModelicaML Validation view as markers.

And live mode validation results are reported in the form dialog prompt.
C1 'ModelicaMLModel::C1::C1_OK' contains packages or classes with same name 'Package1'

☐ Don’t show this dialog again for warning messages

OK
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© [Goutham Gatla]