Grant Agreement: 287829

Comprehensive Modelling for Advanced Systems of Systems

COMPASS

Fourth Release of the COMPASS Tool Symphony IDE Developer Documentation

Deliverable Number: D31.4b

Version: 1.0

Date: September 2014

Public Document

http://www.compass-research.eu

Contributors:

include Joey W. Coleman, Aarhus
Luís Diogo Couto, Aarhus
Anders Kaels Malmos, Aarhus
Richard Payne, Newcastle
Uwe Schulze, Bremen
Klaus Kristensen, B&O
Kewnneth Lausdahl, Aarhus

Editors:

Joey W. Coleman, AU

Reviewers:

Uwe Schulze, Bremen
Richard Payne, Newcastle
Adrian Larkham, Atego

Document History

Ver	Date	Author	Description
0.1	02-12-2013	JWC	Initial document version based on D31.3b
0.2	03-03-2014	LDC	Incorporate D31.3 feedback
0.3	15-07-2014	JWC	Structure check
0.4	21-07-2014	JWC	Incorporate Portions of D32.4; Editing
0.5	23-07-2014	JWC	Editing
0.6	23-07-2014	LDC	Various minor corrections to reflect name, URL changes, etc.
0.7	08-08-2014	JWC	Add last bits; ready for internal review (modulo cosim protocol)
0.8	20-08-2014	JWC	Editing from internal review
0.9	22-08-2014	JWC	More editing from internal review
1.0	01-09-2014	PGL	Final polish after Kenneth and Uwe met in Bremen

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. We also cover the co-simulation protocol used for running simulations with external processes acting as a component in a CML model.

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 details 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 plug-ins and extensions for Symphony. The process by which CML models are simulated is described in Section 6, along with the necessary detail of the co-simulation process and the network protocol that supports it.

[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

This section provides background information on the main technologies with which Symphony is implemented: the Eclipse SDK and the Overture IDE.

2.1 Eclipse

The Eclipse Platform provides core frameworks and services upon which plug-in extensions can be created and 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 that use an OSGi framework in the same way as Eclipse 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 Java jar file containing 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, providing complete control over the classpath of every bundle.

To control and maintain bundles the OSGi framework stipulates that a bundle must have the life-cycle depicted in Figure 1. Essentially, a bundle may be dynamically installed, started, stopped, updated and uninstalled. A bundle must be installed to enter Equinox and uninstalled to leave it. Once installed, a bundle may be resolved through an Equinox command. Resolving consists of looking up and resolving any bundle

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

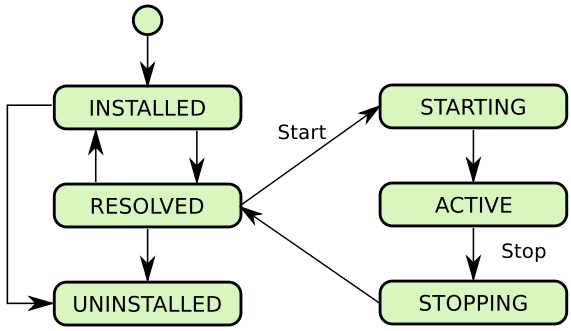


Figure 1: OGSi bundle life-cycle

Source: Wikipedia.org

dependencies. Once resolved a bundle may be started (and stopped) as needed. It is also possible to refresh or update the bundle, returning it to installed state.

2.1.2 Eclipse SDK

Eclipse is more than just an platform to develop plug-ins, it also provides many libraries to help in the creation of whole 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 where other plug-ins can add functionality. These form the base of the framework described in this document.

It is important to notice that the Eclipse developers themselves have extended the platform 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 the Symphony IDE as it is distributed.

The Symphony IDE only includes the base *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. 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 and editors appropriate to a given task. 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, while a Web Design perspective might have a different set of views such as a page designer, an XML editor, and so on. Only one perspective is active at a given time.

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. Example of views include file editors, outlines, and project explorers.

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 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 `plug-in.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 `plug-in.xml` file:

```
<extension point="org.eclipse.ui.editors">
  <editor
```

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

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

The extension in the `plug-in.xml` file stipulates which point it extends and points to a Java class in the plug-in bundle that possibly implements an interface (IEditorPart) associated with the given extension point. In this way Eclipse plug-ins can leverage functionality from the Eclipse platform and other third party plug-ins 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 into two main groups: 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 the core libraries to create an Eclipse-based IDE that provides an interface to the core functionalities. There is also a smaller, third group of components that are meta-components; these include libraries that aid the build process. The most critical of these is ASTCreator which is discussed in Section 4.1.1.

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, reuse allows for updates to the Overture platform to be automatically incorporated into Symphony as they become available. The Symphony IDE is the first full extension of the Overture platform in that it builds a complete new IDE on top of Overture, and our efforts to build Symphony also have the effect of improving the Overture platform.

3 Build Environment

This section assumes that all the necessary tools have been correctly installed and configured (see Section 3.2 for specifics). 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 GitHub.

Write access to the repository is controlled via the GitHub repository permission structure, and developers must first contact the administrator(s) 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 three 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. Also note that the build server builds and publishes developmental versions of the tool based on this branch.

`test` The test branch is a looser version of the development branch. Like the development branch, the build server runs build jobs based on commits to the test branch. Unlike the development branch, the test branch is not supervised by the release manager. The code on the test branch 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 anything committed to this branch will be preserved, so it is not to be used in place of a developer's own initialled branch (see below).

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

The other categories of branch are:

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 each branch. There should be a developer responsible for 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 plug-ins of unsupported CML constructs in the current model; the branch is named `uecollector`. Some developers who work on features, prefer to do so in a personal branch so 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 them. Examples include `jwc` for a branch worked on by the developer "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 to 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 consequence: 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.

The development of Symphony requires the following tools and versions:

- Java Development Kit 1.7[7] — make sure that `JAVA_HOME` is pointing to your JDK installation.
- Maven 3[8] (3.1 recommended)
- Git[9]
- Eclipse 4[10] (4.3 recommended)

[7] `http://www.oracle.com/technetwork/java/javase/downloads/`
[8] `http://maven.apache.org/download.cgi#Installation`
[9] `http://git-scm.com/downloads`
[10] `https://www.eclipse.org/downloads/`

3.3 Checking out and maintaining the code

Note that this section and Section 3.4 assume use of a command line-based environment; use of the Eclipse IDE follows after.

The COMPASS Git repository for the Symphony IDE is at

`https://github.com/symphonytool/symphony.git`

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 local repository cloned and the appropriate branch checked out, 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 (see guidelines in Section 3.1):

```
git merge origin/development
```

By keeping your personal work branches synchronized with `development`, it is much easier for the release manager to merge your work into 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 fetches all of the dependencies and builds all of the core modules. The first run might take upwards of 25 minutes as there are many dependencies to fetch.

After you have successfully built the core modules (see Figure 2), change into the `ide` folder to build the IDE modules:

```
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 use the standalone executable for the Symphony IDE in the following folder: `PROJ/ide/product/target/products/`. Please note that, by default, Maven will only build the IDE for your current platform (Windows, Linux, or MacOSX). If you want to build all versions of the tool, use the following command:

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

```
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] The Symphony IDE Top-Level ......................... SUCCESS [  0.719 s]
[INFO] Symphony Tools Top-Level ........................... SUCCESS [  0.028 s]
[INFO] Symphony Core Top-Level ............................ SUCCESS [  0.034 s]
[INFO] Symphony Core CML Abstract Syntax Tree ............. SUCCESS [ 17.776 s]
[INFO] Symphony Core CML Parser ........................... SUCCESS [ 12.972 s]
[INFO] Symphony Core CML TypeChecker ...................... SUCCESS [ 15.281 s]
[INFO] Symphony Core Analysis Top-level ................... SUCCESS [  0.025 s]
[INFO] Symphony Core Analysis Proof Obligation Generator .. SUCCESS [ 34.030 s]
[INFO] Symphony Core Analysis Theorem Prover .............. SUCCESS [ 17.617 s]
[INFO] Symphony Core Analysis Refinement Tool ............. SUCCESS [  0.066 s]
[INFO] Symphony Core Analysis RTT-MBT Interface ........... SUCCESS [  0.706 s]
[INFO] Symphony Core Analysis Model Checker ............... SUCCESS [  4.611 s]
[INFO] Symphony Core CML Interpreter ...................... SUCCESS [02:12 min]
[INFO] Simple SysML State-Machine to CML translator ....... SUCCESS [  1.171 s]
[INFO] Symphony Core CML Command Line Tool ................ SUCCESS [ 21.158 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 04:18 min
[INFO] Finished at: 2014-07-23T14:33:41+02:00
[INFO] Final Memory: 58M/425M
[INFO] ------------------------------------------------------------------------
```

Figure 2: An example successful maven build for the core modules

from either the IDE folder or from the root folder together with `-PWith-IDE`.

Another important command 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 cause of build failures are test failures. For code that is to be integrated into the main `development` and `master` branches there must be no test failures. However, if the code is not yet ready for integration, it can still be useful build the code regardless of the test status. This is done by the following command:

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

3.5 Importing Projects into Eclipse

In order to develop the Symphony tool, you must import its source code into Eclipse. To do so, you first need the m2e plug-in.

1. In Eclipse go to *Help* → *Install new software*
2. The update site for the plug-in is:
   `http://download.eclipse.org/technology/m2e/releases`
   (see Figure 3).

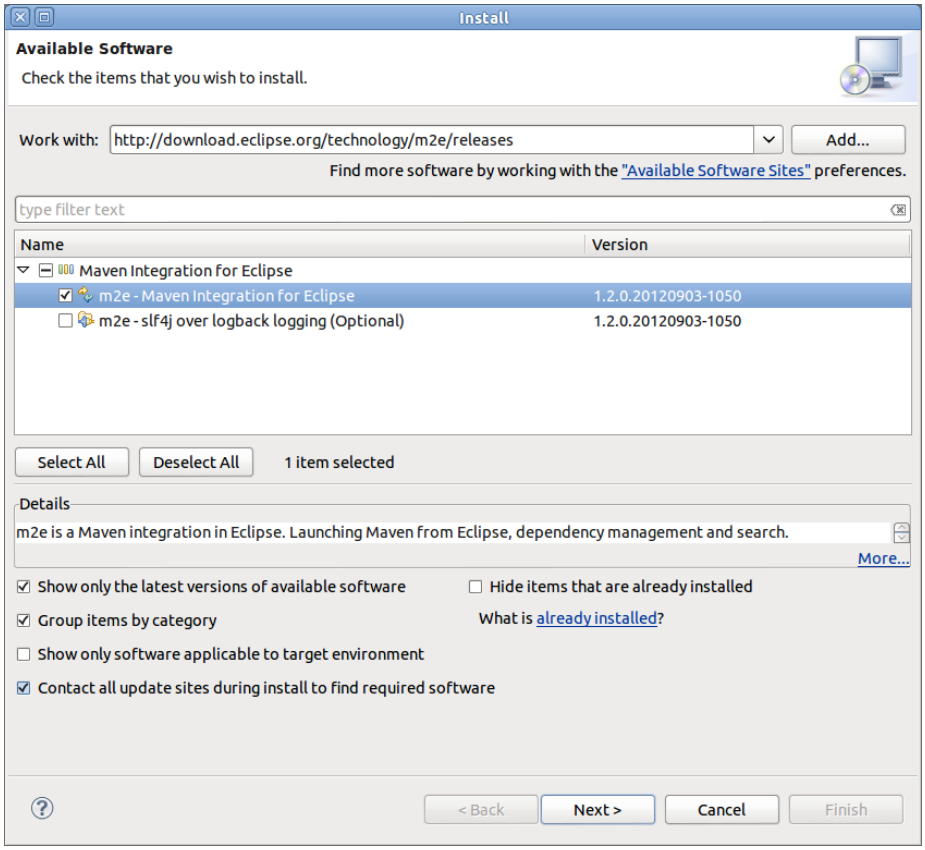


Figure 3: Installing the m2e Eclipse plug-in

3. Import Maven projects into Eclipse: *File* → *Import* → *Maven* → *Existing Maven Projects*. Only import the "leaf" projects of the listing as the "non-leaf" are empty (see Figure 4).

   While we suggest importing all "leaf" projects, the minimal set of projects to import is:

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

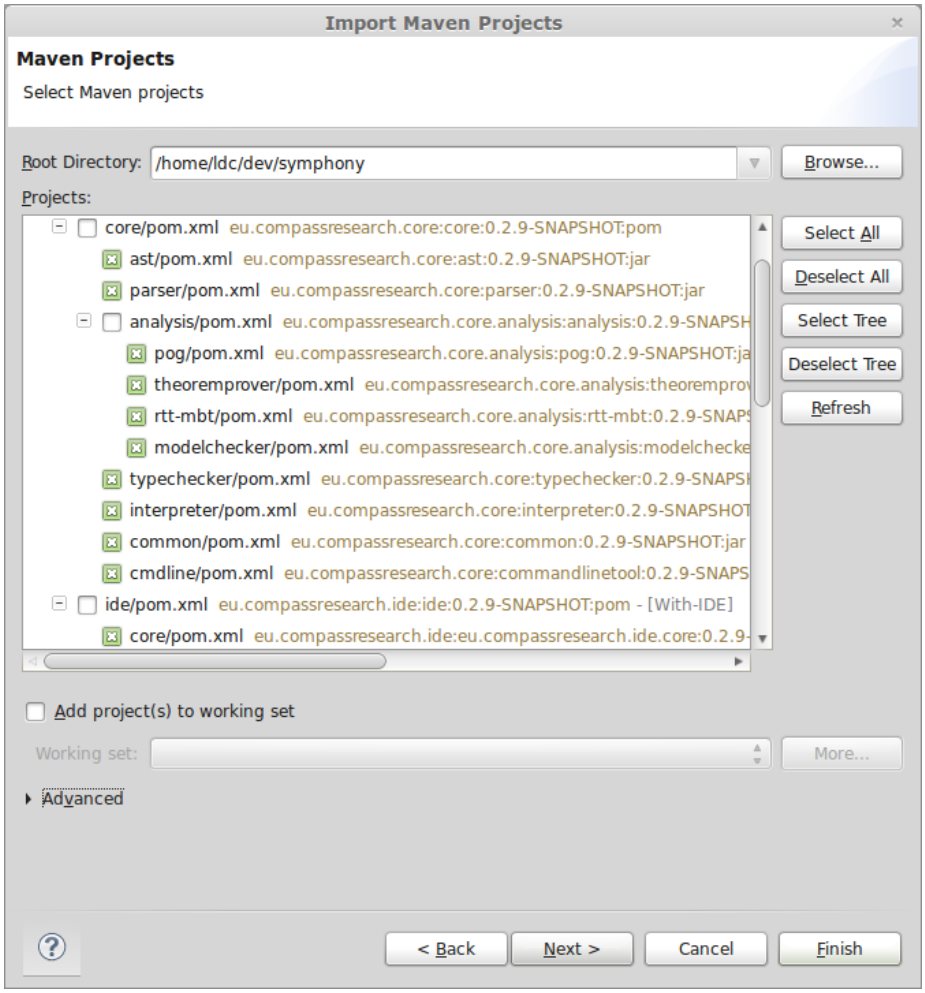


Figure 4: Maven projects to import into Eclipse

4. During the import, you may need to install additional m2e Connectors. Eclipse will prompt you to so (see Figure 5).

3.6 Special Eclipse Dependencies

The Eclipse dependencies that the IDE plug-ins require are handled by the Maven Tycho plug-in.[11] However, during a Maven build, Tycho resolves the dependencies differently than Eclipse does within the IDE. As a result, additional plug-ins in your Eclipse installation to allow Eclipse to find dependencies correctly.

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

[11] This is the maven plug-in for building Eclipse-based software, information is available at `http://eclipse.org/tycho/`.

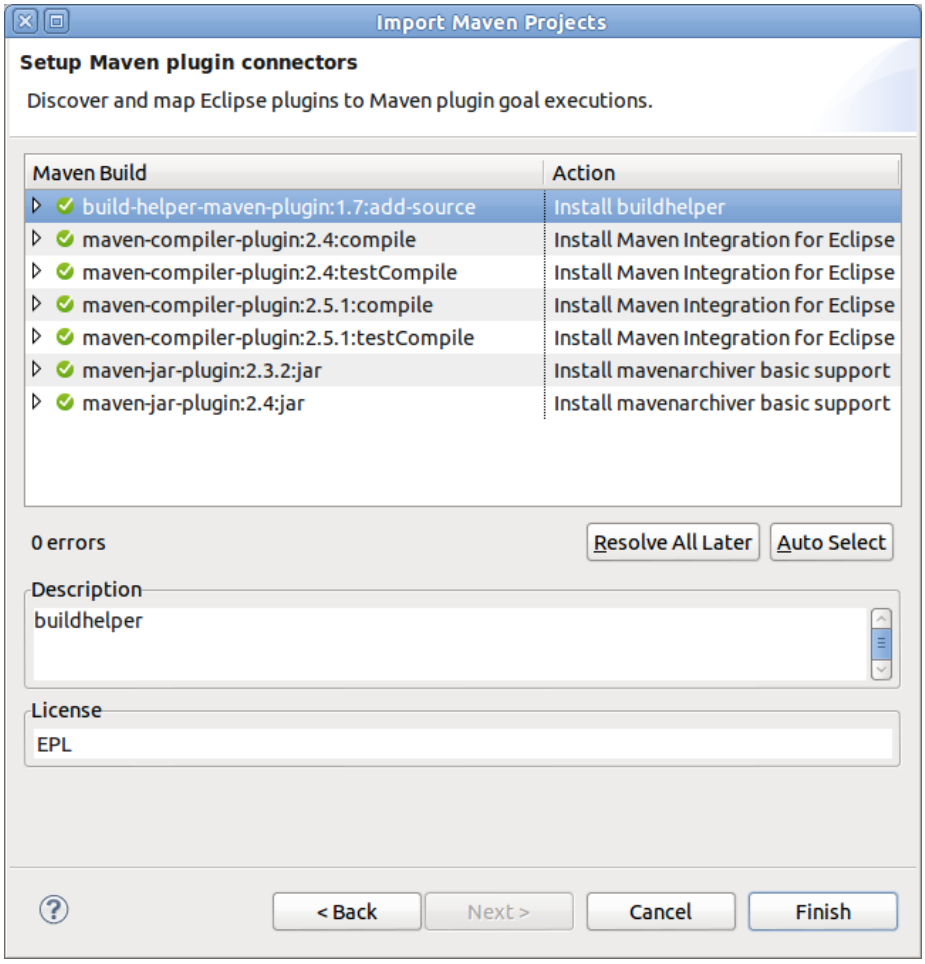


Figure 5: Eclipse prompts the user to install additional m2e Connectors

- Overture (mandatory):
  `http://overture.au.dk/p2/symphony-overture-development/`
- Scala IDE (needed for Theorem Prover Core Module):
  `http://download.scala-ide.org/sdk/helium/e38/scala210/stable/site`
- Isabelle-Eclipse (needed for the Theorem Prover IDE Plug-in):
  `http://overture.au.dk/p2/isabella-eclipse/`

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*). Overture is frequently updated so it's a good idea to check the plug-in on a regular basis.

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 the Scala IDE is already included. Contact the active members of the Symphony development team at GitHub if help is required to set this up.

3.7 Running in Tethered Environment

The Eclipse IDE allows RCP projects, like Symphony, to be run from within the Eclipse environment. Doing this causes the Symphony IDE to appear as though it were launched independently, but it retains a connection to the Eclipse debugger such that the Symphony IDE's runtime behaviour can be debugged like any other Java application.

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 *Plug-ins* 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*.

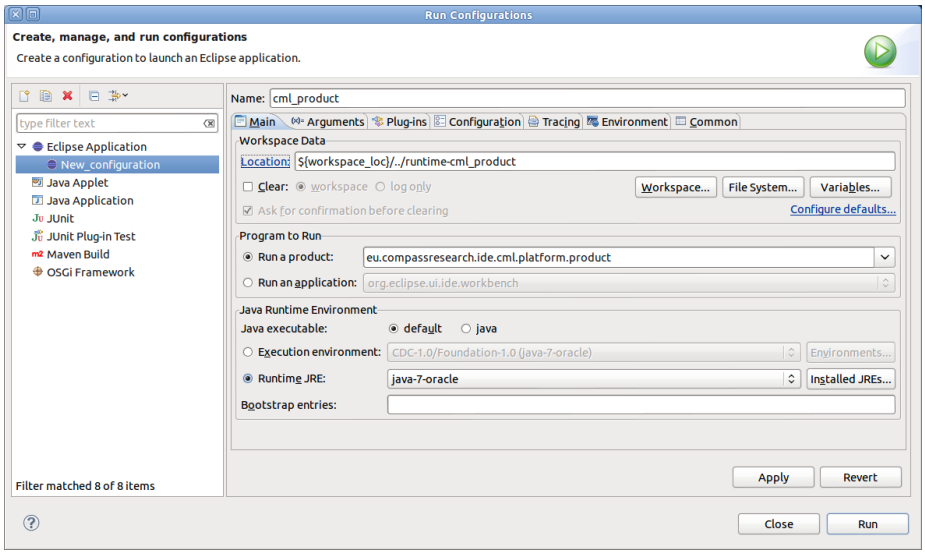


Figure 6: A new run configuration for the Symphony IDE

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, perform the following steps:

1. Run *Maven* → *Update Project* again.
2. The jar error from Section 3.8.5 will appear again; click *Ok*.
3. A series of type errors (package does not exist, etc.) will appear. Run *Maven* → *Update Project* again, but this time untick *Clean projects*.

Bulk updates can cause projects to constantly cycle through the various error stages. As such, it is recommended that the update process be performed on one project at a time.

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.

It is recommended that developers doing Scala work use the standalone IDE instead.[12] This also has the side-benefit of giving you separate and cleaner work environments.

It is may also be necessary, after performing the Maven import, to manually add the Scala nature to the theorem prover projects (Context Menu → `Configure` → `Add Scala Nature`).

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 unresolved 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.

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

3.8.5 Jar is not on its project's build path

This error can periodically appear (see Figure 7) 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 two or three times.

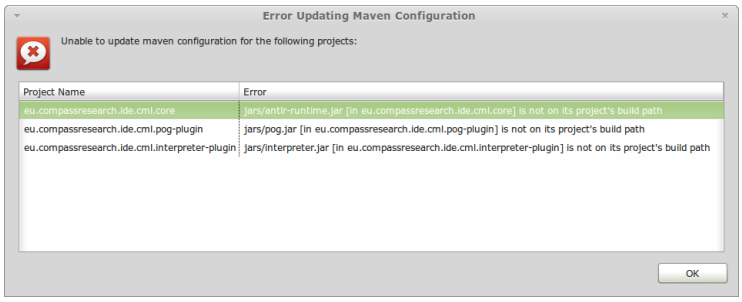


Figure 7: The "jar is not in its project's build path" error

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 does not consistently detect 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 folders 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 completely reliable. It will, on (a very rare) occasion, throw up various compilation errors. These errors will usually be related to the Java classes being used in the Scala code. For example, the following sequence of commands may sometimes trigger the error.

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

The build will fail on the `Isabelle IDE Plug-in`. We have not yet identified the cause of this problem. But we have found a fairly reliable workaround. Simply run `mvn install -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 fails to do so and instead continutes to use the one in the local repository, resulting in various errors when building with Maven.

In theory this should not happen but when you have errors that you cannot trace or fix, it is 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 8 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.

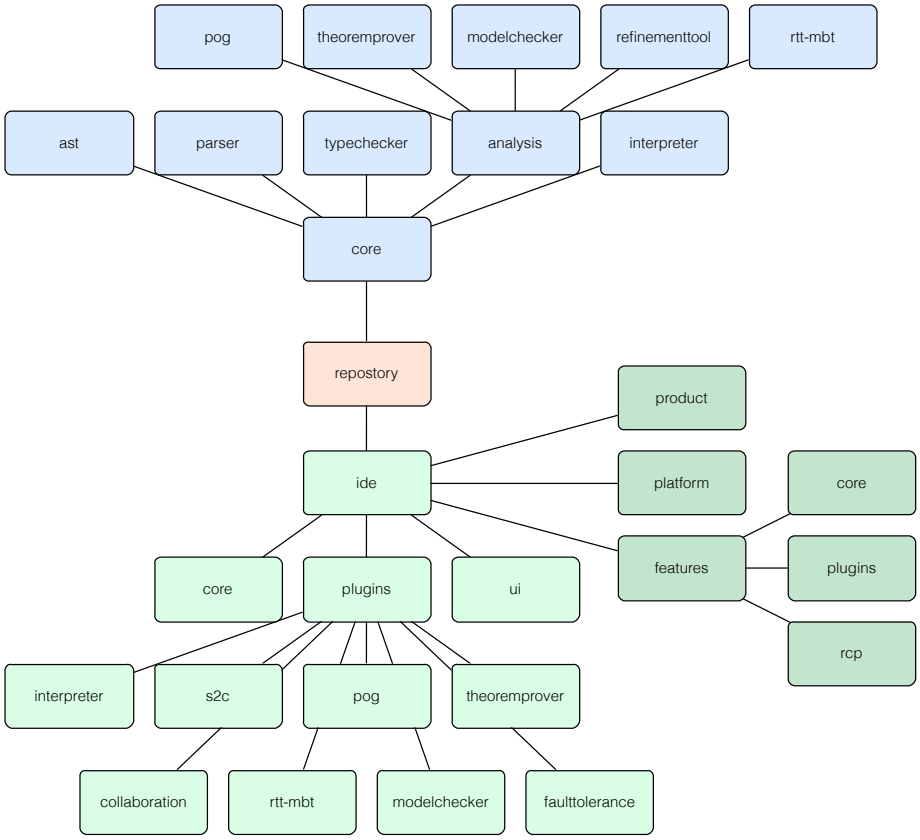


Figure 8: Main projects in the Symphony repository

4.1 Core

The core of the Symphony IDE contains the basic elements of the toolset that are independent of any particular user interface. As their name suggests, they provide the core functionality of loading, validating, and analysing CML files. The Symphony core is responsible for the handling of CML files which involves parsing them and typechecking the resulting AST to ensure its overall consistency. The AST is then used by the various Symphony core plug-ins and the CML interpreter.

4.1.1 The CML AST

The AST represents the CML model that is being worked on. The AstCreator tool is part of the Overture platform and is used to automatically generate the AST for all of the VDM dialects supported by Overture. It is also used to generate the AST for the CML dialect via its extension feature, and it uses the VDM AST as the basis for the CML AST. Note that the AstCreator tool may also be used by plug-in developers to create ASTs for other grammars that are translated to or from the CML AST.

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. 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 them. Note that since the code for the AST is automatically generated, it should not be modified directly.

The main interaction most developers will have with the AST will be done through the visitor pattern [GHJV95]. This patterns allows for separation between a piece of functionality and the data structure it operates on. A visitor implementation will consist of a series of `caseX(...)` methods, where `X` corresponds to a specific AST node and the method simply provides the desired functionality for said node.

Along with the AST files, the AstCreator tool provides a set of Java class files that implement the basic AST visitors within the correct Java packages. These visitors provide basic traversal and method dispatching across the AST and can be subclassed to implement various different kinds of analysis. Plug-in developers should make as much use as possible of the visitors when implementing their desired functionalities. The AST package (along with the visitors) is found in the blue box labelled “ast” in Figure 8.

4.1.2 Parser

The parser (blue box labelled “parser” in Figure 8) is responsible for reading CML files and building the internal representation of the contained model as an AST. It is built using the ANTLR 3.5 parser generator, however, most of the error recovery features are presently disabled in favour of a fail-on-first-error approach. This means that all syntax errors must be resolved before the Symphony IDE will proceed to typecheck a model.

4.1.3 Typechecker

The typechecker (blue box labelled “typechecker” in Figure 8) 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.

4.1.4 Interpreter

The CML interpreter allows for CML models to be executed to show their behaviour. It is a top-level module –the corresponding box is the blue one labelled "interpreter"– rather than an analysis plug-in as its function is execution rather than analysis.

4.2 Analysis Plug-ins

The Symphony IDE was designed as a plug-in-based architecture. Besides the core modules, each of the tool's major features are provided by separate independent plug-ins. Each plug-in 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 is that it allows each plug-in (and individual features) to be independently developed. Thus, the various plug-ins can be implemented in parallel, at different sites both by members of the COMPASS project and third party contributors. This approach 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 four analysis plug-ins: the Proof Obligation Generator (blue box labelled "pog" in Figure 8), the theorem prover core ("theoremprover"), the model checker ("modelchecker"), the RT-Tester core ("rtt-mbt"), and the Refinement Tool ("refinementtool").

4.2.1 Proof Obligation Generator

This section describes the structure of the POG as an example of how plug-ins may be constructed for the Symphony IDE.

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. These objects contains information to recreate the context required for the PO expression. The structure of the POG is depicted in Figure 9.

The Overture package `org.overture.pog.obligation` and similarly-prefixed `....contexts` contain classes that extend 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.

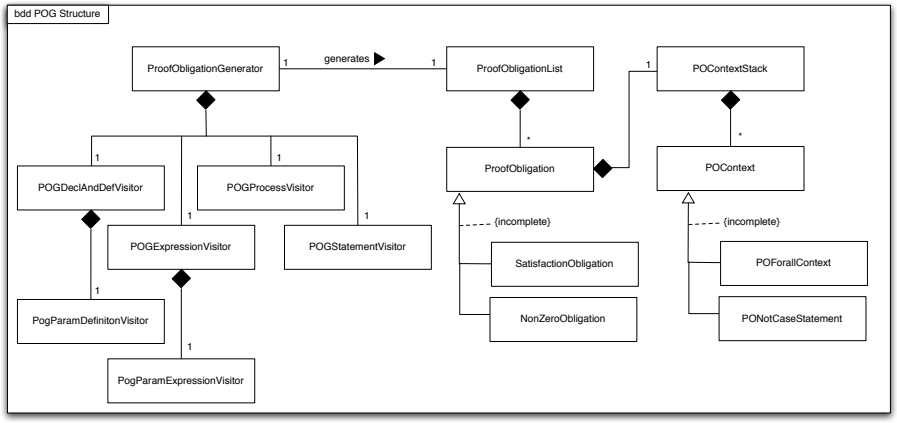


Figure 9: Block definition diagram depicting the main elements of the proof obligation generator

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 plug-ins (particularly the Theorem Prover).

Behaviour

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 are definitions, the POGDeclAndDefnVisitor checker processes the nodes 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.

4.3 User Interface

The structure of the user interface consists of two main groupings under the "ide" box in Figure 8. The lighter green boxes –"core", "ui", and everything below "plugins"– represent the main code for the IDE, while the darker green boxes – "product", "platform", and everything to the right of "features"– correspond to the build structure that is required to use Eclipse as a tool platform.

All of the user interface code (the various Eclipse IDE plug-ins) is packaged under `eu.compassresearch.ide`. These Eclipse plug-ins should hold all user interaction code and provide controls over the actual functionality packages in the core. For example, the Proof Obligation Generator (POG) is split into two sets of packages. The core functionality code is under `eu.compassresearch.core.analysis.pog`. The UI code on the other hand is under `eu.compassresearch.ide.pog`. The UI plug-in should connect into the main Symphony Eclipse IDE while also using the core code. In this way, we get a complete separation between interface and functionality and it should be possible to change the interface from Eclipse to some other IDE platform should it become necessary, by building only another UI plug-in.

There is, however, one exception to separation of concerns: 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 expose core Symphony functionality to command line users. If there is a need to use Symphony without the Eclipse IDE (perhaps as an RCP) then the command-line makes the basic functionality of the tool 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 worth noting that there may be some confusion with the usage of the terms "core" and "plug-in" 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 divided the various functionalities into either core functionalities (required by all others) and plug-in functionalities (independent from each other).

On the other hand, when we use the terms core and plug-in, or better yet UI plug-in, we are not talking about the functionalities of Symphony as a whole but rather a single functionality. Each functionality of Symphony has a core component (that actually implements the functionality) and a UI plug-in (that 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).

4.3.1 Proof Obligation Generator IDE Plug-In

This section describes the structure of the POG IDE plug-in as an example of how to construct an IDE plug-in for the Symphony IDE.

The POG IDE plug-in 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 plug-ins.

Classes

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

In order do run the POG, one should instantiate the `PogPluginRunner` class and invoke the `runPog` method. This will execute the POG on whatever project is currently selected in the CML Explorer. Note that the POG will always analyze the entire CML model contained in the project.

Also of note is the class `PogPluginConstants` which holds the identifiers (ids) for various views and other elements of the POG Plug-in. It is better to centralize these ids as static string constants in a class than to have them spread around the code since it increases maintainability and reduces duplication. 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 Plug-in offers only one command to users: *GeneratePOs*. This command is associated with the CML Explorer's context menu via the `plugin.xml` file. The handler for the command is the `GeneratePOsHandler` class. It simply initializes the `PogPluginRunner` class and executes the `runPog` method. Because the command is only shown when a CML project is selected in the explorer, there is no need to control its availability. Also, before executing the POG, the `PogPluginRunner` calls the parser and typechecker on the model (via `CmlProjectUtil.typeCheck`) and does not proceed if there are any errors in the model.

Views and Perspectives

The POG plug-in provides a single perspective (*pog-perspective*) which is defined in the `plugin.xml` file. This perspective adds two views to the default CML perspective: `POListView` and `PODetailView`.

The `POListView` implements a table that lists 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 takes a `IProofObligationList` and a `ICmlProject` as parameters.

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

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 in a new plug-in.

5.1.1 AST

The AST is one of the largest 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 directly edited. The AST itself is defined in the `cml.ast` file, and therefore any changes to the AST should be done in the `cml.ast` file. However, changes to the AST should never be made without discussion and consent of the other developers. For the sake of demonstration and discussion with other developers, it is fine to make changes in a separate branch in the git repository, with the understanding that any changes made therein may not be adopted.

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 an argument to the `apply` method of the node. A visitor can be created by extending one of the existing versions in order to implement the particular functionality needed. There are currently five predefined visitors:

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 it is necessary to extract and return information.

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.

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. Note that the previous visitors do not implement any kind of traversal.

5.1.2 IDE Utility

The Symphony IDE has a series of classes that provide a mapping between CML files and their respective ATSs as well as some other utilities. They can be used by the various plug-ins to get selected projects and their respective ASTs. Developers are

encouraged to familiarize themselves with these classes and their documentation. Some of the main classes and functionalities are:

ICMLProject: represents a CML project in Symphony. It provides a representation of the project and all its files. Particularly relevant is the 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 commonly used functionalities such as typechecking a project.

Adapters: Eclipse makes extensive use of adapters[13] to handle conversion between the various kinds of project representation. 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 `plug-in.xml` file for the Symphony IDE core project.

5.2 Core plug-ins

This section covers the various steps necessary to build a Symphony plug-in and integrate it into the main toolset. It assumes that the plug-in functionality (e.g., compass-research.core.analysis.pog) is already implemented and available 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 the project. This allows 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: identifies the super package where the plug-in will go.

artifactID: identifies the main package of the plug-in.

dependencies: one of the most important groups. It should typically include the core Symphony libraries and possibly the Overture ones as well. It should also add anything else that the plug-in requires. Listing 1 shows a template for dependency entries.

build: allows for fine-tuning of how Maven builds the project. For example, it can be configured to ignore test failures and keep building.

[13] See `http://www.eclipse.org/articles/article.php?file=Article-Adapters/index.html` for more about Eclipse adapters.

```
<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.

Once the POM has been constructed, if the project is to be built as part of the Symphony Core Modules, it must be added to the `core/analysis/pom.xml` file under `<modules>`.

5.3 Eclipse plug-ins

If a user interface is necessary for the plug-in, it is highly recommended that the a separate UI plug-in is built. This is a separate Eclipse project with its own set of configuration files.

The POM for a UI plug-in 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, the update sites to any extra dependencies of the plug-in must be provided. The main IDE POM already provides update sites for Eclipse and several related dependencies, but if the plug-in has additional ones, they must be added. If there is no update site, then the dependency should not be handled through the UI plug-in but rather the core module (as a normal Maven dependency).

In addition to the update sites, the build rules for the UI plug-in must be configured to ensure the core component is added as a library. This can be done by adding following entry to the UI plug-in 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>
            <outputDirectory>${basedir}/jars</outputDirectory>
            <overWriteReleases>true</overWriteReleases>
            <overWriteSnapshots>true</overWriteSnapshots>
            <overWriteIfNewer>true</overWriteIfNewer>
            <stripVersion>true</stripVersion>
              <!-- The plug-in's core component  -->
              </artifactItems>
                <groupId> </groupId>
                <artifactId> </artifactId>
                <version>${project.version}</version>
```

```
                <type>jar</type>
                <overWrite>true</overWrite>
              </artifactItem>
            </artifactItems>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

This ensures that the relevant core component gets packaged in a jar and placed in the `jars` folder of the plug-in. This means that the build must go through maven every time a change in the core component is made for it to be reflected in the UI plug-in. Therefore, it is usually easier to develop the UI plug-in after the core component is mostly stable.
If the UI project is to be built as part of the Symphony IDE, it must be added to the `ide/plugins/pom.xml` file under `<modules>`.

5.3.1 Feature

Features are an Eclipse mechanism for organizing plug-ins. Symphony makes use of them so there must be a corresponding feature project for the plug-in under `ide/features/plugins/`. A feature project consists mostly of stub files[14]:

pom.xml allows Maven to build the feature. It needs only identify the feature and its parent and configure the packaging as an Eclipse feature (`<packaging>eclipse-feature</packaging>`).

feature.xml used to associate the feature with the respective plug-in (via a `<plugin>` entry).

build.properties is a list of included configuration files. Typically, the default entries for this file are sufficient.

feature.properties configures various attributes of the feature such as description and copyright notice..

Once the feature project has been created, add it at as an entry to the following files:

- `ide/features/plugins/pom.xml` under `<modules>`
- `ide/features/rcp/feature.xml` as a new `<includes>` entry
- `ide/product/category.xml` as a new `<feature>` entry

5.3.2 Manifest Files

In addition to the POM (used for Maven builds), the UI plug-in must be configured to run with the Symphony IDE. This is done by editing (or creating) the `MANIFEST.MF` file 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 the UI plug-in
- `Bundle-SymbolicName`: the main package of the UI plug-in
- `Bundle-Activator`: the activator class for the UI plug-in. This class must implement `BundleActivator`.

[14] Inspect existing Symphony features for more details on the necessary files

- `Import-Package`: the dependencies for the UI plug-in
- `Bundle-ClassPath`: files to extend the UI plug-in's classpath. It usually includes `.` as well as `jars/*.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 plug-in) is shown in Section A.3. The most crucial aspect of the Manifest is to ensure the dependencies are correctly defined. If there are unexpected `class not found` errors or something similar, it is likely that there are missing dependencies.

6 Simulation and Co-Simulation

This section describes a high-level development overview of the CML simulator in Section 6.1, and then gives the overview of how co-simulation works with respect to the simulator in Section 6.2. The second portion is relevant to developers who wish to connect tools to the simulator without having to integrate them into the simulator; the first portion is provided to aid developers' understanding of the overall execution model in these cases.

6.1 Technical Overview of the CML Simulator

This section describes how the CML simulator is structured internally and how it simulates CML specifications, though this does not include a detailed description of every language construct. The intent is to demonstrate, at a high level, how processes from a CML specification are represented in the simulator, and how they are simulated. The example shown in Listing 2 forms the basis of the explanation; it represents a small system (P) consisting of two parallel processes A and B. If time ($tock$) is ignored then only a single execution trace exists, specifically being $\langle c.1 \rangle$ where process A offers to synchronise on $c.1$ and where process B requests[15] a value on channel $c?x$.

The CML simulator uses two different approaches to simulate CML language constructs. Processes and actions are animated using an "inspect and execute" strategy, where the simulator first inspects processes and actions to construct a collection of next possible steps that could be taken. The simulator then selects a construct to execute and performs a re-inspection to select the next construct to execute. The second approach is used for all expressions which do not return the next language construct to execute, but instead execute instantaneously during inspection. This means that they are skipped, from the perspective of a step.

During simulation processes and actions are represented using an abstraction we call *behaviours*. Behaviours are derived from walking the internal simulation representation of the CML model, and are the result of inspecting and executing the language constructs. This means that every process construct is encapsulated in a behaviour, shown as circles in Figure 10. A behaviour may have children that represent nested behaviour

[15] $c?x$ may be thought of as an external choice of of all the possible events on channel c, so B does not strictly make a "request" to A.

```
channels
  c : nat

process P = A [|{c}|] B

process A = begin
  @ c.1 -> Skip
end

process B = begin
  @ c?x -> Skip
end
```

Listing 2: Parallel system P of A and B.

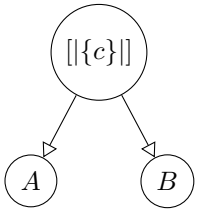


Figure 10: Internal simulator representation of P from Listing 2.

in terms of processes or actions. A behaviour serves as a controller for its children, for whom it filters inspection results and delegates execution instructions.
An example of the system P, shown in Listing 2, is illustrated in Figure 10. It shows process P, which is a parallel composition of processes A and B synchronizing on any event on channel c. The creation and inspection behaviour is illustrated in Figure 11.

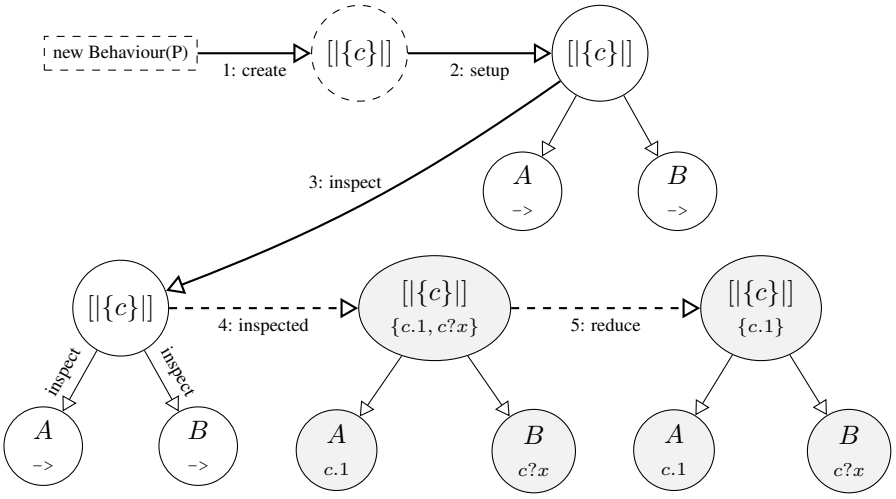


Figure 11: A simulator behaviour, showing creation and inspection.

The process definition P is turned into a behaviour that is then configured and inspected to obtain possible events. When an event is selected it can be executed as shown in Figure 12. The setup and inspection shown in Figure 11 contains the following sequence of steps:

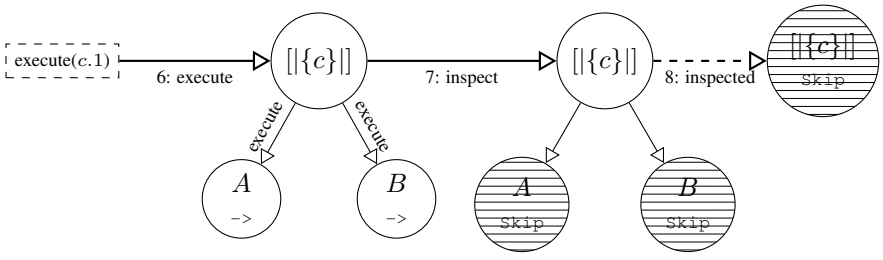


Figure 12: A simulator behaviour, showing execution and completion.

1. Creation of the behaviour from the process definition P.
2. Setup of the behaviour, possibly including the setup of child behaviors and so forth.
3. Inspection of the process. This calculates the next possible transition(s) that the behavior can make, this may include inspection of child behaviors. In this particular case the result depends on the result of its child behaviors and they are therefore inspected.
4. Inspected state. Inspection leads to an inspected state in the behaviour (represented in grey) showing the possible transitions obtained through inspection.

5. Reduction. Since the inspection in this case was performed on a alphabetised parallel composition of A and B, it must reduce the possible child transitions according to the rules of this operator. In this case the $c.1$ event is the only possible transition, so the result of the inspection of P collapses to just $c.1$.

The simulator will use the top behaviour, in this case P, to execute the selected event $c.1$. The behaviour language construct defines how to execute the event $c.1$ by passing it down to the respective children. When a child that represents an action process finishes execution the next inspect phase will change the internal language construct of that child. An action process can generally be considered complete when it reaches a state where the next language construct is a $Skip$ action, illustrated as a shaded areas in Figure 12.[16] If a behaviour has children then it will remove these based on the rules of the language construct it represents and eventually convert itself into a $Skip$ statement and thus become finished.

6.1.1 Process-level Implementation

This section gives an overview of how the simulator implements the principles shown in Figure 11 and Figure 12. The section starts by relating the elements from the figures to the equivalent static structures implemented as Java classes, then the implemented dynamic structure.

The implementation of the simulator implements the behaviour, shown as white circles in Figure 11, as a `Behaviour` class. The responsibility for the control flow is handled by the `Interpreter` class that uses an `ISelectionStrategy` to choose which event to execute if more than one event is available. It can be seen in Figure 13 that the `Behaviour` class has methods corresponding to the arrows in the previous figure. It also has fields that hold either its child processes (a left and right child for binary operators, or just a left child for unary operators) or a node that represents a language construct, such as communication (`->`) or `Skip`. The `Interpreter` class is responsible for implementing the overall simulation protocol that is defined by the following calls to the top behaviour: `inspect`, `isFinished`, `isDeadlocked`, and `execute`.

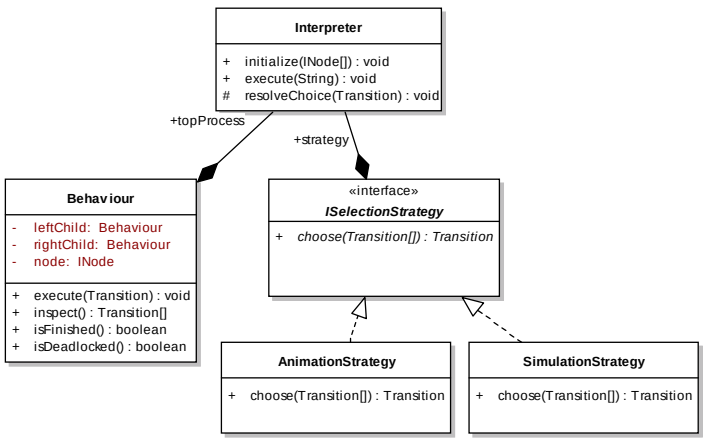


Figure 13: Class Diagram of the core classes in the simulator.

[16] Infinitely recursive processes never resolve to a $Skip$ action, as they never complete.

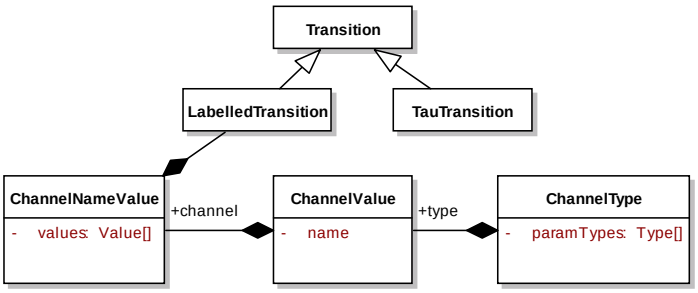


Figure 14: Class Diagram of the transition hierarchy.

The `execute` operation can only be called with a chosen `Transition` and so implies that the behaviour is neither finished nor deadlocked. The events illustrated in brackets within the grey circles in Figure 11 are represented in the implementation by the `Transition` class shown in Figure 14. There are at least two types of transitions where the `TauTransition` class represents internal hidden transitions and the `LabelledTransition` class represents the events from Figure 11. It is the latter transitions that are of interest for co-simulation, as tau transitions are not visible outside the process in which they happen. That these transitions are labelled means that they hold the name of a channel they synchronize on, and a type that may either be void or represent a list of types from the CML language. If a channel has a non-empty list of types then an event must also contain appropriate values that match the types. These values may then be either concrete values such as the value of `1` or `<MY_TOKEN>`, or they may be a special value type that instructs the interpreter to obtain the concrete value from the user.[17]
The dynamic behaviour of the interpreter is shown in Figure 15. It is a simplified representation of the full implementation but covers the essentials. The interpretation starts by invoking the `initialize` method of the interpreter, which creates the necessary initial simulation contexts and prepares the AST represented by the specification. This is then followed by a call to the `execute` operation, which does a full evaluation of the specification by using a process name. This name is then matched and the process definition that matched this becomes the *top* process that will be in control of the simulation. The interpreter then enters a loop that runs as long as the behaviour does not finish or deadlock. The loop starts by inspecting the behaviour and then uses the selection strategy to select one of the possible events the behaviour may return. The event is then executed in the given context and the loop guard conditions are updated before looping.
The inspection call of a behaviour is shown in detail in Figure 16, which outlines how the behaviour uses the language constructs to obtain their children and forwards the inspection calls to them.[18]

6.1.2 State-level Implementation

The previous section describes the mechanism by which the simulator handles the high-level structure of a CML model. However, processes and actions within a CML model

[17] The implemention for obtaining a value from the user has been omitted for clarity.
[18] Note that the apply call to `node` has been simplified for clarity.

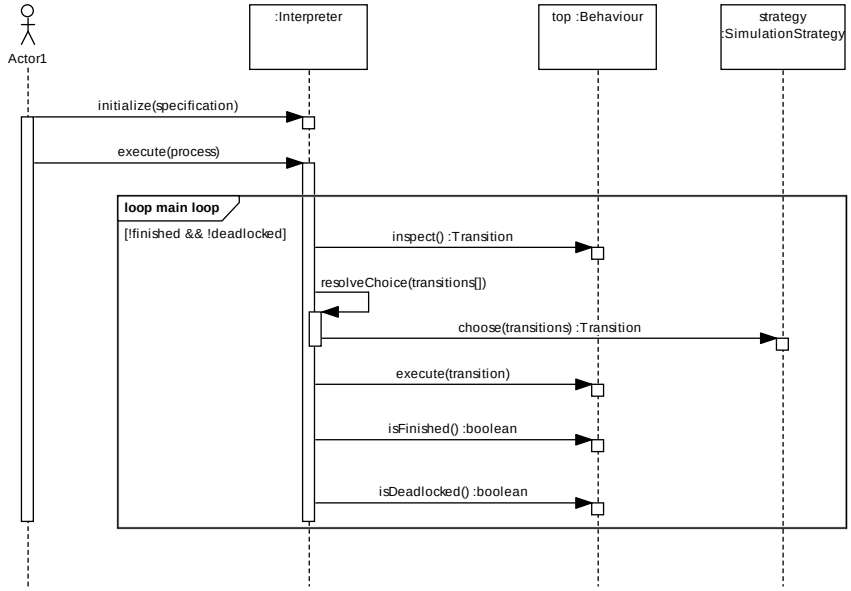


Figure 15: Sequence Diagram of the top level execution in the interpreter class.

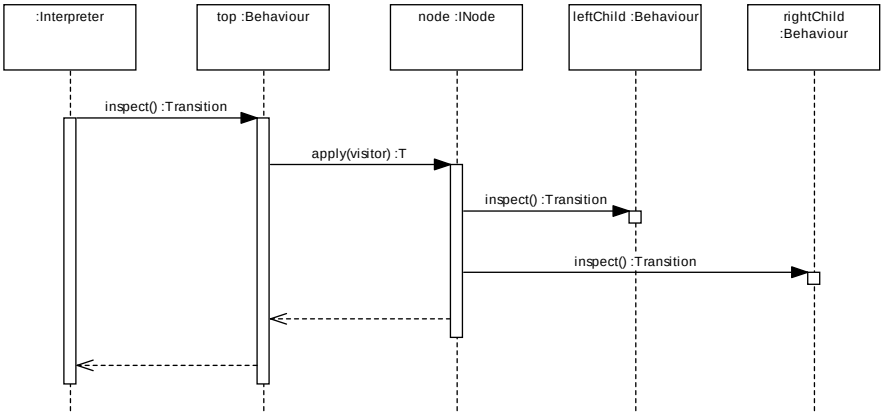


Figure 16: Sequence Diagram of the inspection of the top level behaviour.

are the means of giving the model's communicative and behavioural structure, but not the details of a model's internal operation and state. This is described using the portion of CML that derives from the VDM language.
Here the simulator makes direct reuse of the Overture platform: the CML simulator incorporates the Overture VDM simulator, and any CML language construct recognised as having dervied from VDM is simply handled by the existing mechanisms in the reused code. This includes operations, functions, almost all statements,[19] and all expressions.

[19]CML adds non-deterministic conditional and looping constructs that are not part of VDM.

However, the use of VDM constructs in CML is constrained, and some of the non-deterministic VDM constructs are not supported, in particular the non-deterministic "`let...be`" expression. This restriction means that certain patterns of modelling typically used in VDM models are, while possible, no longer executable in a CML model, as the semantics defined in COMPASS Deliverable D23.5 [BCW14] only allow a subset of VDM's constructs (including the considered removal of the "`let...be`" expression). Although it is still possible to write non-deterministic functions and operations with post conditions that require, for example, that the return value is any arbitrary member of some given set, it is no longer possible to directly execute these functions and operations.
To overcome this constraint, the Overture VDM simulator was extended to use the ProB constraint solver [LB08]. When the Overture VDM simulator encounters an implicit function or operation, it now translates the pre and post conditions into a form suitable for the ProB constraint solver and then uses the result of running ProB on it.
Due to the nature of the reuse that the CML simulator makes of the VDM simulator, this functionality is automatically available for CML models.

6.2 Technical Overview of CML Co-simulation

The term co-simulation will, in this context, mean the delegation of the simulation of a process to an external source. To illustrate this a small pseudo-specification is shown in Listing 3. The specification defines a top level process, P, that consists of the processes A and B, where the latter is composed of processes C and D.

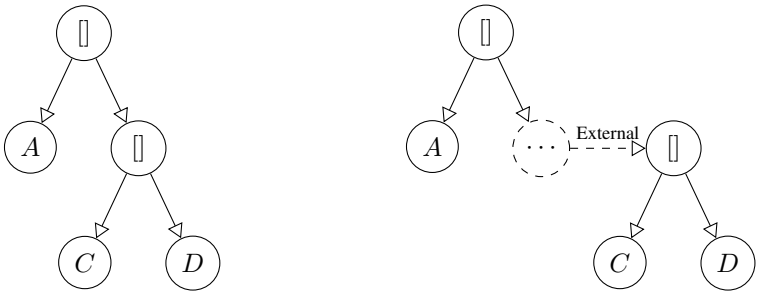
```
process P = A [|{c}|] B

process A = ...

process B = C [|{b}|] D

process C = ...
process D = ...
```

Listing 3: Parallel system P of A and B.

This specification can then be illustrated graphically for a standard simulator as shown in Figure 17a. However, if the process B were externalized and simulated by a remote simulator, then the tree would have to be modified as shown in Figure 17b. The solid lines in both figures represent method calls of the behaviour interface (described in Section 6.1.1). The dashed line represents the same interface accessed over a network connection using a common communication protocol.
In the following, Section 6.2.1 describes the fundamentals of the simulation network protocol, Section 6.2.2 describes how the simulator can be modified to enable behaviours to run externally from the simulator itself, and Section 6.2.3 describes how the simulator can be modified to run as an external simulator for a process, and also how external code can be connected as an implementation of an external process.

6.2.1 Simulation Network Protocol

The simulation network protocol enables the externalization of a behaviour by allowing the calls defined in the `Behaviour` class to be transmitted over a network connection



(a) Standard Simulator setup (b) Externalized Simulator setup

Figure 17: Illustration of the internal simulator behaviour setup.

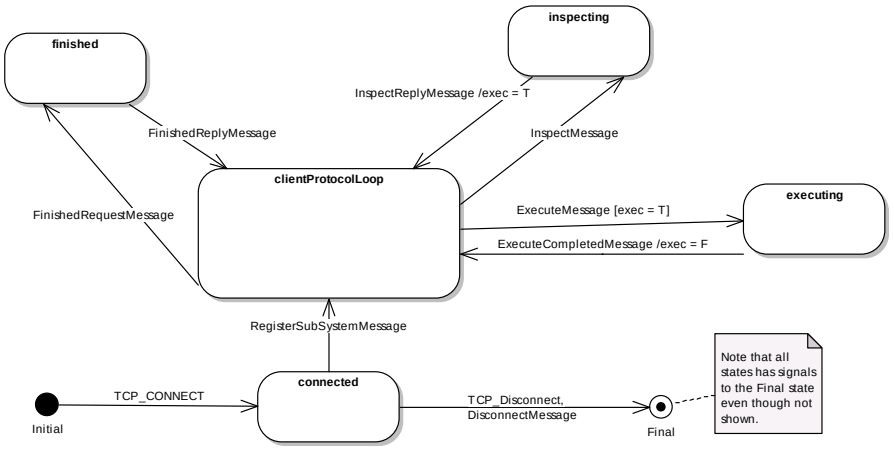


Figure 18: State Diagram of the external simulation protocol.

to a remote implementation of the behaviour. The underlying protocol chosen is Transmission Control Protocol (TCP) [Pos81], because of its reliability and ability to accurately deliver messages. The TCP protocol only handles streams of bytes and therefore a data transmission format must be used to allow the Java implementation of the simulator to communicate with external systems, such as software implemented on different platforms. The JavaScript Object Notation (JSON) [Cro06] language was chosen, to represent the protocol messages for the communication of state and `Transition` values across TCP connections. The format for the different JSON objects is describes in detail in Appendix B.

The protocol handling is described in Figure 18, showing when it is allowed to transmit various messages. The protocol complements the control flow described in Figure 15, and includes an extra control flow state *executing* that only allows an execution message if an inspection has previously been processed. The protocol also has a message to register which process the external system supplies to the coordinator before the client protocol loop starts. It must also be noted that the protocol allows a client to receive a disconnect message or disconnect itself from the TCP connection at any time.

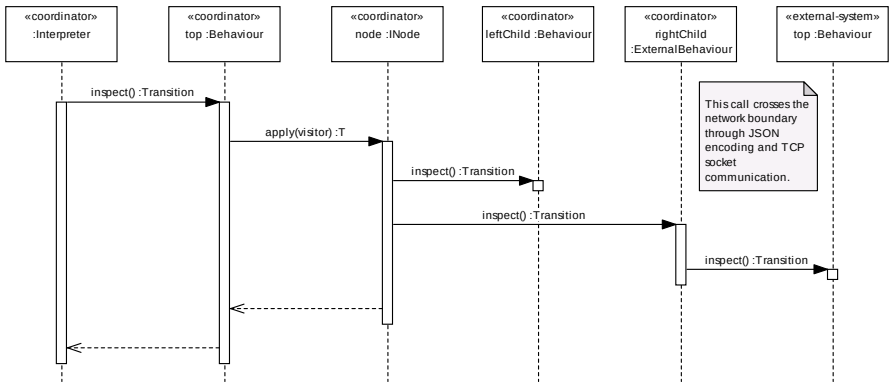


Figure 19: Sequence Diagram of the inspection of the top level behaviour when using external behaviours.

6.2.2 Process Behaviour Delegation (Coordinator)

The simulator has been extended to allow processes to be externalised; however only minimal changes were required to the simulator itself. A new sub class of `Behaviour` that delegates calls to a remote client using the protocol from Section 6.2.1 is needed. A server setup is necessary to enable the simulator to listen on sockets and registering connecting clients such that they can be linked with the `ExternalBehaviour` instances that represents the external processes. The sequence diagram shown in Figure 19 extends Figure 16 with the external behaviour for the `rightChild` in the form of a remote simulator. The external implementation may either be a remote CML simulator as shown in the figure or a custom-developed system implementation, as long as they implement the same protocol as in Section 6.2.1.

6.2.3 Process Behaviour Contribution (Clients)

A process behaviour contribution is represented by either a simulator implementing the client version of the protocol from Section 6.2.1, such as the Symphony IDE, or as a custom implementation in any programming language capable of communicating using TCP. A remote simulator is a simulator that has been modified such that the control is delegated to a coordinator. This section explains how the existing simulator has been modified to allow it to act as a remote simulator. The extended implementation of a system (a process) may also be used for external process simulation. This is achieved by mapping CML events to state changes and method calls in the system. A short overview of this is also provided here; for a more detailed description see [LL14].

Remote Simulator The remote simulator is based on the standard simulator but has a client implementation of the protocol starting its simulation by registering its contribution in the coordinating simulator. Then it follows the execution flow shown in Figure 20. It is similar to Figure 15 but has the main execution loop changed such that it automatically executes internal silent transitions (i.e. `TauTransition`) until none are available. When all available silent transitions are executed then it calls the `Client`, fetching the available transitions from the coordinator, which only ever consists of a single transition. The simulator has another thread the handles the protocol,

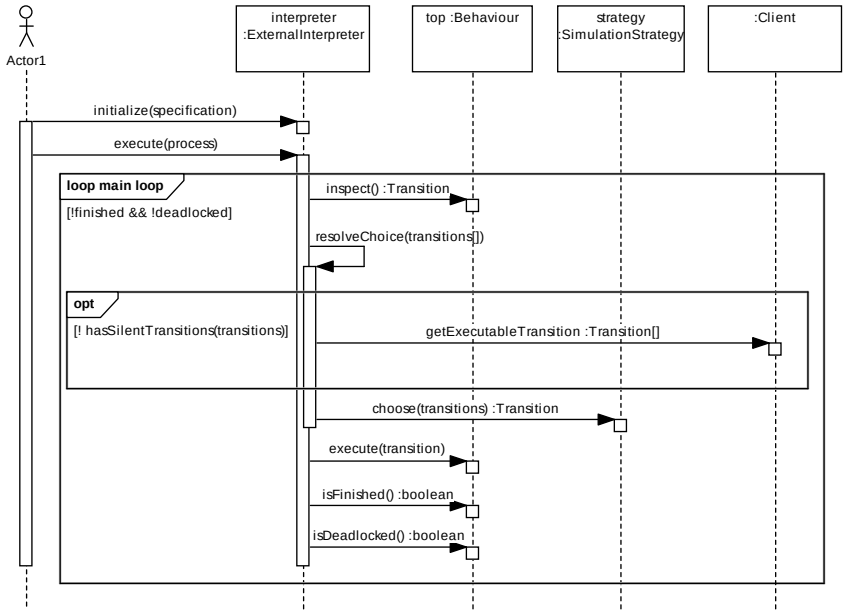


Figure 20: Sequence Diagram of the top level inspection for the remote simulator being connected as a client.

and it is this thread that explicitly calls `interpreter.inspect()` to supply the coordinator with inspection transitions upon request.

External System Implementation An external system implementation may be an implementation of a subsystem in any programming language that is capable of TCP communication. The system must implement the protocol described in Section 6.2.1 in the same manner as the remote simulator. The implementation of the protocol consists of a mapping from the type representation of the concrete programming language to the types used in the simulation interface being CML types described in JSON, and a mapping from the protocol messages to the internal state machine of the external system itself.
As a result of this design, the *inspect* message must be answered with a list of transitions that represent the possible events that are allowed from the current state, and a mapping from the *execute* message to a state change of the system state. These state changes may then lead to a change of events being returned from the subsequent inspection message.
There is a subtle difference in behaviour between the CML simulator and an external implementation. The simulator, run remotely, will never report as available events that are not expected. An implementation may, however, report all possible events as available, but report errors on those that are not expected.
A concrete example of how such mappings can be achieved is described in [LL14], which also presents a framework for managing the type and event mappings.

7 Conclusion

The Symphony platform developed in the COMPASS project is structured to allow for its extension with new features and functionality. This has been used by the various plug-ins made within the project for the Symphony IDE and, though they are integrated into the main tool, it would be possible to distribute them separately from it. This document provides the information necessary to develop further extensions, and serves as a starting point for developers who wish to work on Symphony itself.
The evolution of this document relative to its previous iteration in Deliverable D31.3b [CMCP13] has been relatively minor, with most changes concerned with keeping this document synchronised with the current state of the code and development environment. However, Section 6 is a new addition, giving an explanation of the simulator and the co-simulation protocol developed to interact with it; this provides a new avenue to interact with the Symphony IDE, and provides an alternative way to integrate with the Symphony IDE that does not require integration into the underlying Java virtual machine that it runs on.

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>0.3.5-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.overturetool.core</groupId>
      <artifactId>ast</artifactId>
      <version>${overture.version}</version>
    </dependency>

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

    <dependency>
      <groupId>org.overturetool.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 Plug-in 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>0.3.5-SNAPSHOT</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>

  <developers>
    <developer>
      <id>ldcouto</id>
      <name>Luis Diogo Couto</name>
      <email>ldcouto@eng.au.dk</email>
      <organization>AU</organization>
      <roles>
        <role>architect</role>
        <role>developer</role>
      </roles>
      <timezone>+1</timezone>
    </developer>
  </developers>
</project>
```

A.3 POG UI Plug-in MANIFEST

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Symphony IDE POG Plugin
Bundle-SymbolicName: eu.compassresearch.ide.pog;singleton:=true
Bundle-Version: 0.3.5.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.lex,
 eu.compassresearch.ast.process,
 eu.compassresearch.ast.program,
 eu.compassresearch.core.typechecker.api,
 eu.compassresearch.ide.core.resources,
 eu.compassresearch.ide.ui.editor.syntax,
 eu.compassresearch.ide.ui.utility,
 org.eclipse.core.commands,
 org.eclipse.core.resources,
 org.eclipse.core.runtime,
 org.eclipse.core.runtime.content,
 org.eclipse.core.runtime.jobs,
```

```
 org.eclipse.core.runtime.preferences;version="3.3.0",
 org.eclipse.jface.action,
 org.eclipse.jface.dialogs,
 org.eclipse.jface.preference,
 org.eclipse.jface.resource,
 org.eclipse.jface.viewers,
 org.eclipse.swt,
 org.eclipse.swt.custom,
 org.eclipse.swt.graphics,
 org.eclipse.swt.widgets,
 org.eclipse.ui,
 org.eclipse.ui.handlers,
 org.eclipse.ui.part,
 org.eclipse.ui.plugin,
 org.eclipse.ui.services,
 org.osgi.framework,
 org.overture.ast.analysis,
 org.overture.ast.analysis.intf,
 org.overture.ast.definitions,
 org.overture.ast.expressions,
 org.overture.ast.factory,
 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.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.contexts,
 org.overture.pog.obligation,
 org.overture.pog.pub,
 org.overture.pog.utility,
 org.overture.pog.visitors,
 org.overture.typechecker,
 org.overture.typechecker.assistant,
 org.overture.typechecker.assistant.definition,
 org.overture.typechecker.assistant.type
Export-Package: eu.compassresearch.core.analysis.pog.obligations,eu.co
 mpassresearch.core.analysis.pog.utility,eu.compassresearch.core.analy
 sis.pog.visitors,eu.compassresearch.ide.pog,eu.compassresearch.ide.po
 g.commands,eu.compassresearch.ide.pog.view
Bundle-RequiredExecutionEnvironment: JavaSE-1.7
Eclipse-BuddyPolicy: registered
Eclipse-BundleShape: dir
Bundle-ActivationPolicy: lazy
Bundle-ClassPath: .,
 jars/pog.jar
```

B JSON Co-Simulation Protocol Definition

The following sections define the detailed format of the JSON objects that are used for the different co-simulation message described in Section 6.2.1. The messages a given

in an intuitive notation that defines parts of a message in JSON notation. For example, the alternative of the two JSON values `true` and `false` is used in one of the message definitions later and an element `<bool>` is defined for this purpose as follows:

```
<bool> ::= true| false
```

Listing 4: Boolean values

The character '|' denotes an alternative and syntactic element are written inside '<' and '>'. The rest of the definitions is JSON syntax. For example, the term {"finished":<bool>} defines a JSON object that for a key `finished` contains the value `true` or `false`.

B.1 Messages

```
<message> ::= <RegisterSubSystemMessage>
  | <FinishedRequestMessage>
  | <FinishedReplyMessage>
  | <InspectMessage>
  | <InspectReplyMessage>
  | <ExecuteMessage>
  | <ExecuteCompletedMessage>
  | <DisconnectMessage>
```

Listing 5: Top level messages as shown in Figure 18

```
<RegisterSubSystemMessage> ::=[
  ["eu.compassresearch.core.interpreter.cosim.
     communication.RegisterSubSystemMessage", {
      "processes" : ["java.util.Vector", string],
      "version" : "3.0.0"
    }
  ]]
```

Listing 6: Register sub-system message

```
<FinishedRequestMessage> ::= {
  "FinishedRequestMessage" : {
    "process" : string
  }
}
```

Listing 7: Finished request message

```
<FinishedReplyMessage> ::= {
  "FinishedReplyMessage" : {
    "process" : string,
    "finished" :  < bool >
  }
}
```

Listing 8: Finished reply message

```
<InspectMessage> ::= {"InspectMessage":{"process" :
   string}}
```

Listing 9: Inspect request message

```
<InspectReplyMessage> ::= {
  "InspectReplyMessage" : {
    "process" : string,
    "transitions" :  < transistions >
  }
}
```

Listing 10: Inspect reply message

```
<ExecuteMessage> ::= {"ExecuteMessage": T}
```

Listing 11: Execute message

```
<ExecuteCompletedMessage> ::= {"ExecuteCompletedMessage":
    {}}
```

Listing 12: Execute reply message

```
<DisconnectMessage> ::= {"DisconnectMessage": {}}
```

Listing 13: Execute reply message

B.2 Transitions

```
<transistions> ::= [] | [<transition>]
```

Listing 14: Array of transitions

```
<transistion> ::= <tau_transition>
                | <timed_transistion>
                | <labelled_transistion>
```

Listing 15: Definition of transitions

```
<tau_transition> ::= {
  "type" : "TauTransition",
  "sources" : int,
  "Skip"}
```

Listing 16: Tau-Transition

```
<timed_transition> ::= {
  "type" : "TimedTransition",
  "sources" : int,
  "timelimit" : int}
```

Listing 17: Timed-Transition

```
<labelled_transistion> ::= {
  "type" : "LabelledTransition",
  "sources" : int,
  "name" : string,
  "values" :  <values>}
```

Listing 18: Labelled-Transition

B.3 Values

```
<values> ::= [] | [<value>]
```

Listing 19: Array of values

```
<value> ::= <quote_val>
          | <token_val>
          | <bool_val>
          | <numeric_val>
          | <any_val>
```

Listing 20: Definition of values

Note that the interpreter will choose the most precise value type for numbers such that 1 will be `nat1`, 2 is `nat`, and 1.2 will be `real`.

```
<numeric_val> ::= {"nat1" : number}
            | {"nat" : number}
            | {"int" : number}
            | {"rat" : number}
            | {"real" : number}
```

Listing 21: Numeric values

```
<bool_val> ::= {"bool" : <bool>}
```

Listing 22: Boolean value

```
<token_val> ::= {"token" : <value>}
```

Listing 23: Token value

```
<quote_val> ::= {"quote" : string}
```

Listing 24: Quote value

```
<any_val> ::= {"?" : <type>}
```

Listing 25: Any value used to specify a sync on any value of a particular type.

The type `<type>` is defined by the “type” definition in the CML syntax definition described in [Col14].

References

[BCW14] Jeremy Bryans, Samuel Canham, and Jim Woodcock. CML definition 4. Technical report, COMPASS Deliverable, D23.5, March 2014. Available at `http://www.compass-research.eu/`.

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

[CMCP13] Joey W. Coleman, Anders Kaels Malmos, Luís D. Couto, and Richard Payne. Third release of the COMPASS tool — developer documentation. Technical report, COMPASS Deliverable, D31.3b, November 2013.

[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.

[Col14] Joey W. Coleman. Fourth release of the COMPASS tool — CML grammar reference. Technical report, COMPASS Deliverable, D31.4c, September 2014.

[Cro06] D. Crockford. The application/json media type for javascript object notation (JSON). RFC 4627, Internet Engineering Task Force, July 2006.

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

[GHJV95] E. Gamma, R. Helm, R. Johnson, and R. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, 1995.

[LB08] M. Leuschel and M. J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.

[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.

[LL14] Kenneth Lausdahl and Adrian Larkham. Linkage to executable software. Technical report, COMPASS Deliverable, D32.3, June 2014.

[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.

[Pos81] J. Postel. Transmission control protocol. RFC 793, Internet Engineering Task Force, September 1981.