Grant Agreement: 287829

Comprehensive Modelling for Advanced Systems of Systems

CML Interpreter Design Document

Technical Note Number: DXX

Version: 0.2

Date: Month Year

Public Document

http://www.compass-research.eu

Contributors:

Anders Kaels Malmos, AU

Editors:

Peter Gorm Larsen, AU

Reviewers:

Document History

Ver	Date	Author	Description
0.1	25-04-2013	Anders Kaels Malmos	Initial document version
0.2	06-03-2014	Anders Kaels Malmos	Added introduction and domain description

Abstract

This document describes the overall design of the CML interpreter and provides an overview of the code structure targeting developers. It assume a basic knowledge of CML.

Contents

1 Introduction

This document is targeted at developers and describes the overall design of the CML simulator, it is not a detailed description of every part of the source code. This kind of documentation is done in Javadoc and can be generated automatically from the code. It is assumed that common design patterns are known like **??** and a basic understaning of CML.

1.1 Problem Domain

The goal of the interpreter is to enable simulation/animation of a given CML **??** model and be able to visualize this in the Eclipse IDE Debugger. CML has a UTP semantics defined in **??** which dictates how the interpretation progresses. Therefore, the overall goal of the CML interpreter is to adhere to the semantic rules defined in those documents and to somehow visualize this in the Eclipse Debugger.

In order to get a high level understanding of how CML is interpreted without knowing all the details of the semantics and the implementation of it. A short illustration of how the interpreter represents and progresses a CML model is given below.

In listing 1 a CML model consisting of three CML processes is given. It has a R (Reader) process which reads a value from the inp channel and writes it on the out channel. The W (Writer) process writes the value 1 to the inp channel and finishes. The S (System) process is a parallel composition of these two processes where they must synchronize all events on the inp channel.

```
channels
inp : int
out : int

process W =
begin
 @ inp!1 -> Skip
end

process R =
begin
 @ inp?x -> out!x -> Skip
end

process S = W [|{$inp$}|] R
```

Listing 1: A process S composed of a parallel composition of a reader and writer process

The interpretation of a CML model is done through a series of steps/transitions starting from a given entry point. In figure 1 the first step in the interpretation of the model is shown, it is assumed that the S process is given as the starting point. Process are represented as a circle along with its current position in the model. Each step of the execution is split up in two phases, the inspection phase and the execution phase. The dashed lines represents the environment (another actor that invokes the operation e.g a human user or another process) initiating the phase.

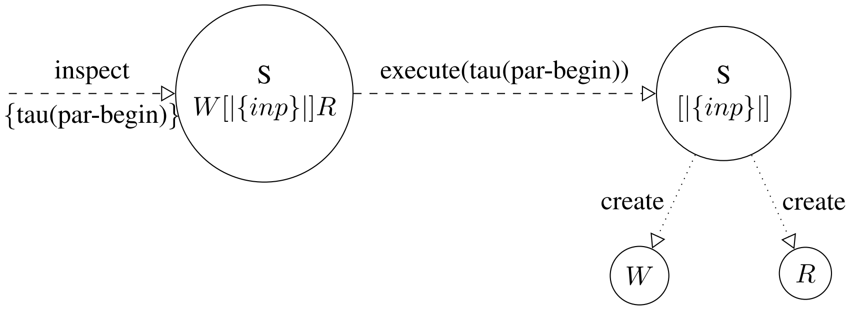

Figure 1: Initial step of Listing 1 with process S as entry point.

The inspection phase determines the possible transitions that are available in the next step of execution. The result of the inspection is shown as a set of transitions below "inspect". As seen on figure Figure 1 process P starts out by pointing to the parallel composition constructs, this construct has a semantic begin rule which does the initialization needed. In the figure Figure 1 that rule is named tau(par-begin) and is therefore returned from the inspection. The reason for the name tau(..) is that transitions can be either observable or silent, so in principle any tau transition is not observable from the outside of the process. However, in the interpreter all transitions flows out of the inspection phase. When the inspection phase has completed, the execution phase begins. The execution phase executes one of the transitions returned from the inspection phase. In this case, only a single transition is available so the tau(par-begin)) is executed which creates the two child processes. The result of each of the shown steps are the first configuration shown in the next step. So in this case the resulting process configuration of Figure 1 is shown in figure Figure 2.

The second step on Figure 2 has a more interesting inspection phase. According to the parallel composition rule, we have that any event on the inp channel must be synchronized, meaning that W and R must only perform transition that involves inp channel events synchronously.

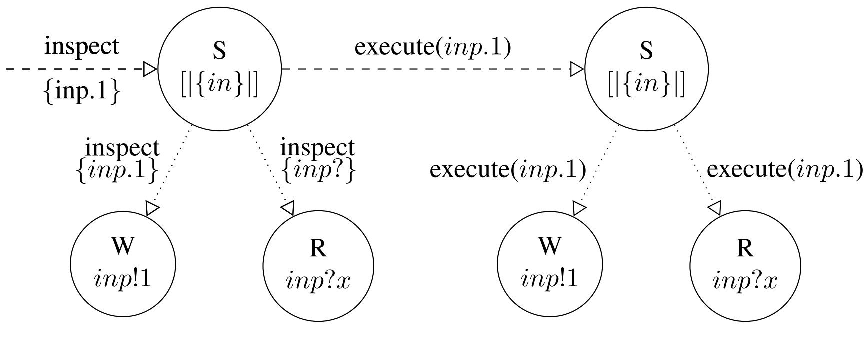

Figure 2: Second step of Listing 1 with S as entry point.

Therefore, when P is inspected it must inspect its child processes to determine the possible transitions. In this case W can perform the $inp.1$ event and R can perform any event on inp and therefore, the only possible transition is the one that performs the $inp.1$ event. This is then given to the execution phase which result in the $inp.1$ event and moves both child processes into their next state.

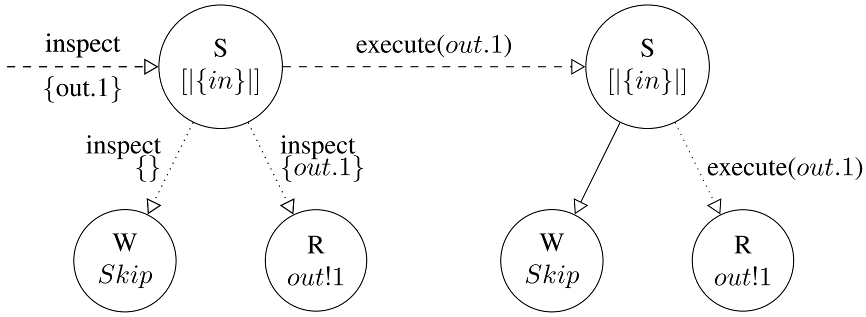

Figure 3: Third step of Listing 1 with S as entry point

In the third step on figure Figure 3 W is now Skip which means that it is successfully terminated. The inspection for W therefore results in an empty set of possible transitions. R is now waiting for the $out.1$ event after 1 was writting to x in the last step and therefore returns this transition. The execution phase is a little different and S now knows only to execute R.

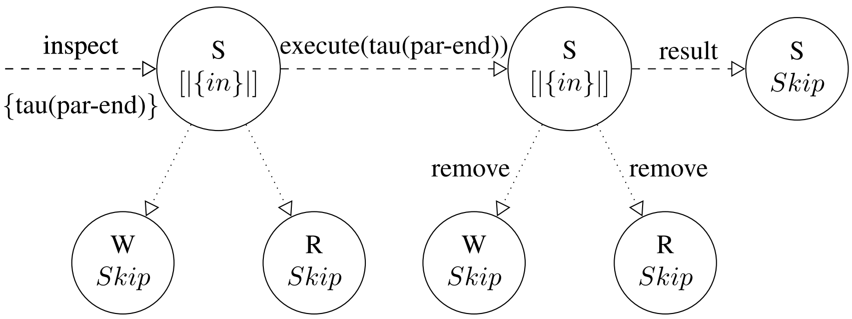

Figure 4: Final step of Listing 1 where the parallel composition collapses unto a Skip process

The fourth and final step shown in Figure 4 of the interpretation starts out with both W and R as Skip, this triggeres the parallel end rules, which evolves into Skip. S therefore returns the silent transition the triggers this end rule.

1.2 Definitions

CML Compass Modelling Language

UTP Unified Theory of Programming, a semantic framework.

Simulation Simulation is when the interpreter runs without any form of user interaction other than starting and stoppping.

Animation Animation is when the user are involved in taking the decisions when interpreting the CML model

2 Software Layers

This section describes the layers of the CML interpreter. As depicted in figure 5 two highlevel layers exists.

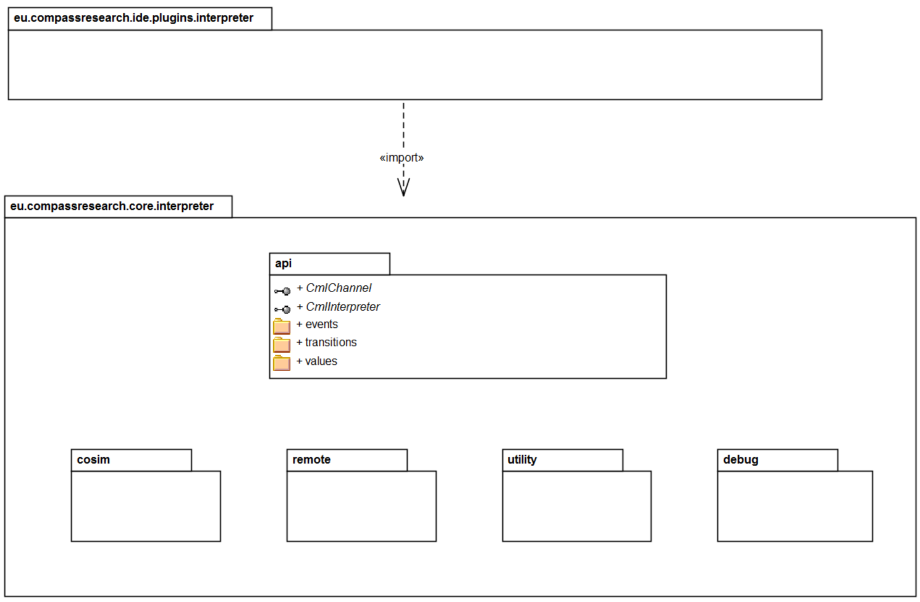

Figure 5: The layers of the CML Interpreter

Each of these components will be described in further detail in the following sections.

2.1 The Core Layer

This layer has the overall responsibility of interpreting a CML model as described in the operational semantics that are defined in [**?**] and is located in the java package named *eu.compassresearch.core.interpreter*. The design philosophy of the top-level structure is to encapsulate all the classes and interfaces that makes up the implementation of the core functionality and only expose those that are needed to utilize the interpreter. This provides a clean separation between the implementation and interface and makes it clear for both the users, which not necessarily wants to know about the implementation details, and developers which parts they need to work with.

The following packages defines the top level structure of the core:

eu.compassresearch.core.interpreter This package contains all the internal classes and interfaces that defines the core functionality of the interpreter. There is one important public class in the package, namely the **VanillaInterpreteFactory** factory class, that any user of the interpreter must invoke to use the interpreter. This can creates instances of the **CmlInterpreter** interface.

eu.compassresearch.core.interpreter.api This package and sub-packages contains all the public classes and interfaces that defines the API of the interpreter. Some of the most important entities of this package includes the main interpreter interface **CmlInterpreter** along with the **CmlBehaviour** interface that represents a CML process or action. It corrosponds to the circles in the figures of Subsection 1.1.

eu.compassresearch.core.interpreter.api.events This package contains all the public components that enable users of the interpreter to subscribe to multiple on events (this it not CML channel events) from both **CmlIntepreter** and **CmlBehaviour** instances.

eu.compassresearch.core.interpreter.api.transitions This package contains all the possible types of transitions that a **CmlBehaviour** instance can make. This will be explained in more detail in section 3.1.2.

eu.compassresearch.core.interpreter.api.values This package contains all the values used by the CML interpreter. They represent the values of variables and constants in a context.

eu.compassresearch.core.interpreter.cosim Has the responsiblity of running a co-simulation. A co-simulation can be either between multiple instances of the CML intepreter co-simulating a CML model, or a CML intepreter instance co-simulating a CML model with a real live system.

eu.compassresearch.core.interpreter.remote This has the responsiblilty of exposing the CML interpreter to be remote controlled.

eu.compassresearch.core.interpreter.debug Has the responsibility of controlling a debugging sessions either from the console or in our case the Eclipse debugger.

eu.compassresearch.core.interpreter.utility The utility packages contains components that generally reusable classes and interfaces.

The **eu.compassresearch.core.interpreter** package are split into several folders, each representing a different logical component. The following folders are present

behavior This folder contains all the internal classes and interfaces that implements the CmlBehaviors. The Cml behaviors will be described in more detail in in section 3.1.1, but they are bassically implemented by CML AST visitor classes.

2.2 The IDE Layer

Has the overall responsibility of visualizing the outputs of a running interpretation a CML model in the Eclipse Debugger. It is located in the *eu.compassresearch.ide.plugins.interpreter* package. The IDE part is integrating the interpreter into Eclipse, enabling CML models to be debugged/simulated/animated through the Eclipse interface. In Figure 6 a deployment diagram of the debugging structure is shown.

An Eclipse debugging session involves two JVMs, the one that the Eclipse platform is executing in and one where only the Core executes in. All communication between them is done via a TCP connection.

Before explaining the steps involved in a debugging session, there are two important classes worth mentioning:

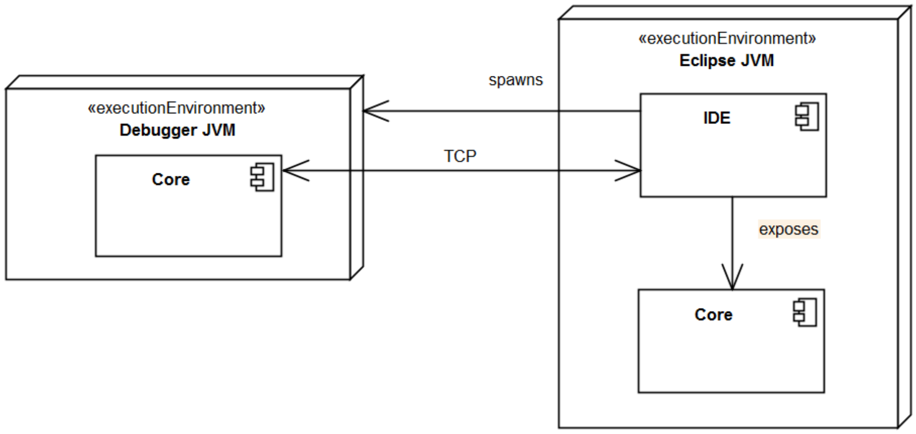

Figure 6: Deployment diagram of the debugger

- **CmlDebugger**: This is responsible for controlling the CmlInterpreter execution in the debugger JVM. All communications to and from the interpreter handled in this class.
- **CmlDebugTarget**: This class is part of the Eclipse debugging model. It has the responsiblity of representing a running interpreter on the Eclipse JVM side. All communications to and from the Eclipse debugger are handled in this class.

A debugging session has the following steps:

1. The user launches a debug session
2. On the Eclipse JVM a **CmlDebugTarget** instance is created, which listens for an incomming TCP connection.
3. A Debugger JVM is spawned and a **CmlInterpreterController** instance is created.
4. The **CmlInterpreterController** tries to connect to the created connection.
5. When the connection is established, the **CmlInterpreterController** instance will send a STARTING status message along with additional details
6. The **CmlDebugTarget** updates the GUI accordingly.
7. When the interpreter is running, status messages will be sent from **CmlInterpreterController** and commands and request messages are sent from **CmlDebugTarget**
8. This continues until **CmlInterpreterController** sends the STOPPED message

3 Layer design and Implementation

This section describes the static and dynamic structure of the components involved in simulating/animating a CML model.

3.1 Core Layer

The core layer is responsible for the overall interpretation of a given CML model. In the following section both the static and dynamic model will be described in details.

3.1.1 Static Model

The top level interface of the interpreter is depicted in figure 7, followed by a short description of each the depicted components.

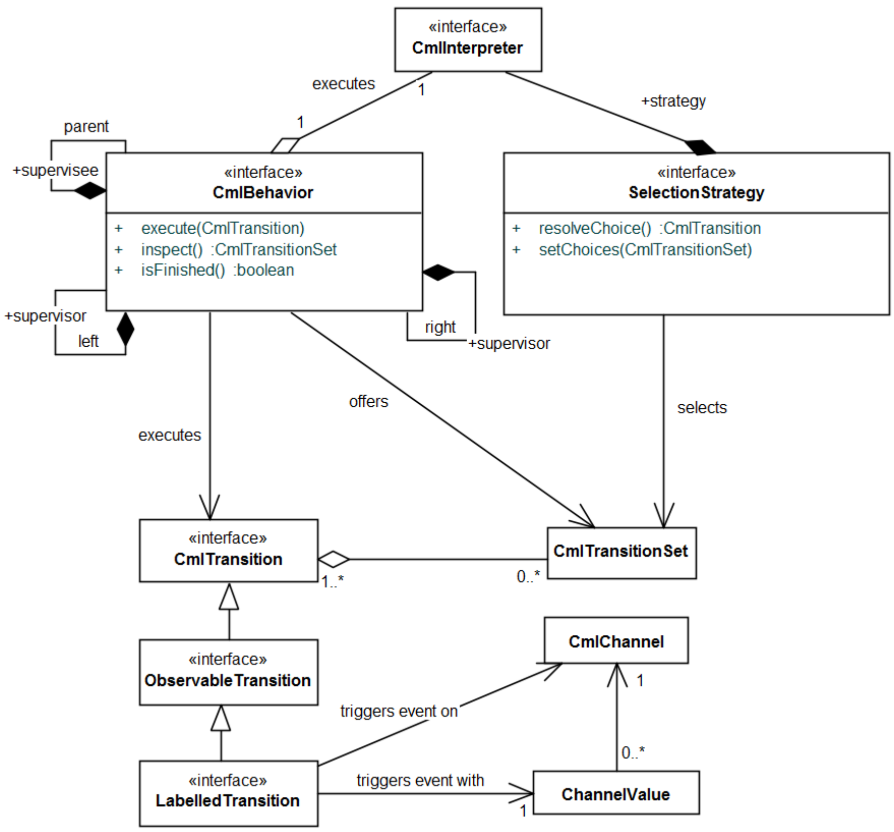

Figure 7: The high level classes and interfaces of the interpreter core component

CmlInterpreter The main interface exposed by the interpreter component. This interface has the overall responsibility of interpreting. It exposes methods to execute, listen on interpreter events and get the current state of the interpreter. It is implemented by the **VanillaCmlInterpreter** class.

CmlBehaviour Interface that represents a behaviour specified by either a CML process or action. It exposes two methods: *inspect* which calculates the immediate set of possible transitions that the current behaviour allows and *execute* which takes one of the possible transitions determined by the supervisor. A specific behaviour can for instance be the prefix action “a -¿ P”, where the only possible transition is to interact in the a event. in any

CmlSupervisorEnvironment Interface with the responsibility of acting as the supervisor environment for CML processes and actions. A supervisor environment selects and exposes the next transition/event that should occur to its pupils (All the CmlBehaviors under its supervision). It also resolves possible backtracking issues which may occur in the internal choice operator.

SelectionStrategy This interface has the responsibility of choosing an event from a given CmlAlphabet. This responsibility is delegated by the CmlSupervisorEnvironment interface.

CmlTransition Interface that represents any kind of transition that a CmlBehavior can make. This structure will be described in more detail in section **??**.

CmlAlphabet This class is a set of CmlTransitions. It exposes convenient methods for manipulating the set.

To gain a better understanding of figure 7 a few things needs mentioning. First of all any CML model (at least for now) has a top level Process. Because of this, the interpreter need only to interact with the top level CmlBehaviour instance. This explains the one-to-one correspondence between the CmlInterpreter and the CMLBehaviour. However, the behavior of top level CmlBehaviour is determined by the binary tree of CmlBehaviour instances that itself and it’s child behaviours defines. So in effect, the CmlInterpreter controls every transition that any CmlBehaviour makes through the top level behaviour.

3.1.2 Transition Model

As described in the previous section a CML model is represented by a binary tree of CmlBehaviour instances and each of these has a set of possible transitions that they can make. A class diagram of all the classes and interfaces that makes up transitions are shown in figure 8, followed by a description of each of the elements.

A transition taken by a CmlBehavior is represented by a CMLTransition. This represent a possible next step in the model which can be either observable or silent (also called a tau transition).

An observable transition represents either that time passes or that a communication/synchronization event takes place on a given channel. All of these transitions are captured in the ObservableTransition interface. A silent transitions is captured by the TauTransition and HiddenTransition class and can respectively marks the occurrence of a an internal transition of a behavior or a hidden channel transition.

CmlTransition Represents any possible transition.

CmlTransitionSet Represents a set of CmlTransition objects.

ObservableTransition This represents any observable transition.

LabelledTransition This represents any transition that results in a observable channel event

TimedTransition This represents a tock event marking the passage of a time unit.

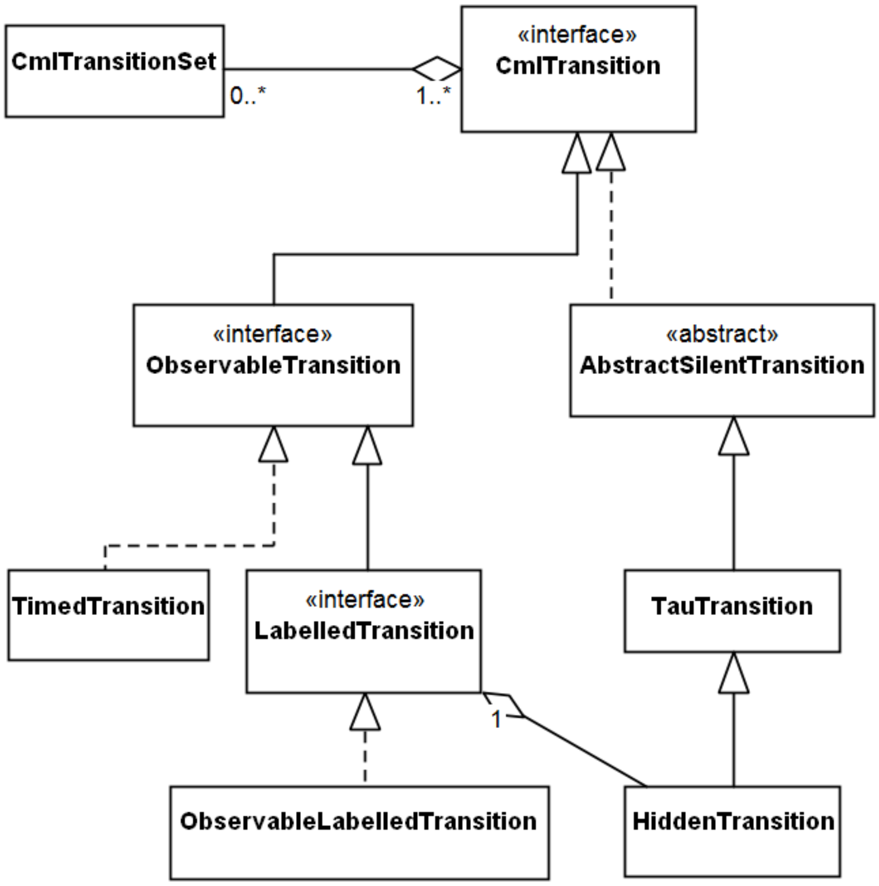

Figure 8: The classes and interfaces that defines transitions/events

ObservableLabelledTransition This represents the occurrence of a observable channel event which can be either a communication event or a synchronization event.

TauTransition This represents any non-observable transitions that can be taken in a behavior.

HiddenEvent This represents the occurrence of a hidden channel event in the form of a tau transition.

3.1.3 Action/Process Structure

Actions and processes are both represented by the CmlBehaviour interface. A class diagram of the important classes that implements this interface is shown in figure 9

As shown the **ConcreteCmlBehavior** is the implementing class of the CmlBehavior interface. However, it delegates a large part of its responsibility to other classes. The actual behavior of a ConcreteCmlBehavior instance is decided by its current instance

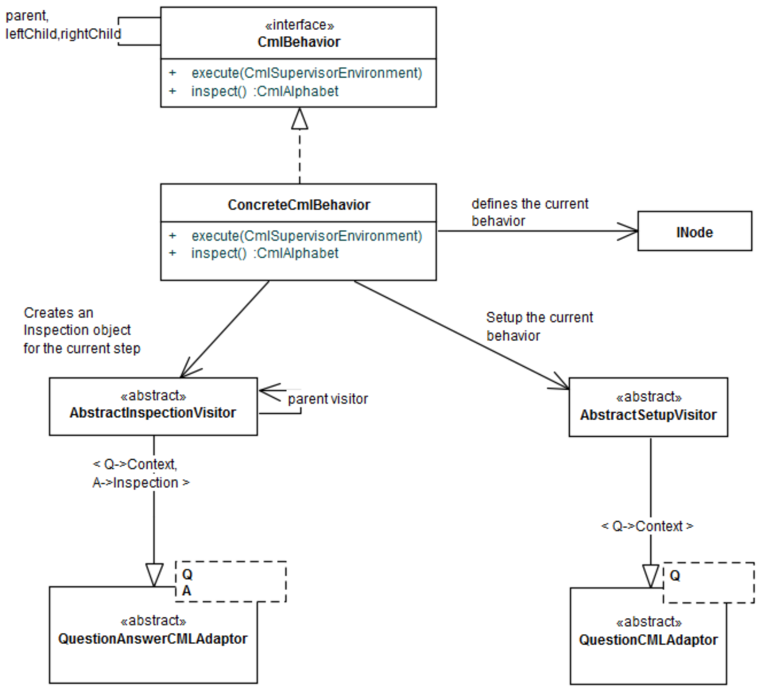

Figure 9: The implementing classes of the CmlBehavior interface

of the INode interface, so when a ConcreteCmlBehavior instance is created a INode instance must be given. The INode interface is implemented by all the CML AST nodes and can therefore be any CML process or action. The actual implementation of the behavior of any process/action is delegated to three different kinds of visitors all extending a generated abstract visitor that have the infrastructure to visit any CML AST node.

The following three visitors are used:

AbstractSetupVisitor This has the responsibility of performing any required setup for every behavior. This visitor is invoked whenever a new INode instance is loaded.

AbstractEvaluationVisitor This has the responsibility of performing the actual behavior and is invoked inside the **execute** method. This involves taking one of the possible transitions.

AbstractAlphabetVisitor This has the responsibility of calculating the alphabet of the current behavior and is invoked in the **inspect** method.

In figure 10 a more detailed look at the evaluation visitor structure is given.

As depicted the visitors are split into several visitors that handle different parts of the languages. The sole reason for doing this is to avoid having one large visitor that handles all the cases. At run-time the visitors are setup in a tree structure where the top most visitor is a **CmlEvaluationVisitor** instance which then delegates to either a

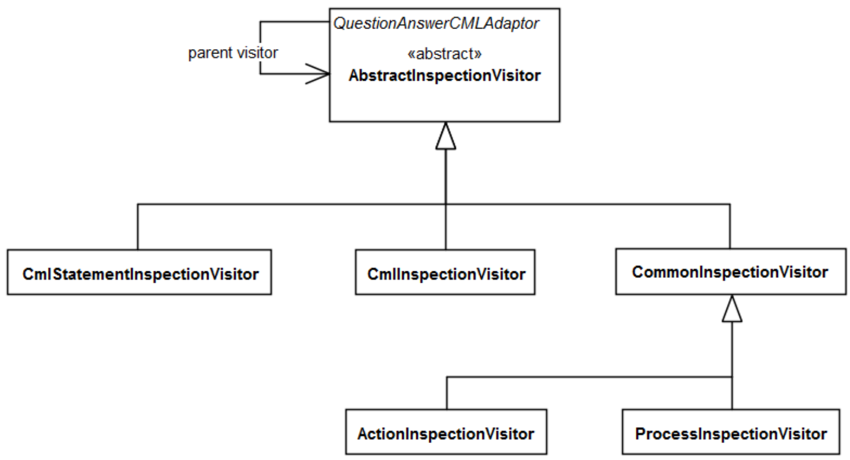

Figure 10: Visitor structure

ActionEvaluationVisitor and **ProcessEvaluationVisitor** etc.

3.1.4 Dynamic Model

The previous section described the high-level static structure, this section will describe the high-level dynamic structure.

First of all, the entire CML interpreter runs in a single thread. This is mainly due to the inherent complexity of concurrent programming. You could argue that since a large part of COMPASS is about modelling complex concurrent systems, we also need a concurrent interpretation of the models. However, the semantics is perfectly implementable in a single thread which makes a multi-threaded interpreter optional. There are of course benefits to a multi-threaded interpreter such as performance, but for matters such as the testing and deterministic behaviour a single threaded interpreter is much easier to handle and comprehend.

To start a simulation/animation of a CML model, you first of all need an instance of the **CmlInterpreter** interface. This is created through the **VanillaInterpreterFactory** by invoking the **newInterpreter** method with a typechecked AST of the CML model. The currently returned implementation is the **VanillaCmlInterpreter** class. Once a **CmlInterpreter** is instantiated the interpretation of the CML model is started by invoking the **execute** method given a **CmlSupervisorEnvironment**.

In figure 11 a high level sequence diagram of the **execute** method on the **VanillaCmlInterpreter** class is depicted.

As seen in the figure the model is executed until the top level process is either successfully terminated or deadlocked. For each

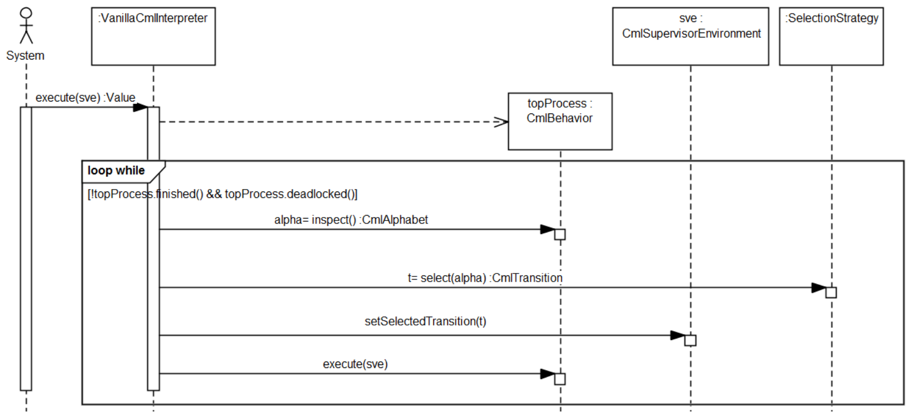

Figure 11: The top level dynamics

3.1.5 CmlBehaviors

As explained in section **??** the CmlBehavior instances forms a binary tree at run-time.

3.2 The IDE Layer