Grant Agreement: 287829

Comprehensive Modelling for Advanced Systems of Systems

COMPASS

Third Release of the COMPASS Tool
Symphony IDE Developer Documentation

Deliverable Number: D31.3b

Version: 1.0

Date: January 2013

Public Document

http://www.compass-research.eu

Contributors:

Joey W. Coleman, Aarhus
Luís Diogo Couto, Aarhus
Anders Kaels Malmos, Aarhus
Richard Payne, Newcastle

Editors:

Joey W. Coleman, AU

Reviewers:

Document History

Ver	Date	Author	Description
0.1	29-10-2013	JWC	Initial document version based on D31.2b
0.2	03-11-2013	LDC	Initial revision of all sections
0.3	08-11-2013	JWC	More revision of all sections
0.4	11-11-2013	LDC	Cleared remaining fixmes.
0.5	11-11-2013	JWC	Prep for internal review
1.0	27-11-2013	JWC	Final version

Contents

1 Introduction

The purpose of this document is to introduce developers unfamiliar with the Symphony IDE[1] to the structure and practicalities of developing extensions to the core functionality of the tool. We cover the basic context in which development happens, the functional and architectural structure of the primary components, the practicalities of configuring a build environment for the tool, and the initial steps required to start a new extension.

Excluded from the scope of this document is general information about software development in the Eclipse environment as that is documented by the Eclipse project on their website.[2] We do, however, provide detail about how the Symphony IDE components fit into the Eclipse framework.

Also excluded from the scope of this document is general information about the Overture tool[3] (see [LBF+10]), the open-source tool for the VDM formal method (see [FLM+05, LFW09]) that uses the Eclipse platform as its basis. As the Symphony IDE builds upon the Overture tool, we do provide details on its integration.

The overall context of development for Symphony is explained in Section 2, including the use and integration of Eclipse and the Overture tool. The structure of the build environment and the steps necessary to set it up are described in Section 3. In Section 4 the structure of the Symphony components is described along with their relation to each other. Section 5 explains the necessary steps to programmatically access the core functionality provided by the Symphony IDE, and gives the necessary configuration templates to start developing plugins and extensions for Symphony.

[1] The Symphony IDE is produced in the COMPASS project, and was previously named the COMPASS tool. It has been rebranded for better continuation of the product after the end of the COMPASS project.

[2] `www.eclipse.org`

[3] `www.overturetool.org`

2 Context

2.1 Eclipse

The Eclipse Platform provides core frameworks and services upon which plug-in extensions can be created. The main purpose of the Platform is to enable other developers to easily build tools. Eclipse is designed to run on multiple operating systems (OSs). This enables plug-ins to be programmed in the Eclipse portable Application Programming Interface (API) and run unchanged on any of the operating systems that Eclipse supports.

The Eclipse architecture supports dynamic discovery, loading and running of plug-ins. The platform handles the logistics of finding and running the right code based on the plug-ins which have been installed. Eclipse is, in itself, only a runtime kernel with basic UI and source navigation. All of its functionality is provided as plug-ins. The plug-in architecture enables developers to contribute their own plug-ins to supply the functionality needed. Plug-ins are structured bundles of code and/or data that contribute functionality to the system. Eclipse provides many libraries that can be used or extended for developing Integrated Development Environments (IDEs). This includes libraries for facilities such as editors, outlines, project explorers and debuggers.

2.1.1 Eclipse, OSGi and Equinox

Conceptually, Eclipse is a combination of two things:

1. An OSGi framework implementation called Equinox.[4]
2. A set of bundles running inside the Equinox OSGi framework.

OSGi is an abbreviation for Open Services Gateway initiative and aims at providing a dynamic component model for Java applications. Most importantly for this context is that programs leveraging an OSGi framework in the way Eclipse does consist only of a set of bundles. Each bundle is a tightly-coupled, dynamically loadable collection of classes, jars, and configuration files that explicitly declare their external dependencies (if any). In Equinox a bundle is a jar-file (may be a fat jar containing other jars) with an extended `MANIFEST.MF` file stating external dependencies and packages offered by the bundle. Bundles in the Equinox OSGi framework are loaded in complete separation from each other, each being loaded in separate Java class-loaders. In this way Equinox has complete control over the run-time classpath that every bundle has. Furthermore, to control and maintain bundles the OSGi framework stipulates that a bundle must have the life-cycle depicted in Figure 1.

2.1.2 Eclipse SDK

Eclipse is more than just an easy platform to develop plug-ins to, it also provides many libraries to help in the creation of IDEs. Functionality can be contributed in the form of code libraries of Java classes with a public API, platform extensions or even as documentation. Plug-ins can define extension points, which are well defined places

[4] See `http://www.osgi.org` and `http://eclipse.org/equinox/`

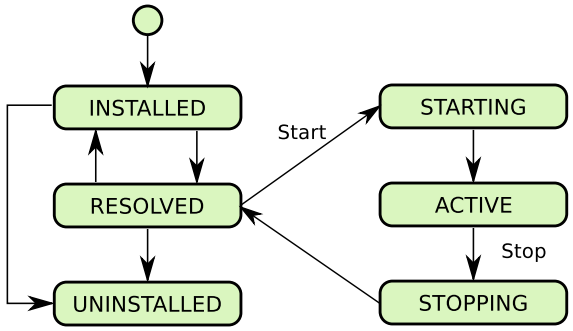

Figure 1: OGSi bundle life-cycle

where other plug-ins can add functionality. These form the base of the framework described in this document.

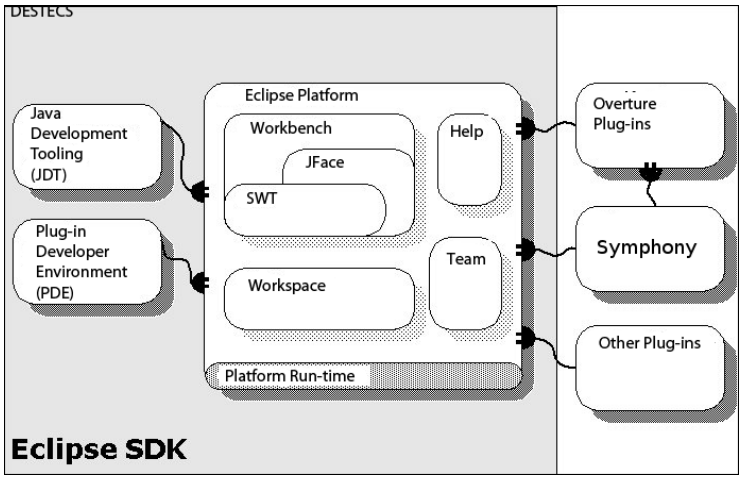

Figure 2: Eclipse SDK in detail

Figure 2 shows the components of the Eclipse SDK. It is important to notice that the Eclipse platform has been extended with the Java Development Tools (JDT) and Plug-in Development Environment (PDE) plug-ins enabling development of new Eclipse plug-ins. The Eclipse SDK is used to develop the Symphony IDE, in the coding of all the plug-ins. However, the complete SDK does not need to be included in the standalone version of Symphony. The standalone only includes the *Eclipse Platform* and the Symphony plug-ins. The Symphony IDE is built on top of the Rich Client Platform (RCP) framework which provides much of the functionality described above. Basically the framework provides plug-ins in the form of a base editor in which the extra functionality needed can be added easily through extension points, and by supplying custom code or configurations such as syntax colouring and so on.

2.1.3 Terminology

This section introduces the Eclipse terminology that will be used throughout the report (see also [MT09]).

Workspace A workspace is the basis for Eclipse platform resource management. There is only one workspace per platform[5] and all the *resources* exist in the context of the workspace. The workspace is present in the computer file system. A *resource* can be a: file, folder, project or the workspace itself.

Perspectives A perspective is a fixed collection of views (for example, editors and project explorers) according to their aim. For example a Java perspective contains a navigator view, a source editor and normally a view showing the errors contained in the source being edited as shown in Figure 3, while a Web Design perspective might have a a different set of views like a page designer, an XML editor, and so on. Only one perspective can be active at a given time.

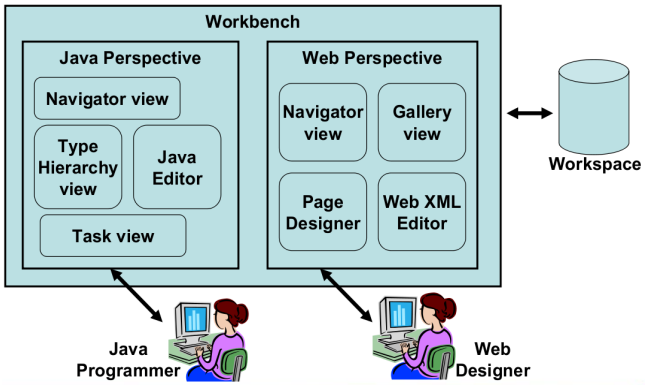

Figure 3: Eclipse Perspective Concept

Views A view often helps the user in the development process. Views do not need to have the open/edit/save behaviour; changes are immediately applied. Often they work together with an editor or show information related to the current selection. All user interfaces for views are organised in the same way, they have their own menu and toolbar.

Editors Editors in Eclipse are user interface elements where the user can modify information depending on the project type. The editor is not just a "text editor": the information that is being edited can be a group of related files, a database entry or similar information where the presentation fits into the traditional model of open/edit/save user interactions. Each editor has a tab where the name of the input that is being edited is shown. It shares the main menu bar and a common toolbar. It is possible to show more than one editor at a time, the position of these editors can be stacked or tiled in

[5]There may be multiple copies of the platform running, however.

the editor area as the user prefers. The modified information is only saved when the user requests it.

Each editor has an associated process called the *reconciler*. This process is activated every time a user types something in the editor and is responsible for updating the IDE with regards to things such as the outline, and the warning and error markers in the editor.

Plug-ins and Extension Points The Eclipse platform consists of layers of plug-ins, each layer defining extensions to extension points of lower layers. Plug-ins are components that provide a certain type of service within the context of Eclipse. Extensions are the central mechanism for contributing behaviour to the platform. Plug-ins can define their own extension points for further customisation.

The `plugin.xml` file contains the extension points that a plug-in provides and the extensions a plug-in provides functionality for. One example of an extension point is *org.eclipse.ui.editors* provided by the *org.eclipse.ui*-plug-in. To provide functionality for this extension point a plug-in must declare an extension for this extension point in its `plugin.xml` file:

```
<extension point="org.eclipse.ui.editors">
  <editor
    class="eu.compassresearch.ide.ui.editor.core.CmlEditor">
    [...]
  </editor>
</extension>
```

The extension in the `plugin.xml` file stipulates which point it extends and points to a java-class in the plugin-bundle that possibly implements an interface (IEditorPart) associated with the given extension point. In this way Eclipse plugins can leverage functionality from the Eclipse platform and other third party plugins to provide new functionality. The Symphony IDE implements a set of extensions to provide Symphony features in Eclipse.

Projects, Builders and Natures A *project* is a group of resources that are related. Each project has a project description, which defines a set of natures, builders and projects that this project references. A *nature* classifies the type of the project such as Java or CML, and binds behaviour and functionality with the project. The nature often tells which builder to use with a particular project and can even determine what the user interface will look like, which icons should be used, and other user interface details.

A *builder* is a mechanism that allows tool-specific logic to process changed files at specific times, this is often the transformation of resources from one form to another. The Java-builder transforms source files to binary class files using the Java-compiler.

2.2 The Overture Platform

The Overture Platform is a set of components that, together, form a VDM development environment based on the the Eclipse platform [CMNL12]. The set of Overture components can be split mainly into two groupings: the core libraries and the Eclipse IDE libraries. The core libraries contain all that is needed to read and manipulate VDM models in a programmatic fashion, and the IDE libraries access these to create an IDE in the usual manner, based in Eclipse. There is also a smaller, third set group of components that are meta-components; these include libraries that aid the build process. The most critical of these is ASTCreator, and is mentioned later in this document.

The Symphony IDE makes heavy reuse of the Overture components, including the ASTCreator, the VDM typechecker, the VDM proof obligation generator, and some of the IDE components. Among other benefits from reuse, this allows for fixes in the Overture platform to be automatically incorporated into Symphony as they become available. The Symphony IDE is the first deep extension of the Overture platform, and our efforts to build Symphony also have the effect of improving the Overture platform.

3 Build Environment

This section assumes that all of necessary tools –such as Git, Maven, Eclipse, etc.– have been correctly installed and configured. Also, `PROJ` refers to the root directory of the COMPASS/Symphony code repository.

3.1 Distributed Version Control using Git

The version control system used by the COMPASS project for the source code of the Symphony IDE is called Git, and documentation on its usage is freely available on the Git website.[6] We recommend the first few chapters of the book *Pro Git* [Cha09] as an introduction to using Git; the book is published by Apress, but is also available online. Hosting for the COMPASS/Symphony Git repository is provided by SourceForge.net, and read access is open to everyone.

Write access to the repository is controlled via the SourceForge.net project permission structure, and developers must first contact the Theme 3 lead to be added as a contributor. For project members, this access is always granted; developers outside the project are required to indicate their reasons for joining as a contributor.

Since development on the Symphony IDE is done by many developers across several of the project sites, we have adopted certain conventions regarding the use of the Git repository.

The basic development process for the Symphony IDE relies on the branching capabilities of Git to track multiple streams of development. We keep to a convention of using three specially-named branches, and two categories of branch that follow a particular type of naming pattern.

The two special branches are:

`master` The master branch is used to track releases of the Symphony IDE. A developer who clones the Git repository and checks out the master branch will be able to compile the latest release of the Symphony IDE. The only person, by convention, who makes changes to this branch is the designated release manager.

`development` The development branch is used to track the latest compilable features under development. A developer who checks this branch out should always be able to cleanly compile this branch, though it may have serious bugs in the resulting functionality. It is preferred that any changes to this branch be made with the release manager's knowledge ahead of time.

`test` The test branch is a looser version of the development branch. Like the development branch, the build server runs build jobs based off test. Unlike the development branch, test is not supervised by the release manager. The code on test should always build (again, the build server monitors this branch) so anyone pushing to it must take care to properly merge their code. On the other hand, there is no guarantee that your work will be preserved on this branch so do not use it as a primary work branch.

The other categories of branch are:

[6] See `http://git-scm.org`

feature Feature branches are named based on a particular feature or bug that they are intended to address, and may have multiple developers working on that branch. There should be a developer responsible for the maintaining the branch until it is ready to be merged into the development branch, and the release manager should be aware of who this is. One such feature branch, present at the time of writing, is for notification by plugins of unsupported CML constructs in the current model; the branch is named `uecollector`. As it may be that multiple developers are working on a specific feature, it may be that branches named according to the feature and individual developer's initials exist such as `uecollector-ldc`.

initialled Branches that are named starting with the initials of a developer's name are the responsibility of that developer, and carry no constraint as to what may be done in those branches. Examples include `jwc` for a branch worked on by JWC. There is no guarantee that these branches are stable or even that they they will build. On the other hand, it is considered bad form to commit code on another person's branch without their permission.

It is expected that individual developers will monitor the `development` branch and keep their branch up to date with respect to it. In all cases a merge from development into the initialled branch should happen before a request is made to the release manager to merge work from the initialled branch back into `development`. Ideally, developers should always merge with development prior and after working on their own branches to ensure they are always in synch.

This structure, and the fact that branch management in Git is well-supported and easy to use, has a significant side-effect: developers have the freedom to easily experiment and either merge the experiment into the rest of their work or simply throw it away. Furthermore, the merge functionality that Git provides allows developers to reconcile their changes against the rest of the developers in a relatively easy way.

3.2 Requirements

The three tools used for setting up the development environment for the Symphony IDE are Git, Maven and Eclipse. We assume users have some familiarity with all three of them.

Symphony requires the following tools and versions:

- Java 1.7 — make sure that `JAVA_HOME` is pointing to your JDK installation.
- Maven 3
- Git
- Eclipse 4 (4.3 recommended)

3.3 Checking out and maintaining the code

The COMPASS Git repository for the Symphony IDE is at

```
ssh://<sfusername>@git.code.sf.net/p/compassresearch/code
```

You can access the code by cloning the repository with:

```
git clone <git repository> <target directory>
```

Once you have cloned the repository, you should check out the branch you want. To check out (and track a branch) use:

```
git checkout <branch name>
```

Once you have your repository up and running, you should use:

```
git fetch origin
```

```
git merge origin/<branch name>
```

to keep your repository up to date with the latest changes. In addition you should keep yourself synchronized with the main development branch:

```
git merge origin/development
```

By keeping your personal work branches synchronized with development, it is much easier for the release manager to pull in your work for each new release of the tool.

3.4 Building with Maven

Symphony is set up for a two stage build. The core modules are built separately from the IDE ones.

To build Symphony, go to the project root folder and use:

```
mvn install
```

This should fetch all dependencies and build all the core modules. The first run might take quite a while since there are quite a few dependencies to fetch.

After you have successfully built the core modules, change into the `ide` folder and build again:

```
cd ide
```

```
mvn install
```

The first run of the IDE build process is also quite lengthy (there are even more dependencies to fetch).

For convenience, there is a special version of the build command that runs both steps in a single operation. From the main project directory, run:

```
mvn install -PWith-IDE
```

and Maven will build first the core and then the IDE.

Once you have built the IDE, you can grab a standalone executable from the following folder: `PROJ/ide/product/target/products/`. Please note that, by default, Maven will only build the IDE for your current platform (Windows or Linux or MacOS). If you want to build all versions of the tool, use the following command (again, from the IDE folder):

```
mvn install -Pall-platforms
```

Last, but not least, is the clean command. When you are rebuilding the project after some changes to the codebase, you will sometimes need to clean out some project folders (such as generated code and local dependency jars). To do this, use the following command:

```
mvn clean
```

Finally, a possible possible cause for build failures are test failures. At the production level, this is as it should be though there should be no failures from the `master` branch in a correctly configured build environment. But in other branches, if you need to build the code regardless of the test status, you can use the following command:

```
mvn -Dmaven.test.skip=true install
```

```
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] The Symphony IDE Top-Level ....................... SUCCESS [0.351s]
[INFO] Symphony Tools Top-Level ......................... SUCCESS [0.016s]
[INFO] Symphony Core Top-Level .......................... SUCCESS [0.011s]
[INFO] Symphony Core CML Abstract Syntax Tree ........... SUCCESS [11.469s]
[INFO] Symphony Core CML Parser ......................... SUCCESS [3.160s]
[INFO] Symphony Core Common ............................. SUCCESS [0.520s]
[INFO] Symphony Core CML TypeChecker .................... SUCCESS [10.483s]
[INFO] Symphony Core Analysis Top-level ................. SUCCESS [0.007s]
[INFO] Symphony Core Analysis Proof Obligation Generator . SUCCESS [12.043s]
[INFO] Symphony Core Analysis Theorem Prover ............ SUCCESS [2.552s]
[INFO] Symphony Core Analysis RTT-MBT Interface ......... SUCCESS [0.120s]
[INFO] Symphony Core Analysis Model Checker ............. SUCCESS [1.638s]
[INFO] Symphony Core CML Interpreter .................... SUCCESS [8.420s]
[INFO] Symphony Core CML Command Line Tool .............. SUCCESS [3.675s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 55.037s
[INFO] Finished at: Fri Nov 08 13:08:40 CET 2013
[INFO] Final Memory: 36M/99M
[INFO] ------------------------------------------------------------------------
```

Figure 4: An example successful maven build for the core modules

3.5 Importing into Eclipse

To import the project into Eclipse, you need the m2e plugin.

1. In Eclipse go to *Help* → *Install new software*
2. The update site for the plugin is:
   `http://download.eclipse.org/technology/m2e/releases`
   (see Figure 5).
3. Import Maven projects into Eclipse: *File* → *Import* → *Maven* → *Existing Maven Projects*. Do not import any root projects as they are empty (see Figure 6).

   The minimal set of projects to import is:

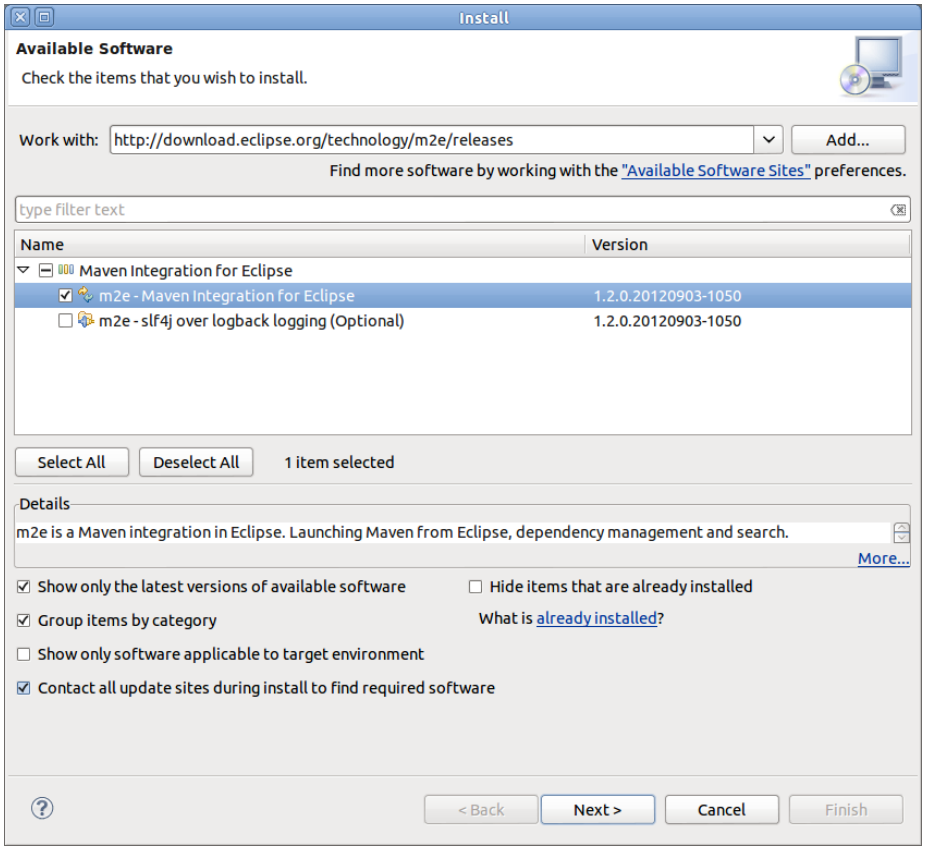

Figure 5: Installing the m2e Eclipse plugin

- From the core projects: ast, parser, typechecker, common
- From the IDE projects: core, ui, platform, product

4. During the import, you may need to install additional m2e Connectors. Eclipse will prompt you to so (see Figure 7).

3.6 Special Eclipse Dependencies

The Eclipse dependencies that the IDE plugins require are handled by the Maven Tycho plugin[7] However, Tycho detaches some of the dependency solving between Eclipse and Maven. This mean that Eclipse-based dependencies are actually solved by Eclipse so you need additional plug-ins in your Eclipse installation.

They are installed in the same way as the m2e plug-in (*Help* → *Install new software*). The required plug-ins and update sites are:

- Overture (mandatory):
  `http://build.overturetool.org/builds/overture-compass-p2repo/`

[7]This is the maven plugin for building Eclipse-based software, available at `http://eclipse.org/tycho/`.

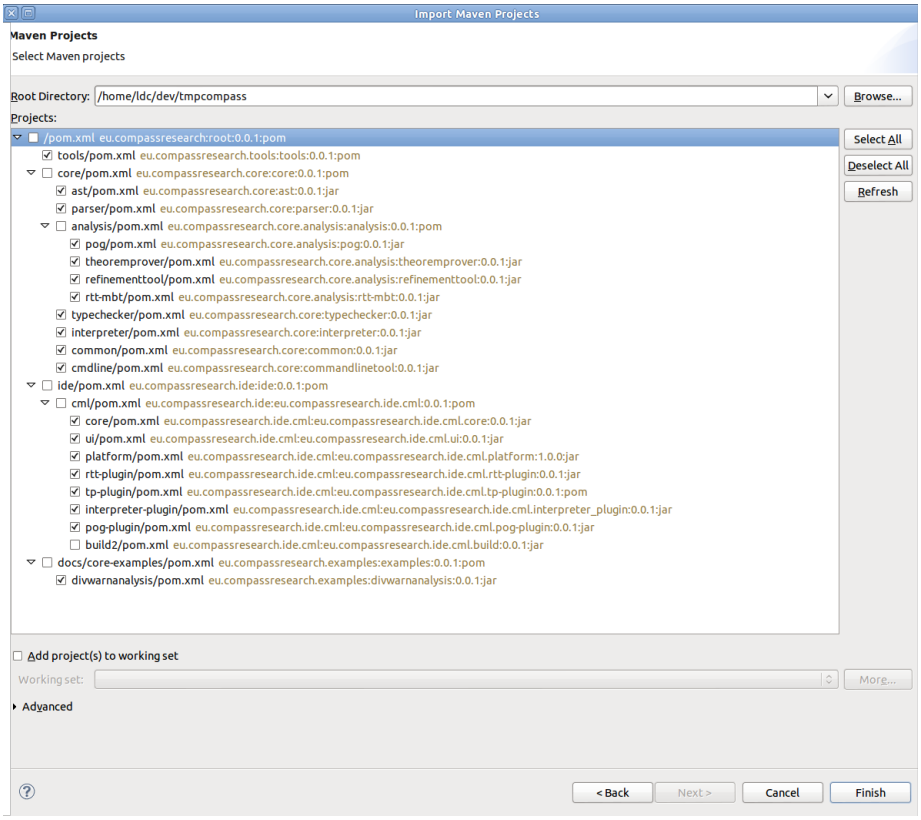

Figure 6: Maven projects to import into Eclipse

- Scala IDE (needed for Theorem Prover Core Module):
  `http://download.scala-ide.org/sdk/e38/scala210/stable/site`
- Isabelle-Eclipse (needed for the Theorem Prover IDE Plugin):
  `http://andriusvelykis.github.io/isabelle-eclipse/updates/isabelle2013/releases/`

An important note regarding the Overture plug-in: Sometimes, Symphony requires that changes be made to the Overture code. While Maven will update and fetch the dependency, Eclipse might not. In this case, you must manually update the Overture plug-in (*Help* → *Check for Updates*). AU will do it best to notify all developers of when such an update might be needed but in general it's a good idea to update the Overture plug-in regularly.

An alternative to installing these plug-ins is to check out their code and have their projects open in the same Eclipse workspace as Symphony. Keep in mind though that this does not apply to the Scala IDE. We don't recommend this unless it becomes necessary; contact the COMPASS projects members at AU if help is required to set this up.

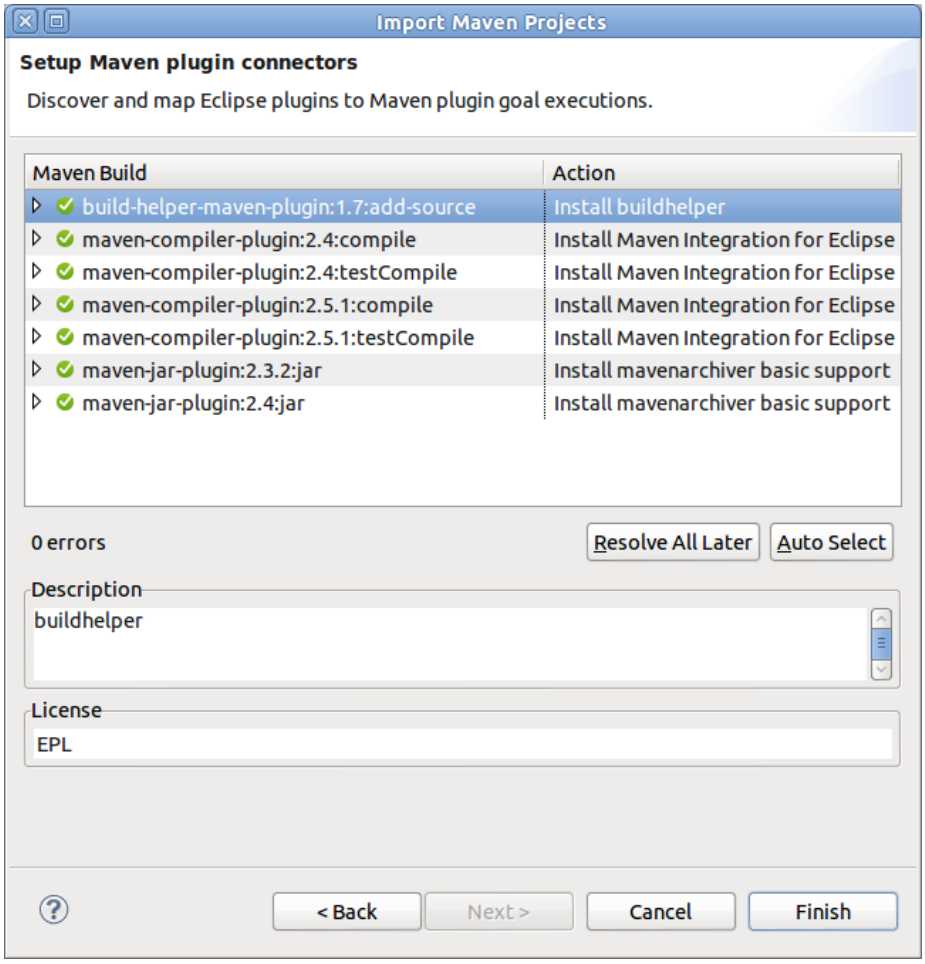

Figure 7: Eclipse prompts the user to install additional m2e Connectors

3.7 Running in Tethered Environment

In order to run Symphony in tethered mode (under the Eclipse debugger), follow these steps:

1. Go to *Run* → *Run Configurations*
2. Create a new Eclipse Application configuration
3. Choose *Run a product* and select `eu.compassresearch.ide.platform.product`
4. In the *Plugins* tab make sure that you tick *Validate plug-ins automatically prior to launching*.
5. You can also use this tab to customize which plug-ins you actually want to run, though we recommend selecting *all workspace and enabled target plug-ins*.

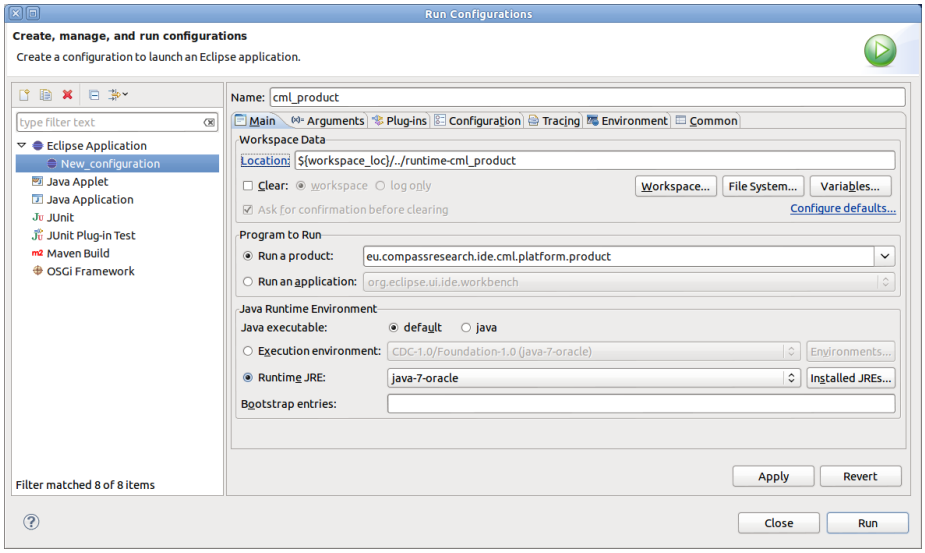

Figure 8: A new run configuration for the Symphony IDE

3.8 Troubleshooting the Development Environment

This section lists common build problems and their solutions.

3.8.1 Artifact has not been packaged yet

This error will show up in Eclipse after importing maven projects or doing *Maven* → *Update Project* from the context menu. To solve it, run maven update project again. You will get the jar error from Section 3.8.5. Click ok. You will get a series of type errors (package does not exist, etc.). Run maven update project again but this time untick *Clean projects*.

3.8.2 Scala Classes in Eclipse

This error occurs when Eclipse (with the Scala plug-in) is unable to recognize Scala code. Scala classes referenced in Java code will give type errors. The problem can be traced to the plug-in version of the Scala IDE which is known to have some issues. Multiple developers have had problems getting it to even run on Eclipse 4.

It is recommended that developers doing Scala work use the standalone IDE instead.[8] This also has the nice side-benefit of giving you separate and cleaner work environments.

[8] Available at `http://scala-ide.org/download/sdk.html`

3.8.3 Project is out of synch errors in Eclipse

This error usually happens after updating the code base in Git and getting some changes to POM. In the Context menu select *Maven* → *Update project*.

If you are still getting errors, clean and rebuild in Maven, then refresh all projects in Eclipse and run the Maven update again. Finally, clean all projects in Eclipse.

3.8.4 Import or bundle not resolved errors in eclipse

This error usually means that you have some unmet dependencies because you have not imported all the necessary projects. In addition, to the minimal set, some projects depend on others (for example, the TP plug-in depends on the POG plug-in). Check the pom.xml files in each of your projects with errors and make sure that you have imported all of their dependencies.

Alternatively, you may need to install some of the Eclipse plug-ins mentioned in Section 3.6.

3.8.5 Jar is not on its project's build path

This error will pop up sometimes (see Figure 9) while doing a Maven update in Eclipse. Its cause is linked to the two stage build process we use and an inability in the m2e plug-in to determine whether or not the core modules have been built. You can simply ignore and run the Maven update again. Sometimes you may have to force it 2 or 3 times. If it persists beyond that, contact the AU developers for help.

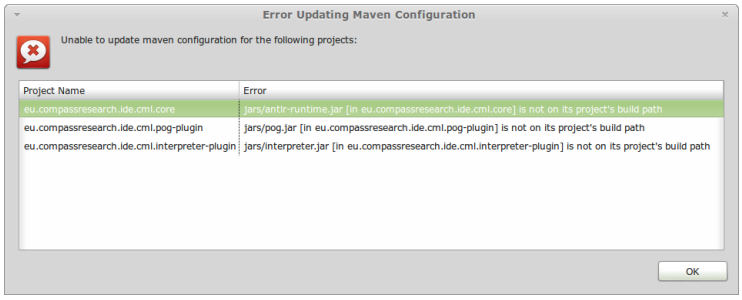

Figure 9: The "jar is not in its project's build path" error

3.8.6 Problems with Eclipse source folders

Maven has a standard structure for source code folders (`src/main/java` and etc.) and Eclipse/m2e can pick it up quite easily. But some projects (namely the interpreter) use non standard source folder structures. Eclipse is not terribly consistent in picking up these folders.

This can manifest in internal project errors (type not found, etc.). To solve this problem, go to *project* → *Properties* → *Java build path* → *Source*. Remove all the source fold-

ers and then re-add them (Eclipse will automatically set the proper nesting of source folders).

3.8.7 Scala Errors while building with Maven

The Maven Scala plug-in is not as reliable as one would like. It will, on (a very rare) occasion, throw up various compilation errors. These errors will usually be around the Java classes being used in the Scala code.

This error occurs frequently if you run the following sequence of commands:

1. `mvn clean`
2. `cd ide`
3. `mvn clean`
4. `cd ..`
5. `mvn -PWith-IDE`

The build will fail on the `Isabelle IDE Plugin`. We have not yet identified the cause of this problem. But we have found a fairly reliable workaround. Simply run `mvn -PWith-IDE` again and the project will build past the error successfully.

3.8.8 Cleaning out the local Maven repository

Sometimes, your local repository may become corrupted and Maven will be unable to properly solve dependencies. Usually, this occurs when Maven must update a dependency but refuses to do so and keeps using the one in the local repository, which can cause various errors when building with Maven.

In theory this, should not happen but when you have errors that you cannot trace of fix, it's worth deleting your local maven repository. To do so, delete the following directory:

- On Linux/Mac: `~/.m2/repository/`
- On Windows: `C:\Users\<username>\.m2\repository` or `C:\DocumentsandSettings\<username>\.m2\repository`

The next invocation of maven will repopulate the directory with the necessary files.

4 Functional Structure

Figure 10 gives an overview of the Symphony repository and its main projects, based on the repository's directory structure. The core components are shown in blue boxes, and the IDE components are in green boxes. The rest of this section will explain the various modules and identify their locations in this diagram.

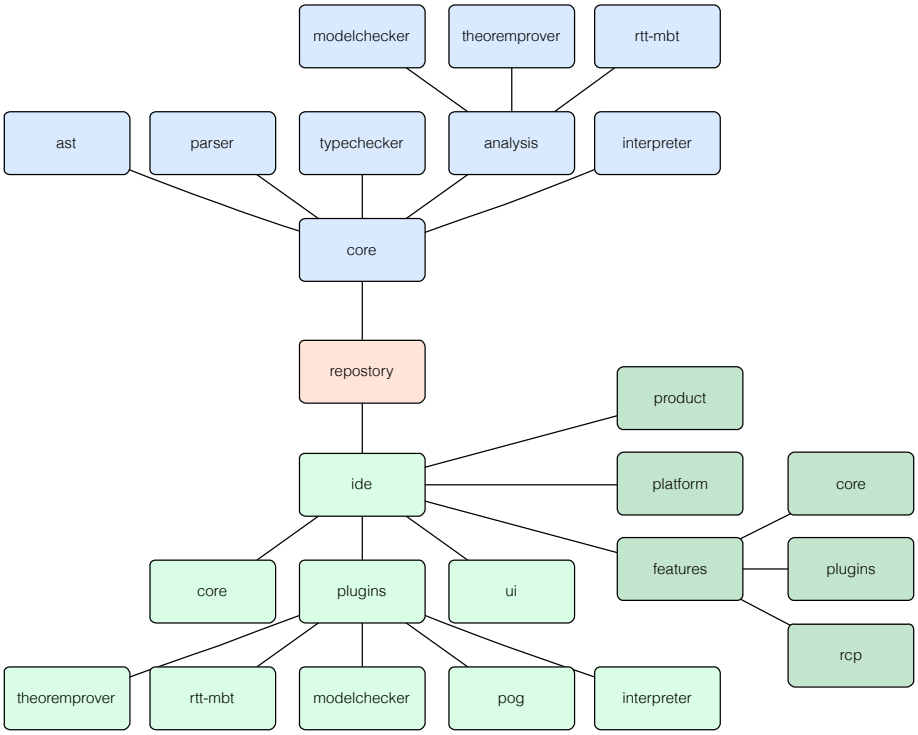

Figure 10: Main projects in the Symphony repository

4.1 Core

The core of the Symphony IDE is the basic elements of the toolset that are idependent of any particular user interface. As their name suggests, they provide only the core functionality of loading, validating, and analysing CML files. The Symphony core is responsible for the handling of CML files from parsing them to build an internal representation –the AST– and then typechecking that to ensure its overall consistency. This AST is then used by the various Symphony core plugins and the CML interpreter.

4.1.1 The CML AST

The AST represents the CML model that is being worked on. The code for the AST is automatically generated by the ASTCreator tool and as such should not be modified.

The main interaction most developers will have with the AST will be done through its visitors. There are existing AST visitors that can be subclassed for any form of analysis that is needed. The AST is found in the blue box labelled “ast” in Figure 10.

As a part of the Overture platform there is a tool called ASTCreator that is used to automatically generate the AST for all of the VDM dialects supported by Overture. This tool is also used to generate the AST for the CML dialect via its extension feature, and it uses the VDM AST as its basis for the CML AST. This usage allows us to directly reuse much of the interpreter and typechecker source code from the Overture platform in the CML tool without needing to change the VDM typechecker or interpreter.

The structure of a dialect is defined in an AST script that is read in by the ASTCreator tool. The ASTCreator tool then resolves the necessary structure of the target AST from this script and generates the Java class files that implement the AST, and in addition it provides a set of Java class files that implement the basic AST visitors within the correct Java packages. These visitors are used to manipulate the ASTs and should be used by plug-in developers to implement additional functionalities on the ASTs.

Note that ASTCreator may also be used by plugin developers to create ASTs for other grammars that are translated to or from the CML AST.

4.1.2 Parser

The parser is responsible for reading in CML files and building the raw representation of the contained model as an AST. It was built using the ANTLR parser generator, however, most of the error recovery features are presently disabled in favour of a fail-on-first-error approach. We will re-enable and elaborate upon the error recovery features as the project progresses beyond M26. The parser is found in the blue box labelled “parser” in Figure 10.

4.1.3 Typechecker

The typechecker is the second phase in processing CML models. It takes the untyped AST produced by the parser and determines the types of the various nodes. The typechecker is built mostly through AST visitors and reuses the Overture typechecker whenever possible. Specifically, most of the VDM-derived elements are type checked using the Overture typechecker. The typechecker is found in the blue box labelled “typechecker” in Figure 10.

4.1.4 Interpreter

The CML interpreter, like the analysis plugins described in the following section, allows for CML models to be simulated and animated. It is a toplevel module –the corresponding box is the blue on labelled “interpreter”– rather than an analysis plugin as the interpreter represents major functionality in its own right.

4.2 Analysis Plugins

The Symphony IDE was designed as a plugin-based architecture. Besides the core modules, each of the tool's major features are provided by separate independent plugins. Each plugin connects to the Symphony IDE and the core by using the AST (through its visitors) and the registry.

One of the advantages of this approach was that it allows for the development of each plugin (and individual features) to be independent. Thus, the various plugins can be implemented in parallel, at different sites within the project. This also provides the structure for future extension of the tool. Since the CML language has a complex structure, a tool set with a rich feature-set is likely to have a large codebase; this further reinforces the need for such an approach to the design.

There are presently three analysis plugins: the Proof Obligation Generator ("pog" in Figure 10), the theorem prover core ("theoremprover"), and the RT-Tester core ("rtt-mbt"). We describe the structure of the POG as an example of how plugins may be constructed for the Symphony IDE.

4.2.1 Proof Obligation Generator

Core Module

The Proof Obligation Generator (POG) is a component in the Symphony core analysis libraries, bundled in the `eu.compassresearch.core.analysis.pog` package. The POG makes heavy reuse of the Overture POG, to handle various CML syntactic elements inherited from VDM.

Structure

The POG is based around a collection of classes extending the QuestionAnswerCMLAdaptor. The POG visitors generate IProofObligationList objects, containing a number of IProofObligations. When generating an IProofObligation object, an IPOContextStack (a stack of nested of IPOContexts) is used. This objects contains information to recreate the context required for the PO expression. This is depicted in Figure 11.

The reused Overture `org.overture.pog.obligations` package contains two hierarchies of classes extending the POContext and ProofObligation classes: each proof obligation type has a corresponding ProofObligation class, and each place where context information must be resolved has a corresponding POContext class.

Additional POG visitors and Proof Obligation classes are defined to support the reuse of the Overture POG, and also to handle the new CML syntax, not supported by Overture.

It is also worth mentioning how the PO predicates are represented. This is done using a subset of the CML AST. The predicate, in the form of a PExp can be accessed through the valuetree field of the ProofObligation class. By reusing the AST for representing PO predicates, significant advantages can be gained in terms of integration with other Symphony plugins (particularly the Theorem Prover).

Behaviour

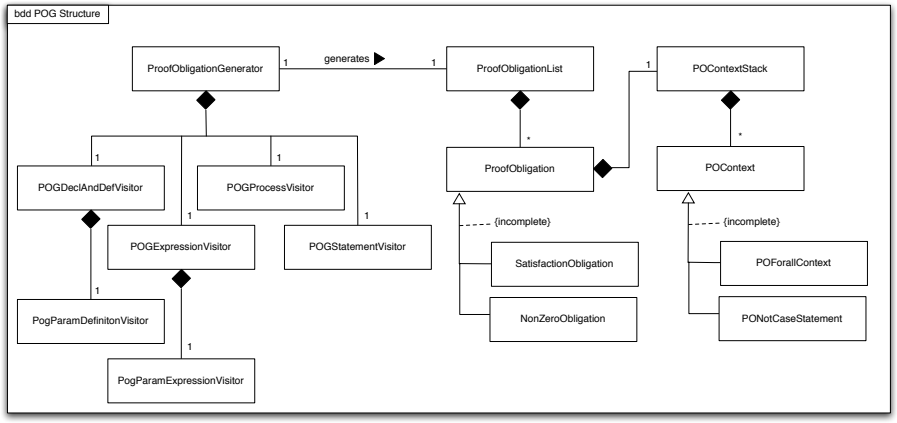

Figure 11: Block definition diagram depicting the main elements of the proof obligation generator

The flow of execution is as follows:

Initialisation Constructor initialises sub checkers – for expressions, declarations and definitions, statements, actions and processes.

Paragraph Splitting The POG iterates through each top level CML declaration, dispatching to the relevant sub checker. As these will be definitions, the POGDeclAndDefnVisitor checker will process the node initially.

Paragraph Processing The POGDeclAndDefnVisitor obtains each CML definition (types, functions, channels, etc.), splits it into its constituent parts (for example, a ATypesParagraphDefinition is split into a number of ATypesDefinition objects), which are passed either to the Overture POG (when the constituent object is a VDM element) or back to the parent POG.

Subsequent Node Processing Here, the parent POG iteratively dispatches each child node to the relevant visitor, which in turn processes the node and returns the output to the parent. Where required, context information is added to the POContextStack - for example, in a function, the parameters are added to the stack so that any subsequent POs may refer to the function parameters.

IDE Plugin

The POG IDE Plugin is responsible for wrapping the core module and exposing its functionality to the user. It is also responsible for providing a UI for generation and inspection of Proof Obligations. Finally, it exposes the POG's functionality to the remaining IDE plugins.

Classes

The POG Plugin is built around a few small classes. The most relevant ones are *PogPluginRunner* and *PogPluginUtils*. Runner holds the main POG functionality (run the POG on a project). Utils holds a series of utility methods for the POG such as opening the POPerspective and getting all the CML files in a project. Runner is designed to be instantiated whereas Utils is set up as a series of static methods.

The only way to run the POG at the moment is by instantiation of the Runner class and invocation of the runPog method. This will execute the POG on whatever project is currently selected in the CmlNavigator. At the moment the POG will always analyze the entire CML project. A more fine grained approach (where it's possible to run the analysis on a single file) is planned for the future.

Also of note is *POConstants* which holds the id values for various views and other elements of the POG Plugin. It is better to centralize these ids as static string constants in a class than to have them spread around the code. The remaining classes are explained in the following sections (note that *Activator* is a default class for all OSGi bundles and can be ignored).

Commands

The POG Plugin offers only one command to users: GeneratePOs. This command is associated with the CmlNavigator's context menu via the `plugin.xml` file. The handler for the command is *GeneratePOsHandler*. It simply initializes the *PogPluginRunner* class and executes the *runPog* method. Because the command is only available when a CML project is selected in the navigator, there is no need to control its availability. Also, before executing the POG, the Runner will call the parser and typechecker on the model (via *CmlProjectUtil.typeCheck*) and will not proceed if there are any errors in the model.

Views and Perspectives

The POG plugin provides a single perspective (*pog-perspective*) which is defined through the `plugin.xml` file. This perspective simply adds two views to the normal workbench.

The POListView implements a table listing all POs. It is implemented as a subclass of the Overture version of the same view. The content of the view can be set with the *setDataList* method which is prepared for receiving a *IProofObligationList* and a *ICmlProject*.

The PODetailView implements a basic text viewer which shows the predicate for the PO currently selected in POListView. It is also implemented as a subclass of the respective Overture view.

4.3 User Interface

The structure of the user interface consists of two main groupings under the "ide" box in Figure 10. The lighter green boxes represent the main code for the IDE, while the darker green boxes correspond to the necessary build structure required to use Eclipse as a tool platform.

Much like the functionality of the Symphony IDE was split into core and plugins, the interface with the tool is also split into two types of elements. All of the core functionality is implemented in "pure Java classes, without any interface concerns. Most of these are subpackages of `eu.compassresearch.core`.

All of the user interface code (the various Eclipse IDE plugins) is packaged under `eu.compassresearch.ide`. These Eclipse plugins should hold all user interaction code and provide controls over the actual functionality packages. For example,

the Proof Obligation Generator (POG) is split into two sets of packages. The main code is under `eu.compassresearch.core.analysis.pog`. The UI code on the other hand is under `eu.compassresearch.ide.pog`. The UI plugin should connect into the main Symphony Eclipse IDE while also using the core code. In this way, we get a complete separation between both concepts and it should be possible to simply change the interface from Eclipse to some other IDE platform should it become necessary by building only another UI plugin.

There is however one exception to separation of concerns and that is the command line tool. The command line tool is housed in `eu.compassresearch.core` even though it is user interface code. The command line tool exists to enable the core Symphony IDE to export its functionality in some way. If there is a need to use Symphony without the Eclipse IDE (perhaps as an RCP) then the command-line will allow for the basic functionality of the tool to be made available. Otherwise, the only way to use the core functionality would be as Java packages imported into another project. In addition the command line tool code itself is quite simple and small and does not pose any major maintenance concerns.

It is be worth noting that there may be some confusion with the usage of the terms "core" and "plugin" since they they have two separate meanings depending on what we are talking about. In Section 4, the terms were used to refer to functionalities in terms of concepts. Simply put, they were dividing the various functionalities into either core functionalities (required by all others) and plugin functionalities (independent from each other).

On the other hand, when we use the terms core and plugin, or better yet UI plugin, we are not talking about the functionalities of Symphony as a whole but of a single one. Each functionality of Symphony will have a core component (which actually implements the functionality) and a UI plugin component (which implements the interface). It is important not to get these two uses of the terms confused (for example, the Symphony core functionality has a core component).

5 Development Templates

5.1 Libraries

This section will describe the various packages that compose each of the core functionalities of the Symphony IDE focusing on how to use them with your own plugin.

5.1.1 AST

The AST is one of the larger libraries in the tool. Most of the code of the AST is a series of classes representing the various nodes that comprise the AST. These classes are automatically generated and as such should never be edited. The AST itself is defined in the `cml.ast` file, therefore any changes to the AST should be done in `cml.ast` file and a new set of Java files should be generated. That said, changes to the AST should never be made without discussion and consent of the other developers.

Most interactions with the AST should be done with visitors. Their usage is relatively simple. Every Node in the AST contains an `apply` method. One simply needs to declare the visitor and pass it as argument to the apply method. As for creating a visitor, this can be done by extending one of the existing versions. Note that these visitors offer no existing functionality in terms traversing the AST (or anything else). It is up to the developer to implement/override for all the cases he needs.

AnalysisCMLAdaptor The most basic of all visitors. It has a specific case to be applied to each node in the ast. It also has a few default cases that can be applied to multiple nodes.

AnswerCMLAdaptor Similar to the previous visitor but the `apply` method returns a class (that can be parametrized). This visitor is useful when we need to extract and return information in most cases.

QuestionCMLAdaptor Similar to the basic visitor but the `apply` method receives a second argument (the question - which can be parameterized). This visitor is useful when each case needs some contextual information to do its job.

QuestionAnswerCMLAdaptor A combination of the two previous visitors. It is useful when both contextual information and results are needed for each case.

DepthFirstAnalysisCMLAdaptor A special visitor that implements a depth first traversal of the AST. Useful when one needs to go through the entire AST.

5.1.2 IDE Utility

The Symphony IDE has a series of classes that provide mapping between CML files and ATSs and other utilities. They can be used by the various plug-ins to get selected projects and their respective ASTs. Developers are envouraged to familiarize themselves with these classes and their documentaiton. Some of the main classes and functionalities are:

ICMLProject Represents a CML project in Symphony. It provides a mapping a representation of the project and all its files. Particularly relevant is a method to get the model represented by the project: `getModel()`.

ICMLModel Internal representation of a CML model. It has methods to ensure it has been properly type checked and, particularly, to access the AST: `getAST()`.

CMLProjectUtil an abstract class with static methods exposing useful functionalities.

Adapters Eclipse makes extensive use of the adapter pattern to transition between the various project representations. The most important adapter is from a generic Eclipse project to a CML project:

```
ICmlProject cmlProj = (ICmlProject) proj.getAdapter(
    ICmlProject.class);
```

However, other useful adapters also exist. They are listed in the `plugin.xml` file for the Symphony IDE core project.

5.2 Core Plugins

This section will take you through the various steps necessary to build your own Symphony plugin and integrate it into the main toolset. It is assumed that you already have your plugin as an existing Java project in Eclipse.

5.2.1 Initial Maven Setup

The first thing to do is to create the POM file for your project. This will allow Maven to build it. A complete POM example (the POG) can be found in Section A.1. The main elements of a POM are:

parent group: this identifies the super package where your plugin will go.

artifactID: this identifies the main package of your plugin

dependencies: one of the more important groups. You will typically want the core Symphony libraries and possibly the Overture ones as well. You should also add anything else that your plugin may require. See Listing 1 for an example

build: this entry allows you to fine-tune how maven builds your project. For example, you can configure it to ignore test failures and keep building.

```
<dependency>
  <!-- the group id of the dependency, from its own POM -->
  <groupId> </groupId>
  <!-- the id of the dependency as specified in its own POM -->
  <artifactId> </artifactId>
  <!-- the specific version of the dependency;
       variables such as ${project.version} are allowed -->
  <version> </version>
</dependency>
```

Listing 1: A maven dependency entry.

5.3 Eclipse Plugins

If you want to add a user interface to your plugin, it is highly recommended that you follow the standard separation of concerns and build its own UI Plugin. This will be a separate Eclipse project with its own set of configuration files.
The POM for a UI Plugin is similar to the POM for its core component but there are a few additional considerations. In particular, as dependency management is handled with Tycho, you have to provide the update sites to any extra dependencies of your plugin. The main IDE pom already provides update sites for Eclipse and a few other dependencies. But if your plugin has additional ones, you must add them. Check the example for the POG UI plugin in Section A.2 for more details. If there is no update site, then the dependency should probably not be handled through the UI plugin but rather the core module (as a normal Maven dependency).
In addition to the update sites, you will also want to configure the build rules for the UI plugin to ensure the core component is added as a library. To do this, add the follow entry to the UI Plugin POM:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-dependency-plugin</artifactId>
      <executions>
        <execution>
          <id>copy</id>
          <phase>package</phase>
          <goals>
            <goal>copy</goal>
          </goals>
          <configuration>
            <artifactItems>
             <!-- Your plugin's core component  -->
              <artifactItem>
                <groupId> </groupId>
                <artifactId> </artifactId>
                <version>${project.version}</version>
                <type>jar</type>
                <overWrite>true</overWrite>
              </artifactItem>
            </artifactItems>
            <outputDirectory>lib</outputDirectory>
```

```
            <overWriteReleases>true</overWriteReleases>
            <overWriteSnapshots>true</overWriteSnapshots>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

This will ensure that your core component gets packaged in a jar and placed in the lib folder of your UI Plugin so that it can be used. This does mean that you will have to build through maven every time you want a change in the core component to be reflected in the UI Plugin. But you should already be building through maven most times. On the other hand, this does mean that it is probably easier to develop the UI Plugin after the core component is mostly stable.

5.3.1 Manifest Files

In addition to the POM (which is for maven builds), you must also configure your UI Plugin in order for it to run with the Symphony IDE. This is done with the `MANIFEST.MF` files in the `META-INF` folder. This is a simple text file with a basic syntax (`Header: Value`). The most important entries in the manifest files are the following:

- `Bundle-Name`: the name of your UI Plugin, per your choice
- `Bundle-SymbolicName`: the main package of your UI Plugin
- `Bundle-Activator`: the activator class for your UI Plugin. You must create this class and make sure it implements `BundleActivator`.
- `Import-Package`: the dependencies for your UI Plugin
- `Bundle-ClassPath`: files to extend your UI Plugin's classpath. You will usually want `.` as well as `lib/*.jar`

Indentation and format is important in Manifest files. Each value must be in its own line and indented with one space. Also, use of commas is mandatory to separate values. An example manifest file (for the POG UI plugin) can be found in Section A.3. The most crucial thing with the manifest is to make sure you get the the dependencies (Import-Package) right. If you are having unexpected `class not found` errors or something similar, there is a good chance you have forgotten to add something there.

6 Conclusion

As of Month 26 in the COMPASS project we have the structure in place to allow developers –both within the project and outside of it– to create extensions that add functionality to the Symphony IDE. Furthermore, plugins for Symphony have been developed within the project and have been integrated into the main tool distribution using this structure. This document provides the information necessary to begin such work, and serves as a starting point for developers.
This document's evolution from M16 has been minor, with changes directed towards keeping this document synchronised with the current state of the code and development environment. The overall structure of the Symphony IDE is relatively static at this point, and future changes to this document will remain minor.

A Configuration Examples

A.1 POG Core Component POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
      http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>eu.compassresearch.core.analysis</groupId>
    <artifactId>analysis</artifactId>
    <version>0.2.5-SNAPSHOT</version>
    <relativePath>../pom.xml</relativePath>
  </parent>

  <artifactId>pog</artifactId>
  <name>Symphony Core Analysis Proof Obligation Generator</name>


  <dependencies>

    <dependency>
      <groupId>eu.compassresearch.core</groupId>
      <artifactId>ast</artifactId>
      <version>${project.version}</version>
    </dependency>

    <dependency>
      <groupId>eu.compassresearch.core</groupId>
      <artifactId>typechecker</artifactId>
      <version>${project.version}</version>
    </dependency>

    <dependency>
      <groupId>org.overture.core</groupId>
      <artifactId>ast</artifactId>
      <version>${overture.version}</version>
    </dependency>

    <dependency>
      <groupId>org.overture.core</groupId>
      <artifactId>typechecker</artifactId>
      <version>${overture.version}</version>
    </dependency>

    <dependency>
      <groupId>org.overture.core</groupId>
      <artifactId>pog</artifactId>
      <version>${overture.version}</version>
    </dependency>

    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
    </dependency>

    <dependency>
```

```
      <groupId>commons-io</groupId>
      <artifactId>commons-io</artifactId>
      <version>2.4</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-source-plugin</artifactId>
        <executions>
          <execution>
            <id>attach-sources</id>
            <goals>
              <goal>jar</goal>
            </goals>
          </execution>
        </executions>
      </plugin>

      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-javadoc-plugin</artifactId>
        <executions>
          <execution>
            <id>attach-javadocs</id>
            <goals>
              <goal>jar</goal>
            </goals>
          </execution>
        </executions>
      </plugin>

      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>
```

A.2 POG UI Plugin POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
           http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>eu.compassresearch.ide</groupId>
    <artifactId>eu.compassresearch.ide.plugins</artifactId>
    <version>0.2.5-SNAPSHOT<!--Replaceable: Main Version--></version>
    <relativePath>../pom.xml</relativePath>
  </parent>

  <packaging>eclipse-plugin</packaging>
```

```
<artifactId>eu.compassresearch.ide.pog</artifactId>
<name>Symphony IDE POG Plugin</name>

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-dependency-plugin</artifactId>
      <executions>
        <execution>
          <id>copy-dependencies</id>
          <phase>process-sources</phase>
          <goals>
            <goal>copy</goal>
          </goals>
          <configuration>
            <outputDirectory>${basedir}/jars</outputDirectory>
            <overWriteReleases>true</overWriteReleases>
            <overWriteSnapshots>true</overWriteSnapshots>
            <overWriteIfNewer>true</overWriteIfNewer>
            <stripVersion>true</stripVersion>
            <artifactItems>
              <artifactItem>
                <groupId>eu.compassresearch.core.analysis</groupId>
                <artifactId>pog</artifactId>
                <version>${project.version}</version>
              </artifactItem>
            </artifactItems>
          </configuration>
        </execution>
      </executions>
    </plugin>

    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-clean-plugin</artifactId>
      <configuration>
        <failOnError>false</failOnError>
        <filesets>
          <fileset>
            <directory>${basedir}/jars</directory>
            <followSymlinks>false</followSymlinks>
          </fileset>
        </filesets>
      </configuration>
    </plugin>
  </plugins>

  <pluginManagement>
    <plugins>
      <plugin>
        <groupId>org.eclipse.m2e</groupId>
        <artifactId>lifecycle-mapping</artifactId>
        <version>1.0.0</version>
        <configuration>
          <lifecycleMappingMetadata>
            <pluginExecutions>
              <pluginExecution>
                <pluginExecutionFilter>
                  <groupId>org.apache.maven.plugins</groupId>
                  <artifactId>maven-dependency-plugin</artifactId>
```

```
                    <versionRange>[1.0.0,)</versionRange>
                    <goals>
                      <goal>copy</goal>
                    </goals>
                  </pluginExecutionFilter>
                  <action>
                    <execute>
                      <runOnIncremental>false</runOnIncremental>
                    </execute>
                  </action>
                </pluginExecution>
              </pluginExecutions>
            </lifecycleMappingMetadata>
          </configuration>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>

  <!-- The POG UI Plugin has no special dependencies but -->
  <!-- if it did, here is where we would add them -->
  <!-- <repositories> -->
  <!--   <repository> -->
  <!--     <id>eclipse-juno</id> -->
  <!--     <url>http://download.eclipse.org/releases/juno</url> -->
  <!--     <layout>p2</layout> -->
  <!--   </repository> -->
  <!-- </repositories> -->

</project>
```

A.3 POG UI Plugin MANIFEST

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Symphony IDE POG Plugin
Bundle-SymbolicName: eu.compassresearch.ide.pog;singleton:=true
Bundle-Version: 0.2.5.qualifier
Bundle-Activator: eu.compassresearch.ide.pog.Activator
Bundle-Vendor: The COMPASS Project
Import-Package: eu.compassresearch.ast.actions,
 eu.compassresearch.ast.analysis,
 eu.compassresearch.ast.declarations,
 eu.compassresearch.ast.definitions,
 eu.compassresearch.ast.expressions,
 eu.compassresearch.ast.process,
 eu.compassresearch.ast.program,
 eu.compassresearch.core.common,
 eu.compassresearch.core.typechecker.api,
 eu.compassresearch.ide.core.resources,
 eu.compassresearch.ide.ui.utility,
 eu.compassresearch.ast.lex,
 org.eclipse.core.commands,
 org.eclipse.core.resources,
 org.eclipse.core.runtime,
 org.eclipse.core.runtime.content,
 org.eclipse.core.runtime.jobs,
 org.eclipse.jface.action,
```

```
 org.eclipse.jface.dialogs,
 org.eclipse.jface.resource,
 org.eclipse.jface.viewers,
 org.eclipse.swt,
 org.eclipse.swt.graphics,
 org.eclipse.swt.widgets,
 org.eclipse.ui,
 org.eclipse.ui.handlers,
 org.eclipse.ui.part,
 org.osgi.framework,
 org.overture.ast.analysis,
 org.overture.ast.analysis.intf,
 org.overture.ast.definitions,
 org.overture.ast.expressions,
 org.overture.ast.intf.lex,
 org.overture.ast.lex,
 org.overture.ast.modules,
 org.overture.ast.node,
 org.overture.ast.patterns,
 org.overture.ast.statements,
 org.overture.ast.types,
 org.overture.ast.factory,
 org.overture.ide.core,
 org.overture.ide.core.resources,
 org.overture.ide.plugins.poviewer,
 org.overture.ide.plugins.poviewer.view,
 org.overture.ide.ui.utility,
 org.overture.pof,
 org.overture.pog.obligation,
 org.overture.pog.pub,
 org.overture.pog.utility,
 org.overture.pog.visitors,
 org.overture.typechecker,
 org.overture.typechecker.assistant.type,
 org.overture.typechecker.assistant.definition
Export-Package: eu.compassresearch.core.analysis.pog.obligations,
 eu.compassresearch.ide.pog
Bundle-RequiredExecutionEnvironment: JavaSE-1.7
Eclipse-BuddyPolicy: registered
Eclipse-BundleShape: dir
Bundle-ActivationPolicy: lazy
Bundle-ClassPath: .,
 jars/pog.jar
```

References

[Cha09] Scott Chacon. *Pro Git*. Apress, August 2009.

[CMNL12] Joey W. Coleman, Anders Kaels Malmos, Claus Ballegaard Nielsen, and Peter Gorm Larsen. Evolution of the Overture Tool Platform. In *Proceedings of the 10th Overture Workshop 2012*, School of Computing Science, Newcastle University, 2012.

[FLM+05] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object–oriented Systems*. Springer, New York, 2005.

[LBF+10] Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. The Overture Initiative – Integrating Tools for VDM. *SIGSOFT Softw. Eng. Notes*, 35(1):1–6, January 2010.

[LFW09] Peter Gorm Larsen, John Fitzgerald, and Sune Wolff. Methods for the Development of Distributed Real-Time Embedded Systems using VDM. *Intl. Journal of Software and Informatics*, 3(2-3), October 2009.

[MT09] David Holst Møller and Christian Rane Paysen Thillermann. Using Eclipse for Exploring an Integration Architecture for VDM. Master's thesis, Aarhus University/Engineering College of Aarhus, June 2009.