VDM2UML in VDM VSCode Bachelor Thesis

Lucas Bjarke Jensen, Jonas Lund

February 2022

Contents

1 Background **2**
1.1 Vienna Development Method 2
1.2 Unified Modeling Language 2
1.3 Diagram Interchange Methods 3
1.4 Overture 3
1.5 VDM VSCode 3
1.6 Language Server Protocol 4

2 Balancing Future Proofing and Usability **5**

3 UML Transformations' Dependency on the Eclipse IDE **6**
3.1 Moving the Plugin from the IDE to the Core Side 6

4 Repurposing the Overture Code for UML Transformation to enable JAR Packaging **8**
4.1 Analysis of the Old Vdm2Uml Handler 8
4.2 The New Vdm2Uml Handler 9
4.3 Analysis of the Old Uml2Vdm Handler 11
4.4 The New Uml2Vdm Handler 12

5 Packaging the JAR using Maven **13**
5.1 Running the JAR File 13

1 Background

1.1 Vienna Development Method

The Vienna Development Method (VDM) is the leading formal method, focused on the modeling of computer-based systems. The goal of the method is to help enforce stringent system requirements using formal semantics, to ensure safety and security in the software before it is deployed for either industrial use or as an aid in the research process. [7]

At the core of the VDM methods usefulness for software development lies the VDM model. This model encapsulates some data and some amount of functionality. The data can, along with the usual assortment of basic types for any kind of programming language, be in the form of structured data types such as union and composite types as well as collection types such as sets, sequences and maps.

What is important for this project is that VDM++ and VDMSL models are object oriented and can thus be portrayed as classes. A convenient method of visualising a model is to take these classes and make them into a class diagram (CD). The visualisation of class diagrams is possible with the widespread Unified Modeling Language.

1.2 Unified Modeling Language

UML is an industry standard visual tool for designing and modeling software-based systems [9]. It is specified by the Object Management Group (OMG) to provide a semi-formal visual language to give an abstract representation of object-oriented systems. UML is therefore very relevant as an aid in The Vienna Development Method, as a tool to design and especially to communicate a system's architecture. This is also why CDs are the primary focus of integration, since they depict the structure of object oriented systems the best.

The transformations between VDM and UML 2.1 have already been developed on the Overture Project as a plugin in Overture Tool. The interest of this paper lies in the repurposing of that implementation onto the new VDM VS Code extension currently in development.

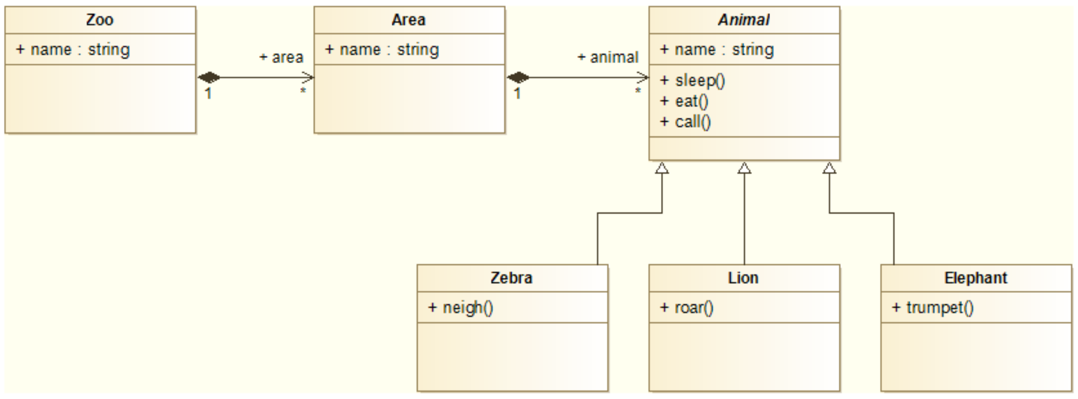


Figure 1: Zoo toy example of a Class Diagram in the Modelio modeling tool.

1.3 Diagram Interchange Methods

The UML transformations implemented as plugins in the Overture Tool use the XML Metadata Interchange (XMI) standard. It is created by OMG [9] with the purpose of exchanging information using the Extensible Markup Language (XML). XMI is often used as a standard for UML models but can be used for any meta model that is expressed in the OMG Meta-Object Facility (MOF).

A problem using XMI is that the newest versions are not widespread, especially on the level of Diagram Interchange (XMI[DI]) [10]. Different tool vendors' implementations of XMI for UML vary quite a bit making the goal of free interchange of UML models difficult. The VDM to UML transformation implemented in Overture was developed with Modelio 2.2.1 in mind as the tool for visualisation, but even then it is difficult to import models correctly and it does not seem like there is much spacial information in the standard as for how classes and links are placed in the diagram. In part 2, other methods for diagram interchange are taken into consideration.

1.4 Overture

In the mid 2000s the Overture project lead an initiative to provide an open source integrated development environment (IDE) for VDM and its various dialects [6]. This included the object oriented version of VDM, called VDM++. The overture tool extends the VDMJ command line tool with a graphical user interface (GUI) provided by the Eclipse framework. The core functionality of VDMJ is coupled with the Eclipse GUI components using a plugin architecture. This architecture leads to a high level of extensibility which enables plugins to be developed that extend the functionality of Overture, one of these being the bidirectional mapping between object oriented VDM languages (VDM++ and VDMSL) and UML 2.1.

1.5 VDM VSCode

Overture is currently the most complete and popular platform for VDM. However, it is getting competition from the Visual Studio Code (VS Code) platform in recent years. Using the VDM VSCode extension, VS Code can connect to a Language Server and work like an IDE while also having the pro of being extendable to other specification languages.

VDM VSCode is an extension for the free source code editor Visual Studio Code developed by Jonas Rask. It provides language support for VDM++ among other VDM dialects in VS Code using the Language Server powered by VDMJ developed by Nick Battle.

The extension currently already contains a lot of the features of its Overture counterpart such as java code generation, debugging, combinatorial testing, etc. [12] This paper will look into adding translation between VDM and UML CDs as well as visualisation of the generated models.

The translation command will be similar to the already implemented Java Code Generation command. This command was implemented by taking the existing code generation plugin from the Overture Tool and packaging it as a JAR executable. This JAR can then be run from VS Code using a command containing desired output folder, VDM dialect and other settings. Currently, this JAR file is located on the client side of VDM VSCode, but in the future, it will be moved to the VDMJ Language Server to fulfill the goal of decoupling the specification language functions from the VS Code extension. The UML translation JAR files will also be located on the client side for now.

1.6 Language Server Protocol

"The idea behind a Language Server is to provide the language-specific smarts inside a server that can communicate with development tooling over a protocol that enables inter-process communication.

The idea behind the Language Server Protocol (LSP) is to standardize the protocol for how tools and servers communicate, so a single Language Server can be re-used in multiple development tools, and tools can support languages with minimal effort." [8]

This is what Microsoft writes about the essence of the Language Server and LSP. Using traditional methods, developers would have to provide language support and functionality for each individual IDE resulting in a lot of repeated work. However, using LSP, all the language neutral information — such as data types, code complection etc. — can stay on the server and the client API can request this information through the LSP, thus getting rid of the need to do such implementations for every code editor.

Specification LSP (SLSP)

An extension of the LSP has to be made to support specification language specific features. An SLSP has been developed to support Proof Obligation Generation, Combinatorial Testing and translation features. [11]

A DAP (Debug Adapter Protocol) is also an important counterpart to the LSP, however as we will not be needing debugging functionality for our implementation, we will provide no further description in this paper.

For the core of this project, the Language Server will simply be for storing the executable JAR file that contain the UML transformation functions. However, it may also be possible to have the visualisation be part of the Language Server. If the visualisation were to happen through some third party extension, such as PlantUML (see 2), the parsing of either VDM or UML to a language the third party extension can read could happen on the Language Server. Though it might be smarter to have it on the client side of VDM VSCode if the extension for visualisation is too dependent on the VS Code platform specifically.

2 Balancing Future Proofing and Usability

To support coupling between VDM++ and UML inside the VS Code platform, a number of different approaches were weighted against each other and the following information was taken into consideration.

The connection between Modelio and Overture's UML transformation has drifted over time as Modelio has changed its implementation of the XMI standard to the point that importing a VDM model into Modelio only presents the classes of the model with no relationship between the classes, such as inheritance or dependency. Trying to show these relations however leads to a tangled web of connection which does not give an overview of the models architecture. And that is if one is even successful in importing said model, as conflicting EMF formats leads to incompatibility between versions.

This problem is the same for other UML tools, as each tool as has a different way of implementing the XMI standard. The standard therefore fails in enabling interoperability between the different tools.

One could choose a specific tool and develop VDM VSCode's UML import/export functions to suit the tools way of implementing the XMI standard. This would mirror what was done for Modelio and would require continually updating the coupling if the XMI implementation of the chosen tool changed.

However, there unfortunately does not exist any suitable UML tool extensions for VS Code that can import a UML file which adheres to the XMI standard. To enable a seamless visualisation of CDs inside VS Code, one would need to create a UML visualisation tool form scratch that supports this functionality, which is not in the scope of this project.

A solution to this is finding a VS Code extension that creates UML from a text based language and create a translator from VDM to that language. It then would not depend on that UML tool to have an import function as one could simply output some text or a file that describes the VDM model in the tools language.

One such tool is PlantUML [14] PlantUML fulfill the previously stated criteria as it is a text based diagram tool that exists as a VS Code extension [5]. As an extension it appears as a tab inside VS Code, has a continually updating live view, and can be worked on by multiple team members as it integrates with versioning tools like Github. It is even cited in books pertaining to critical[2] and cyber-physical[1] systems. It is because of this one of the more relevant options with regards to the goals of this project.

This project could chose to forego the XMI standard entirely and focus on a connection between Plant UML and VDM++. Achieving this would result in higher usability that what was achieved with the Modelio connection as users would only need to open a new tab in VS Code as opposed to opening a new program to import or export VDM++ models.

This would however greatly decrease the maintainability of the UML to VDM++ connection and not be particularly futurep roof. Although the creator of PlantUML updates the component regularly [15] and is promising backwards compatibility [13], this is no guarantee and the tool might be abandoned in the future or changes in the language might occur even if definitions of fundamental diagram types like CD's are unlikely to change. The UML connection would then be dependent on PlantUML to an even higher degree than Overtures UML connection depending on Modelio.

Because of this the decision was made to first at foremost port the already existing implementation of the UML connection from Overture to VDM VSCode. From there, steps will be taken to create a link from the Overture/VDM VSCode XMI implementation to the PlantUML natural language. This has several advantages, the main one being that since the link will be tailored to the current way of doing UML transformations, there will be no risk that the link fails because of incompatible EMF formats or differences in XMI implementations.

3 UML Transformations' Dependency on the Eclipse IDE

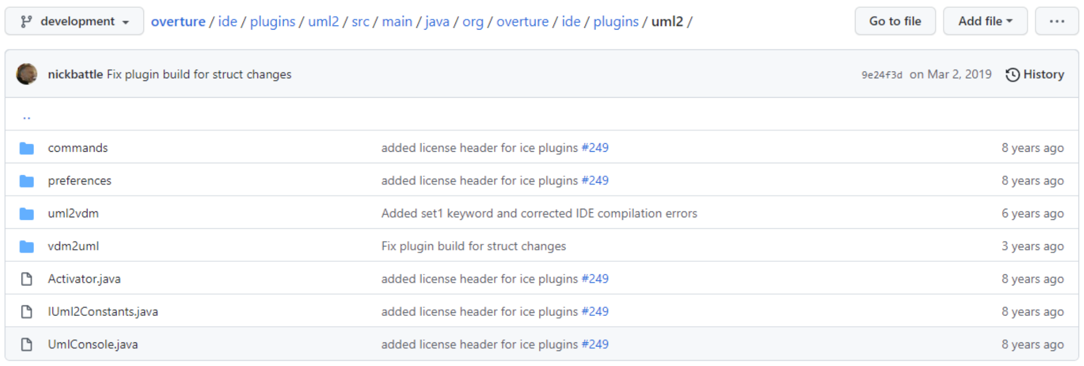
Our method of implementing the UML transformations hinge on packaging JAR files that contain the desired main function as well as all the necessary dependencies for that function which is then run with a command in VS Code when a transformation context button is clicked. Doing so will make the plugin future proof, since whether the Java environment or the Eclipse package repositories are updated, the JAR will contain all dependencies necessary to run that specific main function.

However, getting Maven to package these JARs correctly needs some considerations.

3.1 Moving the Plugin from the IDE to the Core Side

The VDM to UML transformation is implemented in Overture on the Eclipse side. However, contrary to eg. the java code generation, this transformation is implemented as a plugin on the IDE side of the Overture Tool. The plugin being dependent on the GUI aspects of Eclipse makes it incompatible with jar packaging through Maven. This is because the project has dependencies that are specific to the Eclipse IDE as well as Eclipse dependencies that Maven has trouble finding because they have in some way become obsolete.

To circumvent this we will attempt to move the entirety of the UML transformation from the IDE to the core of the Overture Tool, as well as remove all functionality from the implementation that is dependent on the Eclipse IDE. In figure 2, it is shown how the plugins previously were located on the IDE side of Overture Tool. In the code, changes are made to avoid use of



overture / ide / plugins / uml2 / src / main / java / org / overture / ide / plugins / uml2 /

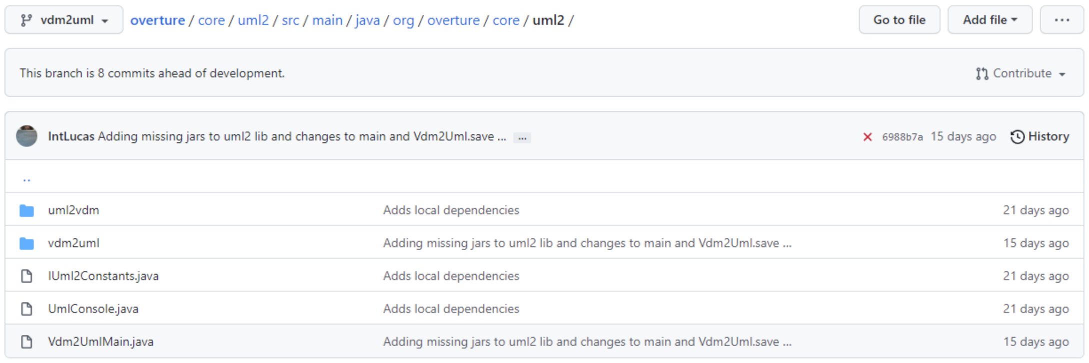
nickbattle Fix plugin build for struct changes 9e24f3d on Mar 2, 2019

..

commands	added license header for ice plugins #249	8 years ago
preferences	added license header for ice plugins #249	8 years ago
uml2vdm	Added set1 keyword and corrected IDE compilation errors	6 years ago
vdm2uml	Fix plugin build for struct changes	3 years ago
Activator.java	added license header for ice plugins #249	8 years ago
IUml2Constants.java	added license header for ice plugins #249	8 years ago
UmlConsole.java	added license header for ice plugins #249	8 years ago

Figure 2: UML transformations as plugins in the IDE directory.

methods specific to the Eclipse IDE, eg. the `UmlConsole.java` class is changed from using the `org.eclipse.ui` extension point to simply using the generic `java.io.PrintWriter` as seen in the following code snippet:


vdm2uml / overture / core / uml2 / src / main / java / org / overture / core / uml2 /

Go to file | Add file | ···

This branch is 8 commits ahead of development.

Contribute

IntLucas Adding missing jars to uml2 lib and changes to main and Vdm2Uml.save ... ✕ 6988b7a 15 days ago History

..		
uml2vdm	Adds local dependencies	21 days ago
vdm2uml	Adding missing jars to uml2 lib and changes to main and Vdm2Uml.save ...	15 days ago
IUml2Constants.java	Adds local dependencies	21 days ago
UmlConsole.java	Adds local dependencies	21 days ago
Vdm2UmlMain.java	Adding missing jars to uml2 lib and changes to main and Vdm2Uml.save ...	15 days ago

Figure 3: UML transformation moved to the core directory.

```
public UmlConsole()
{
  {
    out = new PrintWriter(System.out, true);
    err = new PrintWriter(System.err, true);
  }
}
```

This console is then used in `Vdm2Uml.java` at line 128:

```
private UmlConsole console = new UmlConsole();
```

On the core side we now have all the functionality needed for the VDM2UML transformation. We have also added a main function which is called when the jar file is run from VS Code. There were issues satisfying the same Eclipse resource dependencies that were used for the plugin implementation, such as `org.eclipse.uml2` and `org.eclipse.emf` packages.

A workaround was found by finding the packages and converting them to JAR files, then adding them to the repository in the `lib` folder. The `org.eclipse.uml2` dependency is updated on the Eclipse repository, but it just so happened that the version necessary for this project was uploaded on GitHub [4]. The needed dependencies were then packaged as jars and included in the POM file with a specified system path, eg.

```
<dependency>
  <groupId>org.eclipse.uml2</groupId>
  <artifactId>common</artifactId>
  <scope>system</scope>
  <version>2.5.0.v20200302-1312</version>
  <systemPath>${basedir}/src/lib/org.eclipse.uml2.common_2.5.0.
    v20200302-1312.jar</systemPath>
</dependency>
```

4 Repurposing the Overture Code for UML Transformation to enable JAR Packaging

One way to implement the connection between UML and VDM is to package the existing Overture code for converting VDM to UML as a JAR file and call that within VS Code. Since this method does not involve changing the code which is executed to perform the conversion, only the handler that reads the files, checks the validity of these files and passes the expected arguments to the convert method will need to be replaced.

To do that an analysis of how the Overture implementation calls the converting methods and how the code could be repurposed was conducted. To start off, the direction from VDM to UML is considered.

4.1 Analysis of the Old Vdm2Uml Handler

`Vdm2UmlCommand` is the name of the old handler for the transformation going in the direction of VDM to UML. It is executed when "Convert to UML" is selected in the Eclipse drop-down menu. Upon being called, it is checked whether the selected element is an instance of the `IProject` class, which is an Eclipse interface class for projects. If multiple elements in the GUI are selected the first one is chosen. THE following line is then executed:

```
IVdmProject vdmProject =
    (IVdmProject) project.getAdapter(IVdmProject.class);
```

Here the `IProject` instance is converted, by the Eclipse method `getAdapter`, to the `IVdmProject` class. If the project was not of class type `IVdmProject` the new `vdmProject` is null and the transformation is terminated. A `model` is then extracted from the project, as seen below:

```
final IVdmModel model = vdmProject.getModel();
```

And a check for syntax errors is made. Afterwards it is checked whether the model is empty or the incorrect type and an exit status is prompted if this is the case. Then another check ensures that the model is the correct type and either the VDM-RT or VDM++ dialect. If this is the case, the main body of the transformation is executed. The transformation starts by creating a new instance of `Vdm2uml`, which takes preferences indicated by flags as arguments.

A try-catch statement is then executed, with the try block being the actual conversion and the catch clause being an error message. Another try-catch statement then attempts to save the new UML file on a specified path, with some error handling in the catch clauses.

Considering the end goal of migrating the UML connection to VDM VSCode, expecting that the element to be converted is an instance of the `IProject` class proposes a problem, since this will not be the case when converting a VDM VSCode project to UML.

One way to solve this could be to extract information about the current VS Code project, create an intermediate Eclipse project, and then perform the transformation like `Vdm2UmlCommand`. This however runs into the problem of having to depend on the IDE side as discussed in ...

To circumvent this, one might also simply remove the notion of an `Iproject` and the types `vdmProject` and `model` which are derived from the `Iproject` resource. It would then require finding new ways of providing the same functionality as these classes provide. These classes occur in the code as:

```
model.isParseCorrect()
model.isTypeCorrect()
vdmProject.getDialect() == Dialect.VDM_PP
project.getName(),
model.getClassList()
```

The biggest problem is in the way the `convert` method is called, which is shown below.

```
vdm2uml.convert(project.getName(), model.getClassList());
```

In the call for the method, the name of the project and a list of classes in the project are arguments in the conversion. The name of the project is relatively easy to get, but the `getClassList()` method is harder to acquire. More accurately a method that is not dependent on any eclipse IDE resource which returns a list of type `<SClassDefinition>`, needs to be found. The solution to this is the `TypeCheckResult` class and its `typeCheckPp` method. The `TypeCheckResult` class is part of the VDM type checker. When the `typeCheckPp` method is called, the list of class definitions can be found as an attribute on the `TypeCheckResult` instance, which is named `tcResult`.

```
TypeCheckResult<List<SClassDefinition>> tcResult = TypeCheckerUtil.
    typeCheckPp(files);
```

Furthermore the list of class definitions can simply be found be accessing the result field of `tcResult`:

```
List<SClassDefinition> classList = tcResult.result;
```

As for `model.isParseCorrect()`, using

```
GeneralCodeGenUtils.hasErrors(tcResult),
```

yields the same result, since it is called internally in the method.

4.2 The New Vdm2Uml Handler

`Vdm2UmlMain` is the name of the new handler for the transformation and it is executed when the corresponding JAR file is run.

The `Vdm2UmlMain` method `main`, takes a set of arguments which denote the directory containing the files to be transformed and the directory where the resulting UML file should be placed. The arguments are distinguished by using `-folder` and `-output` as flags, before the arguments.

After some error handling, ensuring that the arguments are in fact present, the arguments are put into to a string list and a linked list of files is created, called `files`. The list of arguments are then iterated over and the files to be transformed are added to `files` and the output folder is stored as `outputDir`. Error messages are given if the path to the input files are not valid or if the no directory is given after an argument flag.

A print statement then signals to the user that the UML transformation has started. A last check makes sure that `files` are not empty and that the `outputDir` is not null. The `handlePp` method is then called with `files` and `outputDir` as arguments.

The `handlePp` method is used when transforming VDM++ files. It starts by creating a new `Vdm2Uml` class, which is used when performing the conversion and saving the UML file. The class constructor is called with two Boolean preference flags as in `Vdm2UmlCommand` but these are now unused and set to false.

The `TypeCheckResult` instance is then created and checked for parsing errors, as shown in the previous chapter. After the list of class definitions is read from the result field and assign to `classList`. The `convert` is then called as shown in the code snipet below.

```
vdm2uml.convert(projectName, classList);
```

A URI is then generated from the output directory path and the name of the project.

```
URI uri = URI.createFileURI(path.getPath() + "/" + projectName);
```

The the `Vdm2Uml` method named `save` then attempts to save the UML file in the specified directory. Changes where made to the `save` method since there were errors trying to run it in its original state. [16]
From:

```
Resource resource = new ResourceSetImpl()
    .createResource(uri.appendFileExtension(UMLResource.FILE_EXTENSION
    ));
resource.getContents().add(modelWorkingCopy);
```

To:

```
ResourceSet resourceSet = new ResourceSetImpl();
resourceSet.getResourceFactoryRegistry()
  .getExtensionToFactoryMap()
  .put("*",new XMIResourceFactoryImpl());
Resource resource = resourceSet
    .createResource(uri.appendFileExtension(UMLResource.FILE_EXTENSION
    ));

resource.getContents().add(modelWorkingCopy);
```

This is because we are developing a standalone application separate from the Eclipse platform. On Eclipse, registrations of the package, the factory map and such are handled automatically, but in a standalone app, those registrations have to be performed programmatically [3]. A similar change is made in the UML2VDM implementation when loading a UML resource.

4.3 Analysis of the Old Uml2Vdm Handler

`Uml2VdmCommand` is the old handler for the transformation in the direction of UML to VDM and works in a similar way as `Vdm2UmlCommand`. Like `Vdm2UmlCommand` its starts of by getting the selected Eclipse element. However, instead of checking for an `IProject` is checks for an `IFile` since only a single UML file is expected as opposed to multiple files in a project. A URI to the file is then created and a new instance of the `Uml2Vdm` class is created.

```
final Uml2Vdm uml2vdm = new Uml2Vdm();
```

This class contains the method for performing the conversion and its constructor is called without any arguments. The class must therefore be initialized at a later point. But before that, calls to various methods that continually show progress to user are made and the correct extension must be found. The extension is used when the class files are written, when the conversion is performed. To find the correct extension, the following code is executed:

```
IVdmProject p = (IVdmProject) iFile.getProject().getAdapter(
    IVdmProject.class);
String extension = null;
if (p != null)
{
  for (IContentType ct : p.getContentTypeIds())
  {
      if (!ct.getId().contains(".external."))
    {
      extension = ct.getFileSpecs(IContentType.
                                        FILE_EXTENSION_SPEC)[0];
      break;
    }
  }
}
```

If the `IVdmProject p` is null, the extension stays unassigned and therefore set to null.
The `uml2vdm` class is then initialized with the UML file URI and the extension of that file as arguments.

```
if (!uml2vdm.initialize(uri, extension))
  {
    return new Status(IStatus.ERROR, Activator.PLUGIN_ID, "Failed
    importing .uml file. Maybe it doesnt have the EMF UML format");
  }
```

The `initialize` method returns a boolean signaling whether the file was successfully obtained. This is why the method is called the a condition in the `if` statement that contains error handling in its body. The method itself consists of the following code:

```
public boolean initialize(URI uri, String extension)
{
  if (extension != null)
  {
    this.extension = extension;
  }
  Resource resource = new ResourceSetImpl().getResource(uri, true);
  for (EObject c : resource.getContents())
  {
    if (c instanceof Model)
    {
      model = (Model) c;
    }
  }

  return model != null;
}
```

As can be gleaned from the code, if the extension is null, the private attribute of the class of type `String` called `extension` is not changed and the default value of `"vdmpp"` is used.

The `convert` method is then called, with a file containing the path to the new directory which contain the generated VDM files.

```
uml2vdm.convert(new File(iFile.getProject().getLocation().toFile(), "
    uml_import"));
```

The progress is then shown to be completed and the `Vdm2UmlCommand` returns with an `OK_STATUS;` event.

4.4 The New Uml2Vdm Handler

5 Packaging the JAR using Maven

Maven is a project building tool for Java based projects. Using Maven is convenient because it abstracts away a lot of the configuration needed to build projects that other build tools require. In the projects POM file we specify what function the JAR will run and the dependencies needed.

Maven is run using the following command:

```
java -classpath core/uml2/target/uml2-3.0.3-SNAPSHOT-jar-with-
    dependencies.jar:core/uml2/src/lib/* org.overture.core.uml2.
    Vdm2UmlMain -folder modelPath -output outputPath
```

5.1 Running the JAR File

Once the JAR file is build, it can be run as a command in a console. For an early version of our VDM2UML transformation, such a command could look like

```
java -classpath core/uml2/target/uml2-3.0.3-SNAPSHOT-jar-with-
    dependencies.jar:core/uml2/src/lib/* org.overture.core.uml2.
    Vdm2UmlMain -folder modelPath -output outputPath
```

The `-classpath` is to specify a class path where the local dependencies can be found. This is because we had trouble getting these dependencies included in the JAR by default, that is, specifying their inclusion the POM file.

All the JARs in the `\lib` directory are included and the main function is specified. The following flags have their functionality specified in the main function where errors are also caught in case of missing directories or if they are badly typed. For the simplest version of the plugin, the `-folder` flag means that the following string tells where the project files for the model are, and the `-output` flag is followed by the directory where the output UML file is saved. This directory will be created if it does not exist.

A flag for choosing the dialect will be added when more dialects are implemented as well as other options if necessary.

References

[1] Andrea Bondavalli, Sara Bouchenak, and Hermann Kopetz. *Cyber-Physical Systems of Systems: Foundations – A Conceptual Model and Some Derivations: The AMADEOS Legacy.* Vol. 10099. Dec. 2016. ISBN: 978-3-319-47589-9. DOI: `10.1007/978-3-319-47590-5`.

[2] Andrea Bondavalli and Francesco Brancati. "Certifications of Critical Systems - The CECRIS Experience". In: Jan. 2017, pp. 1–316. DOI: `10.13052/rp-9788793519558`.

[3] James Bruck, Kenn Hussey, et al. *MDT/UML2/FAQ.* 2011. URL: `https://wiki.eclipse.org/MDT/UML2/FAQ#What.27s_required_to_load_a_UML_.28.uml.29_resource_from_a_standalone_application.3F`.

[4] Kenn Hussey and Christian W. Damus. *Eclipse UML2 v1.7.0 Git Repository.* 2012 (last checked 2022). URL: `https://github.com/creckord/org.eclipse.uml2`.

[5] jebbs. *Rich PlantUML support for Visual Studio Code.* URL: `https://marketplace.visualstudio.com/items?itemName=jebbs.plantuml`.

[6] Peter Larsen and John Fitzgerald. "The evolution of VDM tools from the 1990s to 2015 and the influence of CAMILA". In: *Journal of Logical and Algebraic Methods in Programming* 85 (Oct. 2015). DOI: `10.1016/j.jlamp.2015.10.001`.

[7] Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. "The Overture Initiative Integrating Tools for VDM". In: *SIGSOFT Softw. Eng. Notes* 35.1 (2010), 1–6. ISSN: 0163-5948. DOI: `10.1145/1668862.1668864`. URL: `https://doi.org/10.1145/1668862.1668864`.

[8] Microsoft. *What is the Language Server Protocol?* URL: `https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/`.

[9] OMG. *About the Unified Modeling Language Specification Version 2.5.1.* 2017. URL: `https://www.omg.org/spec/UML/2.5.1`.

[10] OMG. *Diagram interchange v1.0 (2004); version 1.0, formal/06-04-04.* 2004. URL: `http://www.omg.org/cgi-bin/doc?formal/06-04-04`.

[11] Jonas Kjær Rask, Frederik Palludan Madsen, Nick Battle, Hugo Daniel Macedo, and Peter Gorm Larsen. "The Specification Language Server Protocol: A Proposal for Standardised LSP Extensions". In: Proceedings of the 6th Workshop on *Formal Integrated Development Environment,* Held online, 24-25th May 2021. Ed. by José Proença and Andrei Paskevich. Vol. 338. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, pp. 3–18. DOI: `10.4204/EPTCS.338.3`.

[12] Jonas Kjær Rask, Frederik Palludan Madsen, Nick Battle, Hugo Daniel Macedo, and Peter Gorm Larsen. "Visual Studio Code VDM Support". In: *Proceedings of the 18th International Overture Workshop.* Ed. by John S Fitzgerald and Tomohiro Oda. Overture. 2020, pp. 35–49. URL: `https://arxiv.org/abs/2101.07261`.

[13] Arnaud Roques. *Frequently Asked Questions (FAQ).* URL: `https://plantuml.com/faq`.

[14] Arnaud Roques. *PlantUML Language Reference Guide.* URL: `https://plantuml.com/guide`.

[15] Arnaud Roques. *What's new?* URL: `https://plantuml.com/news`.

[16] Anonymous Eclipse users. *null resource?* last checked 2022. URL: `https://www.eclipse.org/forums/index.php/t/123452/`.