Grant Agreement: 287829

Comprehensive Modelling for Advanced Systems of Systems

COMPASS

Initial release of the COMPASS Tool Plug-in Developer Manual

Technical Note Number: D31.1b

Version: 1.0

Date: September 2012

Public Document

http://www.compass-research.eu

Contributors:

Peter Gorm Larsen, AU
Joey Coleman, AU
Anders Kaels Malmos, AU
Rasmus Lauritzen, AU
Stefan Hallerstede, AU

Editors:

Joey Coleman, AU
Stefan Hallerstede, AU

Reviewers:

Alexander Romanovsky, NCL
Margherita Forcolin, Insiel
Simon Foster, York

Document History

Ver	Date	Author	Description
0.1	18-07-2012	Peter Gorm Larsen	Initial document version
0.2	30-07-2012	Stefan Hallerstede	Edited; assigned tasks
0.3	13-08-2012	Joey Coleman	Added some content
0.4	29-08-2012	Joey Coleman	Added structure for chapters 4 & 5
0.5	04-09-2012	Joey Coleman	Fill in detail for the AST chapter
0.6	06-09-2012	Joey Coleman	Cleanup and kill draft notes for internal draft
0.7	07-09-2012	Joey Coleman	Add conclusion; ready for internal draft review
0.8	25-09-2012	Joey Coleman	Incorporate internal draft comments
1.0	28-09-2012	Peter Gorm Larsen	Ready for 1st year project review

Abstract

This deliverable describes the work done in the first year of the COMPASS project in regards to WP 31. Moreover, it describes what is included in terms of functionality in the first iteration of the COMPASS framework. Finally, it includes a guide on how to extend the framework capabilities with separate plug-ins.

Contents

List of Acronyms

API Application Programming Interface

AST Abstract Syntax Tree

CML COMPASS Modelling Language

DoW Description of Work

IDE Integrated Development Environment

JDT Java Development Tooling

MC Model Checker

OS Operating System

OSGi Open Services Gateway initiative

PDE Plug-in Developer Environment

POG Proof Obligation Generator

RCP Rich Client Platform

SDK Software Development Kit

SWT Standard Widget Toolkit

TC Typechecker

TP Theorem Prover

UI User Interface

VDM Vienna Development Method

VDMJ VDM Java interpreter implementation

XML Extensible Markup Language

1 Introduction

The purpose of this report is to introduce the Eclipse[1] environment and how it is used in this project as a basis for an open COMPASS framework to accomplish the incremental inclusion of functionality with plug-ins. The Eclipse environment is designed to serve as a tool platform and it is made so that it is easy to build client applications on it by adding plug-ins to its base. The COMPASS framework comprises a number of plug-ins developed specifically for this project together with the Eclipse base. This report describes how the early-stage COMPASS framework builds upon Eclipse, and how a potential developer can use it to add new plug-in extensions.

The COMPASS framework description and what it should contain can be found in the COMPASS Description of Work (DoW). The COMPASS framework provides the facilities for experimenting with models of systems of systems (SoSs).

The choice to use Eclipse as base was natural since it is open source and designed for easy modular incremental development. Moreover, the Eclipse platform provides a comprehensive set of libraries to build Integrated Development Environments (IDEs). In addition Eclipse is a tool which has been utilized successfully as base for many projects, open source and commercial.

The Overture Tool[2] (see [LBF+10]) uses the Eclipse platform as a basis. This is an open source tool for VDM (Vienna Development Method, see [FLM+05, LFW09]). The Overture Tool was also developed as a set of plug-ins to Eclipse. It is easily included in the framework. Overture contains a VDM interpreter that will be extended to fit the needs of simulating SoS models made in CML.

The structure of this report is the following: Section 2 is a small introduction to Eclipse and its capabilities; Section 3 describes our use of the Overture platform ASTGen tool and why it is important; Section 4 details our use of Git as a version control system for the COMPASS CML tool codebase and the freedom it gains us with distributed development; Section 5 describes how the COMPASS framework was built on top of Eclipse; Section 6 introduces the COMPASS framework architecture; Section 7 shows how to extend the COMPASS framework with further capabilities. Lastly the concluding remarks in Section 8 sum up what has been made and what is planned.

[1] www.eclipse.org
[2] www.overturetool.org

2 Eclipse Platform

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 OSGi framework and Eclipse Architecture

Eclipse is based on Equinox (Eclipse OSGi Framework), an implementation of the OSGi framework.[3] The OSGi framework is a specification of a module system and service platform for Java. It is a complete and dynamic component model in which applications and components can be, in principle, remotely installed, started, stopped, updated and uninstalled without requiring a restart.

Figure 1 shows how the pieces of Eclipse sit on Equinox. The *Enterprise Component Framework* is the component model that supports tool development, tool integration and rich client applications. The *Eclipse RCP* is the set of plug-ins needed to build rich client applications. The *Tools* are libraries provided by Eclipse that facilitate in the construction of applications. The *Runtime Components* and *Rich Client Components* are the basic building blocks for a business application.

From a bird's eye perspective Eclipse is conceptually two things:

1. An OSGi framework implementation called Equinox.[4]

[3] See `http://www.osgi.org/Main/HomePage`

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

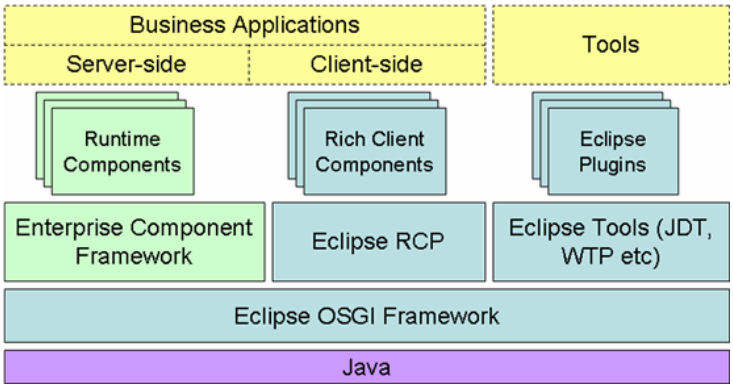

Figure 1: Eclipse on Eclipse OSGi Framework (Equinox)

2. A set of bundles running inside the Equinox OSGi framework.

The following gives a brief introduction to Equinox before Eclipse plug-ins are detailed futher.

2.1.1 Equinox an OSGi implementation

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 2.

To hook the bundle code to the OSGi Life Cycle a bundle must point out a special class in its `MANIFEST.MF` file. This special class is known as an Activator and implements an interface provided by Equinox with methods for handling transitions between life cycle states within the bundle. In addition a set of book-keeping attributes is maintained for each bundle. To get a feeling of how such a bundle is put together and interacts with the OSGi framework, consider the example `MANIFEST.MF` below.

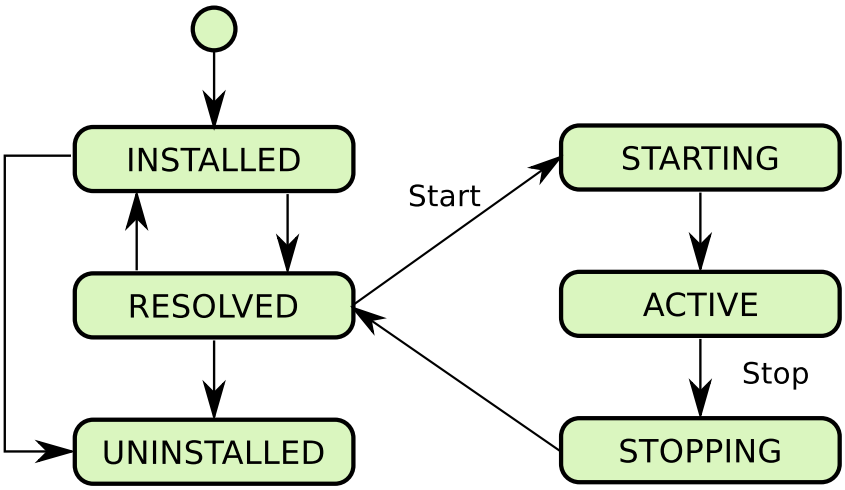

Figure 2: OGSi bundle life-cycle

```
Manifest-Version: 1.0
Bundle-Vendor: The COMPASS Project
Bundle-ActivationPolicy: lazy
Bundle-Version: 0.0.1
Eclipse-BuddyPolicy: registered
Bundle-Localization: plugin
Bundle-Name: COMPASS IDE CML UI
Bundle-ManifestVersion: 2
Bundle-SymbolicName: eu.compassresearch.ide.cml.ui;singleton:=
   true
Require-Bundle: org.overture.ide.core,
 eu.compassresearch.ide.cml.core, [...]
Bundle-Activator: eu.compassresearch.ide.cml.ui.CmlUIPlugin
Bundle-RequiredExecutionEnvironment: J2SE-1.5
Export-Package: eu.compassresearch.ide.cml.ui [...]
Import-Package: eu.compassresearch.ide.cml.core
Bundle-ClassPath: .,
 lib/ast-0.0.1.jar,
 lib/parser-0.0.1.jar,
 [...]
```

Here only the attributes that directly influences run-time behaviour are explained. Many of the attributes are self explanatory and the rest can be looked up on the Equinox website. The important attributes to be considered here are:

Bundle-SymbolicName - This is the name of the Bundle that other bundles depending on this bundle will use in their Require-Bundle attribute.

Require-Bundle - Comma separated list of Bundle-SymbolicNames that are necessary for this bundle to operate correctly. This bundle will have all classes in the listed bundles available in its class-path.

Bundle-Activator - The fully qualified name of the class in the bundle that implements and handles OSGi life-cycle transitions.

Export-Package - The java-packages offered by this bundle.

Import-Package - The java-packages imported by this bundle.

Bundle-ClassPath - The class path relative to the position of the bundle-jar-file (e.g. the plug-in directory) that are available on the run-time class-path when the OSGi framework is running.

It is important to be aware of these attributes in the `MANIFEST.MF` file as these are controlling the run-time environment any bundle has. An important note is that the compile environment is set up elsewhere either as an Eclipse-Build-Path, through Maven dependencies or directly given to javac during compilation. However, these dependencies are not automatically transferred to the run-time environment. Most difficulties with OSGi bundles (and Eclipse plug-ins) occur when there is a difference between the compile-time and run-time environments. It is the responsibility of the bundle developer to manually maintain the run-time environment such that it has the necessary class-path required.

2.1.2 Eclipse Plugins

Eclipse comes with a set of core OSGi bundles for registering and binding Eclipse plug-ins together. Besides these core Eclipse bundles all services and functionallities in Eclipse are provided though Eclipse-Features[5] and Eclipse-Plugins. An Eclipse plug-in is a OSGi bundle with a special configuration file: `plugin.xml`.

2.2 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

[5] A Feature is just a logical grouping of plug-ins cooperating to offer the named feature. Features are not explained in detail here.

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

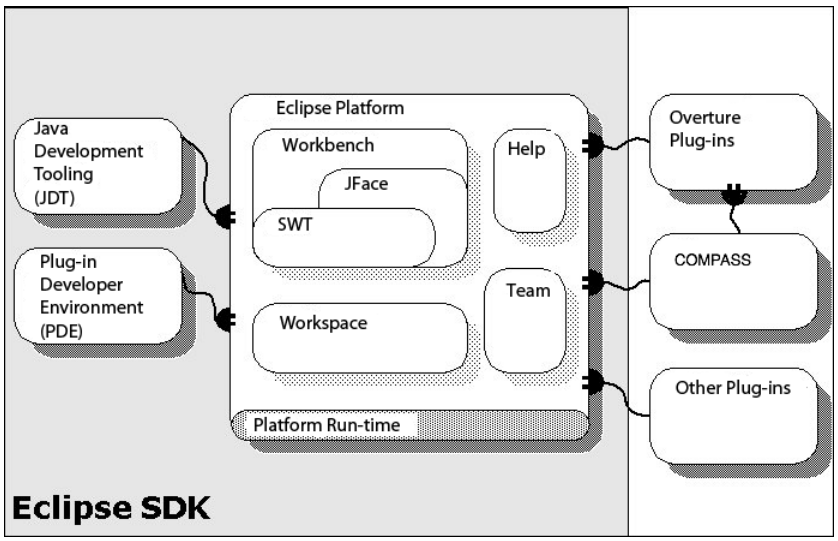

Figure 3: Eclipse SDK in detail

Figure 3 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 COMPASS framework, in the coding of all the plug-ins. However, the complete SDK does not need to be included in the standalone version of COMPASS. The standalone only includes the *Eclipse Platform* and the COMPASS plug-ins. The COMPASS tool is built on top of the 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 etc.

2.3 Eclipse Terminology

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

2.3.1 Workspace

A workspace is the basis for Eclipse platform resource management. There is only one workspace per platform 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.

2.3.2 Perspectives

A perspective is a fixed collection of views (editors, project explorers, etc) 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 4, while a Web Design perspective might have a a different set of views like a page designer, a XML editor, etc. Only one perspective can be active at a given time.

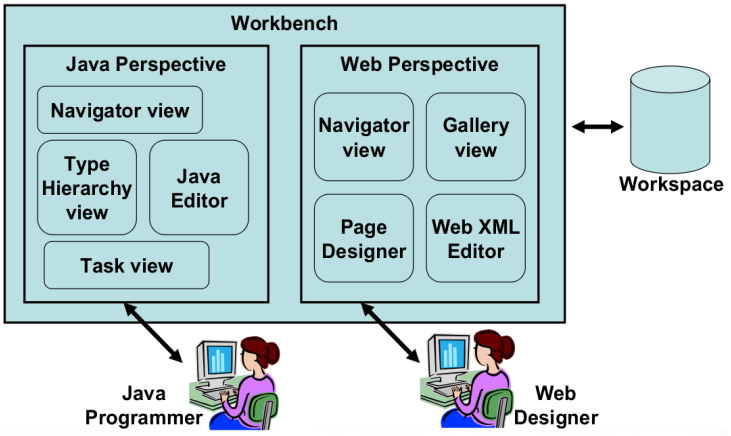

Figure 4: Eclipse Perspective Concept

2.3.3 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 views' user interfaces are organised in the same way, they have their own menu and toolbar.

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

Each editor has a process attached, called the *reconciler*. This process is activated every time a user types something in the editor and is responsible to update, for example, the outline or warning and error markers.

2.3.5 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.cml.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 COMPASS Tool implements a set of extensions to provide COMPASS features in Eclipse.

2.3.6 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 like Java, CML, COMPASS, etc. 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, etc.

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.

3 Abstract Syntax Tree Generation

The core of the COMPASS CML tool platform is the common core Abstract Syntax Tree (AST). This parallels the structure of the Overture platform and is key to the ability of the COMPASS CML tool's ability to reuse portions of the Overture platform source code.

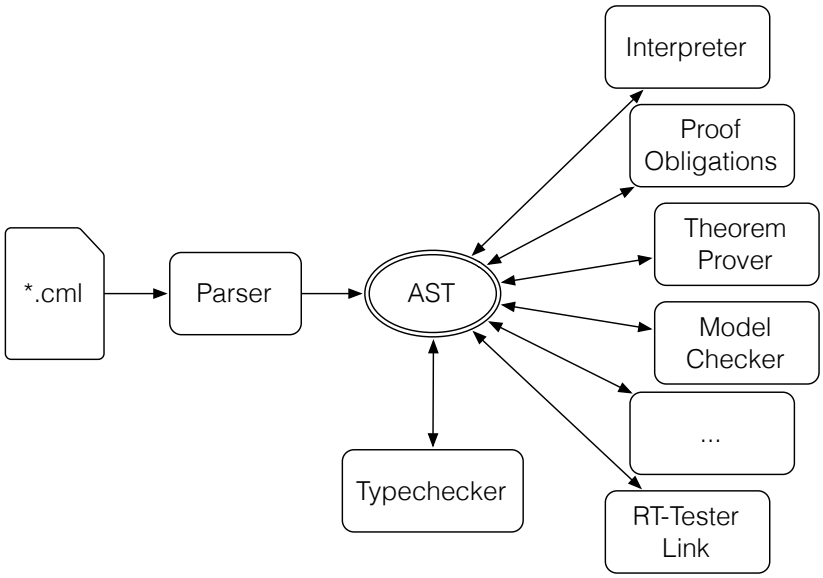

Figure 5: Architecture of the COMPASS CML tool platform

In Figure 5 we show the high-level structure of the COMPASS CML tool. Arrows indicate a relation in terms of dataflow between the modules. Reading from left to right, initially we start with some CML model in a collection of `.cml` files which are read in by the parser, producing a raw AST structure.

This raw AST will then typically be typechecked by the typechecker, ensuring that the models are statically valid and have a certain basic degree of consistency. However, depending on the particular application, some plugins may not need to wait for the typechecker to finish. The process of typechecking will alter the AST in-place, transforming the raw AST into one ready for use by the rest of the plugins.

Each plugin has the ability to add extra data to the AST of a specific CML model. As an example, the proof obligation generator is likely to add the proof obligations that arise from parts of the AST directly to the nodes in the AST that give rise to those obligations. The interpreter could potentially implement memoization for

functions by adding a memoization dictionary to the function nodes. The AST is designed to be flexible for many kinds of uses.

3.1 Using ASTGen

As a part of the Overture platform there is a tool called ASTGen that is used to automatically generate the entire AST of the various VDM dialects used in Overture. This tool is also used to generate the AST for the CML dialects, and uses the VDM AST as a basis for the CML AST. This usage is what allows us to directly reuse much of the interpreter and typechecker source code in the Overture platform without any need to recompile the VDM typechecker and interpreter.

The structure of a dialect is defined in an AST script that is read in by the ASTGen tool. The ASTGen tool then resolves the necessary structure of the target AST from this script and generates the necessary Java files to implement the AST, and it provides a set of extra Java files that implement the basic visitor functionality used to manipulate the ASTs.

When a developer needs to implement some functionality on the ASTs, they do not need to alter the base AST classes at all — in fact, we strongly recommend against doing so. Instead, the developer subclasses one of the visitor classes and writes the necessary functionality in this subclass. When their subclassed visitor is given to the visit method of the AST, the AST will, in turn, invoke a specially-named method in the subclass for each node in the tree, passing that node as a parameter.

This structure makes it possible for a visitor to act upon every element of an AST, and because it is the responsibility of each node-specific method to invoke the visitor for its children in the AST, it is also possible for the developer to control execution of the visitor over the structure of the AST.

Each of the basic visitor classes implements a minimal version of a method for each possible node in the AST, and the functionality of these minimal methods is to simply to visit its children. This allows the developer to subclass one of the basic visitor classes and only write the methods for the nodes of interest, rather than being required to implement a method for every possible node in the AST.

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

4 Version Control of COMPASS Source Files

The version control system used by the COMPASS project for the source code of the COMPASS CML tool is called Git, and documentation on its usage is freely available on the Git website.[6] Hosting for the Git repository is provided by SourceForge.net and the source code is publicly available.

Write access to the repository is controlled by 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.

Due to the fact that development on the COMPASS CML tool is distributed across several of the project sites, we have also adopted certain conventions regarding the use of the Git repository.

4.1 The choice of Git

The particular choice of the Git distributed version control system over the Subversion system used by the project for documentation was due to the extra support Git provides for distributed development. While both Git and Subversion provide a "branch" feature to maintain disparate streams of development, their technical implementations are significantly different.

In Subversion, branching is implemented via a user-managed filesystem hierarchy; in Git, branching is implemented internally in the repository's metadata and managed by the software. The net effect of this is to make both branching and the eventual merging of branches in Git a much simpler process.

4.2 Project development process and conventions

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

The branches are:

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

master The master branch is used to track releases of the COMPASS CML tool. A developer who clones the Git repository and checks out the master branch will be able to compile the latest release of the COMPASS CML tool. 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.

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 will be a developer named as being responsible for the maintaining the branch until it is ready to be merged into the development branch.

initials Branches that are named using the initials of a developer's name, or the initials as a prefix, are semi-private working branches for the individual developers. Changes to these branches should only be made by the named developer.

It is expected that the individual developers and the maintainers of the feature branches will monitor the development branch and keep their branch reasonably up to date with respect to it. In all cases a merge from development into the feature/initials branch should happen before a request is made to the release manager to merge work from the feature/initials branch back into development.

This structure, and the fact that branch management in Git is easy, 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.

4.3 The release process

The release process cycle starts, after a previous release, with the developers working on various features in either feature branches or in their own initialled branches. Occasionally they will merge changes from the development branch into their own branches to keep up to date, and in some cases they may wish to merge from other feature/initialled branches.

As the various new features and bugfixes become stable, they are merged into the development branch, coordinating with the release manager. Merging features into the development branch as they finish also ensures that the complete test suites are run against those features on the build server on a regular basis.

When a planned release approaches, the necessary features for the release are merged into development, and the developers are asked to focus on the stabilisation and integration of the overall tool.

Once the tool is sufficiently stable for use, a release is made and the master branch is updated from the development branch. The cycle then restarts for the next planned release.

5 The COMPASS Framework

This chapter presents the distinctive features of the COMPASS framework. Section 5.1 starts by laying out the intended plug-in structure for the COMPASS Tool. Section 5.2 describes what was done to give a more personalised look to the framework. Section 5.3 is an overview of what the COMPASS framework provides to the end user. Section 5.4 gives more details about how basic Eclipse features have been used to provide the functionality as described in Section 5.3. Section 5.5 introduces the programmable interface for the COMPASS framework, as this is the interface intended for extensions of the framework for the common COMPASS tool integrating the different features in a unified fashion.

5.1 COMPASS Tool structure

The planned plug-ins in the COMPASS tool framework are

COMPASS Core: The COMPASS Core plug-in is a OSGi bundle that wraps the CML AST, lexer, parser and typechecker together, offering these to any other plug-in running in Eclipse.

Interpreter: The interpreter will also be created as a plugin, dependent on the typechecker. The interpreter will give a warning if it detects that not all of the POs have been discharged.

Proof Obligation Generator: This plugin analyses a typechecked CML AST and generates the proof obligations necessary to assert that the CML model is consistent.

Theorem Prover Interface: Takes the proof obligations generated by the POG, translates them into an Isabelle logical model, and tries to get the prover to discharge (prove true) them. We expect it will also be possible to give other constraints to the TP plugin and use it to attempt to prove them as well.

Model Checker: The model checking plugin is planned to take a typechecked CML AST and either show that properties like deadlock-freedom do (or do not) hold. It may also be possible to use the model checking plugin to generate counterexamples for proof obligations before they are given to the theorem prover.

Refinement Checker: This plugin is expected to analyse pairs of CML models and generate a set of proof obligations that, if all true, show that one model is a correct refinement of the other. This will necessarily use the theorem

prover plugin to discharge these obligations, and the model checker may be used to give counterexamples for possible refinements.

RT-Tester interface: This plugin will provide an interface to the RT-Tester tool: a control dashboard for RT-Tester will allow the user to control the behaviour of RT-Tester running CML models.

5.2 Branding of the COMPASS framework

Eclipse provides branding services by which it is possible to transform visually (colours, icons, signs, etc) the final standalone application, giving it a more customised look than the default Eclipse look. This enables application developers to give a different identity to their tool, without compromising on the usability of the well known tool that is Eclipse.

Eclipse supports product branding by customising the window image, *About* dialogue and the welcome experience. The COMPASS IDE benefits from this form of branding because it can be customised with a custom launcher icon as well as a custom welcome page where new users can get information about existing examples and user documentation (see Figure 6).

(a) Executable. (b) RCP Splash.

Figure 6: Branding of the COMPASS IDE.

5.3 Features

The COMPASS framework as delivered in D31.1 provides features for editing CML files, inspecting the structure of the models contained therein, and basic verification of the type-correctness of the models. The framework also provides for management of COMPASS projects and the CML files contained within them.

Future deliverables will include the following features

- editor with error/warning markers
- editor with code completion
- integrated simulation
- all the assorted plugins

5.4 Extension base features in the Eclipse framework

The COMPASS framework is built as a group of plug-ins in the same way as Eclipse is built as described in Chapter 2. All the features described in Section 5.3 are built by extending Eclipse interfaces or built-in standard implementations such as the *TextEditor*. To enable the extensions made in the framework, all must be registered in Eclipse through extension points enabling the new features to be loaded in the RCP. Section 5.4.1 provides a description of how the COMPASS editors are made and which extension points they use. This editor structure is the base for all COMPASS editors. Section 5.4.2 describes how the builder is added to the COMPASS project type.

5.4.1 Editor

The COMPASS framework extends Eclipse with editors that have various features. In this section a description of the main elements of such editors will be described. It will not cover a complete in-depth description of how the editors are made but give an overview of how Eclipse can be extended with editors. Creating a new editor for text editing is simple since Eclipse already provides a built-in text editor that can be extended used the provided extension points for customization. In Listing 1 it is shown what this extension point looks like.

The first thing which is needed is an Eclipse extension point. This will include the new editor into Eclipse and allow it to be associated with a file type:

Extension point: `org.eclipse.ui.editors`

Adding a new editor is simple. The extension point must be filled with the appropriate information as shown in listing 1. This XML needs to be added in a `plugin.xml` file[7], in this case located in the `org.compass.ide.ui` project. The `class` attribute is a fully qualified class name of the implementation of the editor. The attributes `default` and `extensions` (which are omitted in listing

[7] A `plugin.xml` file is present in all plug-in development projects in Eclipse

1) are also important because they set the default editor for a specific file extension. In case `default = "true"` and `extensions=log`, the editor will open when any file with the extension log is opened from the navigator view of the application been developed. The `contributorClass` points to a class that will make additional actions available in the menus when the editor is open.

```
<extension point="org.eclipse.ui.editors">
  <editor class="org.compass.ide.ui.editor.impl.CMLEditor"
                contributorClass="org.eclipse.ui.texteditor.
                   BasicTextEditorActionContributor"
    default="true"
    icon="icons/cview16/cml_file_tsk.gif"
    id="eu.compassresearch.ide.cml.ui.editor"
    name="CML Editor">
</extension>
</editor>
```

Listing 1: Extension point for editor

In listing 1 a new `CML Editor` has been declared. However the actual implementation of it must also be provided. To do this a number of built-in classes and interfaces are used:

Extended classes/interfaces:	• `TextEditor` • `RuleBasedScanner` • `IReconcilingStrategy` • `SourceViewerConfiguration`

The new editor can extend the `TextEditor` which gives the editor all the default features expected in a normal text editor such as copy, paste and undo. However in this framework a more advanced editor is needed, since it must be able to highlight the syntax for comments and keywords. It must also be able to detect when the text has been edited so it can be re-checked for syntax and parse errors. To accomplish this, the editor needs a custom `SourceViewerConfiguration`.

SourceViewerConfiguration A source configuration has three main features that are important for this example.

1. It provides an instance of a reconciler, in this case the reconciler is a `MonoReconciler`.
2. It provides an `IReconcilingStrategy` and lastly
3. A code scanner as an `ITokenScanner`.

IReconcilingStrategy The reconciling strategy is the place where notifications

can be received about edited partitions of the document loaded by the editor. The `reconcile` method must be associated with the parser for the editor to enable the file to be reparsed whenever it is edited.

ITokenScanner This is the code scanner used to highlight keywords and comments. It is based on a `RuleBasedScanner` where rules are given to recognize tokens in the text.

5.4.2 Builder

A builder in Eclipse is normally a process that is run on a project whenever a resource is altered. In the COMPASS framework the builder does static type checking on all CML files. A builder is associated with a project by a *nature*. So, before the builder can be defined, a nature must be defined. Natures are enabled by extensions in the same ways as the editors are. The extension point for natures is:

Extension point: `org.eclipse.core.resources.natures`

A new nature (as shown in listing 2) can be defined in the `plugin.xml` file of the `org.compass.ide.ui` project:

```
<extension point="org.eclipse.core.resources.natures"
    id="nature">
  <runtime>
    <run class="org.compass.ide.core.resources.
       CompassProjectNature" />
  </runtime>
</extension>
```

Listing 2: The COMPASS nature extension

The XML snippet in listing 2 omits a number of elements such as the option for associating a builder directly to the natures and making the nature dependent on other natures.

Defining an incremental builder for eclipse projects with a specific nature must be done through the extension point:

Extension point: `org.eclipse.core.resources.builders`

The COMPASS builder is defined as shown in listing 3. When both a nature and a builder have been defined a project can be created with the nature id and the

builder name. When this is done the builder will be invoked when a resource is changed and saved in the project.

```
<extension point="org.eclipse.core.resources.builders"
    id="org.compass.ide.core.builder"
    name="COMPASS Project builder">
  <builder callOnEmptyDelta="true" hasNature="true"
      isConfigurable="false">
    <run class="eu.compassresearch.ide.cml.ui.builder.
       CmlIncrementalBuilder" />
  </builder>
</extension>
```

Listing 3: The COMPASS builder

The details about how the framework creates projects will not be described here since it is done in the standard Eclipse way. Listing 4 shows a small snippet from the project file illustrating that both the nature and builder have been added to a project:

```
<projectDescription>
  <name>Test Project</name>
  ...
  <buildSpec>
    ...
    <buildCommand>
      <name>org.compass.ide.core.builder</name>
      ...
    </buildCommand>
  </buildSpec>
  <natures>
    <nature>org.compass.ide.core.nature</nature>
  </natures>
</projectDescription>
```

Listing 4: Test project with nature and builder.

The actual builder is an implementation of the incremental project builder interface:

Extended classes/interfaces:	• `IncrementalProjectBuilder`

It consists of one method called `build` which must be implemented. This method is called when a resource in the project has changed. The body of this `build` method contains the code that statically checks the consistency between the con-

figuration files.

5.4.3 Outliner

The COMPASS outliner for M12 is able to outline CML classes and processes with State Actions. It does not yet have decorations. The outline is an existing plug-in coming with the Eclipse workbench. When an editor is activated (shown) the outline will query the editor for an instance of an object implementing the `IContentOutlinePage` interface. The CmlContentPageOutliner is given the root of the CML-AST for the content presented in the CML-editor. The Outliner shows the top level elements in the AST. As the user expands each element the outline dives into the AST rending a textual description of the encountered CML-AST-Node.

The interface between the Outliner plug-in and the CmlEditor for getting the IContentOutlinePage instance is implemented through the method `getAdapter(Class required)` implemented in the CmlEditor.

```
Object getAdapter(Class required) {
  if (IContentOutlinePage.class.equals(required)) {
    if (cmlOutLiner == null) {
      cmlOutLiner = createCmlOutliner();
    }
    return cmlOutLiner;
  }
}
```

5.5 Framework interface provided for extension

The COMPASS framework is compiled into plug-ins which expose all but a few packages allowing other plug-ins to reuse classes from the framework. Only a few internal packages are not exported to ensure consistency, (e.g. the creation and initialization of COMPASS projects).

The COMPASS framework provides:

1. A basic Editor with easy syntax colouring and reconciling (syntax check point).
2. A COMPASS project with framework specific interface.

3. A COMPASS project explorer.
4. A COMPASS perspective.

The framework does NOT restrict usage of the standard Eclipse resources plug-ins such as `IProject` which provides features for manipulating project members such as `IFile` or `IFolder` of the project. However the COMPASS framework provides a convenient way of getting a `ICompassProject` from a `IProject` by use of the adaptor feature of Eclipse. Listing 5 shows how the extension point for adapters has been used to register an adaptor which can convert a `IProject` into a `ICompassProject`. This makes it easy for any plug-in developer to access the dedicated project of this framework.

Extension point: `org.eclipse.core.runtime.adapters`

```
<extension point="org.eclipse.core.runtime.adapters">
  <factory adaptableType="org.eclipse.core.resources.IProject"
      class="org.compass.ide.core.resources.
         CompassProjectAdapterFactory">
    <adapter type="org.compass.ide.core.resources.
       ICompassProject"/>
  </factory>
  <factory adaptableType="org.compass.ide.core.resources.
     ICompassProject"
      class="org.compass.ide.core.resources.
         CompassProjectAdapterFactory">
    <adapter type="org.eclipse.core.resources.IProject"/>
  </factory>
</extension>
```

Listing 5: Adaptor factory for `IProject` and `ICompassProject`.

Listing 6 shows how the `ICompassProject` can be obtained from a `IProject` which is the common resource type of a project in Eclipse.

```
ICompassProject compassProject =
  (ICompassProject) project.getAdapter(ICompassProject.class);
```

Listing 6: Obtaining a COMPASS project from the `IProject` Eclipse type.

The COMPASS project has a number of methods to obtain important source folders or other resources used to configure and run a co-simulation. In Figure 7 a UML class is shown of the `ICompassProject` interface returned by the adapter.

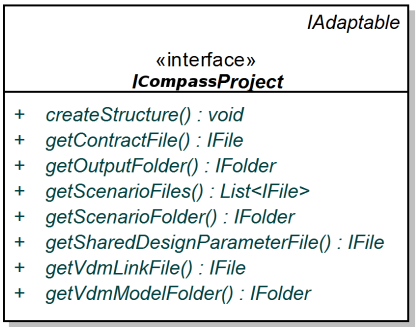

Figure 7: The COMPASS project interface

6 COMPASS Framework Architecture

This chapter provides an overview of the COMPASS framework architecture. The architecture will be described at the source module/project level, relating to the compilation process of the framework.

6.1 Framework artifact structure

The framework is structured as a number of projects/artifacts that may depend on each other. The artifacts are mainly written in Java, and Apache Maven[8] is used to manage dependencies and automate the build process. Maven is a software tool for project management and build automation which is used by the framework. When a Maven project has a parent and references a number of child projects it is mentioned in the literature as a *module*.

The framework is structured in three groups:

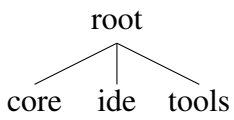

Figure 8: Project structure of the COMPASS framework.

core: All artifacts that are not dependent on Eclipse. This includes lexers, parsers, typecheckers, interpreters, and so on.

ide: All Eclipse plug-ins both graphical and non-graphical, as well as any other artifacts that provide a user interface to the core artifacts.

tools: Any custom tools used during the compilation process. This is presently empty, but may become necessary as development progresses.

6.1.1 Core

The `core` group is a project that has a number of child projects, all of which are used in the framework as fundamental building blocks. This includes the CML parser, interpreter, and the necessary analysis libraries. All of the children of the `core` are developed independently of Eclipse, and dependencies on Eclipse libraries are explicitly disallowed. This allows the core to be built using Maven

[8] See `http://maven.apache.org`

in any standard Java environment. A benefit of this is that we have developed a command line version of the tool, which is useful for both development and direct invocation by other programs. The graphical representation of the children with a description of each sub project is shown in Figure 9.

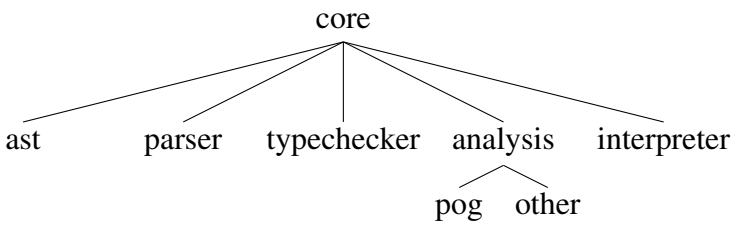

Figure 9: Maven project overview of the core project.

ast: The ast project defines the Abstact Syntax Tree of the CML language using the ASTgen tool from the Overture platform. The AST is defined in the file `cml.ast` and is used during the build process to generate the requisite Java files. There are also a number of Java source files that define helper classes for the AST.

parser: The parser project contains both the jFlex scanner definition and the GNU Bison grammar for the CML language. These are used to generate the Java source files that form the CML parsing library.

typechecker: The typechecker project contains the typechecker for the CML language. It extends the VDM typechecker from the Overture project, allowing usage of the existing VDM typechecker on the subset of CML that is identical to VDM.

analysis: The analysis project groups all of the plugins that do analysis on CML beyond basic typechecking.

- **pog:** The pog project contains the proof obligation generator for the CML language, generating constraints that need to be proven to validate the consistency of a model in CML. At present this project is a placeholder for the D31.2 deliverable.
- **other:** Preparations have been made to receive other analysis plug-ins, for instance:
  - **theoremprover:** The theoremprover project will contain an interface to the Isabelle/HOL theorem prover, and will be capable of taking generated proof obligations and discharging them using Isabelle/HOL. At present this project is a placeholder for the D33.2 deliverable.

refinementtool: The refinementtool project will generate the necessary proof obligations to show that there is a refinement relation between two CML models. At present this project is a placeholder for the D33.4 deliverable.

interpreter: The interpreter project contains the simulator for the CML language. Like the typechecker, this extends the VDM interpreter from the Overture project, allowing us to evaluate the subset of CML that is identical to VDM.

6.1.2 IDE

The Maven `IDE` group is a project with a number of child projects that may depend upon Eclipse, and they may require the use of an Eclipse base installation to be compiled. (The `cmdline` project is an exception.) The tree below shows the child projects excluding the two special projects `platform` and `build`. Both are related to the building of the branded RCP. The `platform` project contains extensions enabling the RCP to be branded and the `build` contains the build scripts to compile the final product.

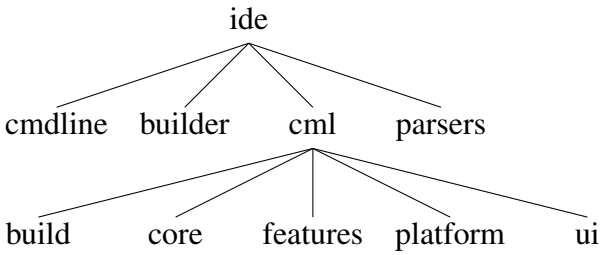

Figure 10: Maven project overview of the ide project.

cmdline: The cmdline project is the sole exception in the IDE group in that it does not depend on Eclipse in any way. This project provides a commandline interface to the core projects.

builder: The builder project is the Eclipse background builder mechanism for CML projects. This allows Eclipse to automatically re-parse, typecheck, and analyse a CML model in the background after a user makes changes.

parsers: The parsers project provides an interface for the core parsing libraries to the IDE plugins.

cml: The cml project contains the necessary projects to create the Eclipse plugin that represents the COMPASS CML tool, as well as the necessary build scripting to create a standalone Eclipse RCP application.

build: The build project contains the scripts that compile the COMPASS tool, resulting in the zip files for the supported platforms and a repository that can be used for the Eclipse self-update feature. Self-update is an Eclipse feature that allows a standalone Eclipse application to check for and install newer versions of the installed plug-ins from a remote repository.

core: The core project is the base project of the COMPASS framework in Eclipse. It contains the code for the COMPASS project and the project nature. This plugin depends only on Eclipse libraries, but all other COMPASS plugins refer to this project.

ui: This project contains all of the basic UI code and extensions for the COMPASS IDE such as project icons, editors and so on.

platform: The platform project contains all extensions to brand the RCP. This includes a splash screen, an "about" dialog, the welcome message and finally it also defines the Eclipse *product* information[9] that specifies what is included in the final COMPASS standalone application.

6.2 Tools

The Maven `tools` group is a project intended to contain Maven tools required in the build process of the COMPASS tool. Maven tools may be created as Maven plug-ins, compiled, then immediately used in the build process. It is important not to confuse a Maven plug-in with an Eclipse plug-in, as Maven plug-ins are not part of the final standalone application and are only useful when building the COMPASS tool.

At present, this directory is empty. However, past experience in other projects has suggested that there will inevitably be a need for this project.

[9] An Eclipse product is a description of the contents of a standalone application and consists of Eclipse features, plugins, and the branding information.

7 Extending the COMPASS Framework

This chapter describes how to extend the framework in both graphical and non-graphical fashions. This chapter uses three example projects to show how a simple `randomnumber` project can be created and made available as: a core project only relying on Java; an Eclipse project with a graphical interface; and a generated project wrapping a core project as an Eclipse plug-in project. In section 7.1 a description is provided on how a new core project can be added and made available from inside Eclipse and in Section 7.2 a description of how to create a new Eclipse plug-in with a simple view is given. Before a new project is created it is important to identify if it will require Eclipse dependencies, since the place where it should be located structure is related with it.

core: This is the simplest type of project. If the new project has only *core* functionality then it should be created here. This includes parsers, interpreters and other simple Java programs that are only used by other projects.

ide: Projects created under ide includes any project which extends the Eclipse platform, but is also intended to include any project that presents a user interface.

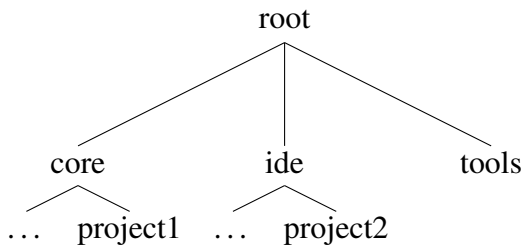

Figure 11: File structure showing two new projects, project1 and project2

The two projects shown in the tree in Figure 11 represent the file structure of where a project must be created:

1. Creating a new core Java project (`randomnumber` in the example in Section 7.1).
2. Creating a new ide Eclipse project (`numberviewer` in the example in Section 7.2).

All projects are Maven projects. This means that, in general, the steps involved in the creation of the project are the same, but there are differences in the projects' configuration. The configuration of a Maven project is done using a XML file named `pom.xml`.

7.1 Extending the framework with a core artifact

The core of the framework can be extended by a new project by navigating to the core folder and creating a new folder, in this section we call it: `randomnumber`.

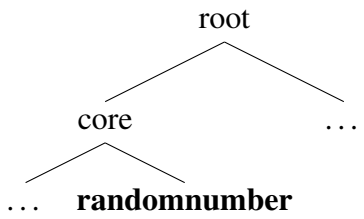

Figure 12: Adding a new project `randomnumber` to the file structure.

When the folder is created the project must be configured. This is done in a `pom.xml` file which gives the project configuration for Maven. The `pom.xml` configuration is explained in Section 7.1.1. However, first the source folder of the Java code must be created as Maven uses a specific set of conventions regarding the structure of folders, and these be followed. The Java code must start its package structure from the `java` folder:

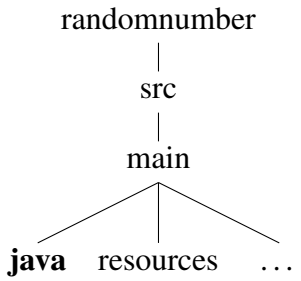

Figure 13: Project folder structure for the `randomnumber` project.

7.1.1 Creation

The `pom.xml` file of a core project must contain the items as shown in listing 7. The first few lines are required to identify this file as a Maven `pom.xml` file. The `parent` block that follows identifies this particular project as part of a larger project, in this case the COMPASS core group. The final lines identify this project as the `randomnumber` artifact of the `eu.compassresearch.core` group (the group is inherited from the parent if not specified), and give this project a human-readable name. Note that the version of this project is also inherited from the parent if not specified.

```
<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</groupId>
    <artifactId>core</artifactId>
    <version>0.0.1<!--Replaceable: Main Version--></version>
    <relativePath>../pom.xml</relativePath>
  </parent>

  <artifactId>randomnumber</artifactId>
  <name>Random Number Generator</name>
</project>
```

Listing 7: Core project configuration for randomnumber

After the project has been properly configured, its parent must be informed of its existence. This requires the addition of the `randomnumber` module to the `modules` block in the core `pom.xml` file as shown in listing 8.

```
<modules>
    <module>randomnumber</module>
</modules>
```

Listing 8: Adding the new project to the core `pom.xml` file.

At this point the developer may begin to add Java source files to `randomnumber/src/main/java` and use Maven to compile the project.

7.2 Extending the IDE with a new Eclipse view

Extending the IDE with a new Eclipse plug-in can be done by creating a new Maven project that has a particular `packaging` option specified. In figure 14, a `numberviewer` project is introduced as an Eclipse project for the `randomnumber` project. The project must be created in the ide folder as shown in figure 14.

When the project is created, the same folder structure must be created as described in Section 7.1.

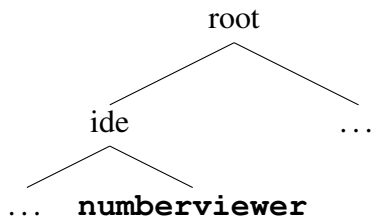

Figure 14: Adding a new project numberviewer to the file structure.

7.2.1 Configuration

A new Eclipse plug-in is created much in the same way as in Section 7.1, however the `packaging` option is different. This option tells Maven that it needs to be compiled in a particular way. The configuration is shown in listing 9 and the addition to the parent (IDE) `pom.xml` is shown in listing 10.

```
<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>ide</artifactId>
    <version>0.0.1<!--Replaceable: Main Version--></version>
    <relativePath>../pom.xml</relativePath>
  </parent>

  <artifactId>eu.compassresearch.ide.numberviewer</artifactId>
  <name>COMPASS NumberViewer IDE plugin for the core
     randomnumber project</name>

  <packaging>source-plugin</packaging>

  <dependencies>
    <dependency>
      <groupId>eu.compassresearch.core</groupId>
      <artifactId>randomnumber</artifactId>
      <version>${project.version}</version>
    </dependency>
  </dependencies>
</project>
```

Listing 9: Project configuration for a new Eclipse plug-in

As with the `randomnumber` plugin, the `numberviewer` plugin must also be added to its parent `pom.xml` file.

```
<modules>
    <module>numberviewer</module>
</modules>
```

Listing 10: Adding the new plug-in to its parent IDE `pom.xml` file

7.2.2 Development of the plug-in

When the Eclipse plug-in has been configured and imported in Eclipse, it is time to develop the actual plug-in. The first step is to configure the `MANIFEST.MF` where dependencies must be added. These are:

- `org.eclipse.ui.views`
- `org.eclipse.ui`
- `org.eclipse.core.runtime`
- `org.eclipse.core.resources`

The second step is to add extension points to the `plugin.xml` file. In listing 11 the views extension point is added and extended by the class specified in the `class` attribute.

```
<extension point="org.eclipse.ui.views">
  <view class="eu.compassresearch.ide.numberviewer.ui.views.
     NumberView"
        id="eu.compassresearch.ide.numberviewer.ui.numberview"
        name="Number View"
        restorable="true">
  </view>
</extension>
```

Listing 11: Extending views

The class (`org.compasside.numberviewer.ui.views.NumberView`) implementing the view must extend `ViewPart` and implement `ISelectionListener`. The method shown in listing 12 `createPartControl` must implement the code to create the SWT elements displaying the content of the view.

```
public void createPartControl(Composite parent) {
  ...
}
```

Listing 12: Create Part control

If any of the COMPASS features available in the `ICompassProject` should be used, an instance of a COMPASS project can be obtained from a Eclipse `IProject` as shown in listing 13.

```
IWorkspace root = ResourcesPlugin.getWorkspace().getRoot();
IProject project = root.getProject(projectName);
if(project != null)
{
  ICompassProject cproject =
    (ICompassProject)project.getAdapter(ICompassProject.class);
}
```

Listing 13: Obtaining in instance of `ICompassProject`

8 Concluding Remarks

At the end of the first year of the COMPASS project, the COMPASS CML tools have advanced to the state where we have: an initial CML AST, based on the D23.1 CML Definition 0 and the AST for VDM from the Overture platform; the corresponding parsing libraries that, with the AST, form the spine of the CML tools; an initial typechecker for CML that uses the CML AST and extends the Overture VDM typechecker; a basic Eclipse-based IDE that provides the basic features necessary for developing CML models. These features form the core of the software component of the D31.1 deliverable.

We also have an executable command-line tool that can be used to invoke the initial version of the CML interpreter; this forms the core of the D32.1 deliverable.

The improvements planned for the immediate future include integrating the initial CML interpreter into the Eclipse-based IDE and updating the core libraries to conform to the D23.2 CML Definition 1 deliverable. We will also have the initial proof obligation generator plugin integrated into the CML tool, and we will have integrated the necessary connections to the RT-Tester tool to enable test automation with CML models. At this point the CML tool will be ready for use by the project partners focused on case studies.

Beyond that point, the work on the CML tool will focus on maintaining conformance with the D23.x CML Definition deliverables, support for debugging and animation of CML models, and on the creation of plugins that provide further functionality in the CML tool. Among the plugins are: connections to a theorem prover and a model checker for the analysis of CML models; plugins for analysis of models such as refinement checking and static fault injection.

References

[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. *ACM Software Engineering Notes*, 35(1), 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.