Grant Agreement: 287829

Comprehensive Modelling for Advanced Systems of Systems

COMPASS

Fourth Release of the COMPASS Tool
Symphony IDE User Manual

Deliverable Number: D31.4a

Version: 1.1

Date: September 2014

Public Document

http://www.compass-research.eu

Contributors:

Joey W. Coleman, Aarhus
Luis Diogo Couto, Aarhus
Kenneth Lausdahl, Aarhus
Claus Ballegaard Nielsen, Aarhus
Anders Kaels Malmos, Aarhus
Peter Gorm Larsen, Aarhus
Richard Payne, Newcastle
Simon Foster, York
Alvaro Miyazawa, York
Uwe Schulze, Bremen
Adalberto Cajueiro, UFPE
André Didier, UFPE

Editors:

Joey W. Coleman, Aarhus

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 from D31.3a
0.2	03-03-2014	LDC	Incorporate D31.3 feedback.
0.3	15-07-2014	RJP	Add initial Fault Tolerance material
—	15-07-2014	JWC	Structure check
0.4	18-07-2014	JWC	Import D32.4 user manual & small editing
0.4.1	21-07-2014	SDF	Updated missing syntax for the theorem prover
0.5	22-07-2014	JWC	Update Cosim & Commandline sections, tidy simulator support
0.6	22-07-2014	AHM	Added screenshots for S2C lite
0.7	23-07-2014	ALRD	In Section 10: Unified document formatting. Updated the examples and the required definitions to run the FT verification.
0.8	23-07-2014	AHM	Added text to the instruction on how to use S2C lite.
0.9	23-07-2014	LDC	Check and clean up sections 3-6 and 8; updated section 9 with new POG-TP interaction.
0.10	24-07-2014	JWC	Edit section 14; fill out Introduction
0.11	23-07-2014	ACF	Adjusted section 9 and Appendix C.
0.12	23-07-2014	LDC	Add POG support appendix.
0.13	25-07-2014	SDF	Update theorem proving material
0.14	28-07-2014	JWC	Intro and Concl editing
0.15	28-07-2014	LDC	Merge POG section into Theorem Proving
0.16	28-07-2014	RJP	Edit Theorem Proving section
0.17	04-08-2014	PGL	Edit Conclusion
0.18	07-08-2014	JWC	Merge Cosim section into Simulation
0.19	07-08-2014	AHM	Added section on refinement tool
0.20	08-08-2014	JWC	Add small section on implicit execution
0.21	10-08-2014	CBN	Added section on Collaborative Modelling
0.22	25-08-2014	JWC	Editing from internal review; §10,12 remain
0.23	25-08-2014	ACF	Editing §9
0.24	25-08-2014	CBN	Editing §7
0.25	25-08-2014	US	Editing §12
0.26	25-08-2014	ALRD	Editing §10
1.0	26-08-2014	JWC	Ready for delivery
1.1	30-08-2014	PGL	Taking review comment into account

Contents

1 Introduction

This document is a user manual for the Symphony IDE (produced in the COMPASS project), an open source tool supporting systematic engineering of System of Systems (SoSs) using the COMPASS Modelling Language (CML). The ultimate target is a tool that is built on top of the Eclipse platform, that integrates with the RT-Tester tool and also integrates with Artisan Studio. This document is targeted at users with limited experience working with Eclipse-based tools. Directions are given as to where to obtain the software.

This user manual does not provide details regarding the underlying CML formalism. Thus if you are not familiar with this, we suggest the tutorial for CML before proceeding with this user manual [WCF+12, BGW12]. However, users broadly familiar with CML may find the Tool Grammar reference (COMPASS Deliverable D31.4c [Col14]) useful to ensure that the they are using the exact syntax accepted by the tool.

Previous versions of the WP31 deliverables included an "Examples Compendium" document that has been omitted from the overall Deliverable D31.4 package. The relevant material, however, is available on the Symphony Tool website[1] and is also bundled with the Symphony IDE itself (see Section 4.2).

This version of the document supports version 0.4.0 of the Symphony IDE. The intent is to introduce readers to how this version of the tool interacts with CML models.

The main tool is the Symphony IDE, which integrates all of the available CML analysis functionality and provides editing abilities. The architectural relationship between Symphony and the rest of the tools used in the COMPASS project is shown in Figure 1.

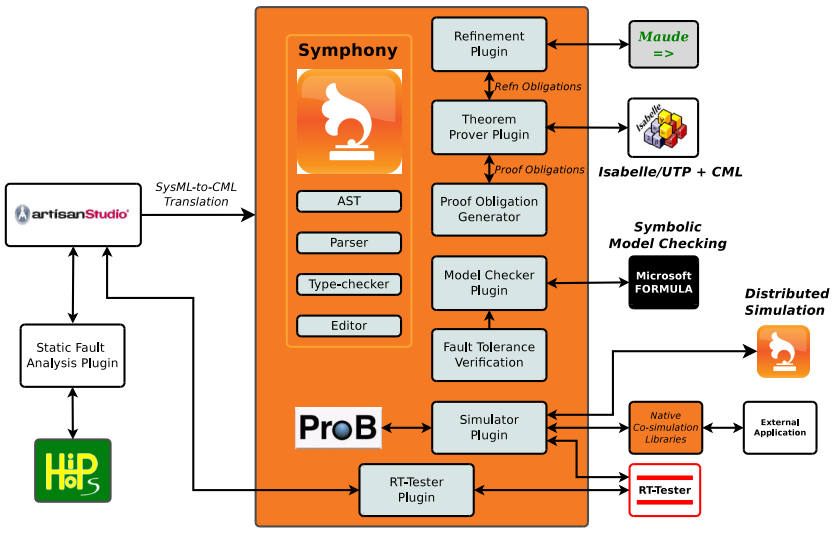

Figure 1: The COMPASS tools

On the left of Figure 1 is Artisan Studio, which provides the ability to model SoSs using SysML. It is possible to generate XMI model files and CML files from SysML models

[1] http://symphonytool.org

in Artisan Studo, and those are recognised within the Symphony tool. SysML models may also have static fault analyses performed on them using the external HiP-HOPS tool, based on work from project task 3.3.3.

In the centre of Figure 1 is the main Symphony tool itself, with many of its submodules identified, and the larger grey boxes correspond to specific plug-in. In particular, the Symphony IDE's connection to the external RT-Tester tool facilitates the use of automated test generation techniques on SoS models, and the Symphony IDE acts as a control console for the RT-Tester platform. There are also dependencies on the Isabelle theorem prover by the theorem prover plug-in, and on the Microsoft FORMULA model checker by the model checker plug-in.

The fault tolerance plugin for CML models uses the model checker plugin to analyse fault tolerance properties of CML models; like the static fault analysis that is done on SysML models using HiP-HOPS, this is based on work from project task 3.3.3. There is support for formal refinement using the refinement plugin; this is a refinement calculator that uses the external Maude tool, and the use of the plugin is documented in [FM14].

The simulator plugin is capable of simulating CML models within the Symphony IDE with no need for external components, but it is also capable of co-simulating a model that does have external components. This is done via a set of libraries that are embedded into the external component, and which allow for communication with the simulator plugin.

Section 2 describes how to obtain the software and install it on your own computer. Section 3 explains the different views in the Symphony Eclipse perspective. This is followed by Section 4 which explains how to manage different projects in the Symphony IDE. Section 5 describes what output the CML typechecker will produce, and where it may be found in the Symphony IDE.

For the situation where a user wishes to simulate a CML model, Section 6 describes the interface to the CML simulator as included in the Symphony IDE. An alternative way of simulating a complete SoS, with some processes external to the Symphony CML simulator, is described in Section 6.4. This external simulation is extended to allow the whole SoS model to be split so that no single constituent owner has the whole model, while still allowing simulation of the complete SoS, as described in Section 7.

Section 8 describes how to use the proof obligation generation and theorem prover plug-ins that can be used together to prove properties about a CML model. CML models may also be model-checked through use of the model checking plug-in described in Section 9. The fault tolerance analysis plugin described in Section 10 uses the model checking plug-in to perform its analysis. The Refinement Tool is described in Section 11, which allows CML models to be transformed via refinement laws. CML models can also be used to drive test generation through the RT-Tester plug-in, as described in Section 12.

It is possible to generate CML models from XMI files exported from a subset of SysML, as described in Section 13[2]. Many of the basic functions of the Symphony

[2]Please note that the translation described here only works for a small subset of the SysML notation but with a simplified semantics. A more elaborate translation has been defined in D33.4 [FM14] and this has been implemented directly from Artisan Studio but since it follows the OMG semantics it produces larger and more complex CML models.

platform are available through a command-line tool, described in Section 14, including basic typechecking, interactive simulation, proof obligation generation, and the SysML to CML generation plugin.

Section 15 provides a few concluding remarks and a perspective on possible future development after the end of the COMPASS project. Finally, Appendices A, B, C, and D list the supported CML language constructs for the interpreter, proof obligation generator, theorem prover and model checker respectively.

2 Obtaining the Software

This section explains how to obtain the Symphony IDE, described in this user manual.

The Symphony tool suite is an open source tool, developed by the universities and industrial partners involved in the COMPASS EU-FP7 project [FLW12]. The tool is developed on top of the Eclipse platform.[3]

The source code and pre-built releases for the Symphony IDE are hosted on SourceForge.net, as this has been selected as our primary mechanism for supporting the community of users of CML and the developers building tools for the Symphony platform. It has facilities for file distribution, source code hosting, and bug reporting.

The simplest way to run the Symphony IDE is to download it from the Symhpony download page at

`http://symphonytool.org/`

This download is a specially-built version of the Eclipse platform that only includes the components that are neccessary to run the Symphony IDE— it does not include the Java development tools usually associated with the Eclipse platform.

Once the tool has been downloaded, in order to run it, simply unzip the archive into the directory of your choice and run the Symphony executable. The tool is self-contained so no further installation is necessary.

The Symphony IDE requires the Java SE Runtime Environment (JRE) version 7 or later. On Windows environments, either the 32-bit or 64-bit versions may be used, on Mac OS X and Linux, the 64-bit version is required.

Artisan Studio and the RT-Tester environment are available from Atego and Verified Systems International, respectively, and are not distributed through the SourceForge.net website. Obtaining those software environments is outside of the scope of this document. The URLs for these are:

http://www.atego.com/download-center/products/category/artisan-studio/

and

https://www.verified.de/products/rt-tester/

In case any issues are discovered with Symphony bugs should be reported at:

https://github.com/symphonytool/symphony/issues/new

Using the *Help* → *Report Symphony bug* menu for the tool will automatically get you to that URL.

[3] `http://www.eclipse.org`

3 Using the Symphony Perspective

When the Symphony IDE is started, the splash screen from Figure 2 should appear. The first time it is started you must decide where you want the default place for your projects to be. Click *ok* to start using the default workspace and close the *welcome* screen to get started for the first time.

Figure 2: The Symphony splash screen used at startup

3.1 Eclipse Terminology

Eclipse is an open source platform based around a *workbench* that provides a common look and feel to a large collection of extension products. Thus, for a user familiar with one Eclipse product it will generally be easy to start using a different product on the same workbench. The Eclipse workbench consists of several panels known as *views*, such as the CML Explorer view at the top left of Figure 3. A collection of panels is called a *perspective*, for example Figure 3 shows the standard CML perspective used in the Symphony IDE. This consists of a set of views for managing CML projects and viewing and editing files in a project. Different perspectives are available in the Symphony IDE as will be described later, but for the moment think of a perspective as a useful composition of views for conducting a particular task.

The *Symphony Explorer view* lets you create, select, and delete Symphony projects and navigate between the files in these projects, as well as adding new files to existing projects.

The *Outline view*, on the right hand side of Figure 3, presents an outline of the file selected in the editor. This view displays any declared CML definitions such as their state components, values, types, functions, operations and processes. The type of the definitions are also shown in the *Outline view*. The *Outline view* is at the moment only available for the *CML* models of the system. In the case another type of file is selected, the message *An outline is not available* will be displayed.

The outline will have an appropriate structure based on the type of CML construct found in the source file that is displayed in the visible CML editor. In Figure 4 a CML class is outlined on the right reflecting the structure of a class. On the left Figure 4 depicts a CML process and lists its actions.

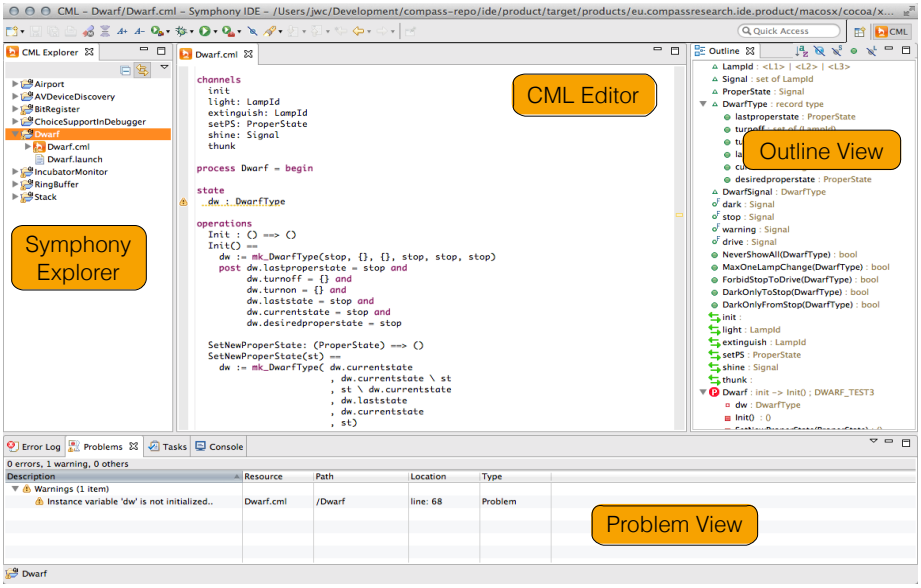

Figure 3: Outline of the Symphony Workbench.

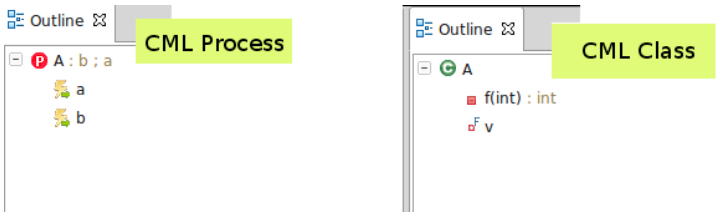

Figure 4: The Outline view showing a *CML* process and its actions on the left. On the right, the Outline view shows a *CML* class.

The higher level elements of the outline can be collapsed or expanded to respectively hide or show their child nodes. For example, a process can be expanded in order to see its actions, operations, and so forth.

Clicking on the name of a definition will move the cursor in the editor to the definition. The outline will also automatically highlight whichever node corresponds to the cursor position as it changes.

Finally, it's important to note that the Outline view, when displayed, is only updated for source-files that parse correctly. Thus, files that have parse errors will not have their Outline view updated.

4 Managing Symphony Projects

This section explains how to use the tool to manage CML projects. Step by step instructions for importing, exporting and creating projects will be given.

4.1 Creating new CML projects

Follow these steps in order to create a new CML project:

1. Create a new project by choosing *File* → *New* → *Project* → *Symphony project* (see Figure 5)
2. Type in a project name (see Figure 6)
3. Click the button *Finish*.

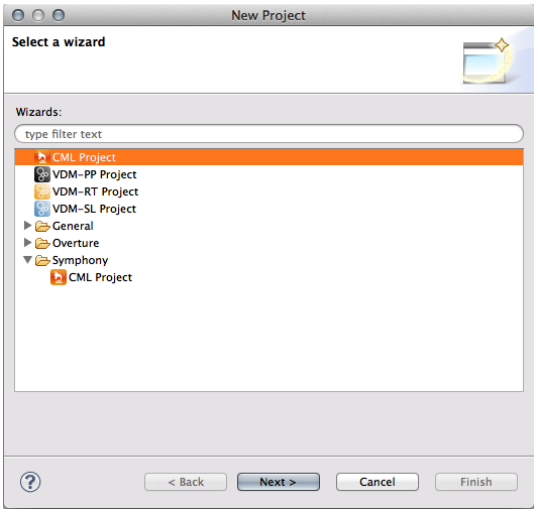

Figure 5: Create Project Wizard

4.2 Importing Symphony projects

4.2.1 Symphony example projects

The Symphony IDE comes bundled with a package of examples that users may experiment with[4]. To import them into the workspace, use the following procedure:

1. Right-click the explorer view and select *Import*, then choose *Symphony* → *Symphony Examples*. See Figure 7 for more details. Click *Next* to proceed.
2. The available example projects will be presented in the next dialog, as shown in Figure 8. Choose the desired examples, then click *Finish* and they will be automatically imported into your workspace.

[4]An overview of all the CML projects that can be imported in this fashion can be seen at `http://symphonytool.org/examples/`.

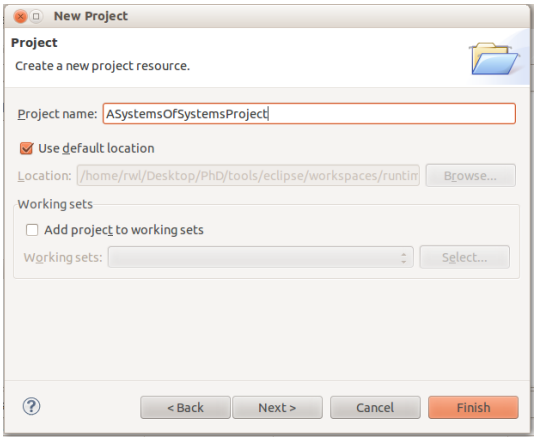

Figure 6: Create Project Wizard

4.2.2 Existing Symphony projects

Follow these steps in order to import an already existing Symphony project:

1. Right-click the explorer view and select *Import*, followed by *General* → *Existing Projects into Workspace*; this can be seen in Figure 7. Click *Next* to proceed.
2. If the project is contained in a folder, select the radio button *Select root directory*, if it is contained in a compressed file select *Select archive file*. These options will be presented in a dialog similar to that in Figure 8.
3. Click on the active *Browse* button and navigate in the file system until the project to be imported is located.
4. Click the button *Finish*. The imported project will appear on the *Symphony explorer view*.

4.3 Exporting Symphony projects

Follow these steps in order to export a Symphony project:

1. Right click on the target project and select *Export*, followed by *General* → *Archive File*. See Figure 9 for more details.
2. A new window like the one shown in Figure 10 will follow. In this case the selected project will appear as root node on the left side of it. It is possible to browse through the contents of the project and select the correct files to be exported. All the files contained in the project will be selected by default.
3. Enter a name for the archive file in the text box following *To archive file*. A specific path to place the final file can be selected through the button *Browse*.
4. Click on the *Finish* button to complete the export process.

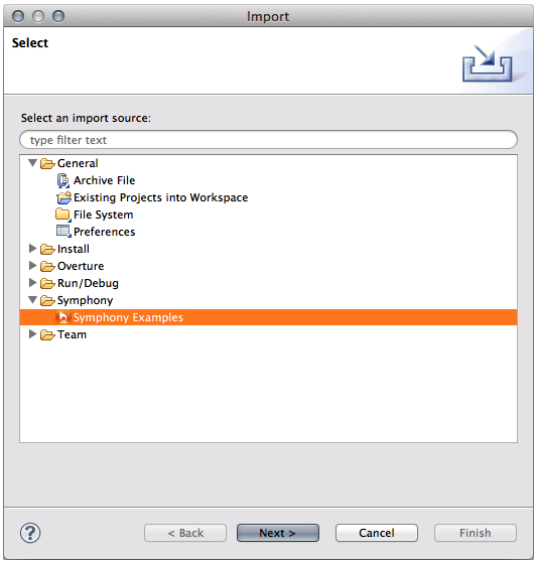

Figure 7: Symphony Example import dialog

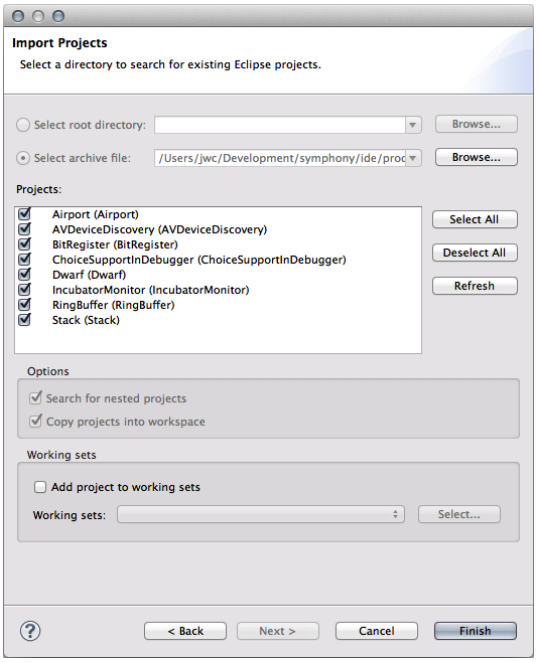

Figure 8: Symphony Example selection

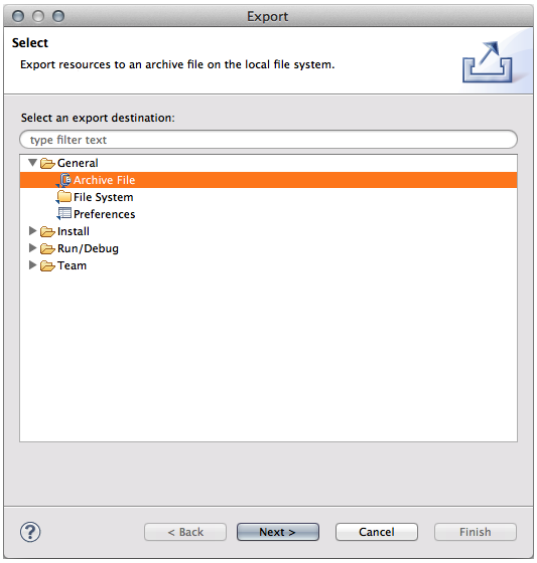

Figure 9: Select an output format for the exported process.

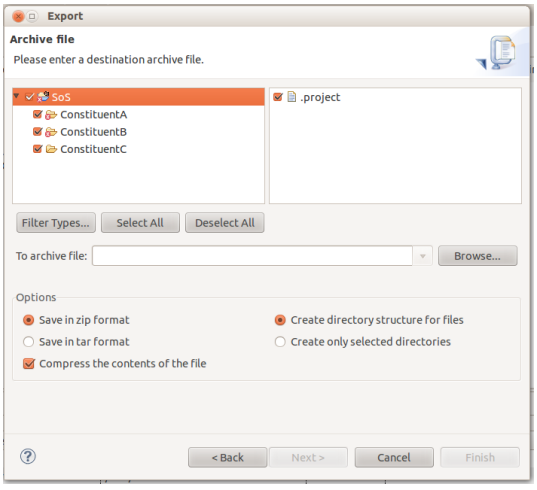

Figure 10: Project ready to be exported.

4.4 Creating Files

Switching to the CML perspective will change the layout of the user interface to focus on CML development. To change perspective, go to the menu *Window* → *Open perspective* → *Other...* and choose the CML perspective. From this perspective you can create files using one of the following methods:

1. Choose *File* → *New* → *CML File* or *CML Class* or *CML process* or
2. Right click on the Overture project where you would like to add a new file and then choose *New* → *CML file* or *CML Class* or *CML process*.

In both cases you need to choose a file name and optionally choose a directory if you do not want to place the file in the home directory of the chosen Overture project. Then a new file with the appropriate file extension (texttt.cml) will be created in the directory.

4.5 Adding Standard Libraries

In addition to adding new empty files it is possible to add existing standard libraries. This can be done by right-clicking on the project where the library is to be added and then selecting *New* → *Add CML Library*. That will make a new window as shown in Figure 11. Here the different standard libraries provide different standard functionalities. In the body of many of these functions/operations are declared as "**`is not yet specified`**" but the actual functionality for all of these are hard-coded into Symphony so the user can get access to this when the respective standard libraries are included. This can be summarised as:

IO: This library provides functionality for input and output from/to files and the standard console.

Math: This library provides functionality for standard mathematical functions such as sine and cosine.

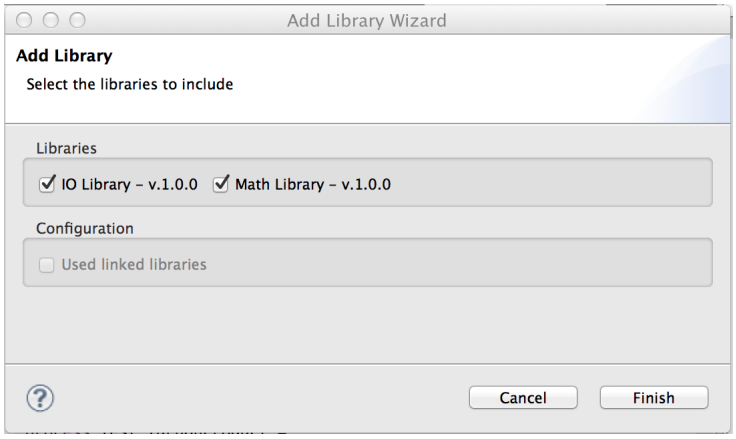

Figure 11: Adding New Libraries

5 The CML Type Checker

The Symphony IDE ships with the CML Type Checker. The Type Checker checks type consistency and referential integrity of your model. Type consistency includes checking that operator and variable types are respected. Referential integrity includes checking that named references exists and have an appropriate type for their context.

5.1 Output

The type checker produces two kinds of artifacts: Type Errors and Type Warnings. Both carry a reference to the offending part of the model, a description of what is ill formed and an exact location of where the issue occurred.

5.2 Representation

In the Symphony IDE user interface, type errors show up in three places. To point the user at the exact piece of CML-source causing an error, an error marker will be shown in the left margin of its Editor. Additionally, the problematic piece of syntax will be underlined with red as seen in Figure 12.

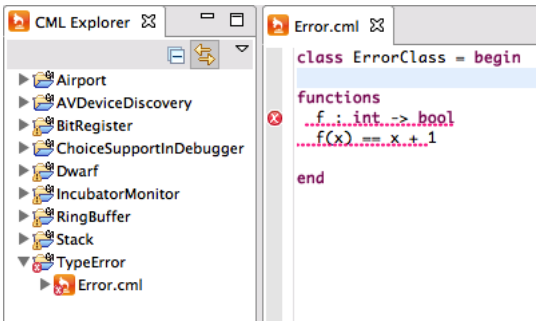

Figure 12: User Interface showing type error markers.

Type errors are also made visible in the user interface through the CML Project Explorer. The CML Project Explorer offers a tree view of CML model file structure. If an error occurs in a CML-source file then all of folders containing that file up through the hierarchy to the project level will have a red error marker (also see Figure 12).

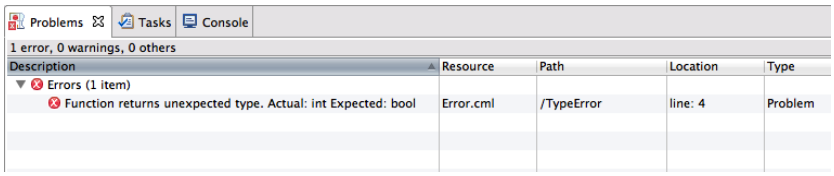

Figure 13: User Interface problems view with type errors.

To give the complete picture for all errors in a given model the problem view shows the list of all generated errors (see Figure 13).

Type error markers will be updated whenever a CML-Source file is saved with changes. To force a re-check of all source files again click the *Project* → *Clean* ... item from the menu bar.

6 The Symphony Simulator

This chapter explains how to simulate/animate a CML model with the Symphony IDE. This includes how to add and configure launch configurations, and how the interpreter is launched and used. Information on which CML constructs are supported by the Simulator can be found in Appendix A.

First, the basic modes of operation are explained. The interpreter operates in two modes, *Run* and *Debug*, and within these modes there are three options *Animate*, *Simulate* and *Remote Control*. These options control the level of user interaction and are described below:

1. **Simulate**: This option will interpret the model without any user interaction. When faced with a choice of several observable events, one will be chosen in an arbitrary but deterministic manner. In other words, the simulation will always make the same choices for every run of the same model.
2. **Animate**: This option will interpret the model with user interaction. All observable events are selected by the user.
3. **Remote Control**: This option enables the interpreter to be remote controlled by an external Java class implementing the *IRemoteControl* interface and located in the "lib" folder of the project.

The interpreter also supports breakpoints. Breakpoints can be set at any line in the model. Once the line with the breakpoint is reached, the execution is suspended and the current state of the model (including variable values) may be inspected. The modes of operation controls the interpreter's behaviour with respect to breakpoints are:

1. **Run**: This will simulate/animate the model ignoring any breakpoints.
2. **Debug**: This will simulate/animate the model and suspend execution at all enabled breakpoints.

6.1 Creating a Launch Configuration

To create a launch configuration, first click on the small arrow next to either the debug button or the run button (depending on the desired mode) as shown in Figure 14.

Once clicked, a drop-down menu will appear with either *Debug configurations* or *Run configurations* (depending on which button you clicked); select the appropriate *configurations* option. This will open a configurations dialog like the one shown in Figure 15. All of the existing CML launch configurations will appear under the *CML Model*. To create a new launch configuration you may double-click on the *CML Model* or on the *New launch configuration* button, then an empty launch configuration will appear as shown in Figure 15 with the name *New Configuration* (possibly followed by a number if this name is already used). To edit an existing configuration, click on the desired launch configuration name and the details will appear on the right hand side of the dialogue.

As seen in Figure 15 a project name and a process name need to be associated with a launch configuration along with the mode of operation as discussed in Section 6. When choosing a project, you can either write the name or click on the *Browse* button which

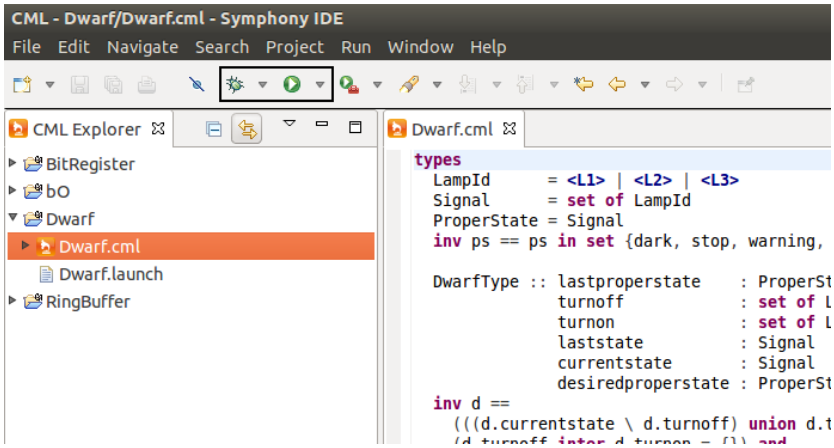

Figure 14: Screenshot of the toolbar of the Symphony IDE showing the debug button (left) and run button (right) highlighted.

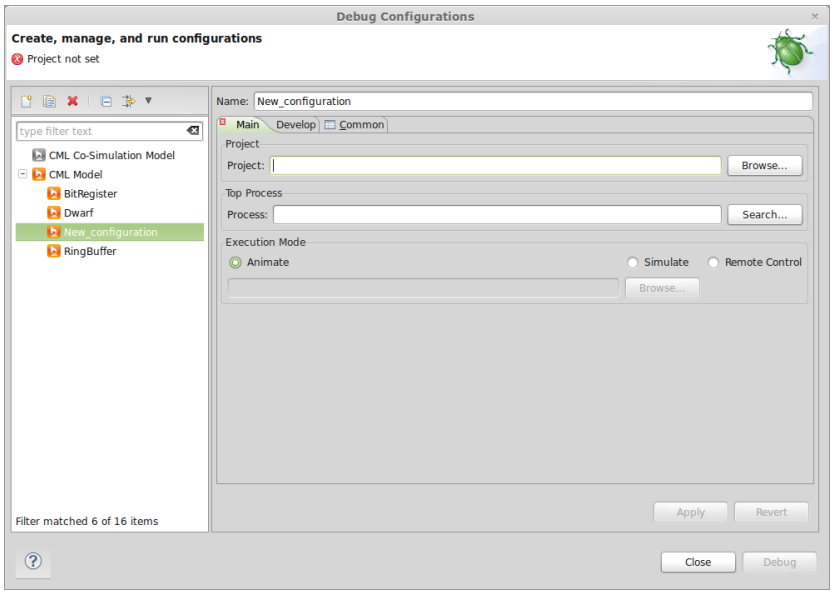

Figure 15: The launch configuration dialog showing a newly created launch configuration

shows a list of all the available projects and choose one from there. The selection of the process name is identical.

The selected project must exist in the workspace, and the process named must exist within it. It will not be possible to launch if they do not. In the left corner of Figure 15 a small red icon with an “X” and a message will indicate what is wrong. In the figure it indicates that no project has been set, so this should be the first thing to do.

After setting the project name and process name, the *Apply* button must be clicked to save the changes to the launch configuration. If the project exists, is open and a process

with the specified name exists in the project, then the *Run* or *Debug* button will be active and it is possible to launch the simulation as shown in Figure 16. Furthermore, the decision of whether to animate, simulate or remote control the model is decided by the radio buttons in the *Execution Mode* groupbox in the buttom, the default setting is to animate.

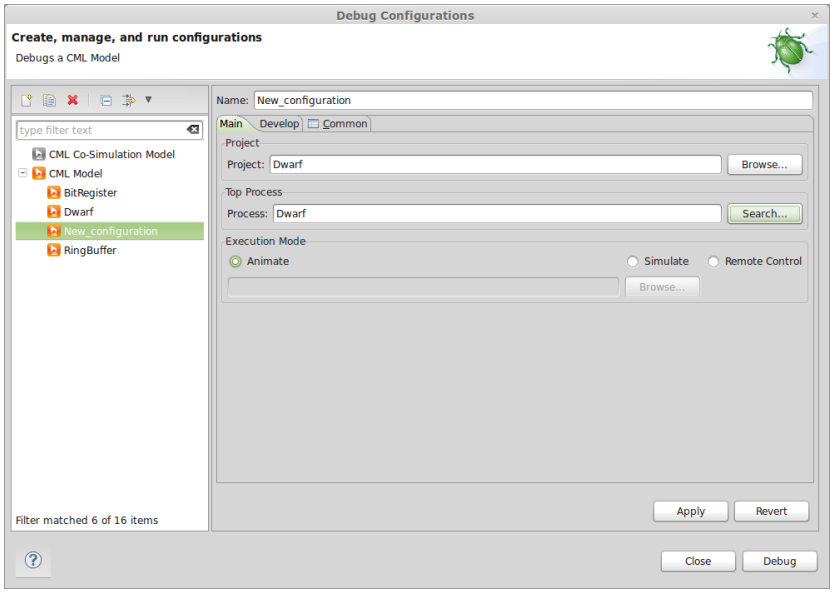

Figure 16: The *configuration dialog* after a project and process have been selected

This launch configuration will now appear in the drop-down menu as described at the beginning of this section. The actual interpretation will be described in Section 6.3.

6.2 Launch Via Shortcut

Another way to launch a simulation is through a shortcut in the Symphony explorer view in the CML perspective. To access this, right click on a cml file to make the context menu appear. From here either choose *Debug As* → *CML Model* or *Run As* → *CML Model* as depicted in Figure 17.

After that, two things can happen: if the CML source file only contains one process then this process will be launched. If however, more than one process is defined, then a process selection dialog appears with a list of possible processes. This is shown in Figure 18.

To launch a simulation, a process must be chosen. This is done by double-clicking one of the process names in the list, or selecting it and pressing ”OK”. This will launch a simulation with that process as the top-level process.

If you launch via a shortcut then a launch configuration named *Quick Launch* (or *Quick Launch(<number>)* if more exist) will be created and launched.

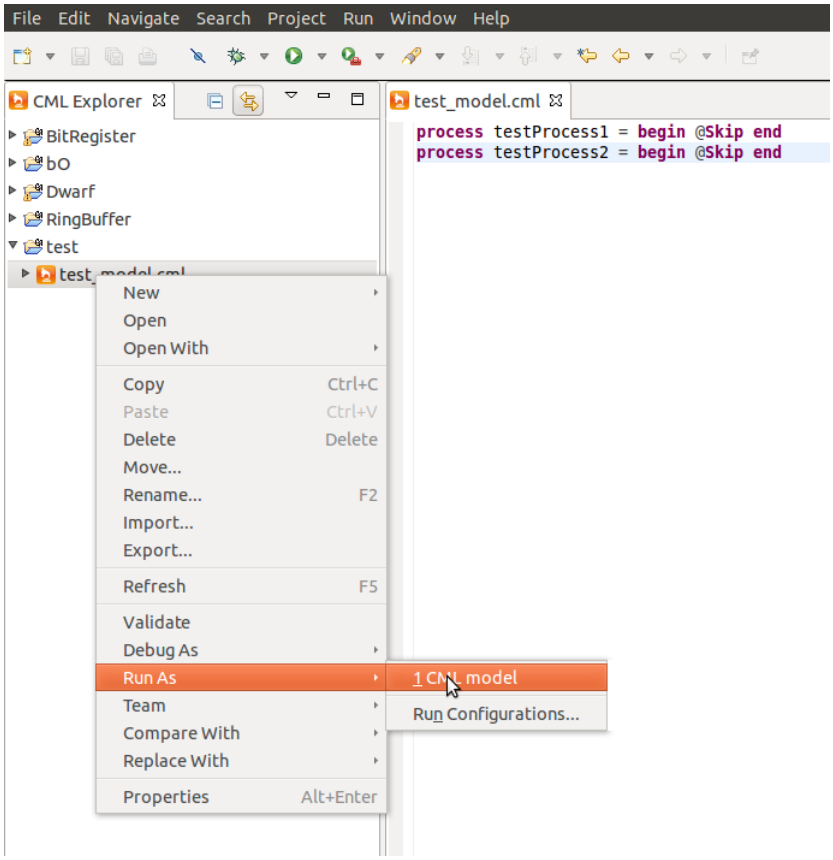

Figure 17: The quick launcher

6.3 Interpretation

As mentioned at the start of Section 6, there are four possible ways to interpret a model, each of them will be described.

6.3.1 Animation

Animating a model is achieved by choosing the *Animate* radio button in the launch configuration as described in the last section, this is also the default behavior. In this mode of operation the user has to pick every observable event before they can occur through the GUI.

In Figure 19 a small CML model is being animated in the debug perspective. The following windows are depicted:

Observable Event History This window is located in the top right corner and shows the observable events that have been selected so far. In Figure 19 only a tock event has occurred so far.

CML Event Options This shows the possible events that can occur in the current state

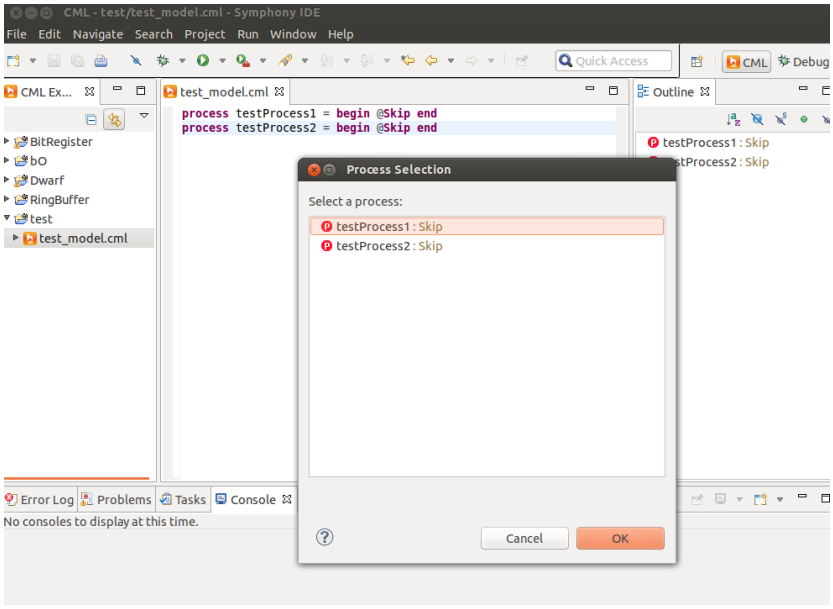

Figure 18: Right after *Run As* → *CML Model* has been clicked, the context menu of the `test.cml` file appears. Since the file defines more than one process, the *process selection dialog* is shown.

of the model. To make a particular event occur you must double-click it. Furthermore, to see the origin of a particular offered event, you must click it and the location of every involved construct will be marked gray in the editor window.

Editor This shows the CML model source code with a twist. As seen in Figure 19 parts of the model is marked with a gray background. This marking is determined by the selected event in the CML Event Options view.

To understand how the views work together a two-step animation is shown in Figure 19 and Figure 20. In Figure 19 *tock* has happened once and a *tock* event is currently selected. Since process A and B both offer *tock* they are both marked with gray in the Editor view. In Figure 20 the *init* event has been double-clicked. Thus, A and B has synchronized on *init* and they both wait for the next event to occur.

6.3.2 Simulation

Simulating without user interaction is achieved by choosing the *Simulate* option in the launch configuration. This mode of operation will interpret the model by taking random decisions when faced with a choice of events. However, the same choices will always be taken if the model is interpreted multiple times (this can be controlled by a seed value). In Figure 21 a simulate interpretation has completed.

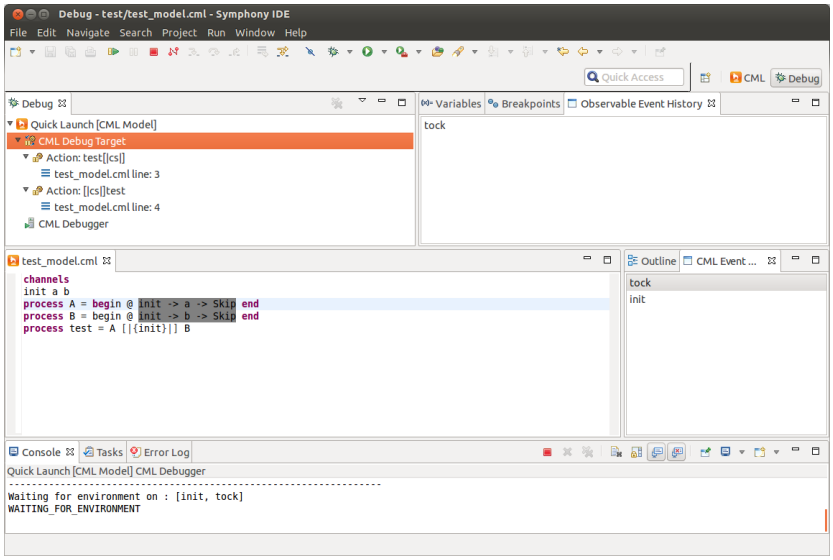

Figure 19: A CML model animated in the debug perspective.

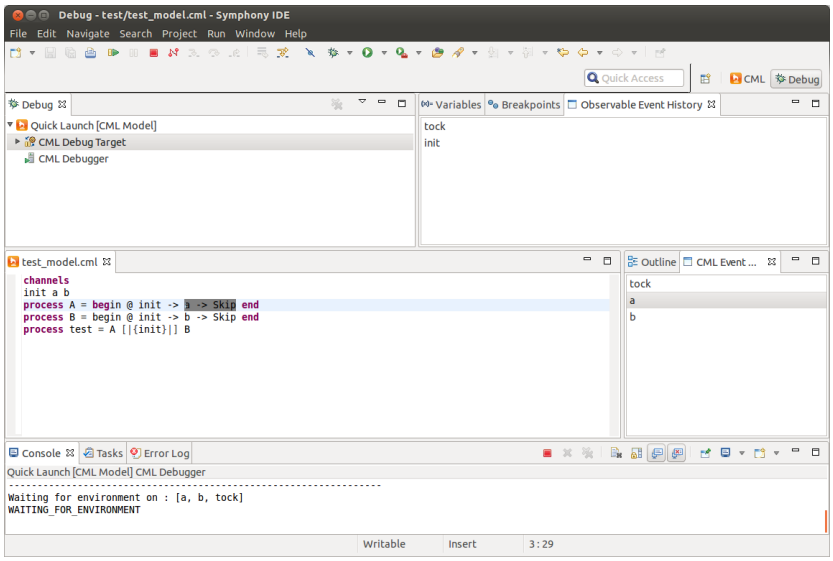

Figure 20: The *init* event has just occurred and it is now currently offering *a*, *b* and *tock*, where *a* is currently selected

6.3.3 Run/Debug

In addition to the two modes of operation *Animate* and *Simulate* the standard modes *Run* and *Debug* also exist. The *Run* mode will interpret the model without ever breaking on any breakpoints. The *Debug* however will stop on any enabled breakpoint in the model.

When a *Debug* configuration is launched, the perspective changes to the Eclipse Debug

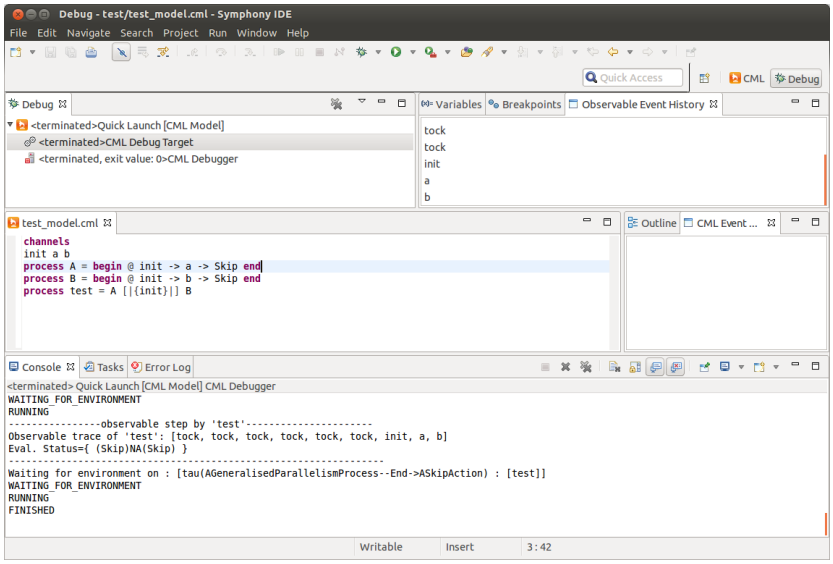

Figure 21: The model has just been simulated

Perspective, however *Run* will stay on the perspective that is currently active.

To create a new breakpoint you have to double-click on the ruler to left in the editor view, if created, this will insert a small dot to the left ruler. Breakpoints can be set on processes, actions and expressions only. Double-clicking on a existing breakpoint dot will remove it. In Figure 22 a debugging session is in progress. Here, a breakpoint on the *a* event in process A has been hit and the interpreter has been suspended. At this point the current state can be inspected in the variables view. From here it is both possible to resume or stop the debugging session. If the resume button is clicked the interpretation is resumed and the stop button stops it.

6.3.4 Error reporting

If an error occurs a dialog will appear with a message explaining the cause of the error. Furthermore, the location of the error will be marked in the editor view. In Figure 23 a post condition has been violated. This is described in the error dialog and a gray marking shows where in the model it happened.

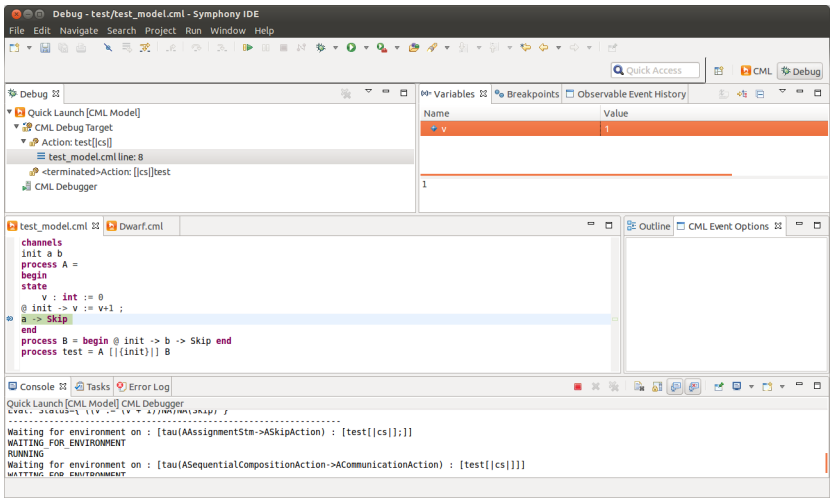

Figure 22: The interpreter is currently suspended because a breakpoint is hit. The line of the breakpoint is highlighted in green and has an arrow in the left ruler. In the variable view in the top right corner the state variable for process A can be seen.

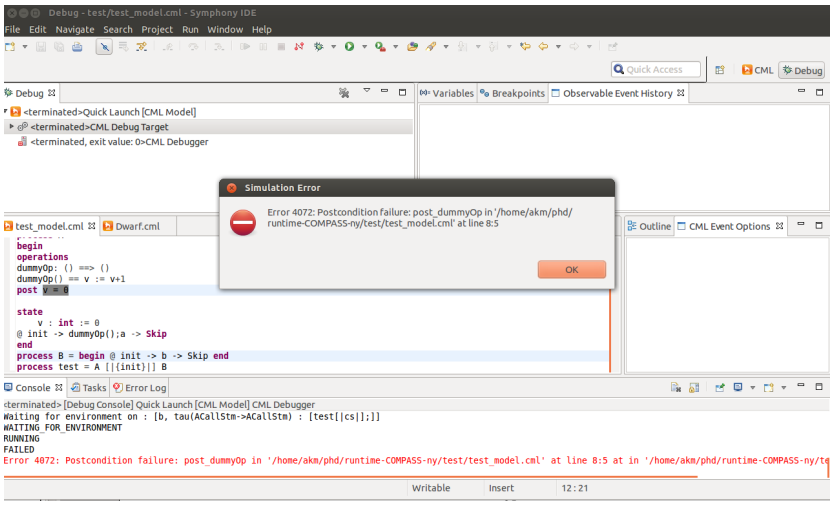

Figure 23: The interpreter has stopped because a post condition has been violated

6.3.5 Interpreting implicit model elements

There are cases where it is useful to be able to simulate models that have a degree of looseness to them, such as functions or operations that are intended to select and return an arbitrary element from a set. To support this, the functionality of the 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.

For example, consider the following example of a `select` function.

```
select(ns : set of nat) n: nat
pre card(ns)>0
post n in set ns
```

This function, given a set of natural numbers, arbitrarily returns one of the numbers in that set. Strictly, this is not a function, but rather a relation, but it could be seen as something that will be a function but is has an incomplete specification. It can still be useful to simulate this, and it is possible to do this in the Symphony IDE.

It is also possible to implicitly invoke the ProB solver in **`for all`** loops that use type bindings, such as:

```
for all n in set { x | x : nat }
do c.n -> Skip
```

where the statement in this example would synchronise on each natural precisely once in some arbitrary order.

There are two major limitations with this feature, however. The first is that union types cannot yet be translated into a form that can be used with the ProB solver, and so will not interpret correctly at this point. The second is that type invariants are not translated, so care must be taken when solving with types that have invariants.

6.4 Co-simulation

It is possible to split the simulation of a model across several instances of the Symphony CML simulator, or even in a mix of CML simulators and other external mechanisms. We call this *co-simulation*. During co-simulation, a portion of the modelled system is actually being simulated by other tools or systems that are external to the CML simulator. This means that a simulator running in one instance of the Symphony IDE may be connected to another system and, as a result, simulate the whole System of Systems (SoS) in co-operation with that system.

Co-simulation is accomplished through the use of the co-simulation configuration available under debug configurations in the Symphony IDE. A co-simulation is a distributed simulation that consists of at least one *coordinator* and some non-zero number of *external systems*. A coordinator is the simulator that simulates the toplevel CML model of the SoS. This means that it conducts its simulation at the process level, and that the

simulation starts with the *top* or *entry* process, which is the process at the highest level that composes the SoS from its constituent systems.

The co-simulation launch configuration shown in Figure 24 and Figure 25 allow the user to select the project, co-simulation URL, and one of two modes:

Coordinator: To run the simulator as a coordinator, the user selects the *Coordinator* option and selects the processes that will be handled by an external system. It is important to note that the process named in the *Top Process* section must be the *top* process at the SoS level. An example of such a configuration is given in Figure 24.

External System: To run the simulator as one of the constituent systems of the SoS, the user selects the *External System* option and, in this case, chooses the specific process that this simulator represents for the *Top Process* section. See Figure 25 for illustration.

Note that the options above are specific to a remote simulator configuration; an alternative configuration for external systems based on native code is described in Deliverable D32.4 [LMN+14].

A co-simulation requires the coordinator to be started first so that it is listening for clients before the external systems are started. This allows any external system to connect and register by informing the coordinator which process the external system implements. Note that current constraints mean that only one instance of a given process may be externalised: neither processes in a replicated operator nor parametrised processes may be externalised.

A Reader-Writer example CML model is provided in Listing 1. The screenshots in Figure 24 and Figure 25 assume this example as a basis for the values in the dialogs. The example itself is a CML specification of a reader-writer system where the top SoS process is `Main`. The top process constructs the SoS using a single `Writer` and `Reader` process in parallel, synchronizing on the channels `a` and `b`. To use the

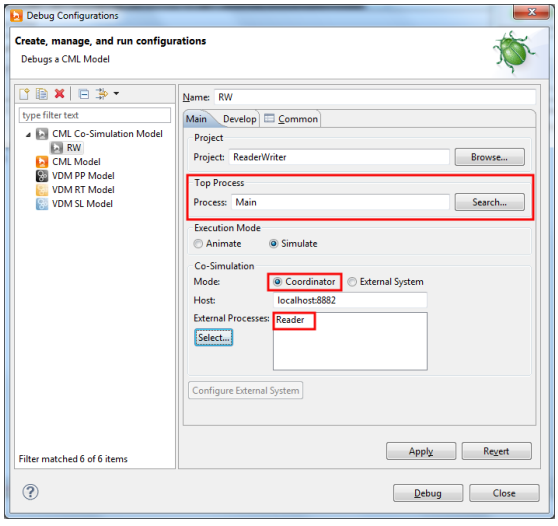

Figure 24: Coordinator launch configuration with the `Reader` process as external.

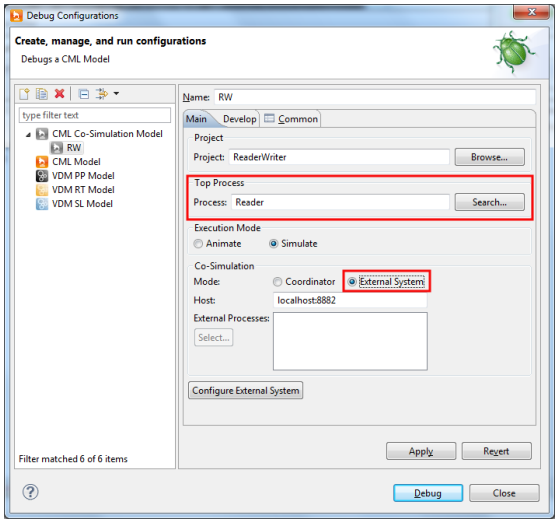

Figure 25: External system launch configuration for the `Reader` process.

co-simulation engine, the configuration must have at least one of either `Reader` or `Writer` designated as external. In Figure 24 a launch configuration is shown for the coordinator, and Figure 25 shows the `Reader` external system.

```
channels
  a, b: int

values
  MAX = 10

process Main = Writer [|{a,b}|] Reader

process Reader = begin
actions
  s = a?x -> (
    [x < MAX] & b!x -> s
    []
    [x = MAX] & b!x -> Skip
  )
@ s
end

process Writer = begin
actions
  s = val x : int @
    a!x -> b?y ->
      let n = y+1 in (
        [n <= MAX] & s(n)
        []
        [n > MAX] & Skip
      )
@ s(1)
end
```

Listing 1: A Reader-Writer model in CML.

7 Collaborative Modelling in the Symphony IDE

An SoS will often contain constituent systems that are developed by different stakeholders, which are geographically distributed. In order to create and develop a model of such an SoS the different stakeholders need to establish a collaboration, through which interface descriptions and the system design can be examined, negotiated and agreed on.

Symphony contains a Collaborative Modelling plugin which enables instances of Symphony IDE, running on different computers, to be connected via the internet in order to exchange modelling data and establish a collaboratively developed SoS model. The Collaborative Modelling plugin can be used by modellers that want to work on a joint CML model of an SoS while being geographical distributed.

The basic principles of the Collaborative Modelling plugin is illustrated in Figure 26. Connectivity is established between instances of the Symphony IDE via a common Collaboration Project that is shared among the collaborating modellers. The collaboration between the modellers is performed by sending and exchanging configurations, as illustrated by the configurations moving in the direction of the arrows. A Configuration can be seen as a snap-shot of either a whole CML model or parts of a CML model that a modeller proposes as a potential system design. A Configuration contains any number of files from the workspace of the CML project to which the Collaboration Project is attached. Other collaborators can then either Approve or Reject a proposed Configuration, indicated by red and green colors, or they can choose to evolve the model further by making alterations to the model data and placing it in a new Configuration that can be send to the other participating collaborators as an improved proposal.

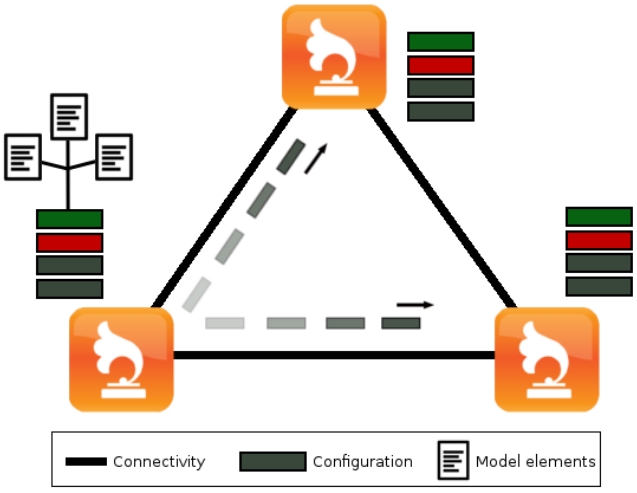

Figure 26: Basic principles of the Collaboration

The key functionality of the Collaborative Modelling plugin can be summed up in four points:

Connectivity between Symphony IDE instances Establishing connections between Symphony IDE instances using instant messaging technology.

Creation of a collaboration environment Creating an environment in the Symphony IDE that enables collaborative work.

Exchange and negotiation of model data Transferring model data between collaboration and reaching consensus on model design.

Distributed simulation Performing distributed simulation between collaborators in the collaboration environment.

These four points are detailed in the respective sections below.

The Collaborative Modelling plug-in defines an Eclipse View called "Collaborative Modelling" that provides a graphical overview of the current collaboration projects that this Symphony IDE instance is related to. This graphical view makes it possible to see other collaborators, view the exchanged configurations and files, see the history of exchanged configurations, and see which collaborators have approved or proposed adjustments to the shared model data. A screenshot of the Collaboration view is shown in Figure 27.

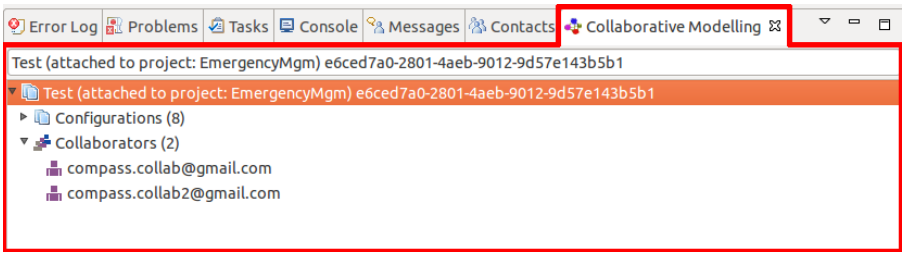

Figure 27: Screenshot of the Collaborative modelling view

7.1 Connectivity between Symphony IDE instances

The connectivity between the Symphony IDE instances is established using the open-source Eclipse Communication Framework (ECF)[5]. The connectivity between all the participating collaborators is used to push notifications to all connected Symphony IDE instances, in order to ensure that collaboration activity is shared rapidly.

The current version of the Collaborative Modelling plug-in makes use of the open standard XMPP[6], formerly Jabber, protocol. This makes use of a centralized server for relaying data between the different participants. An account is used to identify the connected user on the XMPP server. The Collaborative Modelling plug-in has been tested against Google Talk servers for which Google accounts can be used.

Connecting

A connection is established by using the *Account Information* dialog that can be opened by clicking the Collaborative Modelling view's dropdown menu and clicking *Connect to IM Provider*, as shown in Figure 28. In the dialog, XMPP can be selected as con-

[5] http://www.eclipse.org/ecf/
[6] http://www.xmpp.org/

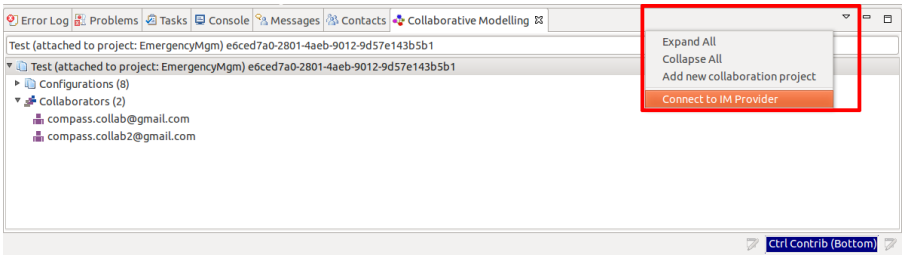

Figure 28: Screenshot of the Connect menu

nection protocol or alternatively XMPPS in order to use an encrypted connection via Secure Sockets Layer.

In the *Account Information* dialog the user account information and server address need to be provided to establish a connection, as show in Figure 29. Once connected, the active view will change to *Contacts*. Only accounts listed in the contacts list can become part of a collaboration project.

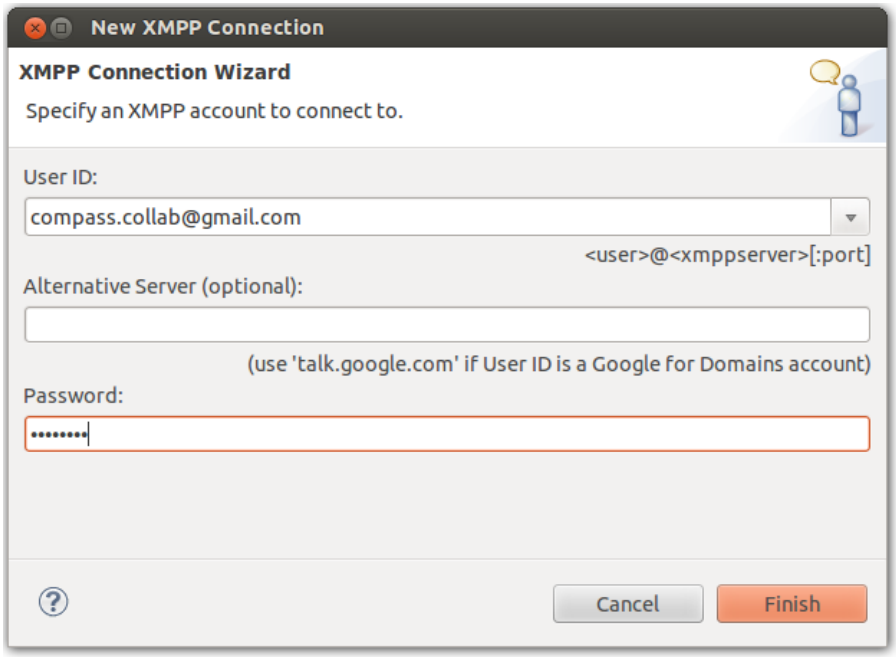

Figure 29: Screenshot of the Account Information dialog

There is currently a limitation in ECF which entails that accounts using Google 2-Step Verification[7] cannot be used for the Collaborative Modelling plug-in, as the logon process cannot be completed.

[7] http://www.google.com/landing/2step/

7.2 Creation of a Collaboration Environment

The basis for establishing the collaborative environment between Symphony IDE instances is the "Collaboration Project". A collaboration project defines the context for the collaboration by keeping track of the different modelling configurations and by defining a "Collaboration Group" which contains the collaborators, i.e. other accounts logged in on other Symphony IDE instances.

Project Creation

A Collaboration Project is created by selecting the *Add new collaboration project*, as shown in Figure 30.

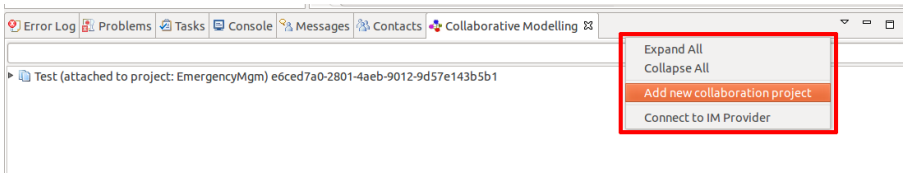

Figure 30: Screenshot of the new project menu

This will open the *New collaboration project* dialog, as seen in Figure 31, in which the title and description of the project is to be entered. The collaboration project must be attached to a CML project in order for the plugin to know in which workspace directory it should operate. Only one Collaboration Project can be attached to a CML project at a time. The plugin will track the files in the workspace directory in order to determine changes to the model.

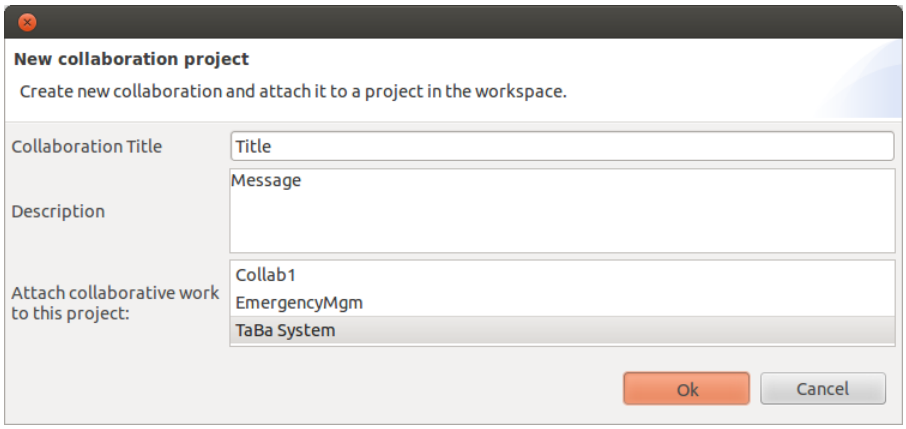

Figure 31: Screenshot of the new project dialog

The creator of the Collaboration Project will initiate the collaboration by adding collaborators to the Collaboration Group. The Collaboration Group is created on the basis of the user's contacts/buddy list from which the wanted collaborators can be selected.

Once other collaborators have joined a collaboration project, the initiator has no more control of the project than the other collaborators. This means that the Collaboration Project is decentralized and is therefore not exclusively managed by any collaborator. All collaborators that are in the Collaboration Group can send requests for new collaborators to join the Collaboration Project. This ensures that no one has the control of the group and the authority is distributed.

Adding Collaborators

The collaborators are selected by right-clicking on the Collaborators node for the relevant Collaboration Project in the Collaborative Modelling view, and choosing the *Add Collaborator* menu item. This will list the online contacts of the account that was used to create the connection (in Section 7.1), as shown in Figure 32.

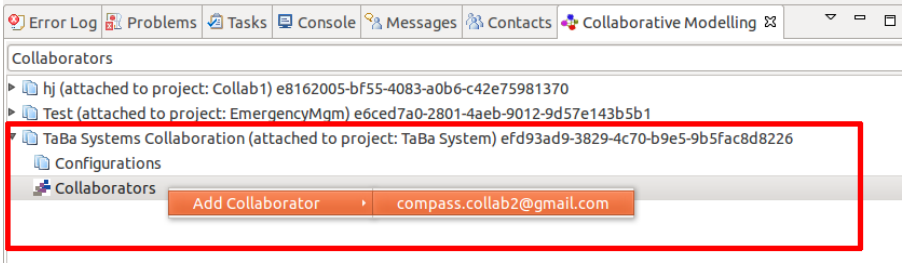

Figure 32: Add collaborator to project screenshot

The *Collaboration Request* dialog will open in which a message to the recipient of the request can be entered, as shown in Figure 33.

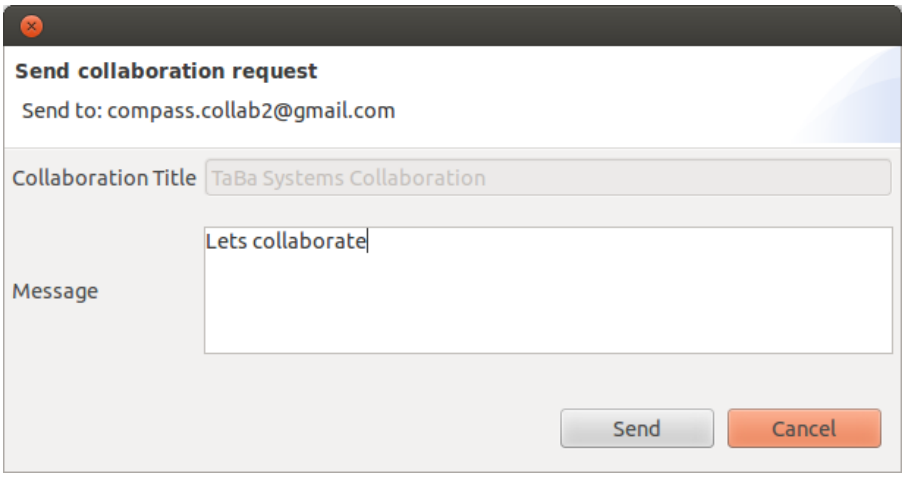

Figure 33: Collaboration Request dialog

The selected collaborator will then receive a join request, with the name of the sender, the title of the collaboration project and the message entered in the *Collaboration Re-*

quest dialog. Figure 34 shows the *Join request* dialog in which the collaborator can choose to join or decline the collaboration request.

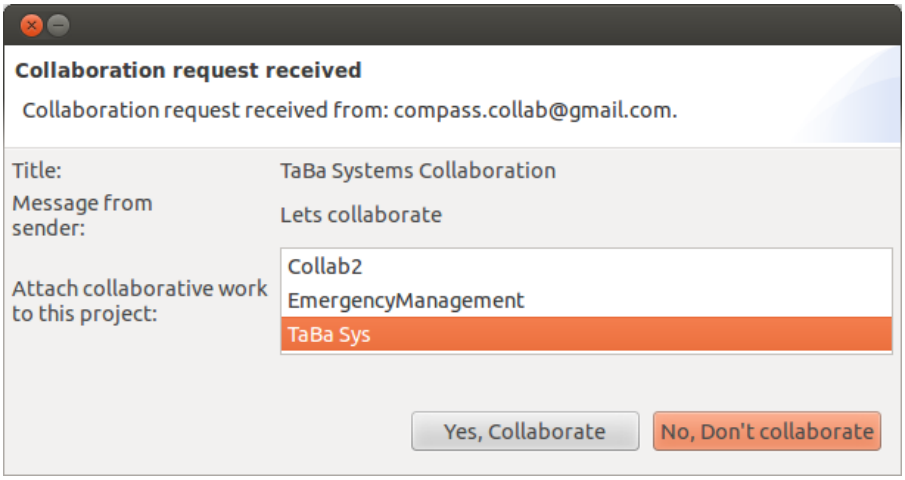

Figure 34: Collaboration Request Received dialog

When the collaborator joins the collaboration, the sender of the request will be notified by a small pop-up box, as shown in Figure 35.

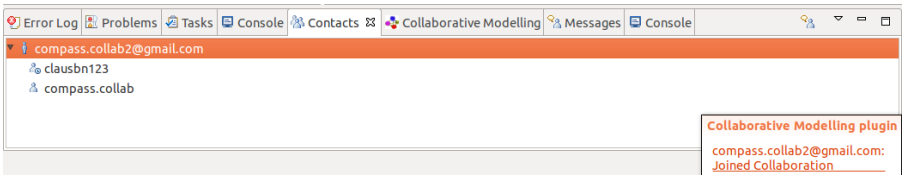

Figure 35: Joined notification

Deleting/Leaving a Collaboration Project

A collaborator can leave a collaboration project by deleting it from the Collaboration View. Deleting a project will remove the project from the list of Collaboration Projects, and it will also notify the other collaborators that this collaborator has left the collaboration and therefore no further data exchanges will be made with this particular collaborator. Deleting a project will not have any effect on the CML project to which it is attached. Deleting a project will also not end the collaboration, even if the original creator of the Collaboration Project deletes the project. The collaboration will remain alive, until the last collaborator deletes their Collaboration Project.

7.3 Exchange and negotiation of model data

Model data is exchanged between instances of the tool by the individual collaborators selecting the parts of their CML model they want to share with the other collaborators

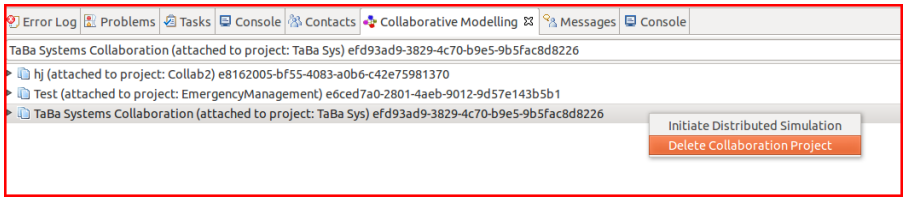

Figure 36: Delete Collaboration Project

in the collaboration group. The exchange occurs at file level, meaning that entire files are being distributed to the collaboration group. This means that the model data that a collaborator wants to share, has to be kept in files separated from the files containing model data that the collaborator does not want to share.

Adding a File to the Collaboration Project

A file is added to the Collaboration Project, by right clicking the `.cml` file in the project explorer and selecting either *Include file in collaboration* or *Include file in collaboration (with limited visibility)*; as shown in Figure 37.

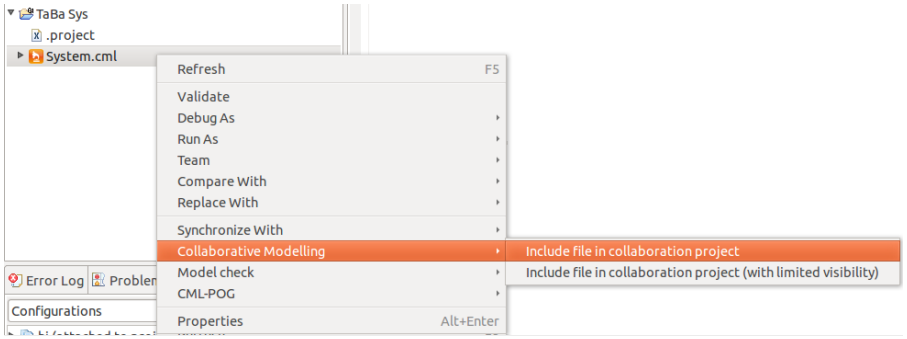

Figure 37: Adding a file to a collaboration project

Adding a file with limited visibility gives each collaborator the option of selecting which collaborators in the Collaboration Group will receive the file. This functionality is included for two reasons, 1) some collaborators may not be willing to show their interfaces and model designs to all collaborators, but only to select few. 2) Given that it is possible for any member of a collaboration group to expand the group by inviting other collaborators, visibility gives the individual collaborators the option of selecting if new members can see their shared models. A constraint of using the visibility functionality, is that once the visibility of a model has been granted to a collaborator, it cannot be revoked. This is done to avoid collaborators from using the visibility functionality to hide interfaces that has already been negotiated or approved by others.

Once a file is added to a Collaboration Project it will become part of a Configuration, as shown in Figure 38.

The contents of the files added to the Collaboration Project can freely be adjusted locally within a Configuration until the collaborator wants to share the model with

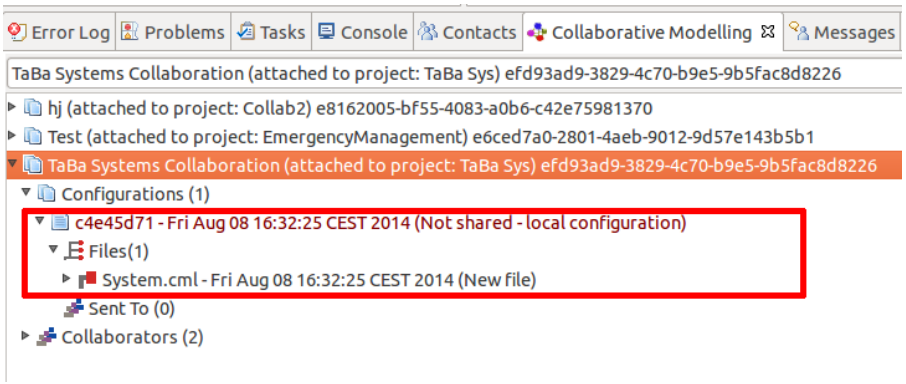

Figure 38: New non-shared configuration

the other collaborators. Once shared, the particular Configuration will become read-only.

Sharing Model Data

A model is shared by right clicking on the Configuration and clicking *Sign and Share Configuration*, as shown in Figure 39.

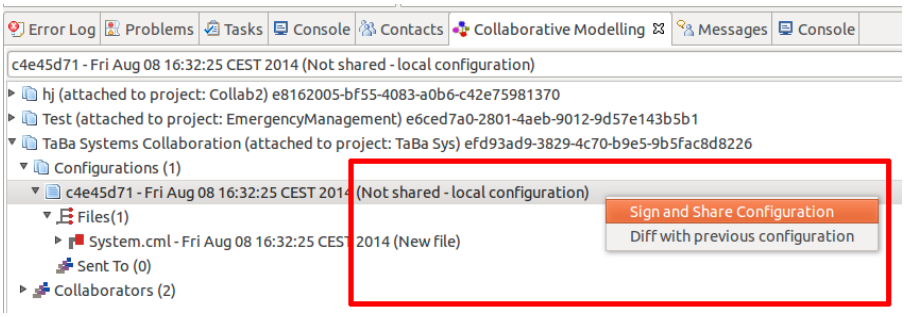

Figure 39: Signing and Sharing configuration

When a Configuration is selected for sharing, it is signed with the Collaborator's name, a snap-shot is taken of the current contents of the files in the Configuration and the Configuration is locked from further changes. The signed and locked Configuration is then pushed to the other collaborators in the Collaboration Group, and as such new changes in the collaboration will appear instantly for all collaborators. This is done in order to show the current development and changes as they occur, as the aim is to support a collaborative work process and not to function as a versioning system.

When a Configuration is received it will appear in the Collaboration View, as a Configuration postfixed with "Received". Received model data will not become part of the overall model before the receiver has approved of it. Meaning that the tool will not include the received model data into the project workspace until the receiver manually includes it into the workspace. As a result, all the files received via Configurations are

stored within the project, but do not become visible in the project explorer, and thereby for analysis in the tool, until requested.

Activating Received Model Data

A received Configuration is made active in the workspace by right clicking the Configuration and selecting the *Activate Configuration* menu, as show in Figure 40.

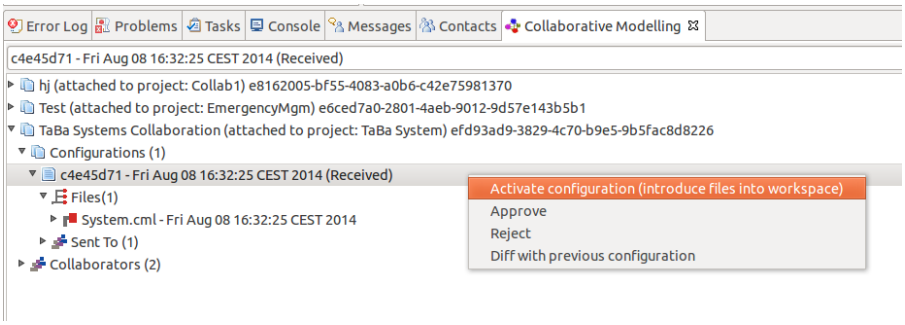

Figure 40: Received Configuration

When activating a Configuration the files it contains will be introduced in the workspace. If a file does not exist in the workspace it will be created. If the files already exist, a dialog will appear that lists the files that will be overwritten and prompt the user to approve that new versions of these files will be created. Please be aware that overwritten files in the workspace cannot be undone by the Collaborative Modelling plugin. As the plugin is not a versioning system, the user should use other technologies for ensuring versioning of the files in the workspace.

Approving, Rejecting and Evolving Model

When a new Configuration is received its contained model data is to be considered as a proposal as to how a particular part of the SoS could be designed. The collaborators that receive the Configuration have the option of either Approving, Reject or to evolve the model further. Approval indicates that the collaborator accepts the entire proposed model. Reject indicates that the collaborator will not use any parts of the proposal, and it is discarded. When a collaborator approves or rejects a Configuration, the other collaborators in the collaboration group will be notified of the decision. The status of a Configuration can be seen under the “Sent To” node of the specific Configuration. An approval by a collaborator is indicated by green, while a rejection is indicated with red, as illustrated in Figure 41.

A Configuration is approved or rejected by right clicking the specific Configuration and selecting either the Approve or Reject menu item, as can be seen in Figure 40. In case a Configuration is rejected, a dialog will open in which the reason for the reject decision can be entered.

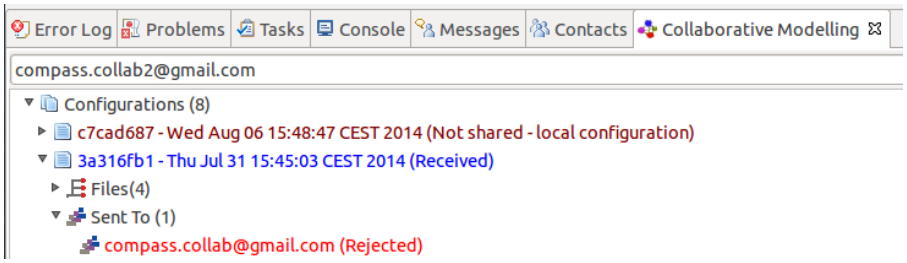

Figure 41: Configuration rejected by collaborator

Evolving and Agreeing on Model Design

Instead of merely approving or rejecting a suggested model design, the collaborator can make adjustments to the files received in a Configuration. When a file is changed, a new Configuration will automatically be created. This new Configuration can then be signed and shared, in the exact same way as described above. Sending updated Configurations back and forward will establish a type of collaboration where the design of the SoS can be evaluated and negotiated by the collaborators, via continuous adjustments of the CML specifications. Each time a new Configuration is received, the new version will be added to a Configuration list, thereby creating a history.

The plugin keeps track of the development and evolution of the SoS model by storing the results of the different modelling sessions performed by the collaboration group. Thereby, the modelling efforts can be resumed and design decisions can be renegotiated and developed as the modelled SoS evolves.

7.4 Distributed Simulation

The Collaborative Modelling plugin also makes use of the co-simulation capabilities, described in Section 6.4. The connectivity that is already established between the collaborators in the Collaboration Group can be used to configure and perform distributed simulation of the collaboratively developed CML models. While the simulation itself makes use of the co-simulation capabilities, described in Section 6.4, the configuration of the simulation is performed without having to manually create co-simulation launch configurations on each of the involved collaborators.

One collaborator will act as the coordinator of the distributed simulation, while other collaborators will act as clients. The collaborator that initiates the distributed simulator will act as the coordinator, and gets to determine which process should be the coordinating process, as well as selecting which processes should be simulated by which collaborator.

The distributed simulation is initiated by right clicking on the Collaboration Project node and clicking *Initiate Distributed Simulation*, as shown in Figure 42. Please note that if the Collaborative Modelling plugin is not connected to an instant messaging account (See Section 7.1) this menu will not appear.

In the appearing *Initialise Distributed Simulation* dialog, shown in Figure 43, the coordinating process has to be selected, as well as the simulate/animate mode in which

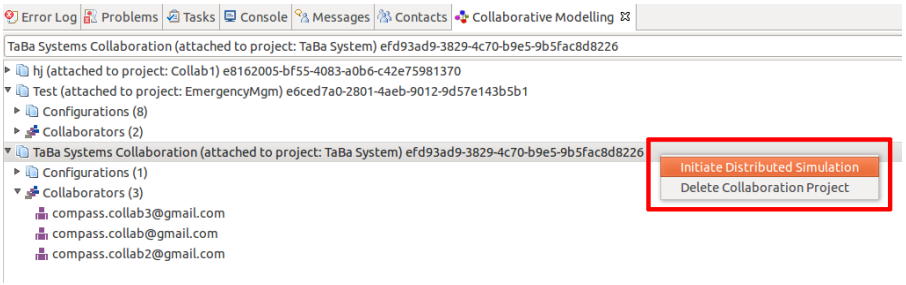

Figure 42: Initiate Distributed Simulation menu

simulator should be running. In the frame below, all the processes defined in the CML project, that the Collaboration Project is attached to, will be listed. Each of these processes can be selected as being an external process that needs to be simulated by another collaborator than the coordinator.

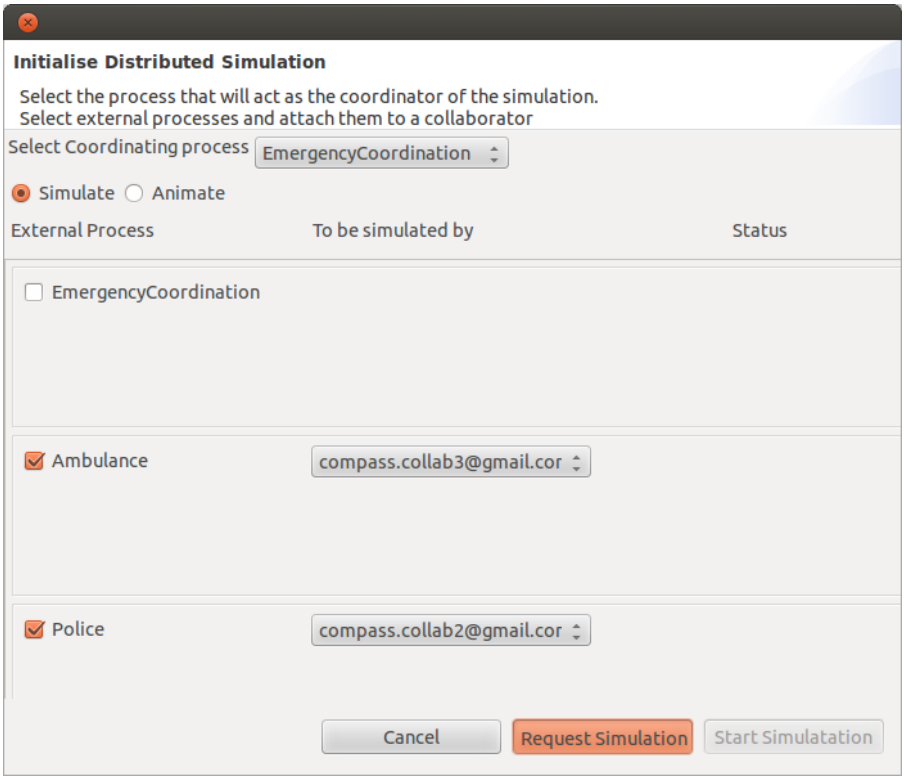

Figure 43: Initiate Distributed Simulation dialog

Once the distributed simulation has been configured, a request can be sent to the selected collaborators, prompting them to participate in the distributed simulation and to simulate specific processes. The collaborators will be presented with the name of the simulations coordinator, the name of collaboration project and the name of the process that needs to be simulated, as illustrated in Figure 44. When receiving a simulation

request, the request can either be declined or accepted. If the request is declined, the dialog will close and nothing more will happen with regards to the simulation. If the request is accepted the dialog will remain open, pending the coordinator starting the distributed simulation.

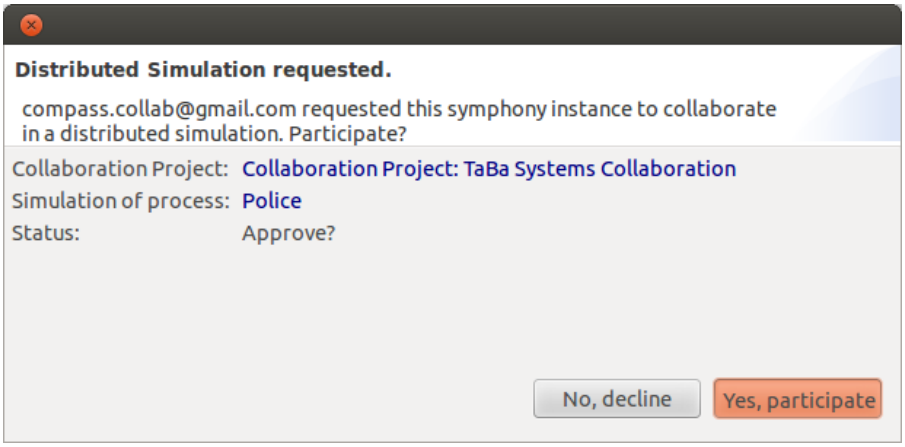

Figure 44: Distributed Simulation requested dialog

As each collaborator replies to the simulation request the status of the reply will be shown on the *Initialise Distributed Simulation* dialog. If all collaborators that have been assigned to simulate a process agrees, the simulation may be started, as shown in Figure 45. If just one of the collaborators rejects participating in the distributed simulation, the whole simulation will be cancelled.

The distributed simulation is started by clicking *Start simulation*, after which the other collaborators will be notified that the simulation is starting. The distributed simulation will then automatically be launched and run as described in Section 6.4.

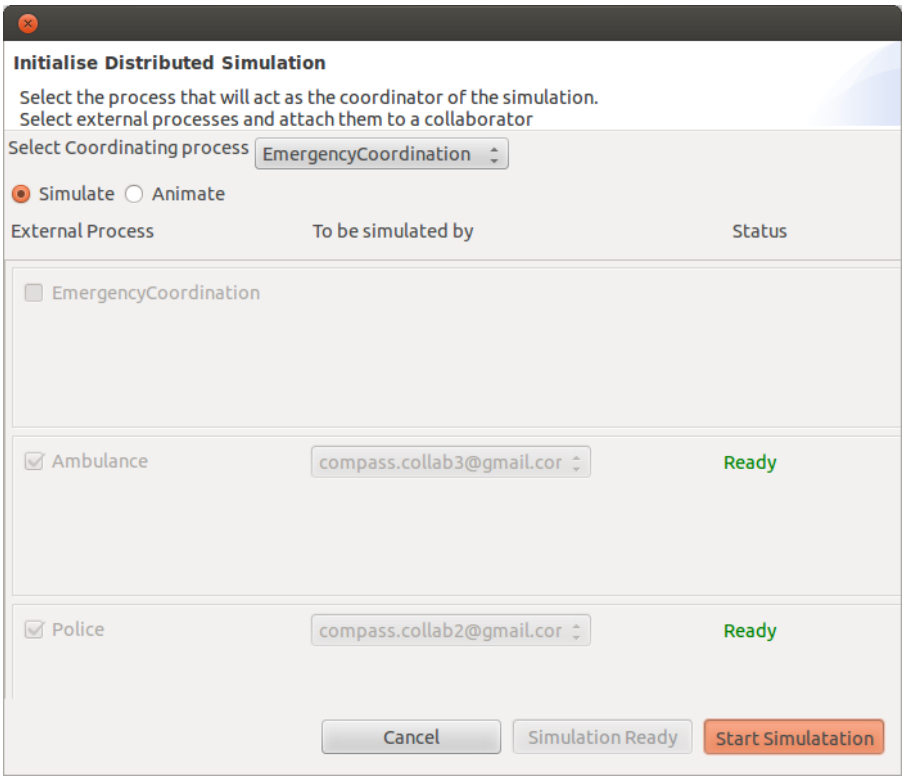

Figure 45: Distributed Simulation ready for start status

8 Proof Support in the Symphony IDE

It should be noted that it is beyond the scope of this user manual to provide detailed descriptions of how to prove theorems in the Isabelle tool, or to provide a tutorial on its use. We therefore recommend that interested parties should read this user manual in conjunction with tutorials on Isabelle and proving in the Isabelle tool, available on the Isabelle website.[8]

Section 8.1 describes how to obtain the software. Section 8.2 describes how to install the software in the Symphony IDE, Section 8.3 explains how to use the Symphony Eclipse perspective and Section 8.4 describes how to prove theorems in the Symphony IDE.

8.1 Obtaining the Software

This section of the user manual assumes that Section 2 has been read and followed.

8.1.1 Isabelle

Isabelle is a free application, distributed under the BSD license. It is available for Linux, Windows and Mac OS X. The tool is available at:

`http://isabelle.in.tum.de`

Instructions for installation for each platform are provided in the following sections:

Mac OS X

Instructions for installation of Isabelle for Mac are as follows:

1. Download Isabelle (version Isabelle 2013-2) for Mac, distributed as a `dmg` disk image.
2. Open the disk image and move the application into the `/Applications` folder.
3. NOTE: Do not launch the tool at this point.

Windows

Instructions for installation of Isabelle for Windows are as follows:

1. Download Isabelle (version Isabelle 2013-2) for Windows, distributed as an `exe` executable file.
2. Open the executable, which automatically installs the Isabelle tool.
3. NOTE: Do not launch the tool at this point.

[8] `http://isabelle.in.tum.de/documentation.html`

Linux

Instructions for installation of Isabelle for Linux are as follows:

1. Download Isabelle (version Isabelle 2013-2) for Linux, distributed as a `tar` bundled archive.
2. Unpack the archive into the suggested target directory.
3. NOTE: Do not launch the tool at this point.

8.1.2 UTP/CML Theories

To prove theorems and lemmas for CML models, Isabelle must have access to the UTP and CML Theories. Instructions for obtaining these theories are given below for different platforms:

Linux, Mac OS X

1. Download the latest version of the `utp-isabelle-x` archive from

   `https://sf.net/projects/compassresearch/files/HOL-UTP-CML/`

   Linux/Mac can choose either `.zip` or `.tar.bz2`.
2. Extract the downloaded theory package and save the utp-isabelle directory to your machine (e.g. `/home/me/Isabelle/utp-isabelle-0.x`).

As the CML and UTP theories are improved, new versions will be made available. As new versions are uploaded, follow the above steps to obtain and unpack the updates.

Windows

1. Download the latest version of the `utp-isabelle-x-windows.zip` archive from

   `https://sf.net/projects/compassresearch/files/HOL-UTP-CML/`
2. Extract the downloaded theory package.
3. Copy the `ROOTS` file from the extracted folder to the `Isabelle2013` application folder (e.g. `C:\ProgramFiles\Isabelle2013-2\`). Windows will warn you a `ROOTS` file already exists. This is ok — choose to replace the existing file.
4. Copy the `utp-isabelle` folder from the extracted folder to the `src` folder in the Isabelle2013 application folder
   (e.g. `C:\ProgramFiles\Isabelle2013\src\`).

As the CML and UTP theories are improved, new versions will be made available. As new versions are uploaded, follow the above steps to obtain and unpack the updates.

8.2 Configuration Instructions for Isabelle/UTP

This section provides the steps required to use Isabelle in the Symphony IDE. This setup procedure is only required on the first use of the theorem prover. However, if a new version of Symphony is installed, then the procedure must be repeated. Instructions for installation with the Symphony IDE are given below:

1. Navigate to the `/contrib/z3-3.2/etc` folder in the Isabelle application folder.
2. Open the `settings` file and uncomment (remove the # character) the line:

   ```
   # Z3_NON_COMMERCIAL="yes"
   ```

   At present, the theorem prover plug-in for Symphony may only be used for non-commercial use.
3. Open the Symphony IDE.
4. From the menu bar, select *Theorem Prover* → *Setup Theorem Prover Configuration*. A Preferences window, as in Figure 46, will appear.

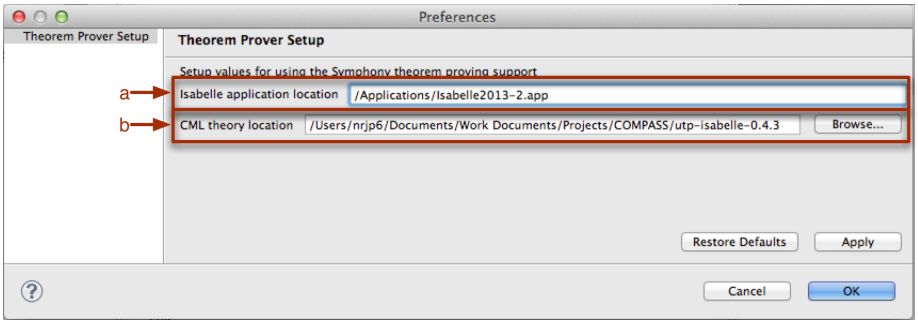

Figure 46: Isabelle configuration setup

5. In the first text box (labelled **a** in Figure 46), supply to the location of the Isabelle application (for example `/Applications/Isabelle2013-2.app`).
6. For Mac and Linux, in the second text box (labelled **b** in Figure 46), navigate to the location of the utp-isabelle folder extracted in Section 8.1.2. Use the *'Browse...'* button to navigate to the correct location if required. This step is not required for Windows.
7. Click the *Ok* button to save the configuration and to finish the setup procedure.
8. Once setup, from the menu bar, select *Theorem Prover* → *Launch Theorem Prover* to run Isabelle. **NOTE:** the first time Isabelle is invoked, several minutes are needed to initialise and build the theories. Subsequent uses of Isabelle will not require this long wait. To monitor progress, click on the button on the bottom right of the tool, as highlighted in Figure 47.

Troubleshooting More detailed instructions are provided at the Isabelle/Eclipse website, which may be of use:

Figure 47: Isabelle configuration in Symphony— initialisation progress.

`http://andriusvelykis.github.io/isabelle-eclipse/getting-started/`

If errors persist, please report them using the Symphony bug tracking facility[9]:

`https://github.com/symphonytool/symphony/issues/new`

8.3 Using the Isabelle perspective with the Symphony IDE

The steps in this section should be followed to begin proving theorems using the Isabelle theorem proving support plug-in for the Symphony IDE. The steps enable the user to prove theorems for a specific CML model.

1. If Isabelle is not already running, from the menu bar, select *Theorem Prover* → *Launch Theorem Prover*. This will run the Isabelle configuration (defined in Section 8.2). If there is already an instance of Isabelle running, an error message will appear, as in Figure 48. This can be safely dismissed.

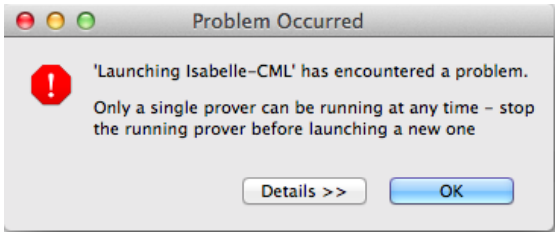

Figure 48: Error messages from Isabelle in Symphony

2. When proving in the Symphony IDE, the Isabelle perspective is used. To open the perspective manually, select the icon labelled **a** in Figure 49, and then select *Isabelle* and then *ok*. If the perspective has been used previously, then select the Isabelle perspective using the button labelled **b** in Figure 49.

3. Once open, the Isabelle perspective will look like Figure 50. There are various panes in the perspective as follows:

   Project Explorer Similar to the CML perspective — this pane shows the projects created in the user's workspace, and their contents.

[9] Please note that a GitHub account is necessary for filing bug reports.

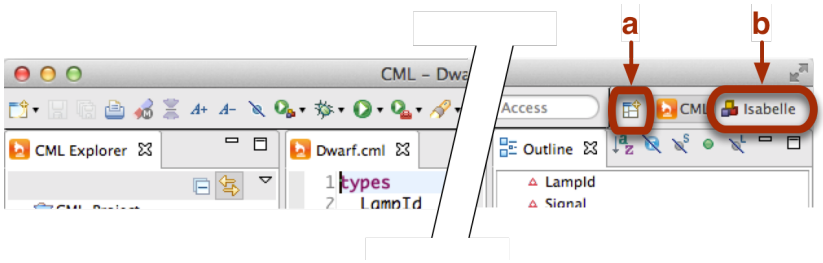

Figure 49: Running Isabelle and selecting Isabelle perspective

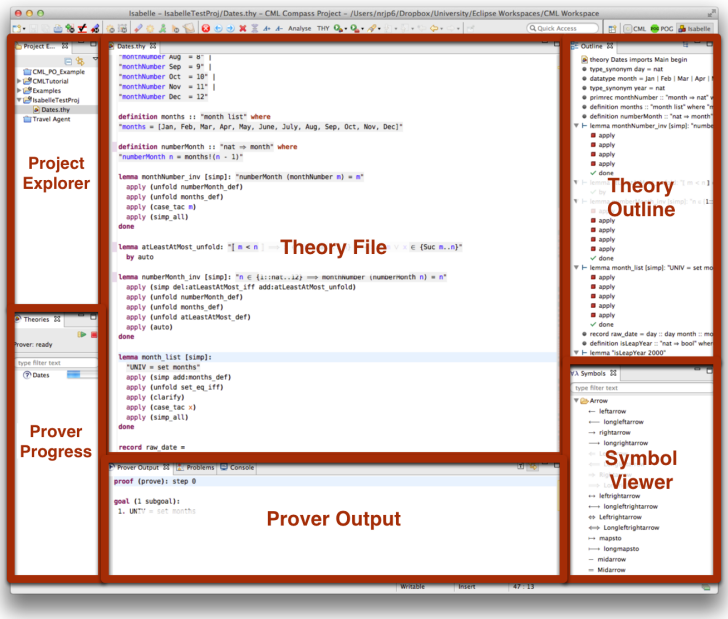

Figure 50: Overview of Isabelle perspective in Symphony

Theory File Editor A text editor which enables the user to interact with the theory script and prove theorems, add additional definitions, lemmas and theorems.

Theory Outline This pane provides an outline to the contents of the selected theory file including definitions, functions, lemmas and theorems and may be used to navigate the theory file.

Prover Progress A collection of status bars for the currently open theory files — shows the progress made by Isabelle in proving the scripts in the theory file.

Prover Output A window to report error messages and the status of the goals of selected theorems.

Symbol Viewer A quick method of adding mathematical symbols to a theory

file. The user can double-click a symbol which will be added to the proof script.

4. Using the theory file editor pane, theorems and lemmas may be defined and proven. The theory editor of Isabelle/Eclipse provides an interactive, asynchronous method for theorem proving, similar to the jEdit interface distributed with Isabelle. The theory file is submitted to Isabelle and results are reported asynchronously in the editor and prover output panes. The editor has syntax highlighting for the Isabelle syntax[10] and problems are marked and displayed in the output pane.

In the next section, we use the steps defined here to use the Isabelle perspective to prove lemmas related to an example CML model.

8.4 Proving CML Theorems

In this section, we provide a brief overview to theorem proving in the Symphony IDE. As proving theorems about a CML model in Isabelle is performed in much the same way as normal theorem proving in Isabelle, the interested reader should refer to tutorials on theorem proving with Isabelle for more details. We consider two main methods of theorem proving in the Symphony IDE; discharging POs and model-specific conjectures. In Section 8.4.2, we describe the initial Proof Obligation Gneration – Thoerem Prover (POG-TP) link and in Section 8.4.3 we consider general theorem proving in CML.

8.4.1 Inspecting Proof Obligations

This section provides brief instructions on how to use the POG plug-in to inspect Proof Obligations (POs). Information on which CML constructs are supported by the POG can be found in Appendix B.

In order to inspect the POs for a given CML model, the, the user must select a CML project (or a CML file), right-click and select *Theorem Proving* → *Inspect Proof Obligations* from the context menu. This is shown in Figure 51.

Once the POG has run successfully, the generated POs are displayed in a view to the right of the editor (if the default POG perspective is enabled). If you click on any PO in this list, its predicate can be seen in a frame below the PO list. This is shown in Figure 52.

Any PO can be double-clicked and the editor will highlight the relevant portion of the CML model that yielded that PO, which is also illustrated in Figure 52.

POs can be displayed in one of two ways: full obligation (the default setting) or definedness predicate only. The full obligation shows all contextual information for each PO whereas the definedness predicate only shows the portion of the PO that enforces definedness with no contextual information. This version can sometime be more intuitive to comprehend. To switch between settings, click *Window* → *Preferences* → *Symphony* → *POG* (see Figure 53).

[10] It is beyond the scope of this document to describe the Isabelle syntax – interested readers are directed to the Isabelle tutorials, as in footnote 8

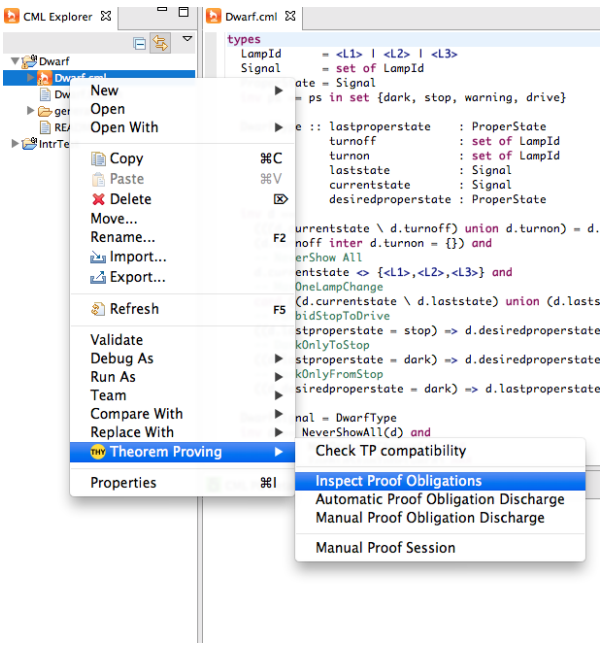

Figure 51: Invoking the Symphony POG.

8.4.2 Discharging Proof Obligations

POs are automatically generated by the POG and can be discharged by the theorem prover. Symphony supports both fully-automated and manual discharge of proof obligations.

Automatic discharge is the preferred method and should be attempted first. To perform automatic discharge, right-click the model in the project explorer and select *Theorem Proving* → *Automatic Proof Obligation Discharge*. This is shown in Figure 54.

This process checks that the CML model is supported by the theorem prover. If there are any parts of unsupported syntax in the CML model, an error message appears which informs the user. A list of unsupported syntax is reported in the warning pane of the Symphony IDE. The individual POs are also checked to ensure they use syntax supported by the theorem prover.

If the model is supported, the POG perspective will be shown, as per Figure 55. From here, users may submit individual POs to the theorem prover by right-clicking them and selecting *Submit PO to Theorem Prover*. Alternatively, all POs can be submitted at once by clicking *Submit All POs* – the button labeled **a** in Figure 55.

Finally, it is possible to terminate the Isabelle session by clicking *Terminate Session* – the button labeled **b** in Figure 55. However, this functionality is still experimental and once a session is terminated, Isabelle may not start up again correctly. If this happens, please restart Symphony.

Automatic discharge is capable of handling a significant number of POs. However, it may be unable to discharge particularly complex obligations. If automatic discharge

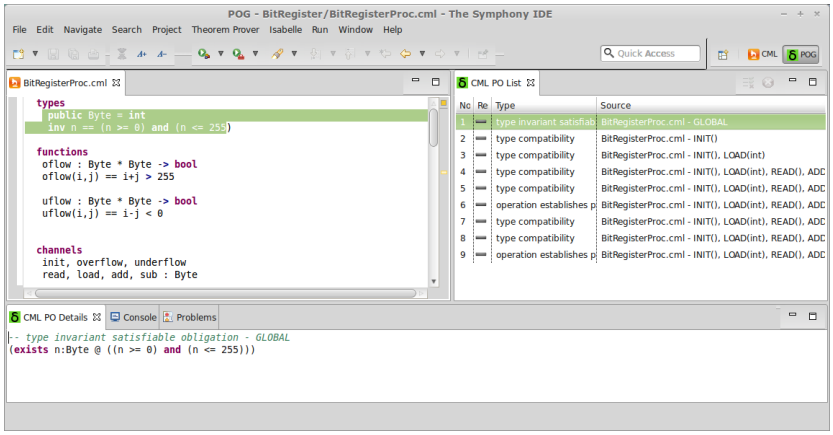

Figure 52: Results of proof obligation generation

fails, users may attempt manual proofs.

To attempt a manual proof, right-click the model in the project explorer and select *CML-THY → Manual Proof Obligation Discharge*. This is shown in Figure 56.

This process checks that the CML model is supported by the theorem prover. If there are any parts of unsupported syntax in the CML model, an error message appears which informs the user. A list of unsupported syntax is reported in the warning pane of the Symphony IDE. The individual POs are also checked to ensure they use syntax supported by the theorem prover.

If the model is supported, the theorem prover plug-in creates two theory files with the `.thy` file extension: a model-specific, read-only, file for the CML model (`<modelname>.thy`) and a user-editable file (`<modelname>_PO.thy`). These files, along with a read-only version of the CML model, are added to a timestamped folder in the `PROJECT\generated\POG` folder of the CML project (see Figure 57). Note — this file is specific to the current state of the model. Any changes made to the CML model will not be reflected in the `.thy` file, and thus the process must be restarted. The generated model `.thy` file uses a combination of regular Isabelle syntax and the Isabelle syntax defined for CML described in more detail in [FP13].

In the `<modelname>_PO.thy` file, each supported PO is represented as an Isabelle lemma with the proof goal being the PO expression. Each lemma initially uses the "`by (cml_auto_tac)`" proof tactic to attempt to discharge the PO automatically. If this is not successful, indicated by red error symbols next to a lemma, the user should either use the keyword `oops` to skip that PO, or attempt to prove the lemma manually. In the next section, we introduce the steps for a more manual approach to theorem proving in the Symphony IDE, including more details on the different `.thy` files generated by the Symphony theorem prover.

8.4.3 Discharging Model-Specific Validation Conjectures

To illustrate the process of proving model-specific conjectures, we use an example introduced in Part B of deliverable D33.2 [FP13] — the Dwarf signal controller.

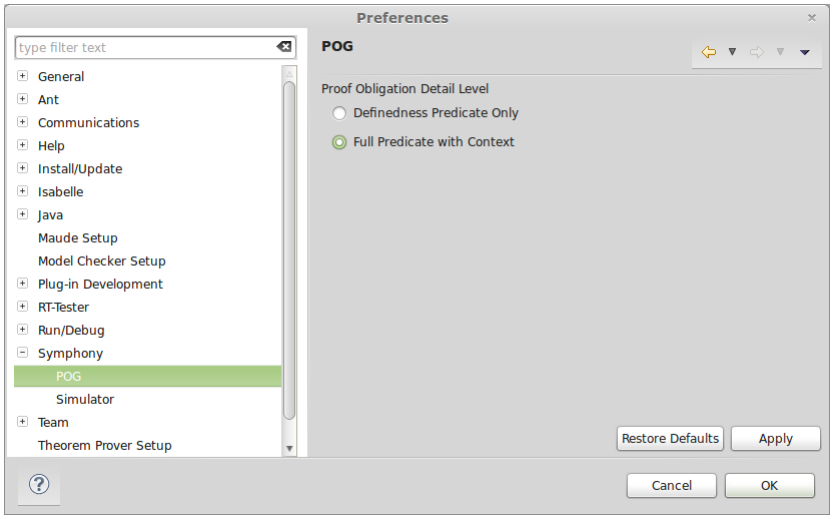

Figure 53: POG preference page

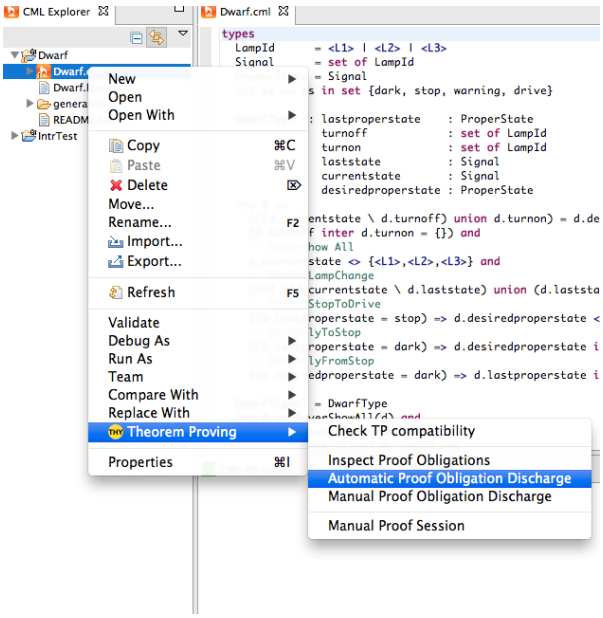

Figure 54: Invoking automatic proof obligation discharge.

With a CML model open, right-click on the model filename in the Project Explorer, and select *Theorem Proving* → *Manual Proof Session*, as shown in Figure 58.

The CML model is first checked to ensure it is supported by the theorem prover. If there are any parts of unsupported syntax in the CML model, an error message appears which informs the user. A list of unsupported syntax is reported in the warning pane of the Symphony IDE.

If the model is supported, the theorem prover plug-in creates two theory files with the

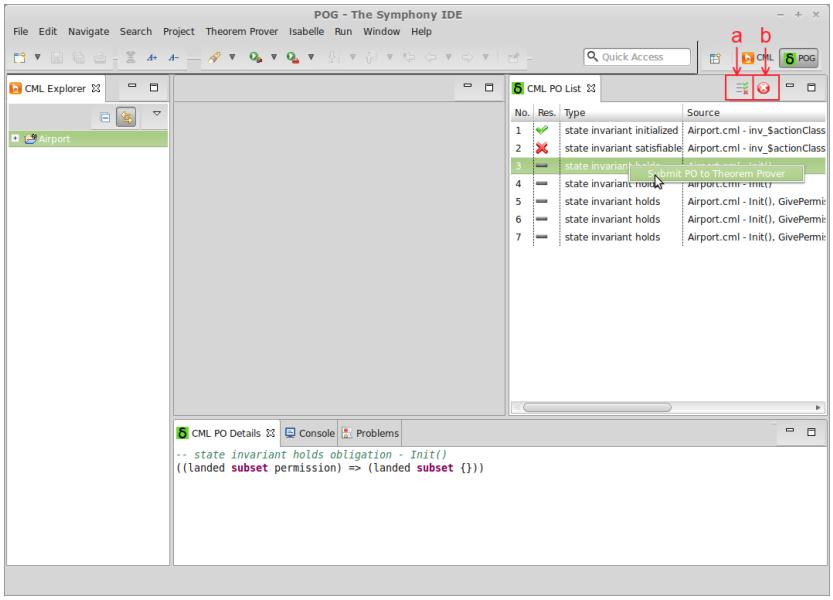

Figure 55: Proof Obligation Generation/Proof Session Perspective

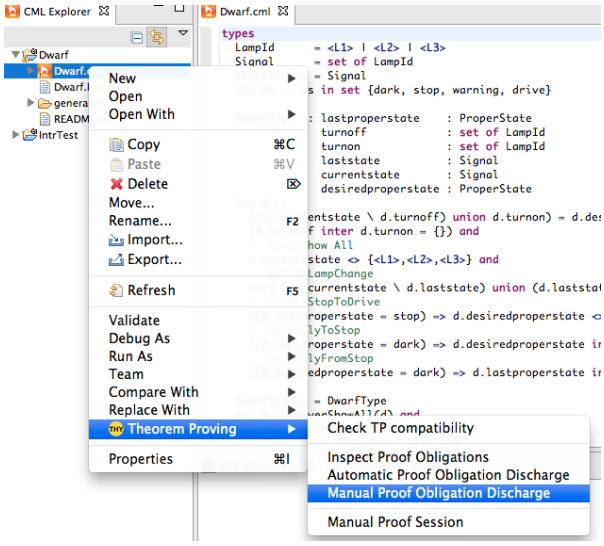

Figure 56: Invoking automatic proof obligation discharge.

`.thy` file extension: a model-specific, read-only, file for the CML model (`<modelname>.thy`) and a user-editable file (`<modelname>_User.thy`). These files, along with a read-only version of the CML model, are added to a timestamped folder in the `PROJECT\generated\Isabelle` folder of the CML project (see Figure 57). As with PO discharging, this file is specific to the current state of the model. Any changes made to the CML model will not be reflected in the `.thy` file, and thus the process must be restarted.

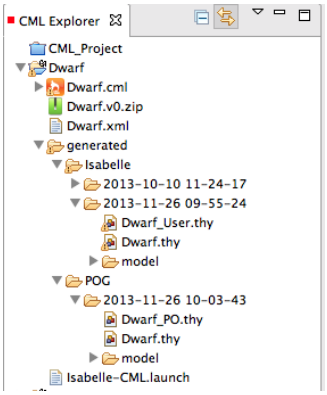

Figure 57: Project explorer with generated `.thy` files

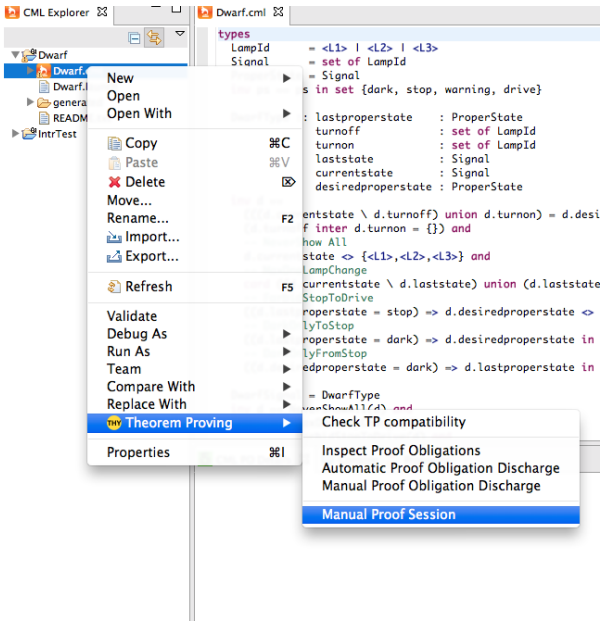

Figure 58: Initiate production of a theory file for a CML model

In Figure 59, the original CML model (`Dwarf.cml`) and the generated `.thy` file (`Dwarf.thy`) are shown in Symphony. The generated `.thy` file uses a combination of regular Isabelle syntax, which is described in various Isabelle manuals and tutorials, and the Isabelle syntax defined for CML. This Isabelle/CML syntax is described in detail in [FP13].

In the corresponding generated timestamped Isabelle directory, a user-editable `.thy` file is produced — in this example, that file is named `Dwarf_User.thy`. This file imports the `utp_cml` theory and the generated Dwarf model theory. We use the Isabelle perspective to start stating and proving theorems and lemmas and can begin to prove some of those theorems introduced in [FP13]. These theorems, named *nsa*, *molc* and *fstd*, are added to the user-editable theory file `Dwarf_User.thy`.

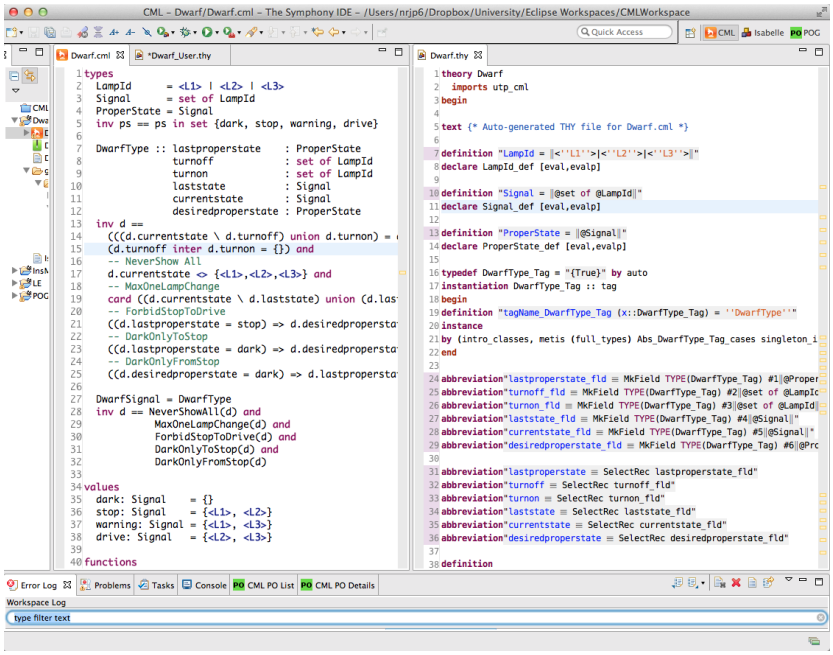

Figure 59: Example Dwarf CML model and generated `.thy` file

The theorems are all simply proved using the `cml_tac` proof tactic. The tactic is applied by using the line "`by (cml_auto_tac)`" on the line below the theorem. This applies rules and tactics defined in the isabelle-utp UTP and CML theories imported during the initial setup of the theorem prover. This tactic is described in more detail in [FP13].

It is beyond the scope of this document to provide detailed descriptions of theorem proving, using the Isabelle tool, or to provide a tutorial on it's use. We therefore recommend that interested parties should read this deliverable in conjunction with tutorials on Isabelle and proving in the Isabelle tool, available on the Isabelle website.

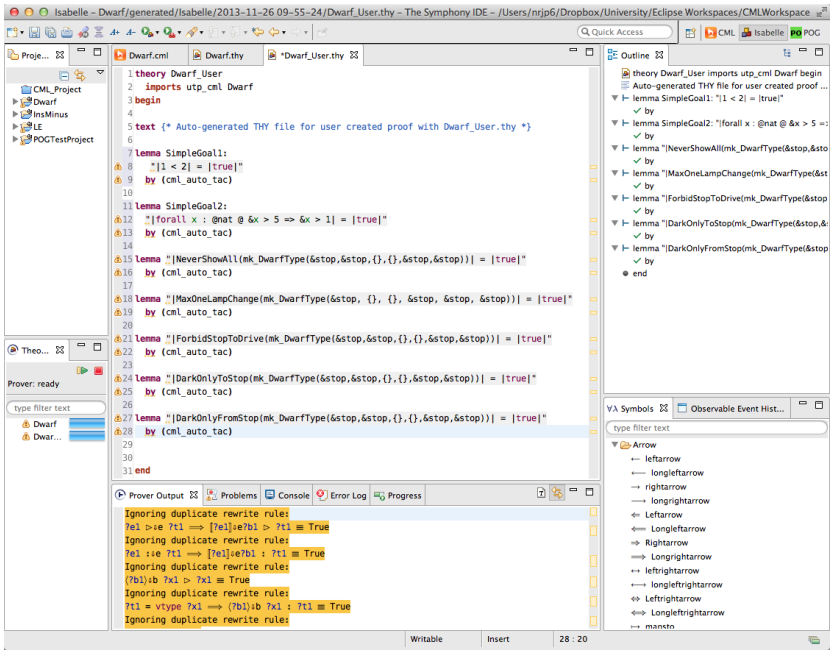

Figure 60: Discharged theorems in Isabelle perspective

9 The Model Checker Plug-In

The CML Model Checker (MC) is part of the Symphony IDE concerned with support for analysing models in terms of classical properties (deadlock, livelock and nondeterminism) and, in the case of a property that is valid, it also provides a useful information about model's behaviour: a trace that validates the analysed property. Furthermore, the model checker provides support for the Fault Tolerance Plugin (detailed in Section 10) in the sense that fault tolerance analysis relies on model checking.

9.1 Installing Auxiliary Software

The CML model checker is developed on top of the Microsoft FORMULA tool and GraphViz. The first is used as the main engine to analyse CML specifications whereas the second is used to show any counterexamples found by the analysis.

The steps to install the CML model checker to work are listed as follows:

1. Download and install the Microsoft FORMULA tool. It is available at

   `http://research.microsoft.com/en-us/um/redmond/projects/formula/`

   Although the tool is free, it requires Microsoft Visual Studio[11]. This makes the current version of the CML model checker platform dependent as the underlying framework is from Microsoft.

2. Download and install the GraphViz software. GraphViz is open source graph visualization software. It allows several kinds of graphs to be written (in a text file) and graphical output generated in several formats to be presented. GraphViz is available at `http://www.graphviz.org/` and can be installed on several platforms. The CML model checker uses specifically the `dot.exe` program, which provides compilation from a textual description to several formats. We use the SVG format that is vectorial and accepted by most of Web browsers.

9.2 Differences from other tools

The main differences of the CML model checker from other popular tools like FDR or PAT include the way these tools are developed, performance and the capability of handling infinite types.

FDR and PAT have been implemented following the usual model checker implementation approach, where search-based algorithms are implemented to provide the desired functionality. The CML model checker, on the other hand, uses the strict relation between the formal semantics of the language and its analysis. Indeed, the CML model checker was created by following the structured operational semantics of CML and is, therefore, semantically well founded. Furthermore, the development cycle of the back-end of the model checker is about a few weeks, assuming knowledge of FORMULA.

[11] `http://www.microsoft.com/visualstudio`.

Concerning performance, FDR and PAT have many internal optimizations that make these tools faster than the CML model checker. Furthermore, as the CML uses SMT solving, it is reasonably expected that it is slower than FDR and PAT.

Regarding manipulation of infinite types, the CML model checker is able to directly handle infinite data types in communication and in predicates, whereas FDR and PAT are not. This is a result of using SMT solving to instantiate and use values based on constraints. Thus, the CML model checker is able to provide answers for cases where FDR and PAT are not.

9.3 Model checker Preferences

The way the model checker instantiates data from infinite domains is configurable in Symphony through the menu `Window -> Preferences`. There is a section "Model Checker Setup" (see Figure 61) that provides a field containing the number of instances that the model checker will use for all infinite types used in any CML model. The default value is 1. The suitable value depends on the specification. For example, if the piece of CML code `ch?x -> Stop` is used, only and `ch` is a channel supporting an infinite type, only one instance is enough to detect the deadlock. However, if the CML code `a?x -> b?y -> if (a <> b) then ...` (`a` and `b` support infinite type) is used, the model checker needs two instances (at least) to perform the correct analysis.

In general guidelines, the number of instances can be determined by looking at the necessary values to explore all branches of the analysed process.

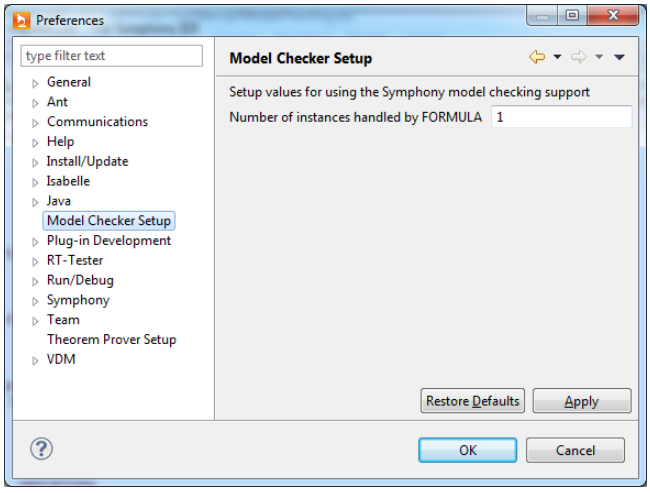

Figure 61: CML Model Checker Preferences

9.4 Using the CML model checker

The model checker functionalities are available through the CML Model Checker perspective (see Figure 62), which is composed by the CML Explorer (1), the CML Editor

(2), the Outline view (3), the internal Web browser (to show the counterexample when invoked) and two further specific views: the CML Model Checker List view (4) to show the overall result of the analysis and the Model Checker Progress view (5) to show the execution progress of the analysis.

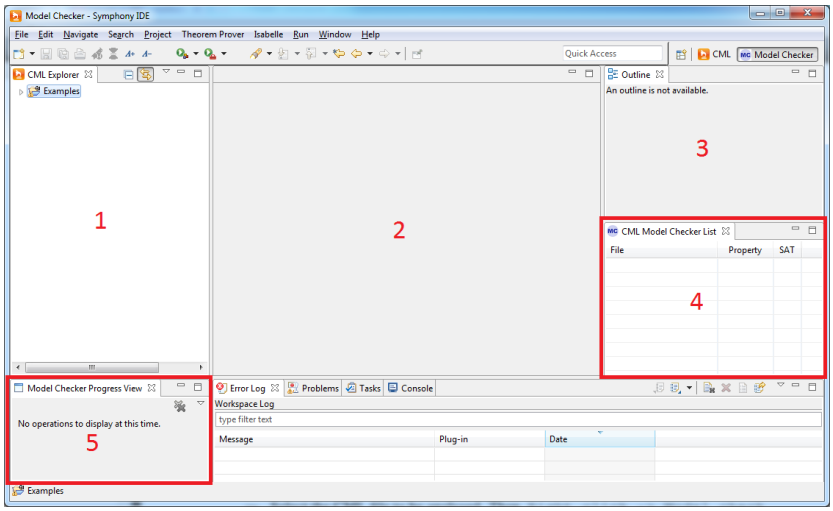

Figure 62: CML Model Checker Perspective

The analysis of a CML model is invoked through the context menu when the CML or the MC perspectives are active (see Figure 63).

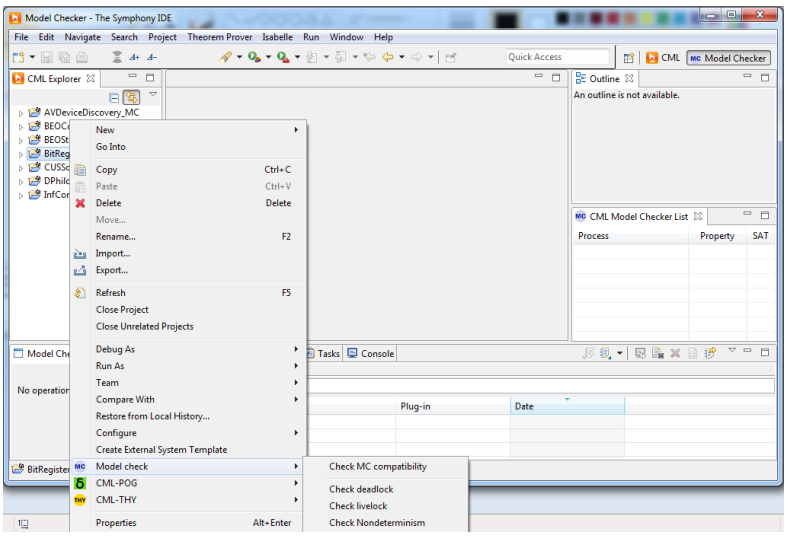

Figure 63: The Model Checker Context Menu

Right click on the CML project (or file) to be analysed and select *Model check* → *Property to be checked*. The option Check MC Compatibility allows a previous check if the constructs used in the model are supported by the model checker. If some constructor is not supported by the model checker, the Symphony IDE shows a warning

message (Figure 64) and the user can see more details by accessing the Problems View (Figure 65).

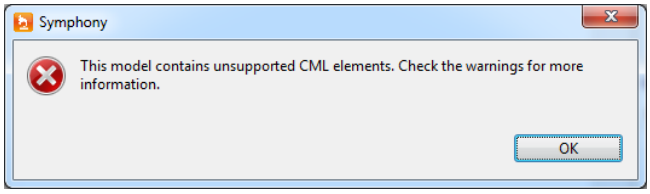

Figure 64: Message about unsupported constructs

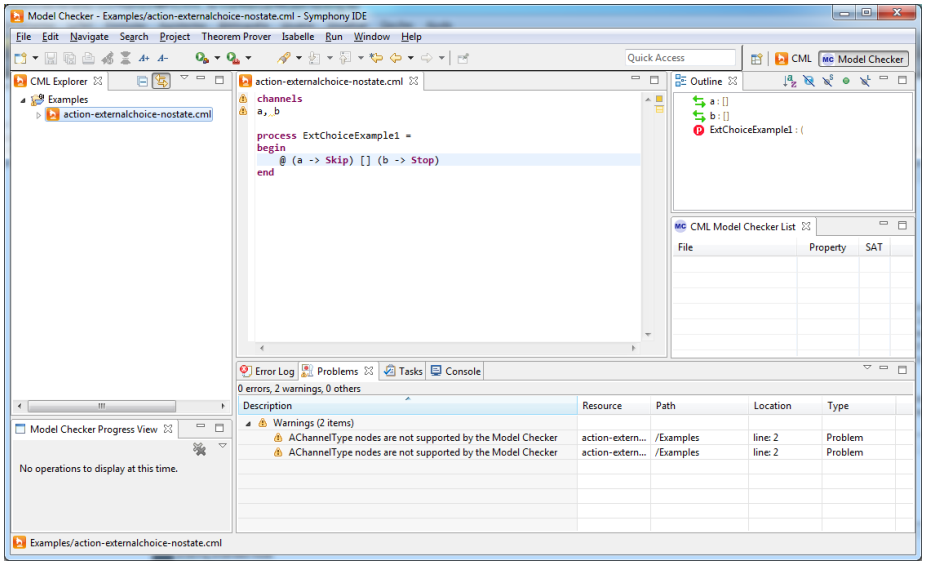

Figure 65: Details about the unsupported constructs

When invoking the analysis, there are several options as result:

- If FORMULA is not installed, the Symphony IDE shows a warning message (see Figure 66), recommending the installation of FORMULA and providing the web site to download the necessary software and installation instructions.

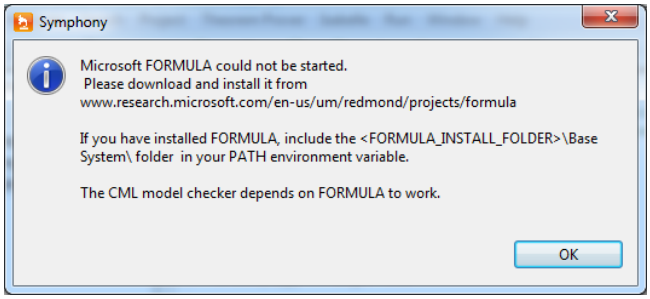

Figure 66: Message: FORMULA not installed

- The analysis is performed and the information is shown in different views. The Model Checker list view shows a ✓ or an **X** as result of the analysis (meaning satisfiable or unsatisfiable, respectively). For each analysis performed over a model there is a corresponding line in the Model Checker List View (see Figure 67). If the user edits the model the results of the previous analysis the results of the previous analysis are still maintained in the Model Checker List View.

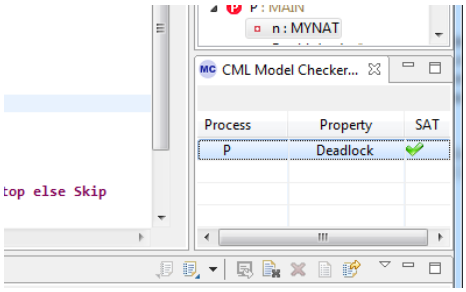

Figure 67: Successful analysis

It is worth mentioning that the user can cancel the analysis by pressing the cancel (or stop) button in the Model Checker Progress View component. Furthermore, for satisfiable models, a graph containing the trace validating the property is accessible by a double clicking the item in the Model Checker List View. If the GraphViz tool is not installed, the Symphony IDE shows a warning message (see Figure 68), recommending the installation of GraphViz and providing the web site to download GraphViz.

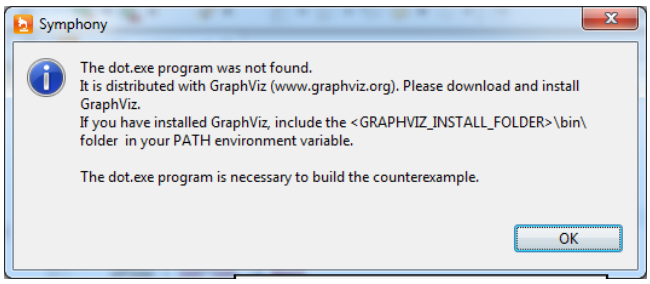

Figure 68: Message: GraphViz not installed

- A conversion error occurs. This kind of error occurs less frequently and represents a translation error from CML to FORMULA and should be reported as explained in Chapter 2. The Symphony IDE shows a popup with this information, as illustrated by Figure 69.
- An internal FORMULA occurs. If some error occurs during the analysis in FORMULA, the Symphony IDE shows a corresponding popup (see Figure 70). Detailed information about this error will be available at the default Error Log (available at the bottom panel of Symphony, as in Figure 71, or through the menu `Window->Show View->Other->General->Error Log`).

The model checker analysis uses an auxiliary folder (`generated\modelchecker`) to generate the FORMULA file (with extension `.4ml`). This file is given as input to

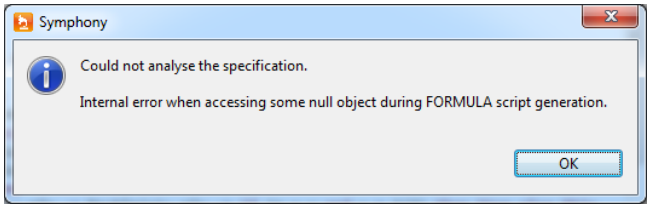

Figure 69: Conversion error

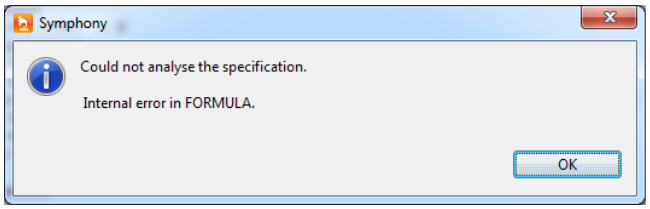

Figure 70: Internal error in FORMULA

the FORMULA tool to be analysed. The graph construction is performed internally (producing a file with extension `.gv`) and using the GraphViz software (actually the `dot.exe` command) to compile the `.gv` file to a graph file (with extension `.svg`). Then Symphony automatically shows the graph file in its internal browser (see Figure 71). All these steps are performed automatically.

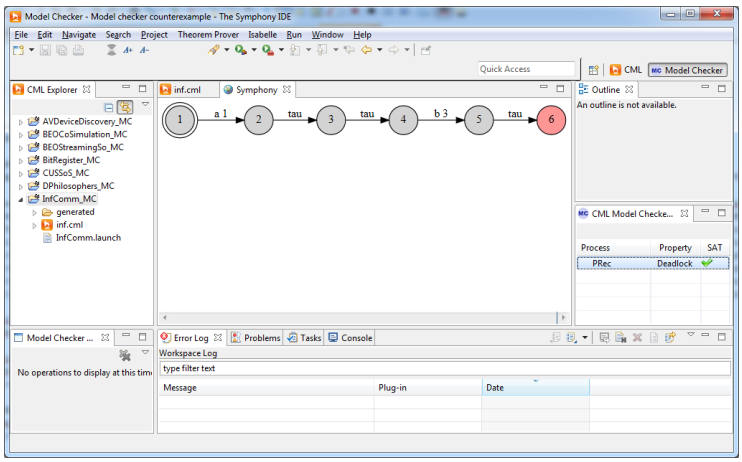

Figure 71: Graph Viewer

The initial state of the graph is two circles; intermediate states are single circles; and the deadlocked state (or other special states related to properties verification) has a different colour (a red tone). Each state has a number and information (hint) about the bindings (from variables to values), the name of the owner process, and the current context (process fragment). To see the internal information of each state just put the cursor over the state number.

Similarly, transitions are labelled with the corresponding event and also have a hint

showing the source and the target states. This feature is useful to provide information about which rule (of the structured operational semantics) was applied.

The internal graph builder of the model checker considers the shortest path that makes the analysed file satisfiable. Thus, although there might be other counterexamples, it shows the shortest one.

9.5 Examples

The Symphony IDE provides some examples that can be imported. This is achieved by a right click on the CML Explorer component and choosing the *Import* option. Then select the option *Symphony* → *Symphony Examples* (see Figure 72). The examples accepted by the model checker contain the suffix "_MC" in their names.

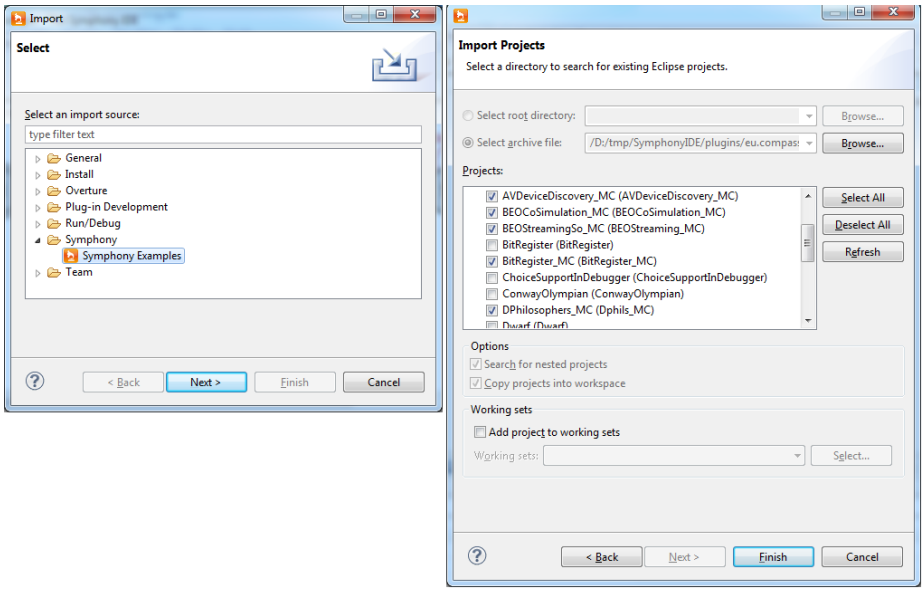

Figure 72: Symphony Examples

Simple Bit Register

The `BitRegister_MC` project contains the model of a bit register. The specification contains operations (to detect overflow or underflow), a state variable, a value to limit the maximum value of the state variable (MAX) and the increment value. The main behaviour is described in the process `RegisterProc` which uses channels to perform increments and decrement until an overflow (or underflow) situation is found. Such a situation generates a deadlock. It is straightforward to see that the `INIT()` operation puts the value `MAX - 1` in the state variable. Depending on the increment value, the trace to deadlock changes. For example, using the value 1, two increments are necessary to cause an overflow (see Figure 73). If one changes the values of increment to 2, only one increment is performed before reaching a deadlock.

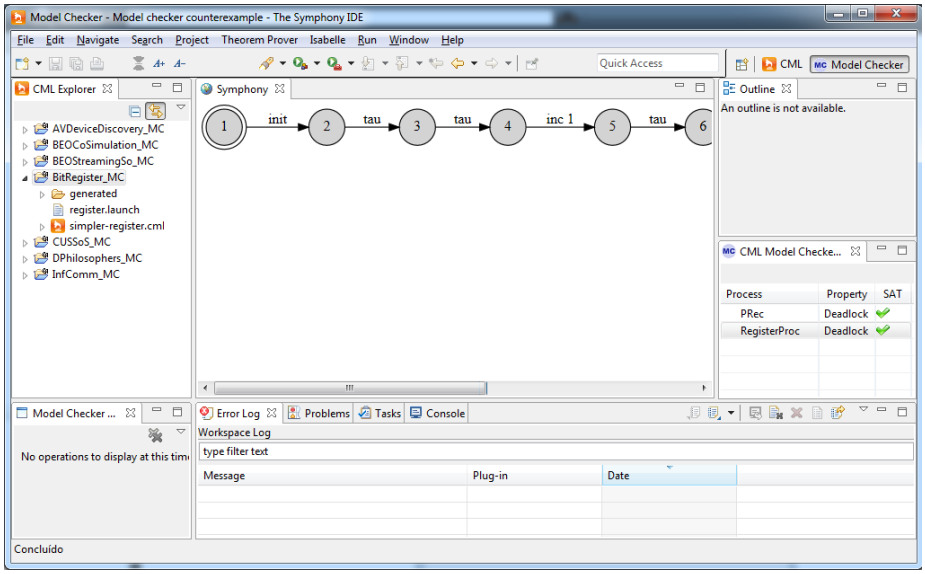

Figure 73: Graph of Bit Register

It is worth noting that the content of any state of the graph is available by putting the cursor over the state. The information of each state has the format `(vars,proc)`, where `vars` contains the manipulated variables (bindings) and `proc` is a process fragment. Furthermore, the generated files can be viewed by refreshing the project. The user can see the content of all files (`.4ml`,`.gv` and `.svg`) as they are text files. Furthermore, a trace provided by the model checker can be exercised directly in the CML animator to provide an interactive way to reach the deadlock.

When the analysed process is unsatisfiable (the property checked is not valid in the model), and the user tries to see the graph, the model checker plug-in returns a message indicating that the graph is available only for satisfiable models (Figure 74). Thus, if the user checks a process for deadlock and it is unsatisfiable, there is no trace leading to deadlock and, hence, does not make sense to build such a trace.

Figure 74: Message when the graph is not available

Models from B & O

There are three models from B & O analysable in the model checker: Streaming, CoSimulation and Device discovery. Concerning the third model, there are constructs like Wait, Timed timeout, interruptions, etc. and there is a a process called

`TargetProduct_DD_SD_InterfaceProtocolView`
that performs an input on channel `IPMulticastChannel`, which supports the type `Device_Record`. Event the invariant of such a type is removed, the process can be analysed and does not presents a deadlock. Furthermore, as these models use timed constructs the model checker will show `tock` events in the graph representation.

Model from Insiel

There is one example from Insiel called CUSSoS. It contains constructs involving parallelism and interleaving of smaller components (processes). The synchronisation between processes is defined by general parallelism where the synchronisation set is defined by specific **`chansets`**.

An Example with Infinite Communications

The `InfComm_MC` project is a simple example containing a process that reads data values in a channel and uses them to update the state variable and to deadlock as long as a predicate involving their values is satisfied. Clearly, this example deals with features not handled by usual tools like FDR and PAT (a corresponding CSP specification is almost the same as the CML file): input events involving values form an infinite domain and predicates involving such values. The values are put in two variables (`x` and `y`) such that there is a deadlock when their values satisfy the expression `n = y and y > 2`, where `n = 2*x`. Furthermore, note that there must be at least two different values manipulated by FORMULA to validate the deadlock (these values are shown in the graph and can be used in the animator to see the deadlock). This property must be set in the Model checker Preferences.

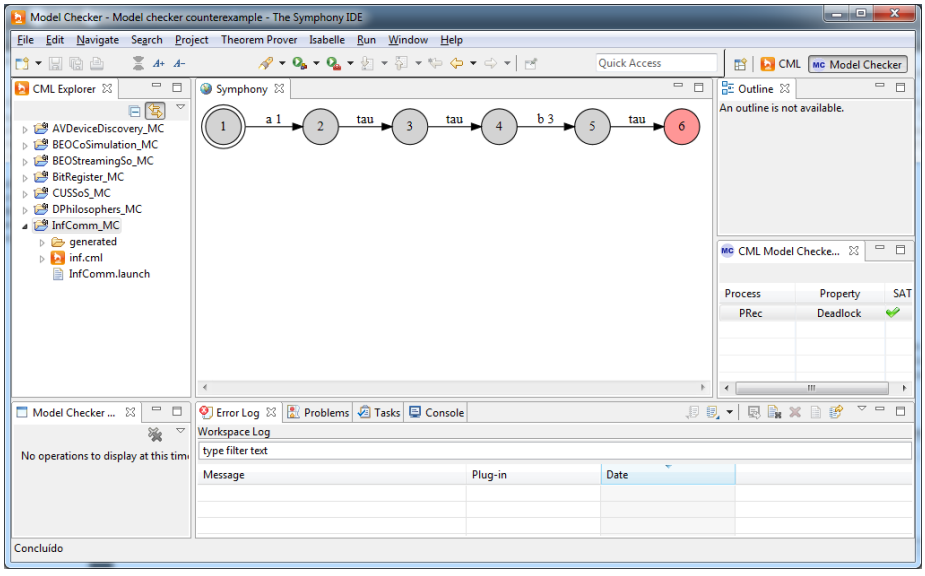

Figure 75: Analysis result with infinite communications

9.6 Syntax Limitations

This section provides tips for users in terms of CML syntax limitations for model checking purposes. Most of these limitations are due to the the embedding of CML in FORMULA and performance. Some of these limitations can also be avoided by the dependency inclusion performed by the model checker when analysing CML processes. That is, even a CML project has many files and processes defined, the model checker includes only the selected process and all other processes it depends on.

Parameters in processes and actions

Processes and actions must have only one parameter to be analysed by the model checker. However, extra parameters can be "emulated" by using local variables (`dcl var:type`) or as state variables, so that their values can be changed and used along the specification. Furthermore, parameters must have primitive type such as int, nat or strings.

Replicated constructs

Replicated constructs can be accepted by the model checker as long as the values used in the indexes can be determined before the translation to FORMULA. This means that sets whose values are dynamically determined when executing the CML model **cannot** be used as indexes of replicated constructs. To overcome this limitation, one can declare the sets of all possible indexes as `values` or `types` in the CML model. For example, the code

```
process Spec =
 begin
  state
   -- this state variable is modified by some actions
   alive : set of NODE_ID := {0,1,2}

  actions
    Alive = []i:alive @ ...
 ...
```

is not allowed as the values in the set `alive` are determined dynamically. On the other hand, if it is possible to use fixed values, the same piece of code can be written as follows[12]

```
values
  alive : set of NODE_ID := {0,1,2}

process Spec =
 begin
  actions
    Alive = []i:alive @ ...
 ...
```

[12]The set `alive` could also be declared as a type in the model.

This causes the conversion to FORMULA to generate an external choice of as many actions as the size of values in the set `alive`.

Names

Names of variables (local, state, communication) must be different because FORMULA does not allow definition of variables (with scope) directly. Some of these constraints are already checked by the parser (for instance, processes using the same names are not allowed).

Concerning state variables, it is not possible (only for model checking purposes) that distinct processes declare state variables with the same name; or it is not possible that two communication (or parameter) variables (in any process or action) have the same name. This was a design decision when implementing the embedding to improve the performance. Manipulation of scope in FORMULA would certainly affect the one-to-one description from the operational semantics as well as performance.

It is worth mentioning that if variables with same names are used in different and independent processes[13] do not represent a problem because only the analysed process will be considered.

Dealing with infinite types

The manipulation of infinite types are implemented in the model checker allows only their use in communication variables. Furthermore, the use of an infinite type is allowed only in input events (`channel?var`) with one parameter. The number of instances of infinite types to be manipulated by FORMULA must be specified in the model checker preferences (see Section 9.3 for details). We recommend to use the the minimum (but sufficient) number of instances. For example, consider a process containing the following example:

```
channels
a:nat

process P =
begin
  state
   n : NATA := 0
  operations
   Double:nat ==> ()
   Double(k) == n := 2*k
  actions
   MAIN = a?x -> Double(x); a?y ->
      if (n = y and y > 2) then Stop else Skip
 @ MAIN
end
```

The analysis of the process `P` can be efficiently analysed considering 2 as the number of instances in the model checker preferences. A brief look at the action `MAIN` shows that: the state value (changed by the input value red by event `a?x`) is compared with

[13] That is, processes that are not reachable from each other.

the second input value (red by the event `a?y`), which must be greater than 2. With only one instance, `x` and `y` would be the same and the condition `n = y` **`and`** `y > 2` would never be valid, disabling therefore, the **`then`** branch (the deadlock).

Another tip is concerned with the use of many input communications in a same channel in recursive processes. This is not recommended because this degrades the performance of SMT solving when choosing them. Instead, one can use different channels to obtain input values so the number of instances in the model checker preferences can be smaller. For example, the process `PRec` defined in the example `InfComm_MC`[14] can be efficiently analysed considering 1 as the number of instances. However, if such a value is set to 2, the analysis becomes more longer.

Expressions and predicates

The use of expressions and predicates in constructs like guarded actions, conditional statements, etc. is restricted to simple expressions. We recommend that only expressions involving the operators `+`, `-`, `*`, `<`, `>`, `<=`, `>=`, `=`, `<>`, **`and`**, **`or`**. The use of complex expressions can make the STM solving activity slower as more constraints are manipulated internally. Furthermore, boolean expressions, involving variables, whose disjunctive normal form have more than two disjuncts are not allowed. This is due to FORMULA limitations on manipulating disjunction. For example, the expressions `value > 5` **`and`** `value < 10` and `a < 5` **`or`** `b = c` are allowed, whereas `a < 5` **`or`** `b = c` **`or`** `b < 3` is not.

Synchronisation values

At the moment the model checker accepts only generic parallelism and only channel names (or set of events) in the synchronisation set. Thus, synchronisation in events with a specific values (for example, `P [| { chan.1 } |] Q` is not allowed).

[14] Importable directly from the Symphony examples

10 The Symphony Fault Tolerance Tool

In this section we present instructions for running the Fault Tolerance plugin via small examples. Section 10.1 shows how to run the Fault Tolerance Plugin. Section 10.2 shows small CML models extracted and modified from Insiel's Emergency Response System (ERS) case study. The Fault Tolerance Plugin works together with the CML model-checker as described in Chapter 9.

In D24.2, we extended the SysML-to-CML mapping rules to allow the verification of fault-tolerance in a SysML model. If these mapping rules are not used, the CML model to be verified must be annotated manually with faults, errors and failures – as is done in the examples in Section 10.2.

When annotating a CML model, several definitions must be provided. The plugin is able to create some automatically, but the following definitions must be provided:

unwanted : <any type> The unwanted events `channel`. It can be of any CML type (for example: `unwanted : int`).

H The set of events (`chanset`) that should be hidden. These events are those related to the recovery mechanisms activation and its internal actions. It can be empty if there is no channel to be hidden.

Alpha_<name-of-the-system-process> The alphabet of the process (`chanset`). It does not include the unwanted events.

Limit_<name-of-the-system-process> This `process` limits the behaviour of the system, avoiding execution of failures, which are those unwanted events that the system is not expected to tolerate. This definition is only used on the Limited Fault Tolerance verification.

In some cases the plugin tries to provide missing definitions, as outlined in Table 1.

Table 1: Cases where the plugin tries to provide missing items.

If not provided	Tries to create it with
`H`	Chansets `ErrorDetectionChannels`, `RecoveryChannels` and `OperationChannels`.
`ChaosE`	CSP's definition of chaos (see [Ros10]).
`NoFault_<system>`	The "no fault" version of `system` (see [APR+13]).
`Lazy_<system>`	Lazy definition of `system` (see [APR+13]).
`LazyLimit_<system>`	Lazy definition of `system` in parallel with the limit process (see [APR+13]).

10.1 Performing Fault Tolerance Verification

To verify a CML model with the Fault Tolerance Plugin, open the CML file in the Symphony IDE, right click on the relevant top-level CML process (either on the project navigator (as shown in Figure 76) or on the CML model (shown in Figure 77)), select *Fault Tolerance* → *Verify*.

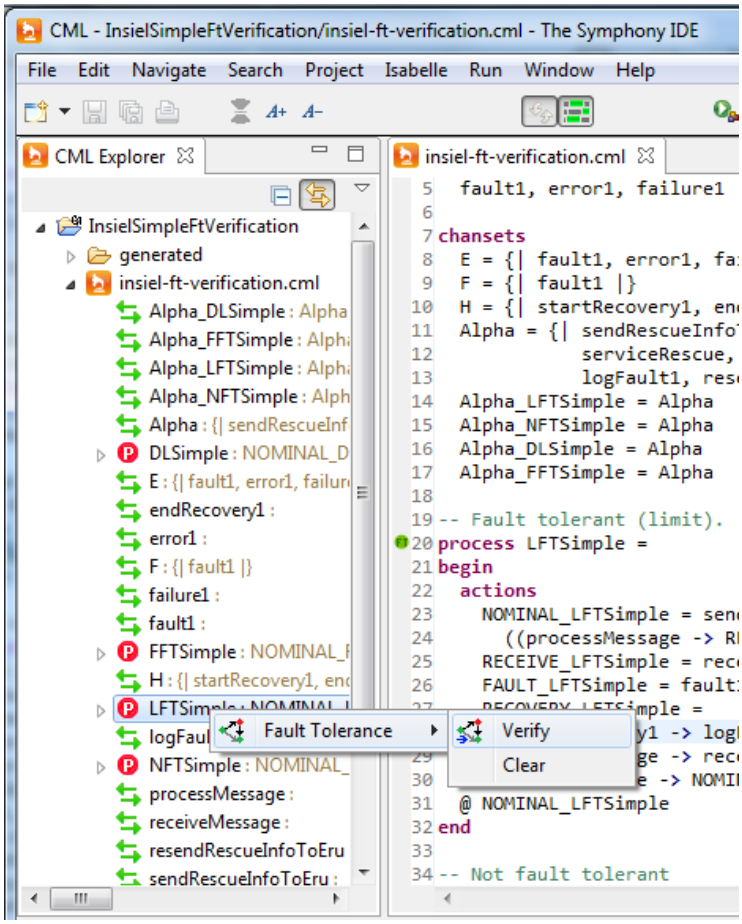

Figure 76: Fault tolerance verification menu on the project navigator

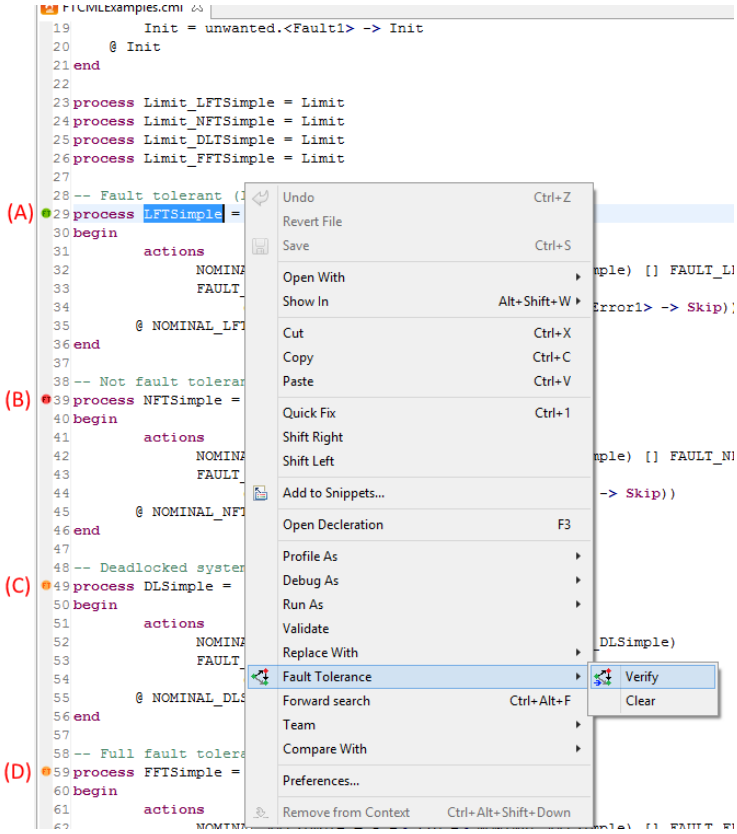

Figure 77: Fault tolerance verification menu on the CML model

Table 2: Messages of fault-tolerance verification plugin

Description	Icon	Message
Missing definition		"To run fault-tolerance verification check: {channels, chansets, processes, namesets}". The tool will show only those missing.
Exception occurred		"Unable to verify fault-tolerance property [property] for process [system] due to an internal error: "[exception message]".
Canceled by user		"The verification of fault-tolerance property [property] for process [system] was canceled by user."
Deadlock		"The system defined by process [system] is NOT deadlock-free. The current version of the model-checker won't enable verification of fault-tolerance of deadlocked systems."
Not divergence-free		"The system defined by process [system] is NOT divergence-free. Total elapsed time: [time elapsed]."
Not semifair		"The system defined by process [system] is NOT semifair. Total elapsed time: [time elapsed]."
Not semifair and not divergence-free		"The system defined by process [system] is NOT divergence-free, nor semifair. Total elapsed time: [time elapsed]."
Not limited fault tolerant		"The system defined by process [system] is NOT fault tolerant with the given limit ([limit process name]). Total elapsed time: [elapsed time]."
Full fault tolerant		"No system should be full fault tolerant. Check design of [system]. Total elapsed time: [time elapsed]."
Limited, but not full fault tolerant		"The system defined by process [system] is fault tolerant with the given limit ([limit process name]). Total elapsed time: [elapsed time]."

If all required definitions are modelled (see Section 10.2), several operations will start running. The first two check the prerequisites explained in [APR+13]: (i) divergence-freedom and (ii) semifairness, and a third will check for deadlock-freedom. If these prerequisites are met, then two further operations are started: one to check if the CML process is fully fault-tolerant and another if the process is fault-tolerant with respect to the limiting conditions (if not limiting conditions are defined, the default considers that the model is able to tolerate all faults). All operations use Symphony's model-checker tool and can be cancelled while they are running or before they run.

The possible messages for errors, warnings and successes are shown as markers beside the top-level CML process, as shown in Figure 77, and are described in Table 2. The elements in square brackets ([]) are replaced by actual values. For example, replace [property] with "Semifairness" and [system] with the process "P". Fault Tolerance Plugin messages are also shown in the Markers pane. If this view is not shown, select the menu *Window* → *Show View* → *Other...*, then select *General* → *Markers*.

To clear any resultant fault tolerance messages, right-click on the relevant CML process or the message itself on the *Markers* view (see Figure 78) and select *Clear fault-tolerant markers*.

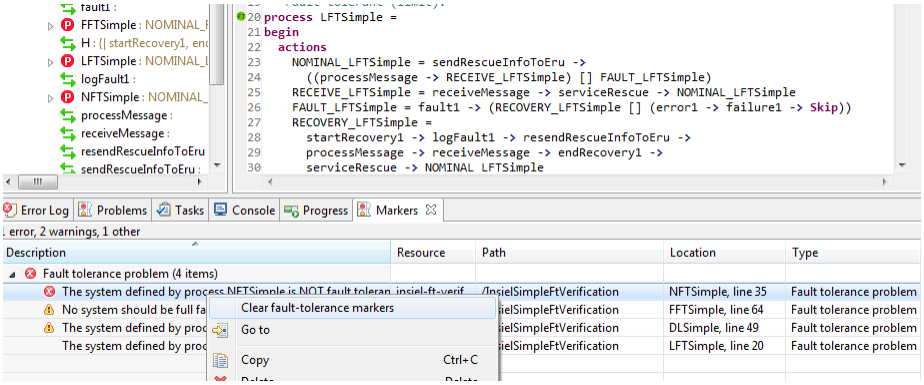

Figure 78: Clear fault tolerance verification messages on the Markers view

10.1.1 Known issues

The Fault Tolerance Plugin has been tested only with small examples (as those shown in Section 10.2) and may, therefore, contain bugs. The only currently known issue is on the first execution of the plugin when Symphony is started. The model-checker preparation phase returns an error when generating the FORMULA files for the CML model. Subsequent executions of the plugin should run normally.

10.2 Fault Tolerance Verification Plugin Example

This section provides small examples to illustrate four key cases of verifying fault tolerance. The examples were extracted from ERS case study and are summarised as follows:

FFT Full Fault Tolerance.

LFT Limited Fault Tolerance.

NFT No Fault Tolerance.

DL The system is deadlocked.

All examples we show use the definitions listed in Listing 2.

```
types
     Unwanted = <Fault1> | <Error1> | <Failure1>

channels
  sendRescueInfoToEru, processMessage,
  receiveMessage, serviceRescue, startRecovery1,
  endRecovery1, logFault1, resendRescueInfoToEru
  unwanted : Unwanted

chansets
  H = {| startRecovery1, endRecovery1, logFault1,
         resendRescueInfoToEru |}
  Alpha = {| sendRescueInfoToEru, processMessage,
             receiveMessage, serviceRescue,
             startRecovery1, endRecovery1,
```

```
            logFault1, resendRescueInfoToEru |}
  Alpha_LFTSimple = Alpha
  Alpha_NFTSimple = Alpha
  Alpha_DLSimple = Alpha
  Alpha_FFTSimple = Alpha

process Limit_LFTSimple = Limit
process Limit_NFTSimple = Limit
process Limit_DLSimple = Limit
process Limit_FFTSimple = Limit

process Limit =
begin
  actions
    Init = unwanted.<Fault1> -> Init
  @ Init
end
```

Listing 2: Basic definitions

10.2.1 Example FFT

The CML model of a system that is fully fault-tolerant is shown in Listing 3[15]. Note that no failure occurs in this system model; only `<Fault1>` occurs, but it is always handled. When the process chooses the `FAULT` action that communicates the `<Fault1>` event, the `RECOVERY` action always communicates the events `processMessage` and `receiveMessage`.

```
process FFTSimple =
begin
  actions
    NOMINAL = sendRescueInfoToEru ->
      ((processMessage -> RECEIVE) [] FAULT)
    RECEIVE =
      receiveMessage -> serviceRescue -> NOMINAL
    FAULT = unwanted.<Fault1> -> RECOVERY
    RECOVERY =
      startRecovery1 -> logFault1 -> resendRescueInfoToEru ->
      processMessage -> receiveMessage -> endRecovery1 ->
      serviceRescue -> NOMINAL
  @ NOMINAL
end
```

Listing 3: Full Fault Tolerance example

The verification of this example raises a warning on the tool (Figure 77 D), as the model does not include any failures. It is recommended that possible failures are modelled – even in the case of fully fault-tolerant systems.

[15] Note: we acknowledge that no *real* system is fully fault-tolerant in general because there is always a non-zero probability of a failure occurring.

10.2.2 Example LFT

The system shown in Listing 4 is not fault-tolerant in general, but it is with regard to a set of limiting conditions representing foreseen faults. Note that the system model includes errors and failures, which are not handled by the defined recovery mechanisms. It represents a real system where detected faults are handled, but for those faults the system is unable to detect, a failure will occur.

```
process LFTSimple =
begin
  actions
    NOMINAL = sendRescueInfoToEru ->
      ((processMessage -> RECEIVE) [] FAULT)
    RECEIVE =
      receiveMessage -> serviceRescue -> NOMINAL
    FAULT = unwanted.<Fault1> ->
      (RECOVERY [] (unwanted.<Error1> -> unwanted.<Failure1> -> Skip ))
    RECOVERY =
      startRecovery1 -> logFault1 -> resendRescueInfoToEru ->
      processMessage -> receiveMessage -> endRecovery1 ->
      serviceRescue -> NOMINAL
  @ NOMINAL
end
```

Listing 4: Limited Fault Tolerance example

When the process chooses the `FAULT` action that communicates the events `<Fault1>`, `<Error1>` and `<Failure1>`, the `RECOVERY` action *can* be chosen. If it does, then it handles `<Fault1>`, communicating both `processMessage` and `receiveMessage`. Otherwise, the system will eventually communicate the event `unwanted.<Failure1>`, so the process is not fault-tolerant in general.

When a limiting condition is imposed, by putting `LFTSimple` in parallel with the `Limit` process shown in Listing 2, then the verification passes[16]. The plugin shows a green marker as depicted in Figure 77 A.

10.2.3 Example NFT

Listing 5 shows a system that is not fault-tolerant, even when considering the limiting conditions. Note that the recovery mechanism does not provide the same events that the nominal behaviour provides. When the `RECOVERY` action is chosen, it will not communicate `processMessage`, nor `receiveMessage`. Figure 77 B shows the corresponding red marker.

```
process NFTSimple =
begin
  actions
    NOMINAL = sendRescueInfoToEru ->
      ((processMessage -> RECEIVE) [] FAULT)
    RECEIVE =
      receiveMessage -> serviceRescue -> NOMINAL
```

[16]The plugin always runs both fault-tolerance verification versions: with and without the limiting condition.

```
    FAULT = unwanted.<Fault1> ->
      (RECOVERY [] (unwanted.<Error1> -> unwanted.<Failure1> -> Skip ))
    RECOVERY =
      startRecovery1 -> endRecovery1 ->
      serviceRescue -> NOMINAL
  @ NOMINAL
end
```

Listing 5: Not Fault Tolerance example

10.2.4 Example DL

The current version of the Fault Tolerance Plugin is unable to check fault tolerance on systems that are deadlocked. Listing 6 shows this example and Figure 77 C its warning mark. The difference to a non-deadlocked system can be subtle (comparing Listing 6 to Listing 4, where both instances of `Stop` replace a recursive call to `NOMINAL_DLSimple`). Both `Stop` action instances cause a deadlock in this system.

```
process DLSimple =
begin
  actions
    NOMINAL = sendRescueInfoToEru ->
      ((processMessage -> RECEIVE) [] FAULT)
    RECEIVE = receiveMessage -> serviceRescue -> Stop
    FAULT = unwanted.<Fault1> ->
      (RECOVERY [] (unwanted.<Error1> -> unwanted.<Failure1> -> Skip ))
    RECOVERY =
      startRecovery1 -> logFault1 -> resendRescueInfoToEru ->
      processMessage -> receiveMessage -> endRecovery1 ->
      serviceRescue -> Stop
  @ NOMINAL
end
```

Listing 6: Deadlocked system example

11 The Refinement Tool

This section describes the use of the refinement tool plug-in, which is distributed with the Symphony IDE. This plug-in takes CML specification and supports the application of refinement laws. This plug-in currently contains a number of simple laws, but can be extended with new refinement laws in two different ways: implementing the refinement law in Java or encoding the refinement law in Maude. In general, the latter option is simpler, more powerful and less error prone. However, for certain types of laws, for instance the copy rule that replaces an action call by its definition, it is easier to implement directly in Java due to the need to search the AST for the definition of the action being called. The details of how the refinement tool can be extended with new laws will be presented in D33.4 [FM14].

11.1 Using the Maude Tool

The Maude tool can be obtained from `http://maude.cs.illinois.edu/`, and installation instructions are provided on that website.

It is important to notice that the use of Maude refinement laws is optional. In order to use Maude in the refinement tool, it is necessary to first install Maude, and then configure the refinement tool in the *Maude Setup* section of the *Preferences* as shown in Figure 79. You must provide the path to the Maude binary, and the path to the Maude encoding `cml-refine.maude` provided with the Symphony IDE. For example, if the Symphony IDE was installed into `/home/user/SymphonyIDE`, then the path in *Maude CML location* should be `/home/user/SymphonyIDE/refinement/cml-refine.maude`.

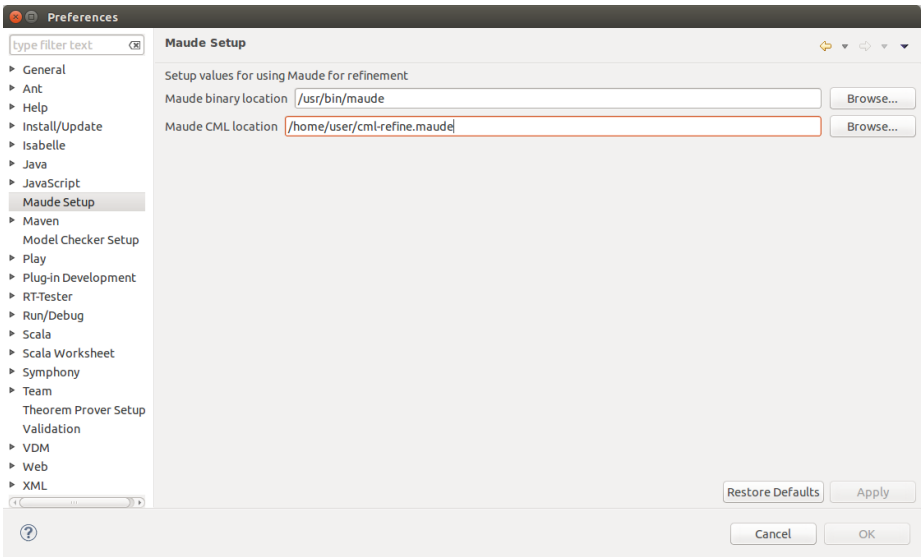

Figure 79: Maude configuration

11.2 The Refinement Perspective

Next, the refinement perspective must be selected as shown in Figure 80.

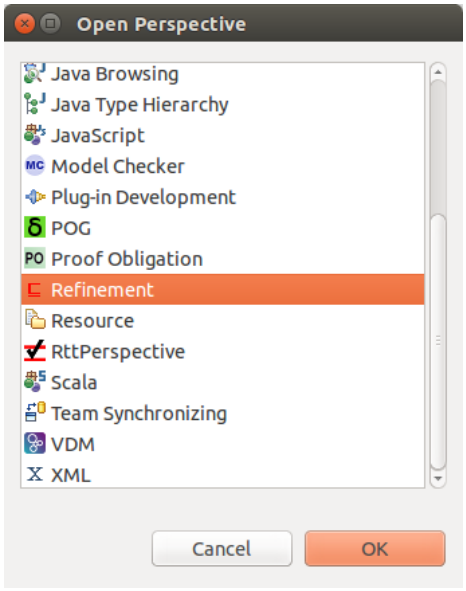

Figure 80: Selection of refinement perspective

The refinement perspective consists of six areas as shown in Figure 81: CML Explorer, CML Editor, Refinement Law Details, Refinement Laws, CML Refinement Proof Obligation (RPO) List, and CML RPO Details. The first two are the same as in the CML perspective. The panel Refinement Laws presents the list of applicable refinement laws, the Refinement Law Details panel shows the details of the selected refinement law, the CML RPO List panel list all the proof obligations generated by the application of refinement laws, and the RPO Details panel presents the details of a selected proof obligation.

11.3 Applying Refinement Laws

In order to apply a refinement law, the excerpt of the specification to which the law is to be applied must be selected as shown in Figure 82.

Next, the applicable laws can be searched by right clicking on the excerpt and selecting *Refinement* → *Refine* (or pressing *Ctrl+6*) as shown in Figure 83.

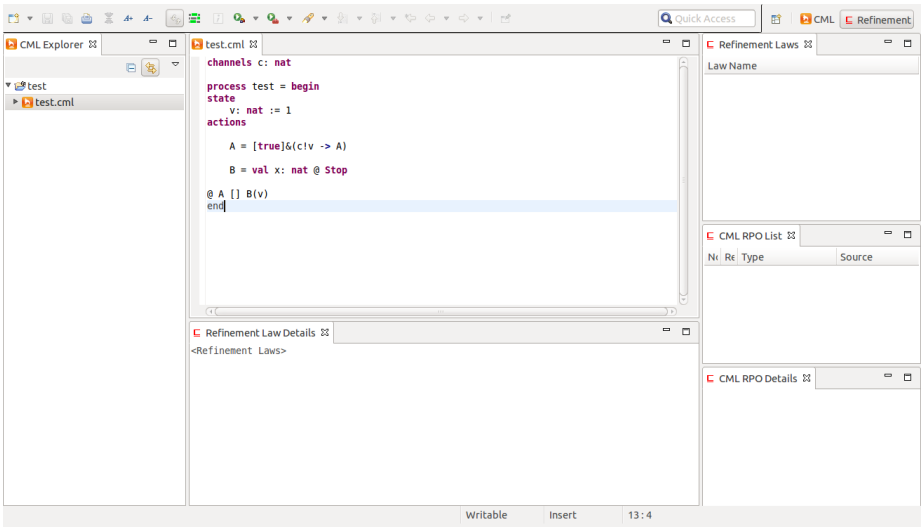

Figure 81: Refinement perspective

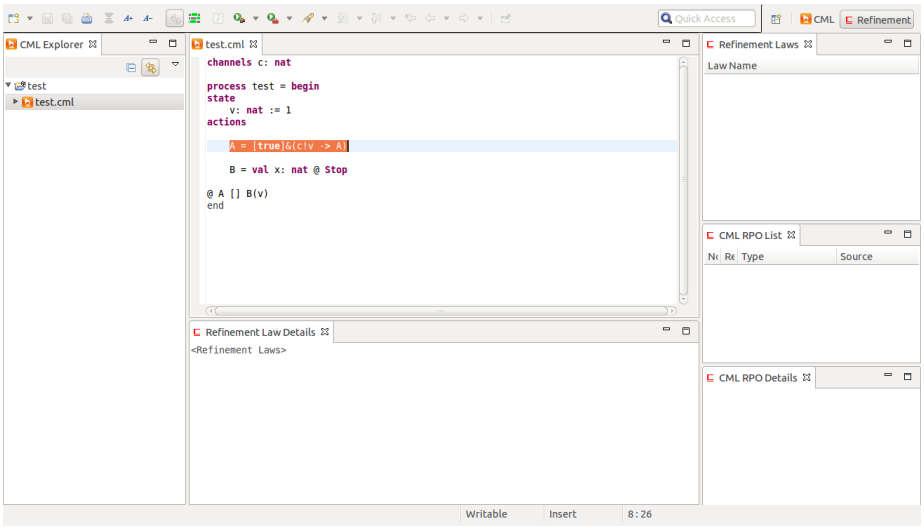

Figure 82: Specification excerpt selection

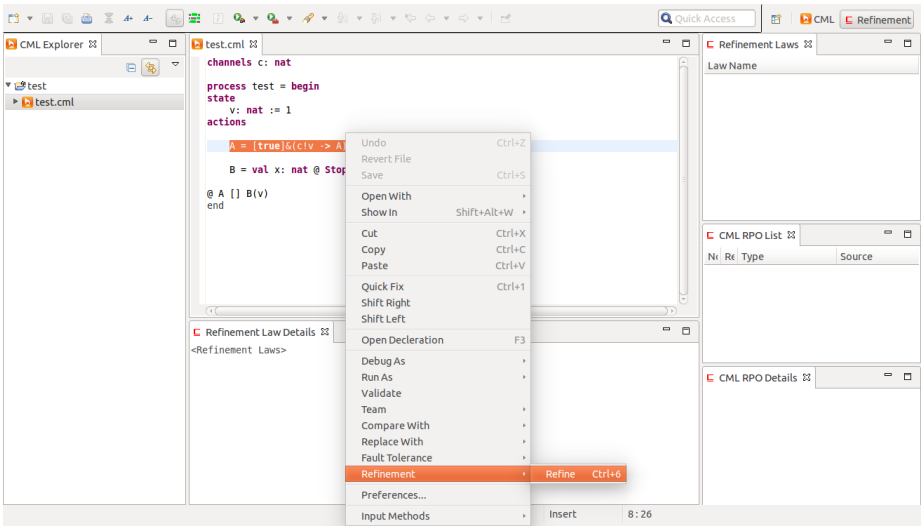

Figure 83: Searching for applicable laws

The applicable laws are then loaded onto the Refinement Laws panel, and a law may be selected as shown in Figure 84. In this case, the details of the law are presented in the bottom panel, and the law can be applied by double clicking it.

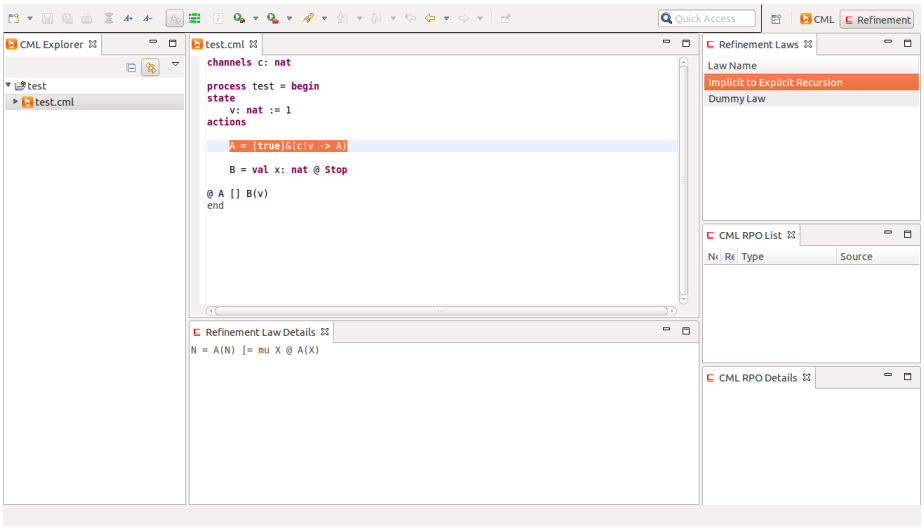

Figure 84: Selection of refinement law

The refined CML specification is shown in the CML Editor, and further refinement laws can be applied (Figure 85). In this example, a refinement law is applied to make the implicit recursion in `A` explicit using the `mu` operator.

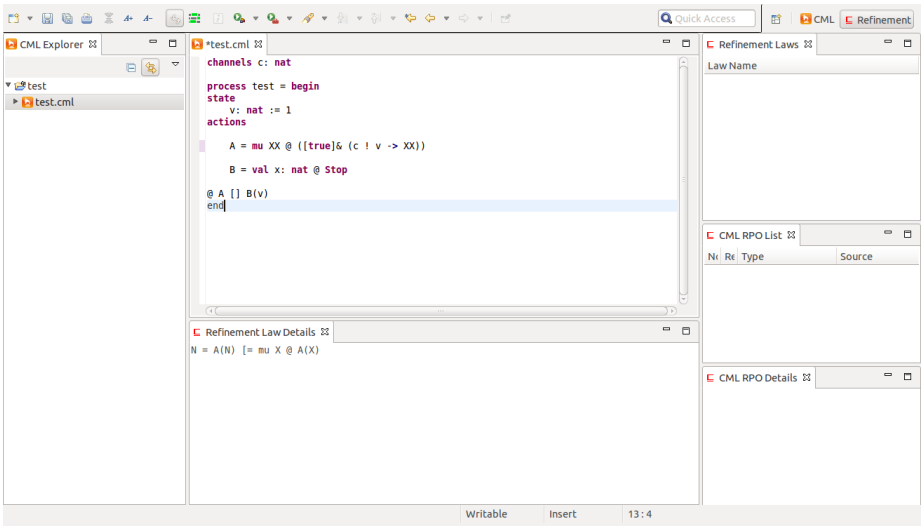

Figure 85: Application of refinement law

As previously mentioned, the details of how the tool can be extended with new refinement laws, as well as the description of other aspects of the tool such as parametrised laws and proviso management is provided in D33.4.

12 RT-Tester Plug-In

This chapter describes the RT-Tester Model-Based Testing (RTT-MBT) functionality provided by Symphony using the RT-Tester plug-in (RTT-Plugin). The user interface is explained together with the basic information about model-based testing with RT-Tester. It covers the areas from creating a test project, selecting and using a specific UML/SysML model to generate test procedures from as well as defining and controlling the test procedure generation process and finally executing and evaluating concrete test procedures.

Model-Based Testing is a complex task and RTT-MBT provides a lot of functionality that can be configured to generate test procedures reaching the desired test goals. The focus of this user manual is to describe the RTT-Plugin and how it can be used as a front-end to RTT-MBT. Additional information about RTT-MBT can be found in a separate manual [Ver13b] that also describes model-based testing with RT-Tester but describes the usage of the RT-Tester graphical user interface from Verified Systems which is not integrated in Symphony. Note that in [Ver13b], you will find additional chapters about generating test models, defining test goals, using the model checking capabilities of RTT-MBT as well as different test strategies and the supported UML and SysML model elements and LTL syntax. These topics are RTT-MBT specific and independent of the graphical user interfaces. While this document concentrates on the RTT-Plugin integrated in the Symphony tool, [Ver13b] and [PVLZ11] are recommended as side reading to this manual.

The tests that are generated by RTT-Plugin within the COMPASS project are RT-Tester test procedures. The RT-Tester manual [Ver13a] provides detailed information about RT-Tester and the test language RTTL, the tests are expressed in.

12.1 RTT-MBT Preferences

A number of preferences can be defined to customize the RTT-Plugin behaviour. If the plugin is installed correctly[17], the Eclipse preferences should contain a separate section RT-Tester with two sub sections Server and Project. The RT-Tester page contains status information about the currently installed RTT-Plugin version, the configured RTT-MBT server, the server version and the uptime of the server. The server version and uptime are unknown (displayed as ”-”) until the `Test server connection` button is activated. This initiates a server connection test and evaluates uptime and server version from the response. The button `Reset server workspace` can be used to erase the data for the configured user id on the RTT-MBT server.[18] [19]

Since RTT-MBT operates in client-server mode, the server name or IP address is required in the RTT-Plugin preferences[20]. Figure 87 shows the Server page of the RT-Tester section in the Eclipse preferences. In addition to the server name or IP address and port number, the name and a user identification has to be provided. A separate server workspace is created for each user and the workspaces are identified with the

[17] The RTT-Plugin is part of the Symphony IDE and should be installed with Symphony automatically.
[18] More information about users and server workspace is given later in this section.
[19] Note: If no server workspace exists for the specified user id, resetting it will fail.
[20] Note: This user manual does not cover the installation of the RTT-MBT server or the RT-Tester backend.

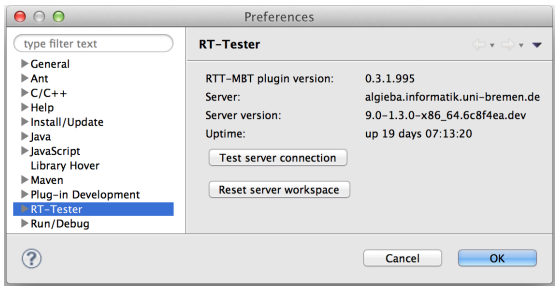

Figure 86: The RT-Tester Preferences Main Page

user id. Therefore the user identification has to be unique within the group of users working on the same RTT-MBT server.

An Apache module can be used to access the RTT-MBT server through a HTTP(S) URL instead of a direct TCP/IP connection. In this case, the complete URL has to be provided in the Server field of the preferences, together with the respective HTTP server port. If the HTTP-Server requires authentication, the credentials must be added in the Username and Password fields of the Server section (see Figure 87 how this is done).

Figure 87: The RTT-MBT Server Settings Page

The RT-Tester core components, the Test Management System (TMS) and the RTT-MBT test generator are executed on the server. The RTT-MBT project files are created and maintained on the client side. For every RTT-MBT task that is to be executed, the client synchronises all required files with the server workspace. It is recommended that you use your real name as the user name and your email address as your user identification. The Server connection can be tested using the main page of the RT-Tester section in the Eclipse preferences.

The Project page shown in Figure 88 of the RT-Tester preferences contains global settings that are used as default values for all RTT-MBT projects, as long as they are not defined in the local project properties of the respective project. The terms Test Generation Context and Test Execution Context are defined and explained in Section 12.3. The Project page contains two fields to define default names for these two directories.

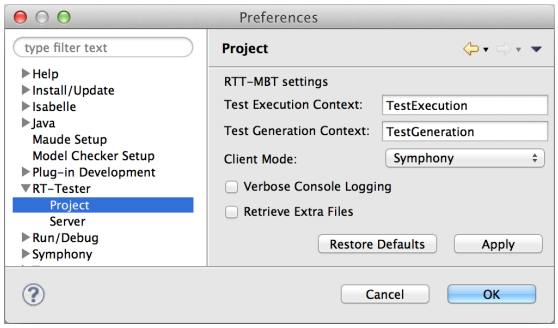

Figure 88: The RTT-MBT General Project Settings Page

The RTT-Plugin can operate in different client modes. The current client mode can be selected using the Client Mode drop-down listbox. Currently only the client modes `Symphony`, `Verified` and `Papyrus` are supported.

The checkbox Verbose Console Logging enables extra output to the RTT-MBT console view during task execution. This output can be helpful to identify problems during test generation or execution. Without this switch, only essential log messages and error messages are sent to the console view.

The Checkbox Retrieve Extra Files enables or disables the download of additional files from the RTT-MBT server after a server task has finished. If this switch is deactivated, all the required files are downloaded. If active, additional log files and other generated files are downloaded that can be useful for debugging.

The Settings for Test Generation Context and Test Execution Context can be overridden by project properties, the settings for the client mode, verbose logging and extra file retrieval can only be defined in the global Eclipse preferences of the RTT-Plugin.

12.2 The RTT-MBT Perspective

RTT-MBT test generation is performed using the RT-Tester perspective (RttPerspective). This perspective is designed to allow model-based test generation and execution of generated test procedures. The perspective shown in Figure 89 consists of a Project Explorer, a Console View, a Progress View, an Outline and a central area for all Editors. The RTT-MBT Toolbar provides quick access to all RTT-MBT commands.

The Project Explorer lets you create, select, and delete projects and navigate between the files in these projects, as well as adding new files to existing projects. It is a central element of the perspective. RTT-MBT commands are normally performed on the selected item in the Project Explorer. The icons in the Toolbar are enabled or disabled with respect to the selected item. If multiple items are selected, only the commands that can be performed for all selected items are enabled. If a command is activated for multiple selected objects, the command is performed for the selected items sequentially, not in parallel[21].

[21] Note that some of the RTT-MBT tasks create or delete files or directories in the local file system. The

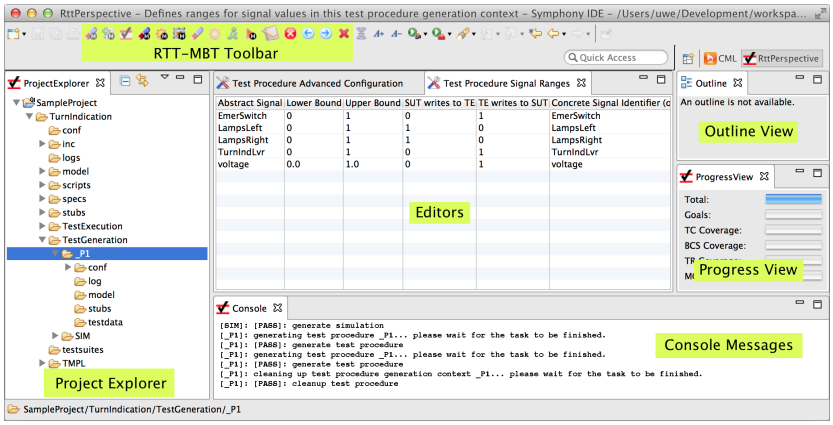

Figure 89: Outline of the RttPerspective Workbench.

The Console View provides feedback in the form of log messages and error messages. The Eclipse build-in progress view provides information about the currently active task. The RTT-Plugin progress view provides information of the status of the current sub-task. For the task of test generation, additional information about the progress of the test generation is given in the RTT-Plugin progress view.

Whenever a RTT-MBT task is started, a console message is given to indicate the start of the action and the progress view is reset. The completion of a task is indicated by the RTT-Plugin progress bar at 100 percent and a message in the console view providing information whether the task as succeeded (`PASS`) or not (`FAIL`).

The Outline is used during the analysis of test procedures. This is explained in detail in Section 12.8.2 and in [Ver13b].

12.3 Terms and Concepts

For the understanding of the rest of this chapter, it is vital that the following concepts are known by the reader. Some of them are just a clarification of how certain terms are used in this document while others are concepts that are used in the rest of this chapter and in the RTT-MBT setting.

Model-based testing The behaviour of the system under test (SUT) is specified by a model elaborated in the same style as a model serving for development purposes. Optionally, the SUT model can be paired with an environment model restricting the possible interactions of the environment with the SUT [Pel13].

Test Model Specifies the expected behaviour of a system under test. This is an important step in model-based testing. Note that a test model can be different from a design model. It might only describe a part of a system under test that is to be tested and it can describe the system at a different level of abstraction.

underlying API that is used for these file operations uses java I/O functions, so that the Eclipse resources are not updated automatically. You can use `Refresh using native hooks or polling` setting from the `Workspace` section of the `General` preferences of Eclipse to perform these updates automatically.

Test Case A test case is a set of input values, execution preconditions, expected results and execution postconditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement. For RTT-MBT (model-based testing with RT-Tester), a test model is used that describes the behaviour of the system that should be tested. Test cases can automatically be derived from this model in form of LTL formulas. These test cases define the precondition and input values, but not the expected outputs, because these are already defined in the model describing the expected behaviour of the system under test. The expected outputs are calculated by test oracles that are executed together with a test procedures covering the test cases.

Test Procedure Detailed instructions for the set-up and execution of a given set of test cases, and instructions for the evaluation of results of executing the test cases (RTCA DOI78B). In RTT-MBT, test procedures can automatically be generated from a test model for a given set of goals (test cases) specified as LTL formulas. These test procedures are separated into a stimulation component that performs a timed sequence of SUT inputs and number of test oracles that observe the stimulations and check for the expected output of the different system components.

Test Oracle A source to determine expected results to compare with the actual result of the system under test. With RTT-MBT, oracles are generated as parts of test procedures. For each component of the SUT in a test model, a test oracle is generated checking for the expected behaviour of the respective component.

Test Generation In this document, test generation describes the process of calculating concrete system under test inputs and expected outputs for a given number of test cases (goals in the form of LTL formulas). An RT-Tester test procedure is created that consists of RTTL (RT-Tester test language) specifications for a stimulator and a number of test oracles. A generic framework for embedding a system under test is generated together with the test procedure.

Test Execution Test execution describes the process of executing a test procedure together with the system under test. Note that a generated RT-Tester test procedure has to be compiled before it can be executed. The result of a test execution is available as soon as the execution terminates, but the test case and requirements tracing information requires to replay the test result against the test model and to create the test procedure documentation. These two steps can be performed automatically using RTT-MBT and RT-Tester.

Generation Context and Execution Context RT-Tester model-based test projects use two contexts which are represented by two project sub-directories.

Generation Context. The **generation context** is the place where the generation of all model-based test procedures of a project is prepared: for every test procedure to be generated there is a separate **test procedure generation context** created in the subdirectory `TestExecution`[22] named as the procedure to be created.

[22]The folder names for the generation context and the execution context can be changed using the Project page of the RT-Tester section in the Eclipse preferences (see Figure 88). The names used here are the default

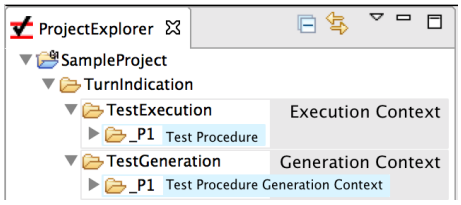

Figure 90: Test procedure generation and execution context.

Test Procedure Generation Context. The **test procedure generation context** is the place in the project where the generation of a single test procedure is prepared. Here the test engineers configure the generation by

- specifying the model portions to be evaluated during the generation,
- specifying the test cases to be covered in the test procedure to be generated.

Execution Context. The **execution context** is the place where the actual **test procedures** which can be compiled and executed against the SUT reside in. This context is contained in directory `TestExecution`. When RTT-MBT creates a new test procedure based on the information provided in the respective test procedure generation context, the resulting test procedure files are placed in a sub-directory of `TestExecution` carrying the same name as the respective generation context . There it can be compiled, executed, evaluated and documented. The execution context can contain both automatically generated and manually created test procedures. The manual development of test procedures is described in [Ver13a].

Figure 90 shows the test procedure generation context `_P1` in the generation context and the respective test procedure `_P1` in the execution context.

12.4 Creating a Project

RTT-MBT projects can be created as a standalone Eclipse project or as a sub-directory inside any existing Eclipse project. Projects are created like any other Eclipse project using a wizard that is activated through the menu entry New → Project from the context menu or from the File menu.

Although creating a project creates local files, a RTT-MBT server connection is required to retrieve project templates and initiate the server workspace. How the server preferences are defined is described in Section 12.1.

On the first page of the wizard, the type of project that is to be created has to be selected. RTT-MBT projects can be found in the category RT-Tester, as displayed in figure 91 On the next page of the wizard, the location for the new project has to be specified. Projects can be created anywhere in your file system[23].

settings.

[23]Note: if you are planning to use RT-Tester as the backend to compile and execute your test, you should not use spaces in directory or file names.

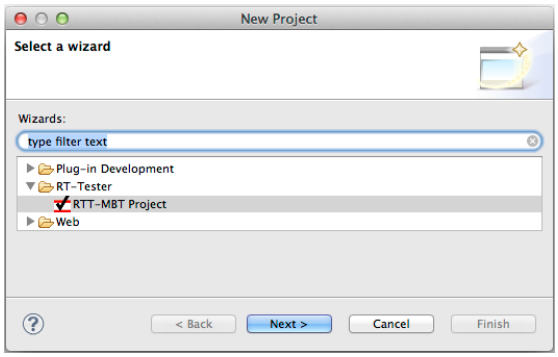

Figure 91: Creating a new RTT-MBT Project - chosing the project type

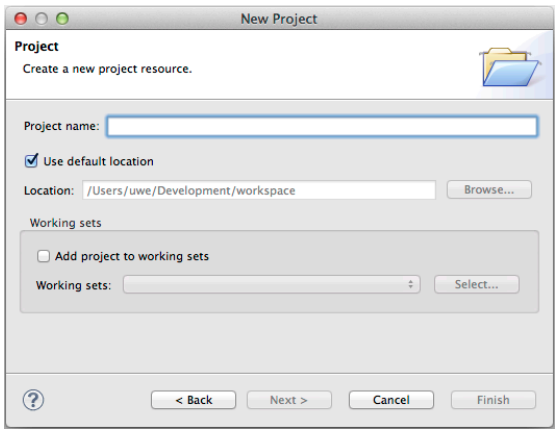

Figure 92: Creating a new RTT-MBT Project - selecting the project location

Creating RTT-MBT projects as sub components of another project can be useful if the RTT-MBT test projects are part of a larger project representing a complete test campaign where several models are used to generate test procedures for different aspects or parts of a complete system. Every RTT-MBT project uses exactly one test model to generate test procedures from. Organising RTT-MBT projects as folders inside another Symphony project provides a flexible way to integrate RTT-MBT projects in your project structure. If one test model is sufficient for your goals, it is recommended to create a stand-alone project.

Creating a RTT-MBT sub-component is also supported by a wizard in the RTT perspective. Select the Symphony project in the project explorer and start the wizard using New → Other from the context menu or the File menu. On the first page of the wizard, select RTT-MBT Folder from the RT-Tester category (Figure 93) and click Next. In the follwing dialog page use the RTT-MBT project name as the folder name for the new RTT-MBT sub-component. The wizards creates an empty RTT-MBT project structure representing the RTT-MBT project.

As stated before, each RTT-MBT project uses exactly one test model. An XMI file containing your test model has to be imported into the project (see [Ver13b] for an explanation how test models are exported to XMI). Typically, this XMI file resides

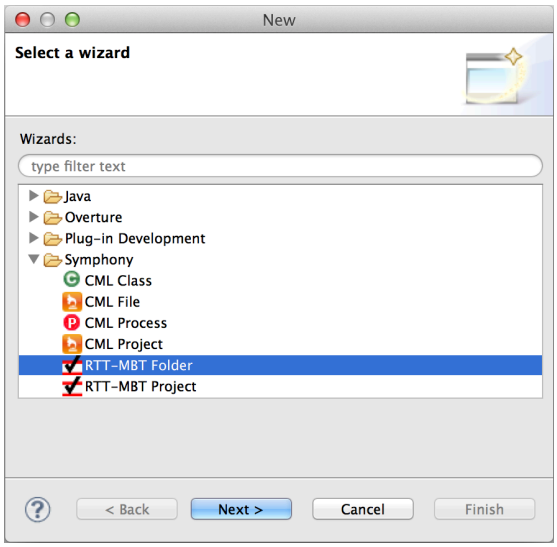

Figure 93: Creating a new RTT-MBT Project - creating a Sub-Directory Project

on your PC where also the test model elaboration took place. Selecting RTT-MBT → Import Model opens a file browser that can be used to navigate to the directory where your XMI file (with extension XML or XMI) is located. Select the file and start the import. As a result of the import, the XMI file is stored as `model_dump.xml` in the `model` directory of the RTT-MBT project, together with a `LivelockReport.log` file. Please check the livelock report for problems in the model.

After a successful model import a first test generation context `_P1` has been created in the generation context (directory `TestGeneration`) which will be used to generate the first test procedure along with an initial project configuration and signal map as explained in Section 12.5. The initial generation context `_P1` will be used to create further contexts to be used for the generation of additional test procedures.

If the test model is changed, it can be imported again using the RTT-MBT pull down menu and selecting Import Model.

Note that re-importing a test model can lead to necessary adjustments in all test procedure generation context definitions and generated test procedures depending on the differences between the old model and the new one.

12.5 Automated Generation of the First Test Procedure

The automated generation of the very first test procedure is performed in the test procedure generation context

```
TestGeneration/_P1
```

which has been created as a result of the model import during project setup, as described in Section 12.4. The configuration of the generation context comprises two steps, before the automated generation can be activated.

- Configuration of the test procedure generation context.
- Configuration of the signal map.

12.5.1 Configure the first test procedure generation context

The **test configuration file** contains information about model components to be used during the generation process and basic test cases to be covered in the test procedure to be created. Before generating the first procedure `_P1` it has to be configured by means of the configuration editor in the RTT-MBT perspective.

- Double-click on

  `TestGeneration/_P1/conf/configuration.csv`

  to open the configuration editor.
- Mark the entry in column DEACT ("Deactivate model component during the generation process") for every model component that should not be considered during the generation.
- Mark at least one state machine transition of the SUT in column TC ("Transition Coverage"). This is a directive to the generator that the model coverage test case *"visit this transition during test execution"* will be covered by the test procedure to be created.

The other columns of the configuration file are explained in more detail in Section 12.7.1. For the initial generation there is nothing more to do.

12.5.2 Check and Edit the Signal Map

The **signal map** specifies the input/output direction of each interface variable, as well as the data ranges admissible for these signals. During project initialisation an initial version of the signal map is created, based on the type information of the interface signals specified in the model. The resulting signal map is placed into the test procedure generation context as file

`TestGeneration/_P1/conf/signalmap.csv`

Since the generator can only guess the appropriate signal ranges and since it may be useful to change the ranges for specific test purposes it is advisable to open this file by double-clicking it in the project browser. Then adapt the pre-defined data ranges where appropriate.

In the turn indication sample project described in [Ver13b], for example, the floating point input voltage has typical value 12V in today's cars, and the model defines $10 \leq \text{voltage} < 15$ as the admissible range. As a consequence, the lower bound 0.0V and upper bound 16.0V are suitable values to be inserted for voltage in the signal map.

Observe that the test data generator will only create inputs to the SUT which are consistent with the data ranges defined in `signalmap.csv`. This fact may be used to influence the generation process in the following ways: if an input to the SUT is

specified with identical lower and upper bounds in the signal map, the generator will leave this value constant over the complete test execution time and try to reach the test objectives by manipulating the other inputs only.

A more detailed explanation of all columns in the signal map is given in Section 12.7.2.

12.5.3 Generate the First Test Procedure

To generate a test procedure from the initial generation context `_P1`, select the context `_P1` in the project browser. Then select RTT-MBT command Generate Test Procedure in the context menu of the project browser (right-click on `_P1`) or from the RTT-Plugin command bar (see Figure 94).

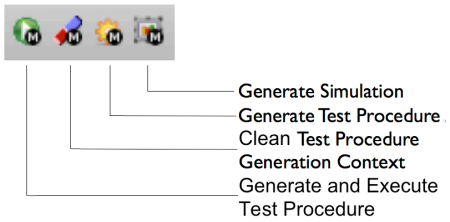

Figure 94: Model-based test commands in the RTT-Plugin command bar.

As a result of this first generation the execution context `TestExecution` is created, and the first test procedure is stored there in directory `_P1`. It is explained in Section 12.9 how the generated procedures can be compiled and executed against a SUT or a simulation. As a side effect of this initial generation, the model-related test cases have been identified.

12.6 Creating Additional Test Procedures

Additional test procedures can be generated by copying the initial (or any other suitable) test procedure generation context _P1 in the generation context. A test procedure generation context can be copied and pasted into the generation context like any other folder in the project.

Rename the new test procedure generation context and

- inspect the signal map `conf/signalmap.csv`, whether the input signals need adaptations for the new test procedure to be generated (see Section 12.7.2),
- adapt the configuration `conf/configuration.csv` as far as needed (see Section 12.7, and/or
- allocate new test cases to be covered by the new procedure that will be generated form this context.
- remove previously generated files in the subdirectories `log`, `model` and `testdata`.

12.7 Test Generation Configuration

Model-based test procedure generation with RTT-MBT can be influenced in many ways, allowing a user to specify different test goals to be reached and different test procedure and solver behaviour. This adds complexity to the configuration of the test procedure generation context, but provides flexibility in test integration and purpose of the generated test procedure. This section describes the different configuration files that are taken into account during the test generation and their purpose.

12.7.1 Detailed Configuration of Test Procedure Generation Contexts

The detailed configuration editor for transition coverage goals is opened by double clicking the

`TestProcedures/<Test Proc. Gen. Context>/conf/configuration.csv`

file in the project explorer[24]. This displays the detailed configuration editor as shown in Figure 95 for the turn indication sample project. The columns of this pane have the following meaning.

Component allows you to browse through the test model components in top-down fashion: when opening the pane for the first time, the complete component structure is unfolded below the top-level components for SUT and TE.

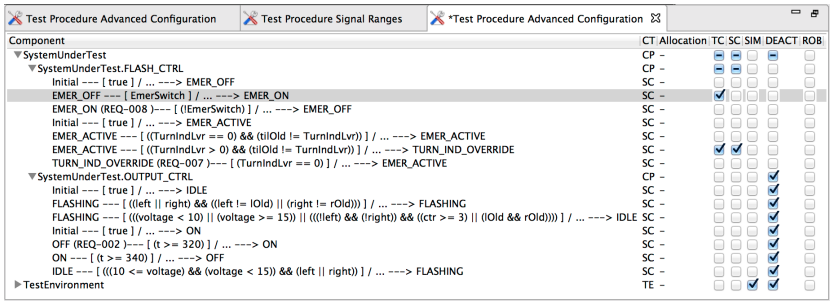

Figure 95: Detailed configuration example from the turn indication project.

CT displays the component type, that is

- CP for a composite structure, block or class,
- TE for the top-level component of the test environment,
- SC for a state machine transition.

In Figure 95, for example, the transitions of the state machines associated with components FLASH_CTRL and OUTPUT_CTRL are displayed for the sample project.

[24] During creation of the initial test procedure the detailed configuration has already been used – see Section 12.5.1.

Allocation is only relevant for hardware in the loop testing: it allows to specify the allocation of TE simulations and SUT test oracles to a specific CPU core.

TC is used for quickly defining transition coverage (TC) goals: marking a state machine transition in the model browser in column TC specifies the goal *"cover this state machine transition in the test procedure to be created"*. In Figure 95, for example, the mark in column TC for transition

```
EMER_OFF[6] --- [EmerSwitch]/...  ---> EMER_ON
```

of state machine FLASH_CTRL will lead to the generation of a test procedure where the emergency switch EmerSwitch is set to 1 at some time during the test execution, so that the SUT – if implemented correctly –performs the equivalent behaviour to the state machine transition from EMER_OFF to EMER_ON.

Observe that every transition coverage goal is also identified as a test case automatically derived from the model. Transition coverage test cases have identifiers

```
TC-<Project Name>-TR-<number>
```

and by selecting the test case equivalent to the state machine transition marked in the TC column, and adding it to the additional goals file (See [Ver13b] for a detailed description how this is performed), the same effect is achieved. The goal identification via column TC is just a work flow abbreviation: in many situations – for example in case of a regression test where just a few transitions have been altered in the test model – testers just need a test procedure verifying these transitions. Moreover, state machine transition coverage is regarded as the typical test coverage criterion to be applied whenever the SUT has to be tested in a reasonably thorough way, but is not safety critical or business critical, so that the more sophisticated coverage criteria described in [Ver13b] should be applied.

SC is used for quickly defining modified condition/decision (MC/DC) coverage goals. SC stands for "safety-critical", because MC/DC coverage is the most common coverage criterion to be applied in the context of safety-critical systems testing (see the avionic standard [WG-11], and [Ver13b] for explanations about this coverage criterion). As shown in Figure 95, transitions have to be marked in both the TC and the SC columns, if MC/DC coverage should be realised by the test procedure to be created.

As for transition coverage, marking MC/DC coverage goals in columns TC and SC is just a short cut for selecting the MC/DC coverage test cases associated with this transition. Note, however, that in contrast to transition coverage, *several* test cases are associated with the goal to cover one transition according to the MC/DC criterion: the transition has to be exercised with differing valuations of the atomic conditions contributing to the guard condition, and several stability conditions, where the state machine remains stable in its current control state, have to be explored. This is illustrated, for example, by the MC/DC test cases related to control state IDLE in state machine OUTPUT_CTRL of the sample project.

Finally observe, that some MC/DC conditions and their associated test cases may be infeasible. The stability condition:

```
(IMR.TurnIndLvr@0 <= 0 &&
  IMR.SystemUnderTest.FLASH_CTRL.tilOld@0 != IMR.TurnIndLvr@0 &&
  IMR.SystemUnderTest.FLASH_CTRL.FLASH_CTRL.EMER_ON.EMER_ACTIVE@0 &&
  IMR.EmerSwitch@0) && (IMR.TurnIndLvr@0 != 0 ||
  IMR.SystemUnderTest.FLASH_CTRL.tilOld@0 == IMR.TurnIndLvr@0)
```

for example, cannot be fulfilled, due to the contradictory atomic conditions

```
IMR.TurnIndLvr@0 <= 0 && IMR.TurnIndLvr@0 != 0
```

and

```
IMR.SystemUnderTest.FLASH_CTRL.tilOld@0 != IMR.TurnIndLvr@0 &&
IMR.SystemUnderTest.FLASH_CTRL.tilOld@0 == IMR.TurnIndLvr@0)
```

because IMR.TurnIndLvr always carries non-negative values. Note, however, that RTT-MBT can identify most infeasible test cases. This is performed when creating the test case database from the test model, and the infeasible test cases are recorded in directory

```
model/unreachable_testcases.csv
```

(where you will find the infeasible MC/DC test case discussed above), so that these do not have to be covered in test campaigns.

SIM marks test model components as simulations. For components residing in the test environment this is mandatory, so un-marking this column for any TE-entry will have unpredictable effects during test procedure generation. In contrast to this, SUT components are normally unmarked in the SIM column, unless

- a simulation of the SUT should be generated: in this case, *all* SUT components have to be marked in the SIM column before the simulation generation command is given, or
- a sub-component of the SUT has not yet been implemented, and should therefore be simulated by the testing environment. In this situation, *only the unavailable sub-components* are marked in the SIM column, and during the test procedure generation process simulation components are created for just these SUT parts.

DEACT allows you to deactivate test model components during the test procedure generation process. This option is practical for incomplete models, where some parts are still under construction. De-activating incomplete components will avoid error messages referring to missing model elements or erroneous syntax. Moreover, sometimes several variants of a model component have been created, and for every test procedure generation a specific variant should be applied. This applies particularly for the testing environment, where several alternative simulations operating on the same interface variables may be developed, and each test procedure requires its specific simulation variant (or no simulation at all).

Observe that it is unnecessary to deactivate test model components in order to speed up the test generation process. Suppose, for example, that the functionality of FLASH_CTRL and OUTPUT_CTRL has been implemented in two different control components of the SUT. If a complete regression test achieving transition coverage should be performed for FLASH_CTRL, and the coverage thereby obtained for OUTPUT_CTRL is without relevance, it suffices to mark all FLASH_CTRL transitions in the TC column, while

leaving the same column unmarked for OUTPUT_CTRL. The RTT-MBT tool analyses the coverage goals configured for the test procedure generation process and performs a so-called **cone of influence reduction** on the model: all model components that do not contribute to the coverage goals defined are automatically de-activated during the generation process, so that the constraint solver operates on a simpler transition relation, in this example the local relation specifying the behaviour of FLASH_CTRL alone.

ROB marks transitions of the test environment as **robustness transitions**. This means that they will only be executed if

- the associated TE-component has been activated for the test procedure generation,
- parameter RB (robustness test generation ON/OFF) has been set to 1 in the advanced configuration (see Section 12.7.3), and
- parameter RP (percentage of robustness transitions) has been set to an integer value in range 1 — 100 in the advanced configuration (see Section 12.7.3).

The typical application of this feature is for the generation of robustness tests, where certain inputs to the SUT are to be produced only in exceptional behaviour situations. These inputs are written in test environment (TE) simulations, and the associated state machine transitions have been marked in the ROB column of the detailed configuration. Then these transitions will only be taken in test procedures generated with RB $= 1$, RP $\in \{1, \ldots, 100\}$ in their advanced configuration.

Test environment state machines can have transitions marked by the stereotype <<Robustness>>. These transitions are *always* processed as robustness transitions, regardless of their status in column ROB in the detailed configuration. With column ROB it is possible to mark *additional* transitions of TE state machines as robustness transitions.

Finally, observe that marking SUT state machine transitions in column ROB has no effect.

12.7.2 Detailed Configuration of the Signal Map

The signalmap editor is opened by double clicking the file

`TestProcedures/<Test Proc. Gen. Context>/conf/signalmap.csv`

The signal map has already been introduced in Section 12.5.2 in the context of test project creation. We will now describe the detailed configuration options provided by the signal map. To this end, consider an example of a signal map from the turn indication sample project, as displayed in Figure 96[25].

[25]The editor for signalmap.csv files contains more columns than are displayed in this figure. Here only the columns that are relevant for this manual are displayed.

Abstract Signal	Lower Bound	Upper Bound	SUT writes to TE	TE writes to SUT	Concrete Signal	Admissible Error	Latency
EmerSwitch	0	1	0	1	EmerSwitch	0	100
LampsLeft	0	1	1	0	LampsLeft	0	100
LampsRight	0	1	1	0	LampsRight	0	100
TurnIndLvr	0	2	0	1	TurnIndLvr	0	100
voltage	0.0	16.0	0	1	voltage	0	100

Figure 96: Signal map example from the turn indication sample project.

Abstract Signal specifies each observable variable of the SUT with its name as declared in the test model. Model variables not occurring in the signal map are automatically unobservable, that is, internal variables of the model without any observable counter part at the SUT interface.

Lower Bound **and** Upper Bound specify the lower and upper bounds, respectively of input variables to the SUT, as they should be observed in the test procedure generation configured in the current context. If no TE simulations have been defined for an input interface variable, only values within the specified range will be generated. If, however, specific input values are generated by a TE simulation, this overrides the lower and upper bound specification in the signal map. For SUT outputs, the specification of lower and upper bounds only has informative value; it does not affect the test procedure generation process.

SUT writes to TE specifies with entry values 1, that the variable is an output of the SUT which may be observed in the testing environment. If this variable cannot be observed by the TE, its SUT writes to TE value has to be set to 0.

TE writes to SUT specifies with entry values 1, that the variable is an input to the SUT which may be written to by the testing environment. If the TE cannot write to a variable, its TE writes to SUT value has to be set to 0.

In some hardware-in-the-loop testing environments, the TE may write outputs in place of the SUT, for example, to override erroneous outputs of SUT components that might affect other parts (e.g., a different controller in an SUT network) of the SUT. In these situations both the SUT writes to TE *and* the TE writes to SUT entries are marked by 1, and further definitions must be provided in columns Concrete Signal Identifier explained next.

Concrete Signal Identifier is only relevant, if the testing environment uses different names for SUT inputs and outputs from the ones occurring in the model. This is frequently necessary in hardware-in-the-loop testing environments, where abstract model signals have to be mapped to concrete hardware interfaces with names depending on the HW drivers used. In these situations, the concrete name of an interface variable, as it should be read by the TE during test execution, is inserted in column Concrete Signal Identifier (only filled in if TE may READ this signal). Insert the variable name to be used by the TE when writing to an interface variable of the SUT into column Concrete Signal Identifier (only filled in if TE may WRITE this signal).

Admissible Error may contain 0 or a positive integer or floating point value ε. It is used to introduce tolerances into test oracles, to be applied when checking SUT outputs against expected results. If for some SUT output variable x an admissible error ε has been specified in the signal map, and an output value $\overline{x}$ is expected at a certain stage of the test execution, outputs x in range

$$\overline{x} - \varepsilon \leq x \leq \overline{x} + \varepsilon$$

are still accepted by the oracles and lead to PASS verdicts. Values outside this range lead to a FAIL.

Latency complements the Admissible Error in the time domain. A latency value $\delta > 0$ (time unit milli seconds) affects the test oracles in such a way that they still accept an output expected at time t_0 according to the model, if it is produced within the range of the admissible error at some point in time interval $[t_0, t_0 + \delta]$.

12.7.3 Test Generation Properties

The test generation properties can be adjusted by editing the configuration files `max_steps.txt` and `advanced.conf`. Both files contain properties that control the RTT-MBT test generation. The files are text files with semicolon separated values. These values are described in this chapter so that they can be adjusted using an editor or any other CSV file editing tool. In addition to this possibility, RTT-Plugin provides the possibility to adjust these settings in the RT-Tester section of the project properties. Because these settings are specific for each test procedure generation context, they are only part of the RT-Tester properties page if a test procedure generation context directory is selected. Figure 97 shows the properties page for a test procedure generation context.

Generation Steps Configuration File The configuration file

`TestProcedures/<Generation Context>/conf/max_steps.txt`

contains the settings for

MAX SOLVER STEPS The maximal number of model execution steps from the current model state that the constraint solver will perform to look for a solution of the test objective to be fulfilled. A value between 20 and 100 is suitable for most projects. Default value is 100.

MAX SIMULATION STEPS The maximal number of simulation steps to be performed by the generator without covering any new model transitions: if this number is greater than 0, the generator will try to find test data for a test objective also by means of random walks through the model, if the constraint solver could not solve the goal within the given number of steps from the current state. A random walk is continued as long as new portions of the model are covered by this walk. If a simulation step fails to cover a new model element the initial value of `MAX SIMULATION STEPS` is

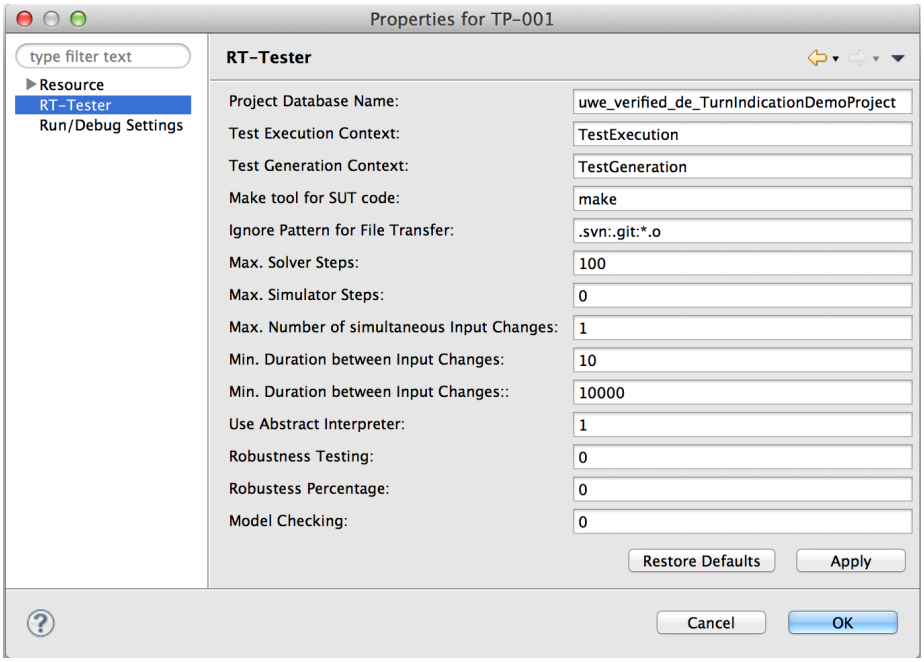

Figure 97: Properties page of a test procedure generation context.

decremented. The following simulation steps continue to decrement this value until a new model element is covered or the value becomes zero. In the former case the value is set back to `MAX SIMULATION STEPS`. In the latter case the simulation is stopped and the constraint solver tries to reach (one of) the remaining test goals from the model state reached by the simulation.

The default value is 0 (no simulation with random data generation).

Default configuration. If the default values are suitable for a new test procedure to be generated, it is not necessary to edit the basic configuration. The initial test procedure generation context `TestProcedures/_P1` is created with such a default configuration. Observe, however, that the basic configuration is also copied when creating a new generation context from an existing one. So changes to the default configuration in the source context also apply to the new target context.

Solver Configuration File The solver configuration file

`TestProcedures/<Generation Context>/conf/advanced.conf`

contains the fields described in Table 3. Each configuration entry in `advanced.conf` consists of a single line; each line is structured into

`<Parameter>;<Value>;<Comment>`

Table 3: Configuration parameters and default values.

	Default	Description
GC	1	If 1, cover all goals in addgoals(ordered).conf, even if they are already covered by other procedures.
BT	0	Switch back tracking on if 1.
LO	0	Produce logger threads instead of checkers if 1.
AI	0	Use abstract interpretation for speed-up of solver, if 1.
MM	0	Maximise model coverage if 1.
SC	0	Perform sanity checks in solver and abstract interpreter if 1.
RB	0	Do robustness testing if 1.
RP	0	If RB=1 RP defines the percentage of robustness transitions to be performed.
CI	1	Maximal number of simultaneous input changes.
DI	10	Minimal duration between two input changes.
LI	10	Upper limit for the duration between two input changes.
MC	0	Perform model checking instead of test generation.

CI – Maximal number of simultaneous input changes. Parameter `CI` ("changed inputs") is a non-negative natural number. It specifies the number of inputs that may be changed simultaneously after a delay. The input vector to the SUT may be changed after time delays, during which the model state remained stable. In hardware-in-the-loop tests it may be desirable to change only a bounded number of inputs at a time, since the SUT reaction may become non-deterministic in presence of too many nearly simultaneous input changes. Therefore parameter `CI` is set to 1 by default, meaning that after a delay at most one input to the SUT is allowed to be changed.

For software testing, it is often allowed and even necessary to change several input variables to the SUT at the same time. If this is the case, `CI` should be set to a bound which is sufficiently high.

DI – Minimal duration between two input changes. In hardware-in-the-loop testing the **interface latency** of the SUT has to be taken into account: if changes to the SUT occur with too high a frequency, the SUT will not be able to process them, because consecutive changes get lost already on input interface boards, or in buffers of the SUT runtime system. Therefore the minimal duration between input changes to the SUT should be respected by the testing environment. To this end, parameter `DI` ("duration between input changes") can be set to a natural number, indicating the minimal duration between two input changes in the time unit milliseconds.

LI – Upper limit for the duration between two input changes. Changes to the SUT should not occur with too high a frequency. Thus it can be useful if the tester can limit the maximum duration between two input changes, to prevent the solver to generate tests that wait unreasonably long between input changes. The parameter `LI` ("Upper limit for the duration between input changes") can be set to a nonnegative natural number, indicating the upper limit for the duration between two input changes in time unit milliseconds. Note that the latency for signals has to be taken into account when defining `LI` or `DI`. Otherwise it would be possible to define configuration in which the checker will never find problems in the SUT outputs.

RB – Robustness Testing ON/OFF. If robustness tests should be performed by the test procedure to be generated, then parameter `RB` has to be set to 1. In this case, the robustness transitions defined in TE state machines are not ignored (as it is the case when `RB` is 0), but are performed with the percentage specified in `RP` (see explanation of `RP` below, and Section 12.7.1). Observe that setting `RB` to 1 only has an effect, if

- `RP` is greater than 0,
- TE components have been modelled, are active, and
- at least one TE state machine associated with an active TE component has robustness transitions.

The default value for `RB` is 0.

RP – Robustness Transition Percentage. If `RB` is set to 1, parameter `RP` is evaluated. It is an natural number in range 0 — 100 and specifies the percentage of robustness transitions to be taken when residing in a TE state machine state from where some emanating transitions have been marked as robustness transitions (see Section 12.7.1), but also ordinary transitions exist. The test generator will fire approximately `RP%` robustness transitions from this state, and `(100 - RP)%` normal behaviour transitions.

For achieving an adequate distribution of normal behaviour and robustness transitions in TE simulations, it is advisable to configure the test procedure generation context as a combination of constraint solving and random simulation. Recall that this is achieved by setting parameters Max. Solver Steps and Max. Simulation Steps to positive values, as described in the basic configuration. The constraint solver alone – no simulation active – always looks for the most direct model trace leading to coverage of a given test goal. Therefore it ignores the percentage of normal behaviour or robustness transitions taken so far, and always chooses the one which is most suitable for the test objective.

The other configuration parameters are currently experimental, and should only be used with their default values, as indicated in Table 3. When the initial test procedure generation context is created in `TestProcedures/_P1` (see Section 12.4), the `advanced.conf` file associates all parameters with their default values, so that it will be unnecessary in most situations to edit this file, unless `CI` and `DI` should be adapted.

12.8 Test Procedure Generation

If a test procedure generation context is configured completely, a test procedure can be generated according to this configuration. Note that exactly one test procedure in the execution context will be generated for each test procedure generation context in the generation context of a project.

12.8.1 Activating the Generation Process

As explained in Section 12.5.3, the generation process is activated by selecting a test procedure generation context in the project explorer and giving the Generate Test command from the RTT-MBT tool bar, from the RTT-MBT menu or in the Test Generation Context context menu of the project explorer (right-click on the selected item).

When a test generation is started, all progress bars are cleared and the following console message is displayed:

```
generating test procedure <name>...
please wait for the task to be finished.
```

During the test generation, the progress for the different coverage goals and for the overall test generation is indicated in the progress view. Successful test procedure generation is indicated through the following console message:

```
[<name>]: [PASS]: generate test procedure
```

If the generation was aborted or did fail, error messages are given in the console view.

12.8.2 Results of Test Procedure Generation – Validation Aspects

Before running a generated test procedure against the SUT it may be desirable to validate it with respect to the original test objectives you had in mind when configuring the test procedure generation context. Therefore RTT-MBT produces additional information during the generation process that will be useful for checking whether your are testing the right thing. This information is described in the following paragraphs.

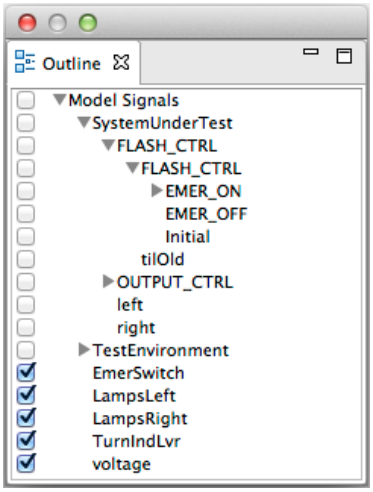

Figure 98: Signal viewer outline showing all signals selectable for display.

Signal flow over time. While generating a test procedure, RTT-MBT records all changes of interface and internal model variables, as well as simple state changes within state machines, as they should arise during test procedure execution against the model. These variable changes over time can be visualised using the **signal viewer**

which is built into the graphical user interface RTT-GUI. After having generated a test procedure, the signal viewer can be opened by double clicking on the generated file signals.json in the model directory of the respective test procedure generation context.

To select the signals (variables and simple states) to be displayed, an **outline view** is applied. In Figure 98, some of the signals selectable for the turn indication sample project are shown: apart from SUT inputs and outputs, also local variable values are selectable, as, for example, the tilOld variable storing the last state of the turn indication lever. Additionally, it is possible to display simple state machine states, such as EMER_OFF of state machine FLASH_CTRL in the sample project. Simple states are displayed like Boolean variables over time, value 1 meaning that the machine resides in this state. Hierarchic composite states – such as EMER_ON in Figure 98 can be unfolded to show their subordinate states.

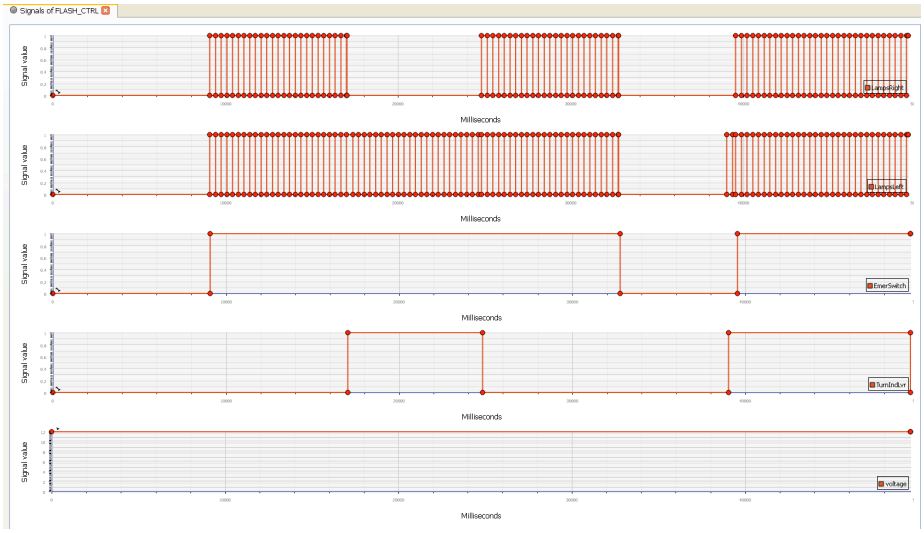

Figure 99: Display of the signal viewer for a test procedure covering all transitions of state machine FLASH_CTRL in the sample project.

Figure 99 shows a typical display of the signal viewer; it is associated with a test procedure generated with the objective to cover all transitions of state machine FLASH_CTRL in the sample project. The signal values are displayed over time[26] for the following signals:

- EmerSwitch (emergency switch on/off = 1/0) – input to the SUT
- LampsLeft (indication lights left-hand side on/off = 1/0) – output of the SUT
- LampsRight (indication lights right-hand side on/off = 1/0) – output of the SUT
- TurnIndLvr (turn indication lever off/left/right = 0/1/2) – input to the SUT
- voltage – input to the SUT

[26]This presentation style is called **y/t diagram**.

The test procedure that is visualized in Figure 99 stimulates different turn indication and emergency flashing conditions. The stimulations as well as the expected SUT indications are visible.

12.9 Test Procedure Execution

The generated RT-Tester test procedures reside in the test execution context[27] of the RTT-MBT project. Recall that a generated test procedure has the same name as the test procedure generation context used to generate this test procedure.

These RT-Tester tasks can be selected through the toolbar, the RTT-MBT menu or the Test Execution Context section of the context menu for a selected test procedure. Note that a test procedure has to be selected in the Project Explorer of the RttPerspective for the RT-Tester actions to be enabled.

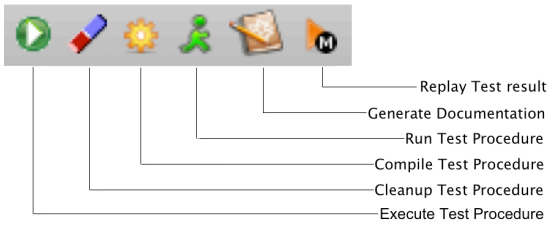

Figure 100: The RT-Tester commands in the toolbar.

12.9.1 Execute Test Procedure

A complete test execution consists of the steps [28]

- Clean Test Procedure
- Compile Test Procedure
- Run Test Procedure
- Replay Test Result
- Generate Documentation

12.9.2 Clean Test Procedure

A test procedure can contain outdated test results from previous test runs or the test procedure executable itself is outdated. In both cases, the Clean Test Procedure command can be used to remove all test data and the test procedure executable.

[27] by default the folder `TestExecution` within the project

[28] The menu command Execute Test from the RTT-MBT menu or the toolbar can be used to perform the test procedure actions Clean Test, Compile Test, Run Test, Replay Test Result and Generate Documentation in the correct order without further user interaction, as long as no errors occur.

12.9.3 Compile Test Procedure

A test procedure is always executed together with the system under test. The stimulations of the test procedure must be connected to the respective input interfaces of the SUT and the output interfaces of the SUT that are relevant for the test must be connected to the test procedure. This is a task for the test environment. A generic test environment is created together with the test procedures during the test generation. Connecting an SUT to this generic interface is part of the duties of a test engineer. RTT-MBT provides functionality to create a simulation for the SUT or components of the SUT, in case that test procedures should be designed, generated and evaluated before the SUT implementation is complete and an actual SUT exists that can take part in the test. Simulations generated with RTT-MBT already contain a connection to the test environment, so that a generated test procedure can be compiled together with a simulated SUT without any further implementation.

The integration of a SUT into a test environment can be a simple task but can also be very complex, depending on the SUT, the test integration level. It can vary from linking the SUT object code for unit level tests to providing a function stub interface or shared memory communication for software integration tests to implementing hardware interfaces and communication protocols for hardware-in-the-loop tests.

Independent of the test configuration, every RT-Tester test procedure has to be compiled into a test procedure executable. This task can be started by selecting the test procedure in the Project Explorer in the test procedure execution context and using the Compile Test command in the context menu or in the RTT-MBT toolbar.

12.9.4 Run Test Procedure

After successful compilation, a test procedure is executed by giving the Run Test command in the context menu or in the RT-Tester toolbar. The results of the test execution are stored in log files in the `testdata/` sub-directory of the test procedure.

12.9.5 Replay Test Result

After a generated test procedure has been executed and documentation has been generated, the test execution log has to be **replayed** against the test model. During replay, the RTT-MBT interpreter processes the test execution log resulting from the test execution against the SUT and checks whether the SUT reactions observed conform to the reactions expected according to the model. Moreover, the test cases covered during the test execution are identified, and the PASS/FAIL results are validated during the replay process.

The reason for performing replays is twofold.

- Some test cases refer to internal model states that cannot be observed during test execution against the SUT, but can be identified by the model interpreter when running the test execution log against the model.
- Replay is a redundant procedure operating “orthogonally” to the automated test procedure generation. An error that may have been produced by the generator will be uncovered during replay with high probability. Therefore replay is

mandatory to be performed when RTT-MBT is applied for the test of safety-relevant software: replay is an essential pre-requisite to RTT-MBT **tool qualification**; this is described in more detail in [BPS12]. For application of test automation tools in a safety-critical context, only qualified tools may be used. More details about tool qualification are presented in [Ver13b].

To replay a test procedure executed before, select the test procedure in the execution context using the Project Explorer and issue the Replay command in the tool bar, the RTT-MBT menu or the Test Execution context menu. As a result of the replay command execution, a log file is stored in the log directory of the test procedure generation context of the selected test procedure:

`TestGeneration/<TestProcName>/log/covered_testcases.csv`

listing all test cases covered during the test execution together with the PASS/FAIL information obtained for this test case during replay. These test case verdicts are added to the test documentation in the next step (Generate Documentation). If replay is not performed, test cases that are assigned to internal test model states might not get a verdict from the test execution and will occur as inconclusive in the test documentation.

12.9.6 Generate Documentation

After a test run, automatic generation of the test result should be triggered. To this end, select the procedure in the test execution context and issue the Document Test command in the context menu, in the toolbar or in the RTT-MBT menu.

This has the following effects:

- A test procedure description document in PDF format is created in the `testdata/` directory of the test procedure.
- A test results description document in PDF format is created in the same directory.

13 SysML to CML Translation

This section describes the use of the SysML to CML translation plug-in (S2C-lite) that is available in the Symphony IDE. This plug-in takes an XML representation of the SysML model in the form of an XMI file exported from Atego's Artisan Studio and produces a number of CML files, one for each block and state machine. It is worth mentioning that this tool is different from the implementation in Artisan Studio of the semantics in D22.4; it differs both in scope (only covers blocks as types and individual state machines), and in the semantics, which takes a synchronous view of events. The distinctive feature of the semantics implemented in this plug-in have been described in COMPASS Deliverable D31.4d [ML14].

Finally, since XMI files produced by different tools are not, in general, compatible, this tool assumes the file has been generated using Artisan Studio version 7.4.

13.1 Exporting SysML models

In order to export a SysML models into an XMI file, we must first open the model on Artisan Studio. Figure 101 shows the Hybrid SUV (HSUV) example model in Artisan Studio.

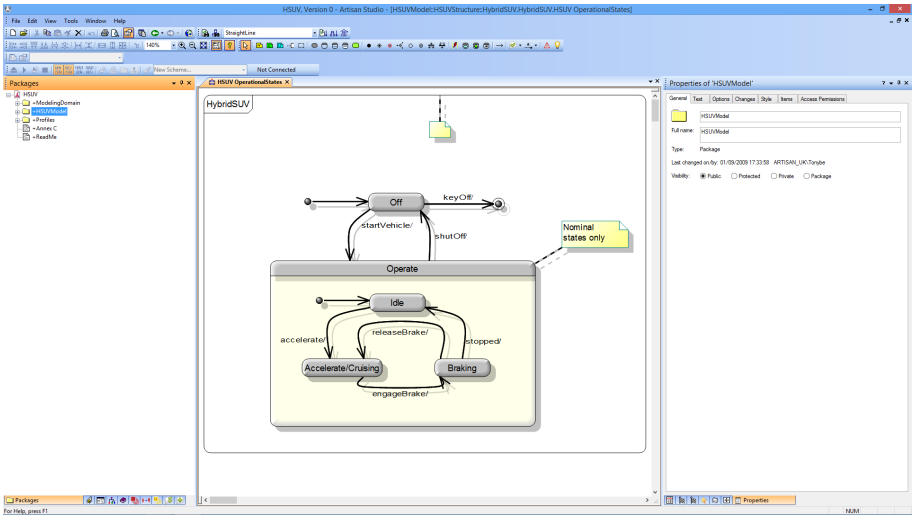

Figure 101: HSUV model in Artisan Studio.

With the model open, we must select the menu item *Tools* → *XMI* → *Export* as shown in Figure 102. If the *XMI* menu is not available, then the tool may not have been installed. In this case, re-run the installation of Artisan Studio and select the XMI feature to be installed.

Next, the destination XMI file must be selected in the XMI Export dialog box and the button *Export* pressed as shown in Figures 103 and 104.

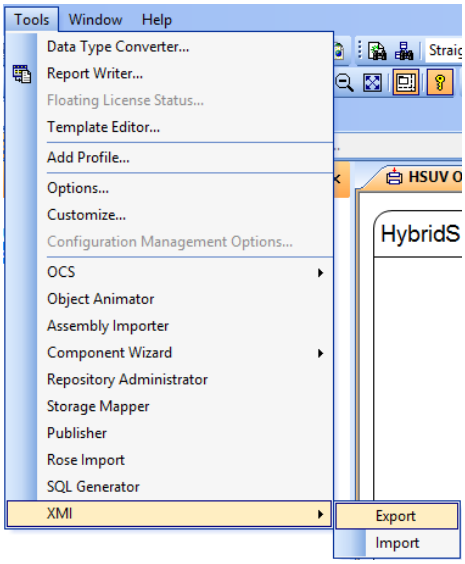

Figure 102: XMI Exporter menu.

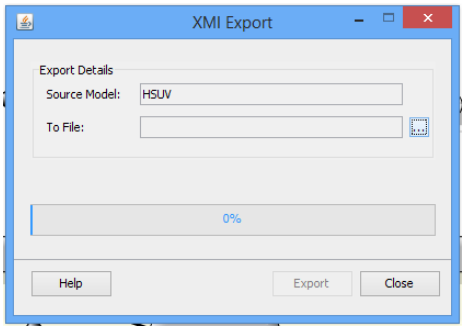

Figure 103: XMI Exporter dialog box.

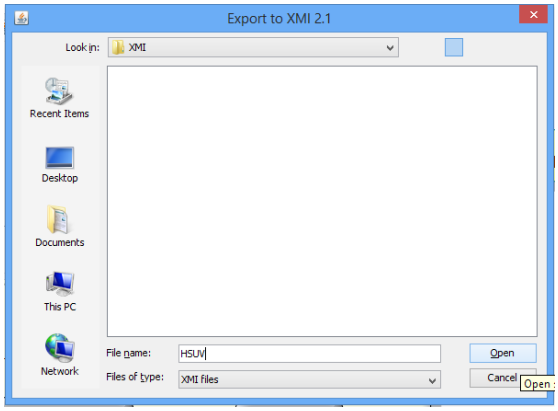

Figure 104: File selection.

13.2 Importing the XMI file

Once the model has been successfully exported, it is necessary to import the XMI file into Symphony. For that, create a new CML project in Symphony, right click on the project and select *Import* (Figure 105), choose *File System* as the import source (Figure 106), navigate to the XMI file (Figure 107), and click *Finish*.

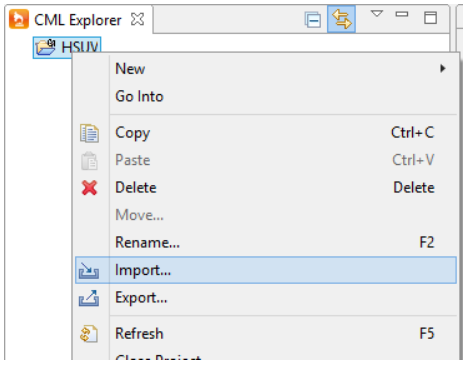

Figure 105: Importing a new file into the CML project.

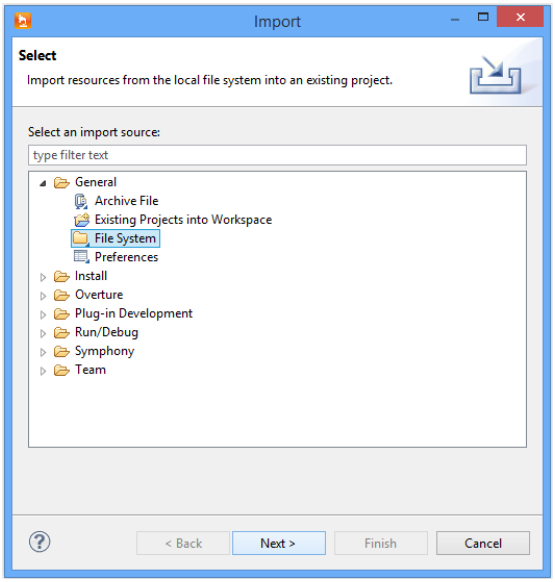

Figure 106: Selecting *File System* as import source.

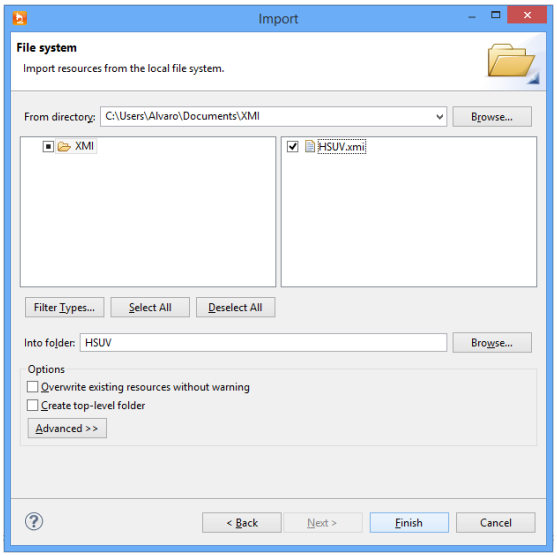

Figure 107: Selecting the XMI file.

13.3 Generating CML and Inspecting the Result

Finally, to generate the CML model right-click the XMI file and select the menu item *SysML → Import SysML state machine* as shown in Figure 108.

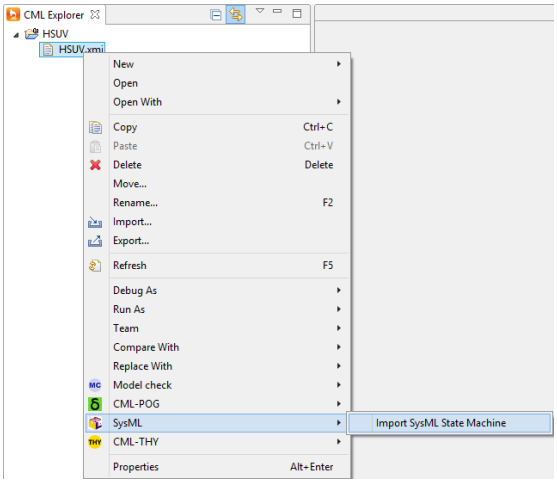

Figure 108: Menu selection for model generation.

A number of CML files will be created in the CML project as shown in Figure 109. In order to inspect the model of the state machine, select the appropriate file (in this case `State-Machine_HybridSUV`, as shown in Figure 109).

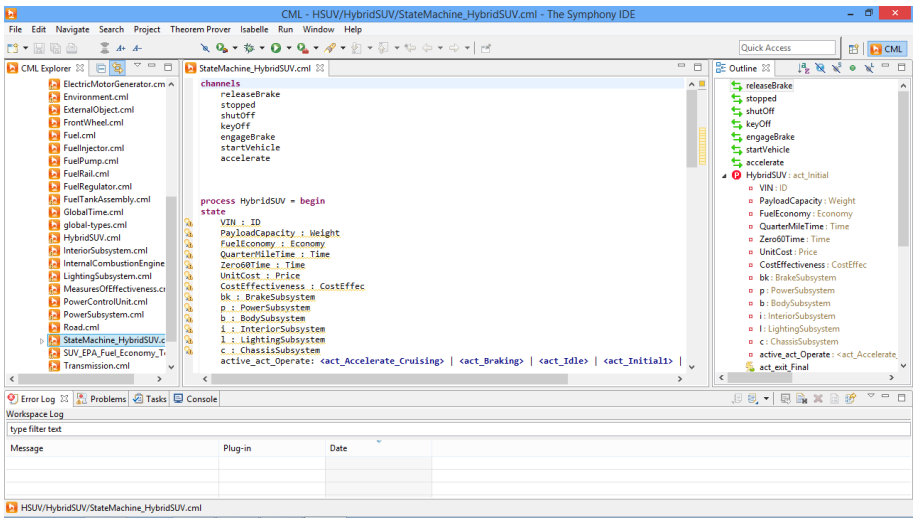

Figure 109: State machine model.

14 The Command-line Tool

The command-line interface to the COMPASS tool was conceived as a tool for developers to quickly allow them to access and test the core libraries. This allows developers of the tool to quickly test new functionality for correctness without having to create the GUI elements that will control the functionality in the integrated IDE. A beneficial side-effect of having this tool is that general users are not required to load the IDE to test CML programs, but instead may invoke them via the command-line.

Presently `cmlc` is available alongside the main Symphony IDE bundle (see Section 2), but as a separate file. The command-line tool in provided as a ZIP file, `cmlc-0.4.0.zip`, and the contents are pure Java, and inherently cross-platform as a result. All that is required to run the tool is a Java runtime environment, version 1.7 (Java 7) or later.

After downloading `cmlc-0.4.0.zip`, decompress it into a folder. In that folder will be the files `cmlc`, `cmlc.bat`, and `cmlc-0.4.0.jar`. Invoke the command-line tool using either `cmlc` (Linux, Mac OS X) or `cmlc.bat` (Windows).

14.1 Available Functionality

The command-line tool provides access to the following features in the core libraries:

- CML model parsing and typechecking
- Proof Obligation Generation (POG) for CML models (see Section 14.3)
- CML model simulation (see Section 14.4)
- SysML to CML translation (see Section 14.5)
- Generation of graph representations of parsed CML models

The CML parser is the primary element of the command-line tool, as nothing can happen without using it. Generally, the tool will read in a (sequence of) CML file(s) and then perform a typecheck on the abstract syntax tree (AST). At this point, the data is ready to be used by the rest of the core libraries and plugins. It is possible to run the core libraries on an AST that has not been typechecked, but doing so is not recommended except to test error reporting or if the user only wishes to generate a graph representation of the AST.

The graph representation generator will output a SVG file containing a representation of the AST generated from the input CML files. This depends on the Graphviz suite of graph visualization utilities.[29] The output is useful for producing a visual representation of the data used internally by the COMPASS tool to represent the static structure of a model of a system of systems. This allows a developer to quickly verify whether the input CML files result in the expected internal data structures. General user use is not recommended.

[29] Found at `http://www.graphviz.org`.

14.2 Basic Invocation

After obtaining the command-line tool package, decompress it into a folder. In that folder will be –among others– the files `cmlc` and `cmlc.bat`. Invocation of the `cmlc` (Linux, Mac OS X) or `cmlc.bat` (Windows) script with no parameters will produce the following output:

```
Symphony command line CML Checker

Usage: cmlc [switches] <file1> ... <fileN>
Switches:
  -dot      -  Enable dot graph output to file <out>, -dot=<out>.
  -dotpath  -  The path to the dot binary, -dotpath=<path>.
  -e        -  Enable simulation of process <procID>, -e=<procID>.
  -i        -  Interactive mode, read input from stdin.
  -no-tc    -  Disable typechecking.
  -pog      -  Enable Proof Obligation Generator output.
  -s2c      -  Enable state machine to CML tranlation, -s2c=<xmiFile>.
```

Assuming some CML model in a file, `example.cml`, loading it into the command-line interface is accomplished by typing `cmlc example.cml`. If run in this manner, the output will be:

```
Symphony command line CML Checker
Parsing files: [example.cml]
Type checking
[model types are ok]
```

If there are errors in the file, they will be reported in a manner similar to:

```
Symphony command line CML Checker
Parsing files: [example.cml]
Type checking
    Error 3430: Non-compatible type <CONECT> detected for communication
         parameter index 0 expected (CoSimulationProtocol | TCP_Event)
        in 'example.cml' at line 77:27
```

Note that, by default, the interpreter is not invoked on input; see Section 14.4 for such usage.

It is also possible to input CML directly into the command-line tool when invoked with the `-i` option. This is useful for quickly cutting and pasting small bits of CML, for example.

To generate a SVG graph representation of a parsed CML model, we use the
`-dot=<file>` and
`-dotpath=<executable-path>`
options. The invocation
`cmlc -dot=example -dotpath=/usr/bin/dot example.cml`
will produce the same output as above, but will also create the file `example.svg` with a representation of the parsed model.

14.3 Proof Obligation Generation

Invoking the command-line tool with the `-pog` option will cause the proof obligation generator to be run over the CML model, if the model passed the type checker. The resulting obligations (if any) are printed directly in the tool's output. They may then be inspected by the user, though nothing else may be done with them at this point.

Running the POG against the Dwarf Signal example (bundled with the Symphony IDE) results in the following (truncated) output:

```
Symphony command line CML Checker
Parsing files: [Dwarf.cml]
Type checking
    Warning 5001: Instance variable 'dw' is not initialized. in '/Users
        /jwc/Documents/compass/Common/PublicLiveCMLCaseStudies/Dwarf/
        Dwarf.cml' at line 68:2
: type invariant satisfiable obligation @ in '/Users/jwc/Documents/
    compass/Common/PublicLiveCMLCaseStudies/Dwarf/Dwarf.cml' at line
    4:2
(exists ps:ProperState & (ps in set {dark, stop, warning, drive}))
: type invariant satisfiable obligation @ in '/Users/jwc/Documents/
    compass/Common/PublicLiveCMLCaseStudies/Dwarf/Dwarf.cml' at line
    7:2
(exists d:DwarfType & (((((d.currentstate) \ (d.turnoff)) union (d.
    turnon)) = (d.desiredproperstate)) and ((((d.turnoff) inter (d.
    turnon)) = {}) and (((d.currentstate) <> {<L1>, <L2>, <L3>}) and
    (((card (((d.currentstate) \ (d.laststate)) union ((d.laststate) \
    (d.currentstate)))) <= 1) and ((((d.lastproperstate) = stop) => ((d
    .desiredproperstate) <> drive)) and ((((d.lastproperstate) = dark)
    => ((d.desiredproperstate) in set {dark, stop})) and (((d.
    desiredproperstate) = dark) => ((d.lastproperstate) in set {dark,
    stop}))))))))))
: type invariant satisfiable obligation @ in '/Users/jwc/Documents/
    compass/Common/PublicLiveCMLCaseStudies/Dwarf/Dwarf.cml' at line
    27:2
(exists d:DwarfSignal & (NeverShowAll(d) and (MaxOneLampChange(d) and (
    ForbidStopToDrive(d) and (DarkOnlyToStop(d) and DarkOnlyFromStop(d)
    )))))
...
```

14.4 CML Simulation

The command-line tool enables simulation of CML models when invoked with the `-e` option. Since the CML model may have more than one process defined, the `-e=<procId>` option must be supplied, where `<procId>` is the name of the process that is to be simulated.

As an example of how this works, consider the following CML model in a file called `example.cml`:

```
channels
  init, a, b

process A = begin
  @ init -> a -> Skip
```

```
end

process B = begin
  @ init -> b -> Skip
end

process C = A;B
```

The following command will simulate the process identified by `C`:

```
cmlc -e=C example.cml
```

This results in the following (truncated) output being printed to the console:

```
Symphony command line CML Checker
Parsing files: [example.cml]
Type checking
[model types are ok]
Simulator status event : INITIALIZED
Simulator status event : RUNNING
Simulator status event : WAITING_FOR_ENVIRONMENT
The system picked: tau('A()'->'begin class $actionClass0...') : [C
    Next: A()]
Simulator status event : RUNNING
Simulator status event : WAITING_FOR_ENVIRONMENT
The system picked: tau('begin class $actionClass0...'->'init -> a ->
    Skip') : [C = A#1
    Next: begin
class $actionClass0
end $actionClass0
 @ init -> a -> Skip
end]
Simulator status event : RUNNING
Top CML behavior: WAITING_CHILD
Waiting for environment on : init, tock
Simulator status event : WAITING_FOR_ENVIRONMENT
[0]init
[1]tock
0
The environment picked: init
Executing: init
Simulator status event : RUNNING
Top CML behavior: RUNNING
Top CML behavior: WAITING_CHILD
Waiting for environment on : a, tock
Simulator status event : WAITING_FOR_ENVIRONMENT
[0]a
[1]tock
0
...
```

Note that the two zeros on lines by themselves are user input: the simulator will run until the environment must choose an observable event to synchronise on, then present the choice of synchronisable events to the user. The user must then enter the number of the chosen event, then the simulator will continue.

14.5 SysML to CML translation

Given an example SysML model that has been exported to an XMI file from Artisan Studio, it is possible to use the command-line tool to convert the model into CML. If the XMI file is named `example.xmi`, and it contains a state machine called "ExampleModel", then the output of the translation will be placed in a subdirectory called `ExampleModel`. Note that the output path is based on the name of the model, not the XMI filename.

Details on how the translation is used within the Symphony IDE may be found in Section 13, and details on the translation process itself may be found in COMPASS Deliverable D31.4d [ML14].

15 Conclusion

This user manual provides a basic guide to the use of the Symphony IDE and where to find and activate the tool's features. Section 2 illustrated how to get hold of the software whereas the following sections introduced the Eclipse terminology with perspectives, projects and basic error handling.

The plug-ins for CML model simulation (Section 6), collaborative modelling using multiple instances of the Symphony IDE (Section 7), automated theorem proving of proof obligations (Section 8), model checking of CML models (Section 9), using the Model Checker for analysis of fault tolerance (Section 9), the refinement support features (Section 11), automated test generation (Section 12) and simplified support for translating SysML to CML models (Section 13) are fully integrated into the main Symphony IDE. Some of the plugins –for fault tolerance verification (Section 10) and refinement calculation (see COMPASS Deliverable D33.4 [FM14])– are explained more elaborately from a theoretical perspective in their own deliverables. The overall capabilities of the Symphony IDE are broad, and have a high degree of reuse of the component functionality.

Some of the plug-ins, due to external dependencies, require the use of software beyond what is distributed by the COMPASS project; these cases are documented in the sections belonging to the individual plug-ins, and core features of the Symphony IDE remain wholly usable without the plug-in functionality.

The Symphony IDE is still not an industrial-strength product, but the focus of the third and last year of the COMPASS project has been dedicated to providing both engineering improvements of the features. It is worthwhile noting that all the features of the Symphony IDE tool are properly integrated and represent a clear progress beyond state-of-the-art of the baseline technology. It is now possible to analyse the same CML models both from simulation, test automation, model checking and theorem proving perspectives. Many external tools have been integrated and the installation of these has been made as easy as possible for users. However, because of licensing concerns it is not possible to make the installation of the external tools fully automated.

References

[APR+13] Z.H. Andrews, R. Payne, A. Romanovsky, A.L.R. Didier, and A. Mota. Model-based development of fault tolerant systems of systems. In *7th International Systems Conference, IEEE SysCon*. IEEE, April 2013.

[BGW12] Jeremy Bryans, Andy Galloway, and Jim Woodcock. CML definition 1. Technical report, COMPASS Deliverable, D23.2, September 2012.

[BPS12] Jörg Brauer, Jan Peleska, and Uwe Schulze. Efficient and trustworthy tool qualification for model-based testing tools. In Brian Nielsen and Carsten Weise, editors, *Testing Software and Systems. Proceedings of the 24th IFIP WG 6.1 International Conference, ICTSS 2012, Aalborg, Denmark, November 2012*, number 7641 in Lecture Notes in Computer Science, pages 8–23, Heidelberg Dordrecht London New York, 2012. Springer.

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

[FLW12] John Fitzgerald, Peter Gorm Larsen, and Jim Woodcock. Modelling and Analysis Technology for Systems of Systems Engineering: Research Challenges. In *INCOSE*, Rome, Italy, July 2012.

[FM14] Simon Foster and Alvaro Miyazawa. Formal refinement support. Technical report, COMPASS Deliverable, D33.4, September 2014.

[FP13] Simon Foster and Richard J. Payne. Theorem proving support - developers manual. Technical report, COMPASS Deliverable, D33.2b, September 2013.

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

[LMN+14] Kenneth Lausdahl, Anders Kaels Malmos, Claus Ballegaard Nielsen, Joey W. Coleman, and Klaus Kristensen. Co-simulation engine. Technical report, COMPASS Deliverable, D32.4, June 2014.

[ML14] Alvaro Miyazawa and Peter Gorm Larsen. Fourth release of the COMPASS tool — sysml to cml translation mitigation effort. Technical report, COMPASS Deliverable, D31.4d, September 2014.

[Pel13] Jan Peleska. Industrial-Strength Model-Based Testing - State of the Art and Current Challenges. *Electronic Proceedings in Theoretical Computer Science*, abs/1303.1006:3–28, 2013.

[PVLZ11] Jan Peleska, Elena Vorobev, Florian Lapschies, and Cornelia Zahlten. Automated model-based testing with RT-Tester. Technical report, 2011. http://www.informatik.uni-bremen.de/agbs/testingbenchmarks/turn_indicator/tool/rtt-mbt.pdf.

[Ros10] A. W. Roscoe. *Understanding Concurrent Systems*. Springer, London Dordrecht Heidelberg New York, 2010.

[Ver13a] Verified Systems International GmbH, Bremen. *RT-Tester 6.0-4.9.8 – User Manual*, 2013. Available on request from Verified System International GmbH.

[Ver13b] Verified Systems International GmbH. RTT-MBT Model-Based Test Generator - RTT-MBT Version 9.0-1.0.0 User Manual. Technical Report Verified-INT-003-2012, Verified Systems International GmbH, 2013. Available on request from Verified System International GmbH.

[WCF+12] J. Woodcock, A. Cavalcanti, J. Fitzgerald, P. Larsen, A. Miyazawa, and S. Perry. Features of CML: a Formal Modelling Language for Systems of Systems. In *Proceedings of the 7th International Conference on System of System Engineering*. IEEE, July 2012.

[WG-11] RTCA SC-205/EUROCAE WG-71. Software Considerations in Airborne Systems and Equipment Certification. Technical Report RTCA/DO-178C, RTCA Inc, 1140 Connecticut Avenue, N.W., Suite 1020, Washington, D.C. 20036, December 2011.

A CML Support in the Simulator

This section gives an overview of the CML constructs that are implemented for interpretation. As all of the expression types are implemented, no detailed overview of them is given here.

The overview is divided into two subsections: actions (and statements which is a subgroup of actions); and processes. Each subsection contains a series of tables that group similar categories. The first column of each table gives the name of the operator, the second gives an informal syntax, and the last is a short description that gives the operator's status. If a construct is not supported entirely (no or partial implementation of the semantics), then the name of operator will be highlighted in red and a description of the issue will appear in the third column.

A.1 Actions

This section describes all of the supported and partially supported actions. Where `A` and `B` are actions, `e` is an expression, `P(x)` is a predicate expression with `x` free, `c` is a channel name, `cs` is a channel set expression, `ns` is a nameset expression.

Operator **Syntax**	**Comments**
Termination `Skip`	terminate immediately
Deadlock `Stop`	only allows time to pass
Divergence `Diverge`	runs without ever interacting in any observable event
Delay `Wait e`	does nothing for `e` time units, and then terminates.
Prefixing `c!e?x:P(x) -> A`	offers the environment a choice of events of the form `c.e.p`, where `p in set x | x: T @ P(x)`.
Guarded action `[e] & A`	if `e` is true, behave like `A`, otherwise, behave like `Stop`.
Sequential composition `A ; B`	behave like `A` until `A` terminates, then behave like `B`
External choice `A [] B`	offer the environment the choice between `A` and `B`.
Internal choice `A |~| B`	nondeterministically behave either like `A` or `B`.
Abstraction `A \\ cs`	behave as `A` with the events in `cs` hidden
Channel renaming `A[[ c <- nc ]]`	behaves as `A` with event `c` renamed to `nc`.
Mutual Recursion `mu X,... @ (F(X,...),...)`	explicit definition of mutually recursive actions.

Table 4: Action constructors.

Operator **Syntax**	**Comments**
Interrupt `A /_\ B`	behave as `A` until `B` takes control, then behave like `B`.
Timed interrupt `A /_ e _\ B`	behave as `A` for `e` time units, then behave as `B`.
Untimed timeout `A [_> B`	behave as `A`, but nondeterministically change behaviour to `B` at any time.
Timeout `A [_ e _> B`	offer `A` for `e` time units, then offer `B`.
Start deadline `A startsby e`	`A` must execute an observable event within `e` time units. Otherwise, the process is infeasible.
End deadline `A endsby e`	`A` must terminate within `e` time units. Otherwise, the process is infeasible

Table 5: Timed action constructors.

Operator **Syntax**	**Comments**
Interleaving (without state) `A ||| B`	execute `A` and `B` in parallel without synchronising. Neither `A` nor `B` change the state.
Interleaving (with state) `A [|| ns1 | ns2 ||] B`	execute `A` and `B` in parallel without synchronising. `A` can modify the state components in `ns1` and `B` can modify the state components in `ns2`.
Alphabetised parallelism (without state) `A [ cs1 || cs2 ] B`	execute `A` and `B` in parallel synchronising in the intersection of `X` and `Y`. `A` in only allowed to communicate on `X` and `B` is only allowed to communicate on `Y`. Neither `A` nor `B` change the state.
Alphabetised parallelism (with state) `A [ ns1 | cs1 || cs2 | ns2 ] B`	as with the "without state" version, but modifications from `A` to the state components in `ns1` and modifications from `B` to the state components in `ns2` are preserved at the completion of the operator.
Generalised parallelism (without state) `A [| cs |] B`	execute `A` and `B` in parallel synchronising on the events in `cs`. Neither `A` nor `B` change the state.
Generalised parallelism (with state `A [| ns1 | cs | ns2 |] B`	execute `A` and `B` in parallel synchronising on the events in `cs`. `A` can modify the state components in `ns1` and `B` can modify the state components in `ns2`.

Table 6: Parallel action constructors.

Operator **Syntax**	**Comments**
Replicated sequential composition `; i in seq e @ A(i)`	`e` must be a sequence, for each `i` in the sequence, `A(i)` is executed in order.
Replicated external choice `[] i in set e @ A(i)`	offer the environment the choice of all actions `A(i)` such that `i` is in the set `e`.
Replicated internal choice `|~| i in set e @ A(i)`	nondeterministically behave as `A(i)` for any `i` in the set `e`.
Replicated interleaving `||| i in set e @ [ns(i)] A(i)`	execute all actions `A(i)` in parallel without synchronising on any events. Each action `A(i)` can only modify the state components in `ns(i)`.
Replicated generalised parallelism `[|cs|] i in set e @ [ns(i)] A(i)`	execute all actions `A(i)` (for `i` in the set `e`) in parallel synchronising on the events in `cs`. Each action `A(i)` can only modify the state components in `ns(i)`.
Replicated alphabetised parallelism `|| i in set e @ [ns(i) | cs(i)] A(i)`	execute all processes `A(i)` in parallel synchronising on the intersection of all `cs(i)`. Each process `A(i)` can only perform events in `cs(i)` and can only modify the state components in `ns(i)`.

Table 7: Replicated action constructors.

Operator **Syntax**	**Comments**
Let `let p=e in a`	evaluate the action `a` in the environment where `p` is associated to `e`.
Block `(dcl v: T := e @ a)`	declare the local variable `v` of type `T` (optionally) initialised to `e` and evaluate action `a` in this context.
Assignment `v:=e`	assign e to v

Multiple assignment
`atomic (v1 := e1;`
`...;`
`vn := en)`
	assignments are executed atomically with respect to the state invariant.

Call (1) `obj.op(p)` (2) `op(p)` (3) `A(p)`	execute operation `op` of an object `obj` (1) or of the current object or process (2) with the parameters `p`. (3) execute action `A` with parameters `p`.

Assignment call
(1)
`v := obj.op(p)`
(2)
`v := op(p)`
	(1) execute operation `op` of an object `obj` or (2) execute operation `op` the current object or process with the parameters `p` and assign the value returned by `op` to a variable.

Return `return` or `return e`	terminates the evaluation of an operation possibly yielding a value `e`.
Specification `[frame ...]`	This cannot be interpreted.
New `v := new C()`	instantiate a new object of class `C` and assign it to `v`.

Table 8: CML statements.

Operator **Syntax**	**Comments**

Nondeterministic if statement
`if e1 -> a1`
`| e2 -> a2`
`| ...`
`end`
	evaluate all guards `ei`. If none are true, then the statement diverges. If one or more guards are true, one of the associated actions is executed nondeterministically.

If statement
`if e1 then a1`
`elseif e2 then a2`
`...`
`else an`
	the boolean expressions `ei` are evaluated in order. When the first `ei` is evaluated to true, the associated action is executed. If no `ei` evaluates to true, the action `an` is executed.

Cases statement
`cases e:`
`p1 -> a1,`
`p2 -> a2,`
`...,`
`others -> an`
`end`
	The expression `e` is matched against each pattern `pi` in order. The result of the cases statement is the first action `ai` whose associated pattern `pi` matches the expression `e`. If not pattern is matched, the `others` clause is executed.

Nondeterministic do statement
`do e1 -> a1`
`| e2 -> a2`
`| ...`
`end`
	if all guards `ei` evaluate to false, terminate. Otherwise, choose nondeterministically one guard that evaluates to true, execute the associated action, and repeat the do statement.

Sequence for loop
`for e in s`
`do a`
	for each expression `e` in the sequence `s`, execute action `a`.

Set for loop
`for all e in set S`
`do a`
	for each expression `e` in the set `S`, execute action `a`.

Index for loop
`for i=e1 to e2 by e3`
`do a`
	execute action `a` for each integer `i` in the range `1,...,e2e1,...,e2` such that `i = e1+(n*e3)` (where `n` is a natural number).

While loop `while e do a`	execute action `a` while the boolean expression `e` evaluates to true.

Table 9: Control statements.

A.2 Processes

This section describes all the supported and partially supported processes. `A` and `B` are both processes, `e` is an expression and `cs` is a channel expression.

Operator **Syntax**	**Comments**
Sequential composition `A ; B`	behave like `A` until `A` terminates, then behave like `B`
External choice `A [] B`	offer the environment the choice between `A` and `B`.
Internal choice `A |~| B`	nondeterministically behave either like `A` or `B`.
Generalised parallelism `A [| cs |] B`	execute `A` and `B` in parallel synchronising on the events in `cs`.
Alphabetised parallelism `A [ cs1 || cs2 ] B`	execute `A` and `B` in parallel synchronising in the intersection of `X` and `Y`. `A` is only allowed to communicate on `X` and `B` is only allowed to communicate on `Y`.
Interleaving `A ||| B`	execute `A` and `B` in parallel without synchronising.
Abstraction (Hiding) `A \\ cs`	behave as `A` with the events in `cs` hidden
Process instantiation `(v:T @ A)(e)` or `A(e)`	behaves as `A` where the formal parameters (`v`) are instantiated to `e`.
Channel renaming `A[[ c <- nc ]]`	behaves as `A` with event `c` renamed to `nc`.

Table 10: Process constructors.

Operator **Syntax**	**Comments**
Interrupt `A /_\ B`	behave as `A` until `B` takes control, then behave like `B`.
Timed interrupt `A /_ e _\ B`	behave as `A` for `e` time units, then behave as `B`.
Untimed timeout `A [_> B`	offer `A`, but may nondeterministically stop offering `A` and offer `B` at any time.
Timeout `A [_ e _> B`	offer `A` for `e` time units, then offer `B`.
Start deadline `A startsby e`	`A` must execute an observable event within `e` time units. Otherwise, the process is infeasible.
End deadline `A endsby e`	`A` must terminate within `e` time units. Otherwise, the process is infeasible

Table 11: Timed process constructors.

Operator **Syntax**	**Comments**
Replicated sequential composition `; i in seq e @ A(i)`	`e` must be a sequence, for each `i` in the sequence, `A(i)` is executed in order.
Replicated external choice `[] i in set e @ A(i)`	offer the environment the choice of all processes `A(i)` such that `i` is in the set `e`.
Replicated internal choice `|~| i in set e @ A(i)`	nondeterministically behave as `A(i)` for any `i` in the set `e`.
Replicated generalised parallelism `[| cs |] i in set e @ A(i)`	execute all processes `A(i)` (for `i` in the set `e`) in parallel synchronising on the events in `cs`.
Replicated alphabetised parallelism `|| i in set e @ [cs(i)] A(i)`	execute all processes `A(i)` in parallel synchronising on the intersection of all `cs(i)`. Each process `A(i)` can only perform events in `cs(i)`.
Replicated interleaving `||| i in set e @ A(i)`	execute all processes `A(i)` in parallel without synchronising on any events.

Table 12: Replicated process constructors.

B CML Support in the POG

This section gives a brief overview of the CML constructs that are supported by the POG. If a construct is supported, then any relevant POs will be generated for said construct. If a construct is not supported, the POG will simply ignore it and generate POs for other constructs. In practice, this means that the POG will run on any CML model, regardless of its contents.

B.1 Processes

No process-specific POs are generated. However, POs are generated for the various constructs inside a process definition.

B.2 Classes

There are no POs specific to classes. However, POs are generated for the various constructs inside a class definition.

B.3 Actions

Actions are not supported by the POG. No POs are generated for actions.

B.4 Channels and Chansets

No POs exist for channels or chansets.

B.5 Namesets

No POs exist for namesets.

B.6 Operations

Operations are fully supported by the POG. Obligations will be generated for any operation that requires them.

B.7 Types

Types are fully supported by the POG. Obligations will be generated for any type that requires them.

B.8 Functions

Functions are fully supported by the POG. Obligations will be generated for any function that requires them.

B.9 Values

Values are fully supported by the POG. Obligations will be generated for any value initialization that requires them.

B.10 State

State (both in processes and classes) is supported by the POG. POs will be generated for any state initializations and invariants that require them.

C CML Support in the Theorem Prover

This section gives an overview of the CML constructs that are implemented. We present the constructs using tables where the first column of each table gives the name of the operator, the second gives an informal syntax, and the last is a short description that gives the operator's status. If a construct is not supported entirely, then the name of operator will be highlighted in red, whilst partially supported operators are written in blue.

C.1 Actions

The following tables describe all of the supported and partially supported actions. Where `A` and `B` are actions, `e` is an expression, `P(x)` is a predicate expression with `x` free, `c` is a channel name, `cs` is a channel set expression, `ns` is a nameset expression.

Operator **Syntax**	**Comments**
Termination `Skip`	terminate immediately
Deadlock `Stop`	It yields a state with no outgoing transition
Chaos `Chaos`	can always choose to communicate or reject any event
Divergence `Div`	It yields a livelock
Delay `Wait e`	Wait for e time units
Prefixing `c!e?x:P(x) -> A`	offers the environment a choice of events
Guarded action `[e] & A`	if `e` is true, behave like `A`, otherwise, behave like `Stop`.
Sequential composition `A ; B`	behave like `A` until `A` terminates, then behave like `B`
External choice `A [] B`	offer the environment the choice between `A` and `B`.
Internal choice `A |~| B`	nondeterministically behave either like `A` or `B`.
Interrupt `A /_\ B`	Interrupt A by B
Timed interrupt `A /_ e _\ B`	Interrupt A by B for e time units
Untimed timeout `A [_> B`	Timeout to B
Timeout `A [_ e _> B`	Timeout to B after e time units
Abstraction `A \\ cs`	behave as `A` with the events in `cs` hidden.

Table 13: Action constructors

Operator Syntax	Comments
Start deadline `A startsby e`	Not implemented
End deadline `A endsby e`	Not implemented
Channel renaming `A[[ c <- nc ]]`	Not implemented
Recursion `mu X @ F(X)`	Definition of a recursive action
Mutual Recursion `mu X,Y@(F(X,Y),G(X,Y))`	Unsupported in the "mu" form, full support when using mutual equations
Interleaving `A [|| ns1	ns2 ||] B`	Not implemented
Interleaving (no state) `A ||| B`	execute `A` and `B` in parallel without synchronising. Neither `A` nor `B` change the state.
Synchronous parallelism `A [| ns1	ns2|] B`	Not implemented
Synchronous parallelism (no state) `A || B`	Not implemented
Alphabetised parallelism `A [ns1|cs1||cs2|ns2] B`	Not implemented
Alphabetised parallelism (no state) `A [cs1 || cs2] B`	Not implemented
Generalised parallelism `A [|ns1	cs	ns2|] B`	Not implemented
Generalised parallelism (no state) `A [| cs |] B`	execute `A` and `B` in parallel synchronising on the events in `cs`. Neither `A` nor `B` change the state.

Table 14: Parallel action constructors

Operator **Syntax**	**Comments**
Replicated sequential composition `; i in seq e @ A(i)`	Execute A for parameter i in sequence over e
Replicated external choice `[] i in set e @ A(i)`	External choice indexed by i in set e
Replicated internal choice `|~| i in set e @ A(i)`	Internal choice indexed by i in set e

Replicated interleaving
`||| i in set e`
`@ [ns(i)] A(i)`
	Not implemented

Replicated generalised parallelism
`[|cs|] i in set e`
`@ [ns(i)] A(i)`
	Not implemented

Replicated alphabetised parallelism
`|| i in set e`
`@ [ns(i) | cs(i)] A(i)`
	Not implemented

Replicated synchronous parallelism
`|| i in set e`
`@ [ns(i)] A(i)`
	Not implemented

Table 15: Replicated action constructors

Operator **Syntax**	**Comments**
Let `let p=e in a`	Local constant declaration
Block `(dcl v: T := e @ a)`	declare the local variable `v` of type `T` (optionally) initialised to `e` and evaluate action `a` in this context.
Assignment `v:=e`	assign e to v
Multiple assignment `atomic (v1 := e1, ..., vn := en)`	assignments are executed atomically with respect to the state invariant.

Call
`(1)op(p)`
`(2)A(p)`
	execute operation `op` of the current or process (1) with the parameters `p`. (2) execute action `A` with parameters `p`.

Assignment call `v := op(p)`	Execute operation op, assign result to v
Return `return e` or `return`	Return value e in an operation

Specification
`[frame`
`wr v1: T1`
`rd v2: T2`
`pre P1(v1,v2)`
`post`
`P2(v1,v1~,v2,v2~)]`
	VDM specification statement

New `v := new C()`	Not implemented

Table 16: CML statements

Operator **Syntax**	**Comments**

Nondeterministic if statement
`if e1 -> a1`
`  | e2 -> a2`
`  | ...`
`  end`
	Not implemented

If statement
`if e1 then a1`
`  [elseif e2 then a2]*`
`else an`
	The condition e1 is used to enable the statement a1 or an. The optional elseif are not allowed.

Cases statement
`cases e:`
`p1 -> a1,`
`p2 -> a2,`
`others -> an`
`end`
	Not implemented

Nondeterministic do statement
`do e1 -> a1`
`| e2 -> a2`
`| ...`
`end`
	Not implemented

Sequence for loop `for e in s do a`	Not implemented
Set for loop `for all e in set S do a`	Not implemented
Index for loop `for i=e1 to e2 by e3 do a`	Not implemented
While loop `while e do a`	repeat a while e evaluates to false

Table 17: Control statements

C.2 Declarations

Operator **Syntax**	**Comments**

Value Declaration
`values`
`  value definitions`
	Definitions of values to be used in a cml model.

Value Definition `N:nat = 2`	It declares the value N as a natural number and assigns the value 2 to it.

Channel Name Declarations
`channels`
`  a, b : type`
	it declares the channels a and b of a specific type

Chanset Declarations
`chansets`
`  chanset definition`
	it declares a set of channels

Nameset Declarations
`namesets`
`  nameset definition`
	Not implemented

State Declarations
`state`
`  value: nat := 0`
`  ...`
	It defines the state structure containing the variable value. Invariants implemented as UTP designs.

Process Declaration
`process P=val x:nat @`
`  process_body`
	parametrised processes fully supported

Action Declarations
`actions`
`A = val i:int @ action`
	Full support including parametrised actions

Table 18: Declarations

C.3 Types

- all basic types including nat, string, token etc. are supported
- quote types exist as a single type in Isabelle such that union types over them can be constructed
- compound types, set, map, seq, seq1 and products are supported
- record types are supported through tagged product types (similar to HOL records)
- union types are currently partially supported for types with a common subtype (e.g. nat and real)

C.4 Expressions

- boolean expressions (with three values): full support
- numeric expressions: full support
- token expressions: full support
- set expressions: partial support (all operators other than *set comprehension*)
- sequence expressions: partial support (*seq comprehension* missing)
- map expressions: partial support
  - *map comprehension* missing
  - *range restriction* missing
  - *iteration* missing
  - *inverse* missing
- product expressions: full support
- record expressions: partial support (*is_* expressions currently missing)

C.5 Operations

Operator **Syntax**	**Comments**

Operation Declaration
 	
`operations`
`Credit: nat ==> ()`
`Credit(n) ==`
`balance:=balance+n`
	It defines a new operation Credit, which receives a natural number and does not return values. The semantics of Credit is to change the value of balance, which should had been defined in the state. Pre and post conditions are nor allowed. The constructs 'frame' 'rd' and 'wr' are not allowed.

Table 19: Operations

D CML Support in the Model Checker

This section gives an overview of the CML constructs that are implemented. We present the constructs using tables where the first column of each table gives the name of the operator, the second gives an informal syntax, and the last is a short description that gives the operator's status. If a construct is not supported entirely (no or partial implementation of the semantics), then the name of operator will be highlighted in red and a description of the issue will appear in the third column.

D.1 Actions

The following tables describe all of the supported and partially supported actions. Where `A` and `B` are actions, `e` is an expression, `P(x)` is a predicate expression with `x` free, `c` is a channel name, `cs` is a channel set expression, `ns` is a nameset expression.

Operator **Syntax**	**Comments**
Termination `Skip`	terminate immediately
Deadlock `Stop`	It yields a state with no outgoing transition
Divergence `Div`	It yields a livelock
Delay `Wait e`	It waits specific for units of time, represented by `tock` events
Prefixing `c!e?x:P(x) -> A`	offers the environment a choice of events of the form `c.e.p`, where `p in set {x | x: T @ P(x)}`. Only forms like `c.x` (where x is an integer) are supported.
Guarded action `[e] & A`	if `e` is true, behave like `A`, otherwise, behave like `Stop`.
Sequential composition `A ; B`	behave like `A` until `A` terminates, then behave like `B`
External choice `A [] B`	offer the environment the choice between `A` and `B`.
Internal choice `A |~| B`	nondeterministically behave either like `A` or `B`.
Abstraction `A \\ cs`	behave as `A` with the events in `cs` hidden.
Channel renaming `A[[ c <- nc ]]`	Not implemented
Recursion `mu X @ ( F(X) )`	definition of a recursive action. Recursive actions must be explicit (e.g. P = `P = a -> P`).

Mutual Recursion
`mu X,Y @`
`(F(X,Y),G(X,Y))`
	Not implemented

Table 20: Action constructors.

Operator **Syntax**	**Comments**
Interrupt `A /_\ B`	The usual CML untimed interruption operator.
Timed interrupt `A /_ e _\ B`	The usual CML timed interruption operator considering `e` units of time
Untimed timeout `A [_> B`	The usual CML untimed timeout operator
Timeout `A [_ e _> B`	The usual CML timeout operator considering `e` units of time
Start deadline `A startsby e`	Not implemented
End deadline `A endsby e`	Not implemented

Table 21: Timed action constructors

Operator **Syntax**	**Comments**
Interleaving (no state) `A ||| B`	execute `A` and `B` in parallel without synchronising. Neither `A` nor `B` change the state.
Synchronous parallelism `A [| ns1 | ns2|] B`	Not implemented
Synchronous parallelism (no state) `A || B`	Not implemented
Alphabetised parallelism `A [ns1|cs1||cs2|ns2] B`	Not implemented
Alphabetised parallelism (no state) `A [cs1 || cs2] B`	Not implemented
Generalised parallelism `A [|ns1 | cs | ns2|] B`	Not implemented
Generalised parallelism (no state) `A [| cs |] B`	execute `A` and `B` in parallel synchronising on the events in `cs`. Neither `A` nor `B` change the state.

Table 22: Parallel action constructors.

Operator **Syntax**	**Comments**
Replicated sequential composition `; i in seq e @ A(i)`	`e` must be a sequence, for each `i` in the sequence, `A(i)` is executed in order.
Replicated external choice `[] i in set e @ A(i)`	offer the environment the choice of all actions `A(i)` such that `i` is in the set `e`.
Replicated internal choice `|~| i in set e @ A(i)`	nondeterministically behave as `A(i)` for any `i` in the set `e`.
Replicated interleaving `||| i in set e @ [ns(i)] A(i)`	Not implemented
Replicated generalised parallelism `[|cs|] i in set e @ [ns(i)] A(i)`	execute all actions `A(i)` (for `i` in the set `e`) in parallel synchronising on the events in `cs`. Each action `A(i)` can only modify the state components in `ns(i)`.
Replicated alphabetised parallelism `|| i in set e @ [ns(i) | cs(i)] A(i)`	Not implemented
Replicated synchronous parallelism `|| i in set e @ [ns(i)] A(i)`	Not implemented

Table 23: Replicated action constructors.

Operator **Syntax**	**Comments**
Let `let p=e in a`	Not implemented
Block `(dcl v: T := e @ a)`	declare the local variable `v` of type `T` (optionally) initialised to `e` and evaluate action `a` in this context.
Assignment `v:=e`	assign e to v
Multiple assignment `atomic (v1 := e1, ..., vn := en)`	Not implemented

Call
`(1)op(p)`
`(2)A(p)`
	execute operation `op` of the current or process (1) with the parameters `p`. (2) execute action `A` with parameters `p`.

Assignment call `v := op(p)`	Not implemented
Return `return e` or `return`	Not implemented

Specification
`[frame`
`wr v1: T1`
`rd v2: T2`
`pre P1(v1,v2)`
`post`
`P2(v1,v1~,v2,v2~)]`
	Not implemented

New `v := new C()`	Not implemented

Table 24: CML statements.

Operator **Syntax**	**Comments**

Nondeterministic if statement
`if e1 -> a1`
`  | e2 -> a2`
`  | ...`
`  end`
	Not implemented

If statement
`if e1 then a1`
`  [elseif e2 then a2]*`
`else an`
	The condition e1 is used to enable the statement a1 or an. The optional elseif are not allowed.

Cases statement
`cases e:`
`p1 -> a1,`
`p2 -> a2,`
`others -> an`
`end`
	Not implemented

Nondeterministic do statement
`do e1 -> a1`
`| e2 -> a2`
`| ...`
`end`
	Not implemented

Sequence for loop `for e in s do a`	Not implemented
Set for loop `for all e in set S do a`	Not implemented
Index for loop `for i=e1 to e2 by e3 do a`	Not implemented
While loop `while e do a`	Not implemented

Table 25: Control statements.

D.2 Declarations

Operator Syntax	Comments

Value Declaration
`values`
`  value definitions`
	Definitions of values to be used in a cml model.

Value Definition `N : nat = 2`	It declares the value N as a natural number and assigns the value 2 to it.

Channel Name Declarations
`channels`
`  a, b : type`
	It declares the channels a and b supporting a specific type

Chanset Declarations
`chansets`
`  chanset definition`
	It declares sets of events at once

Nameset Declarations
`namesets`
`  nameset definition`
	Not implemented

State Declarations
`state`
`  value: nat := 0`
`  ...`
	It defines the state structure containing the variable value. Qualifiers and invariants are not allowed in instance variable definitions.

Process Declaration
`process P = val x: nat @`
`  process_body`
	It declares the process P with a parameter. Only the qualifier val is allowed.

Action Declarations
`actions`
`  A = val i:int @ action`
	It declares the action A with the parameter i. Only the qualifier val is allowed for parameter.

Table 26: Declarations

D.3 Types

Operator **Syntax**	**Comments**

Type Declaration
 	
`types`
`Index = nat`
`inv i==`
`i in set{1,...,10}`
	The type Index is defined as a natural number whose values are limited by the invariant expression. That is, the value of Index can be from 1 to 10. Only basic types are allowed to be used as types of user defined types. Field types are not allowed.

Table 27: Types

D.4 Operations

Operator **Syntax**	**Comments**

Operation Declaration
 	
`operations`
`Credit: nat ==> ()`
`Credit(n) ==`
`balance:=balance+n`
	It defines a new operation Credit, which receives a natural number and does not return values. The semantics of Credit is to change the value of balance, which should had been defined in the state. Pre and post conditions are nor allowed. The constructs 'frame' 'rd' and 'wr' are not allowed.

Table 28: Operations