



Grant Agreement: 287829

Comprehensive Modelling for Advanced Systems of Systems

C O M P A S S

## CML Interpreter Design Document

Technical Note Number: DXX

Version: 0.2

Date: Month Year

Public Document

<http://www.compass-research.eu>

<sup>11</sup> **Contributors:**

<sup>12</sup> Anders Kaels Malmos, AU

<sup>13</sup> **Editors:**

<sup>14</sup> Peter Gorm Larsen, AU

<sup>15</sup> **Reviewers:**

## <sup>16</sup> Document History

	Ver	Date	Author	Description
	0.1	25-04-2013	Anders Kaels Malmos	Initial document version
<sup>17</sup>	0.2	06-03-2014	Anders Kaels Malmos	Added introduction and domain description

## 18 **Abstract**

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

## Contents

23	<b>1 Introduction</b>	<b>6</b>
24	1.1 Problem Domain . . . . .	6
25	1.2 Definitions . . . . .	8
26	<b>2 Software Layers</b>	<b>9</b>
27	2.1 The Core Layer . . . . .	9
28	2.2 The IDE Layer . . . . .	10
29	<b>3 Layer design and Implementation</b>	<b>12</b>
30	3.1 Core Layer . . . . .	12
31	3.2 The IDE Layer . . . . .	17

## 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 understanding 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 the interpretation. Therefore, the overall goal of the CML interpreter is to adhere to the semantic rules defined in those documents and to visualize this in the Eclipse Debugger.

In order to get a high level understanding of how CML is interpreted without knowing all the details, a short illustration of how the interpreter represents and evolves 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.

```

52 channels
53   inp : int
54   out : int
55
56 process W =
57   begin
58     @ inp!1 -> Skip
59   end
60
61 process R =
62   begin
63     @ inp?x -> out!x -> Skip
64   end
65
66 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. Processes 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 represent the environment (another actor that invokes the operation e.g. a human user or another process) initiating the phase.

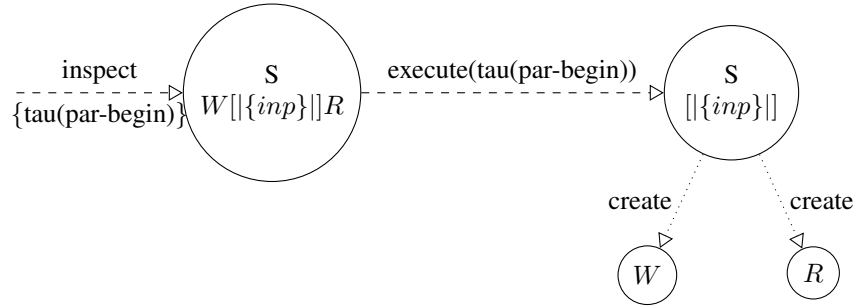


Figure 1: Initial step of Listing 1 with process S as entry point.

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  $\text{tau}(\text{par-begin})$  and is therefore returned from the inspection. The reason for the name  $\text{tau}(\dots)$  is that transitions can be either observable or silent, so in principle any  $\text{tau}$  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  $\text{tau}(\text{par-begin})$  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  $\text{inp}$  channel must be synchronized, meaning that W and R must only perform transition that involves  $\text{inp}$  channel events synchronously.

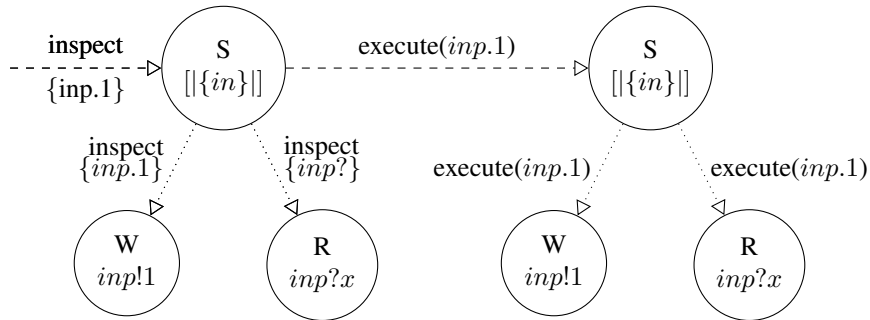


Figure 2: Second step of Listing 1 with S as entry point.

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

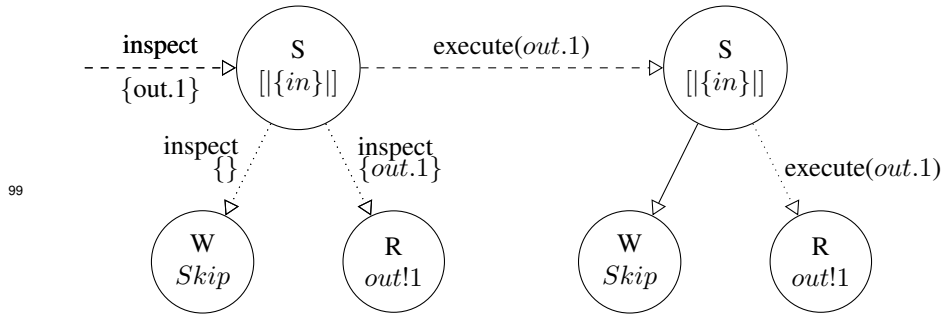


Figure 3: Third step of Listing 1 with S as entry point

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 writing 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.

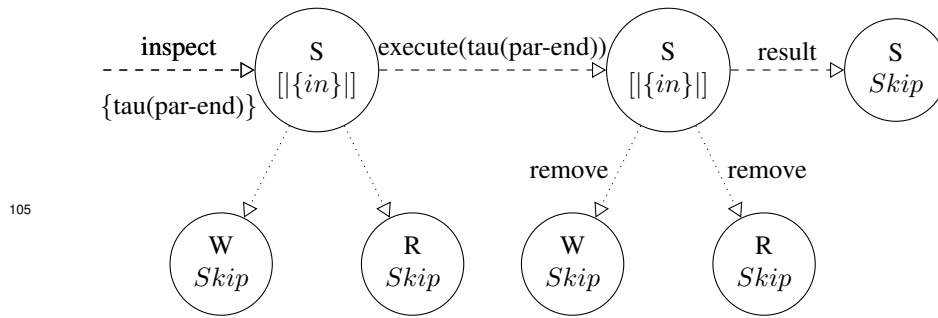


Figure 4: Final step of Listing 1 where the parallel composition collapses unto a Skip process

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

## 1.2 Definitions

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

**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.

**trace** A sequence of observable events performed by a behavior.



## 2 Software Layers

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

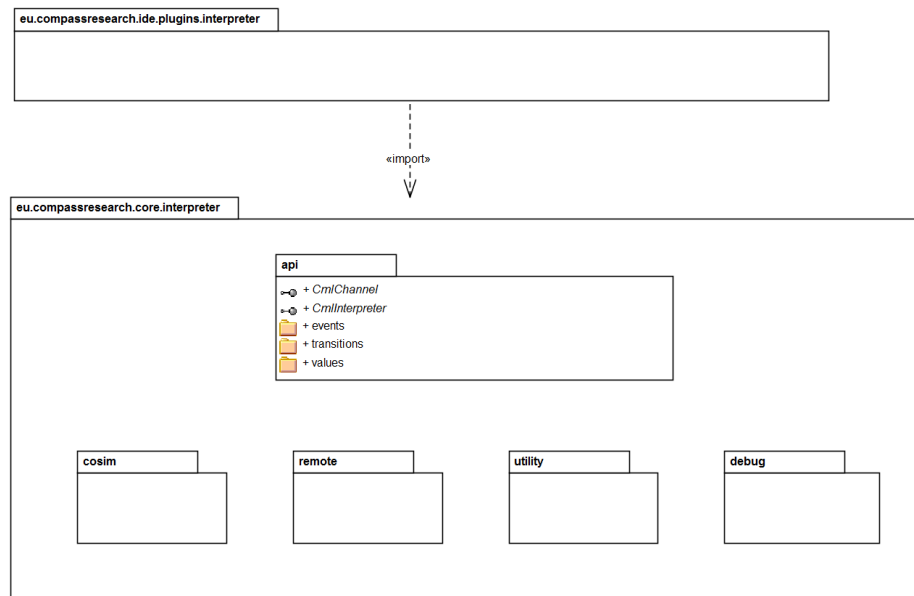


Figure 5: The layers of the CML Interpreter

Each of these components will be described in further detail in the following sections. The major reason behind this layering is that the implementation of the semantics should be independent of the view showing the results.

### 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 package *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 package 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. Furthermore, this pack-

age is split into two separate source folders, each representing a different logical component. The following folders are present:

**src/main/java** This folder contains all public classes and interfaces as described above.

**src/main/behavior** This folder contains all the internal classes and interfaces that the default interpreter implementation is comprised of. This will be described in more details in Subsection 3.1.1.

**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 corresponds 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 events (this is not CML channel events) from both **CmlInterpreter** 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 responsibility of running a co-simulation. A co-simulation can be either between multiple instances of the CML interpreter co-simulating a CML model, or a CML interpreter instance co-simulating a CML model with a real live system.

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

**eu.compassresearch.core.interpreter.debug** Has the responsibility of controlling a debugging sessions, which only includes the Eclipse debugger at this point.

**eu.compassresearch.core.interpreter.utility** The utility packages contains reusable classes and interfaces that are use across packages.

## 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.

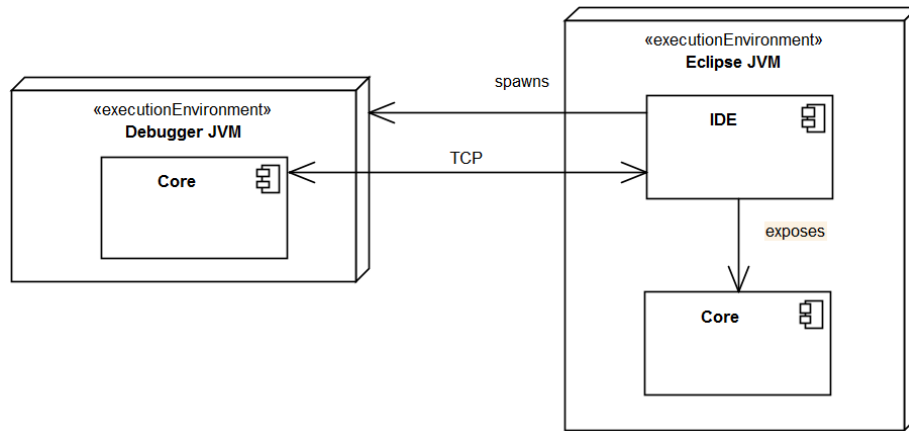


Figure 6: Deployment diagram of the debugger

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

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

187 A debugging session has the following steps:

- 188 1. The user launches a debug session
- 189 2. On the Eclipse JVM a **CmlDebugTarget** instance is created, which listens for  
 190 an incoming TCP connection.
- 191 3. A Debugger JVM is spawned and a **CmlInterpreterController** instance is cre-  
 192 ated.
- 193 4. The **CmlInterpreterController** tries to connect to the created connection.
- 194 5. When the connection is established, the **CmlInterpreterController** instance  
 195 will send a **STARTING** status message along with additional details
- 196 6. The **CmlDebugTarget** updates the GUI accordingly.
- 197 7. When the interpreter is running, status messages will be sent from **CmlInter-  
 198 preterController** and commands and request messages are sent from **CmlDe-  
 199 bugTarget**
- 200 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.

Before going into details with each element on figure 7 a few things needs mentioning. First of all, any CML model 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 along with the selection strategy controls every observable transition that any CmlBehaviour makes.

**CmlInterpreter** The interface exposing the functionality of the interpreter component. This interface has the overall responsibility of interpreting. It exposes methods to inspect and execute and it is implemented by the **VanillaCmlInterpreter** class in the default simulation settings.

**CmlBehavior** Interface that represents a behavior specified by either a CML process or action. Most importantly it exposes the two methods: *inspect* which calculates the immediate set of possible transitions that it currently allows and *execute* which takes one of the possible transitions determined by it's supervisor. This process is described in Subsection 1.1 where a CmlBehavior is represented as a circle in the figures. As seen both in Subsection 1.1 and Figure 7 associations between CmlBehavior instances are structured as a binary tree, where a parent supervises its child behaviors. In this context supervises means that they control the flow of possible transitions and determines when to execute them. The reason for this is that is corresponds nicely to the structure of the CML semantics.

**SelectionStrategy** This interface has the responsibility of choosing a CmlTransition from a given CmlTransitionSet. This could be seen as the last chain in the supervisor hierarchy, since this is where all the possible transitions flows to and the decision of which one to execute next is taken here. The purpose of this interface is to allow different kinds of strategies for choosing the next transition. e.g there is a strategy that picks one at random and another that enables a user to pick.

**CmlTransition** Interface that represents any kind of transition that a CmlBehavior can make. They are not all depicted here and will be described in greater details

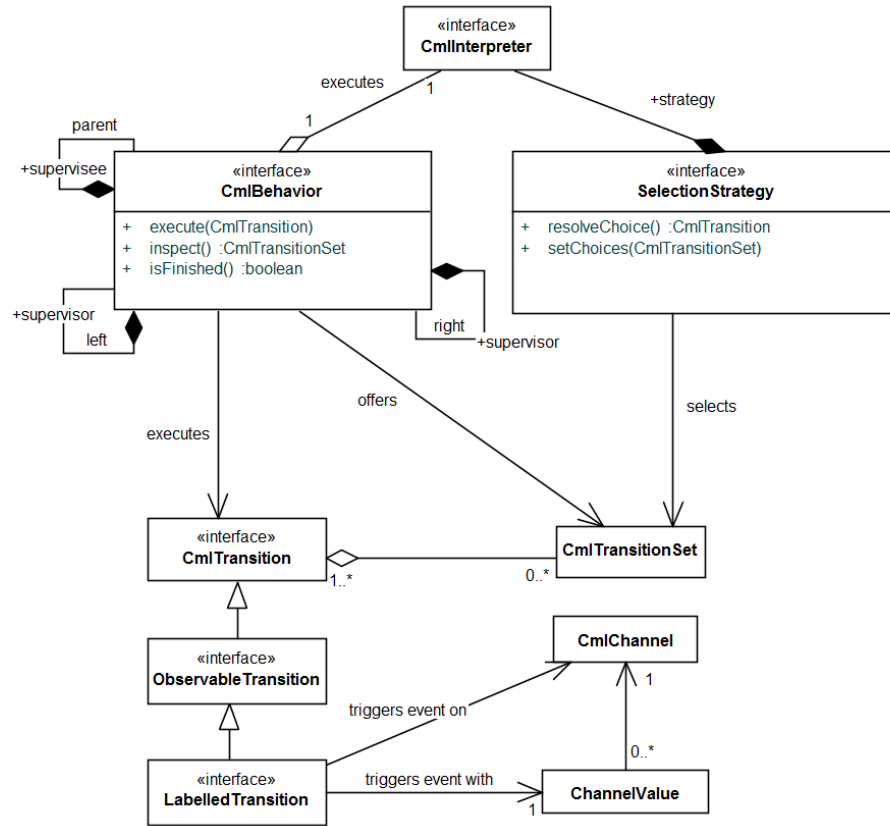


Figure 7: The high level classes and interfaces of the interpreter core component

in ???. But overall, only transitions that implements the **ObservableTransition** interface can produce an observable trace of a behavior.

**CmlTransitionSet** This is an immutable set of **CmlTransition** objects and is the return value of the `inspect` method on a **CmlBehavior**. The reason for it being immutable is to ensure that calculations never change the input sets.

### 3.1.2 Transitions Model

As described in the previous sections 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 ??, 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

255 in the ObservableTransition interface. A silent transitions is captured by the TauTransition and HiddenTransition class and can respectively marks the occurrence of a an  
 256 internal transition of a behavior or a hidden channel transition.  
 257

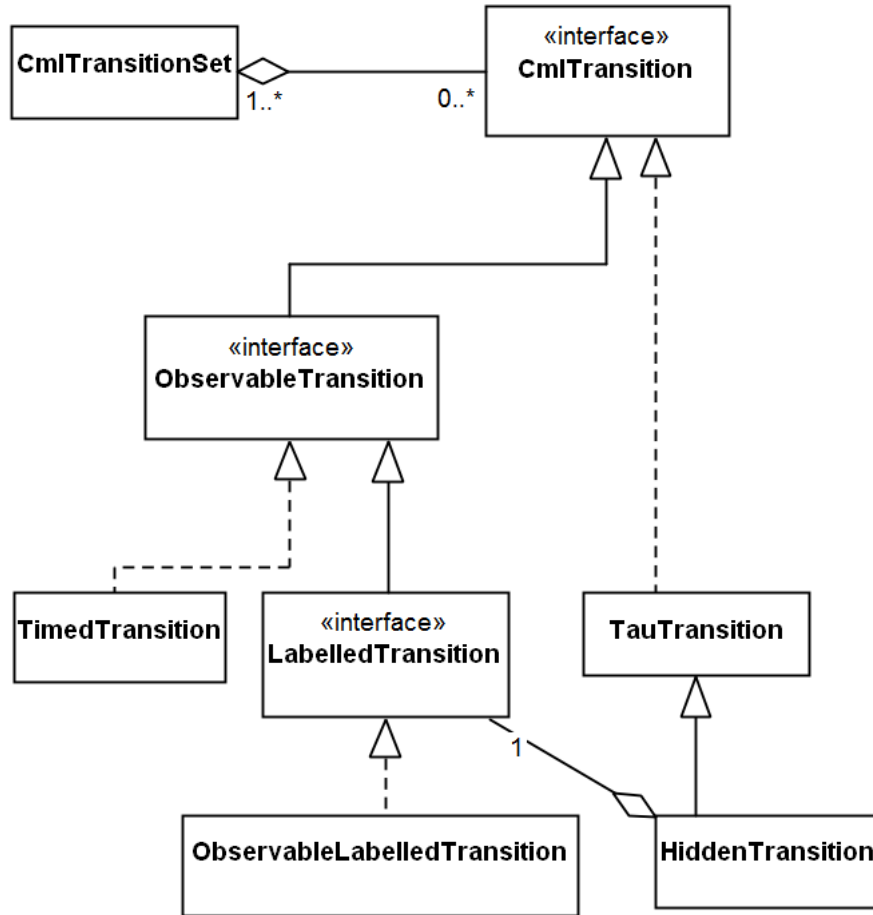


Figure 8: The classes and interfaces that defines transitions

258 **CmlTransition** Represents any possible transition.  
 259 **CmlTransitionSet** Represents a set of CmlTransition objects.  
 260 **ObservableTransition** This represents any observable transition.  
 261 **LabelledTransition** This represents any transition that results in a observable channel  
 262 event  
 263 **TimedTransition** This represents a tock event marking the passage of a time unit.  
 264 **ObservableLabelledTransition** This represents the occurrence of a observable chan-  
 265 nel event which can be either a communication event or a synchronization event.  
 266 **TauTransition** This represents any non-observable transitions that can be taken in a  
 267 behavior.

### 270 3.1.3 The default Implementation of CmlBehavior

```
classDiagram
    class Q_A {
        <<abstract>>
        QuestionAnswerCMLAdaptor
    }
    class Q {
        <<abstract>>
        QuestionCMLAdaptor
    }
    class eu_compass_research_core_interpreter {
        <<package>>
    }
    class CmlExpressionVisitor {
    }
    class AbstractInspectionVisitor {
        <<abstract>>
    }
    class AbstractSetupVisitor {
        <<abstract>>
    }
    class CmlBehavior {
        <<interface>>
        + execute(CmlTransition)
        + inspect() CmlTransitionSet
        + isDeadlocked() boolean
        + isFinished() boolean
    }
    class ConcreteCmlBehavior {
        + execute(CmlTransition)
        + inspect() CmlTransitionSet
    }
    class Inspection {
        - trace : CmlTrace
        - transitionFunction : CmlCalculationStep
        - transitions : CmlTransitionSet
    }
    class CmlCalculationStep {
        <<interface>>
        + execute(CmlTransition) : Pair<Node, Context>
    }
    class Node {
    }

    Q_A <|-- eu_compass_research_core_interpreter
    Q <|-- eu_compass_research_core_interpreter
    eu_compass_research_core_interpreter --> CmlExpressionVisitor : cmlExpressionVisitor
    eu_compass_research_core_interpreter --> AbstractInspectionVisitor : Creates
    eu_compass_research_core_interpreter --> AbstractSetupVisitor : setupVisitor
    CmlExpressionVisitor --> AbstractInspectionVisitor : 1
    AbstractInspectionVisitor --> Inspection : 1
    AbstractSetupVisitor --> ConcreteCmlBehavior : 1
    ConcreteCmlBehavior --> CmlBehavior : 1
    ConcreteCmlBehavior --> CmlCalculationStep : delegates execution
    ConcreteCmlBehavior --> Node : determines current behavior
    Inspection --> CmlCalculationStep : 1
    Node --> ConcreteCmlBehavior : 1
```

15

286 this structure is to be able to utilize the already generated visitors by the AST-creator  
 287 [] that enables traversing of CML AST's.

288 Here a brief description of each new element depicted in Figure 9:

289 **CmlExpressionVisitor** This has the responsibility to evaluate CML expressions given  
 290 a Context.

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

293 **AbstractAlphabetVisitor** This has the responsibility of creating an Inspection object  
 294 given the current state of the behavior, which is represented by a INode and a  
 295 Context object.

296 **Inspection** Contains the next possible transitions (in a CmlTransitionSet) along with  
 297 a transition function in the form of a CmlCalculationStep.

298 **CmlCalculationStep** Responsible for executing the actual behavior that occurs in a  
 299 transition from one state to another. This is where the actual implementation of  
 300 the semantics is.

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

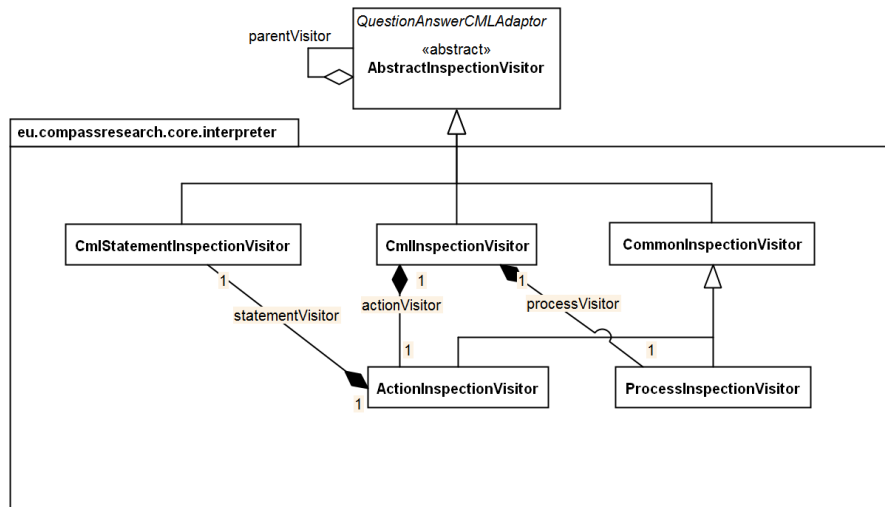


Figure 10: Visitor structure

302 As depicted the visitors are split into several visitors that handle different parts of the  
 303 CML language. The sole reason for doing this is to avoid having one large visitor  
 304 that handles all the cases. At run-time the visitors are setup in a tree structure. For  
 305 the inspection the top most visitor is the CmlInspectionVisitor which then delegates  
 306 to either ActionInspectionVisitor or ProcessEvaluationVisitor depending on the given  
 307 INode. This structures resembles the structure of the setup visitors.

308 The CmlExpressionVisitor is however a little different from the others. It takes care  
 309 of all CML expressions, but delegates the entire subset of VDM expression constructs



that are contained in CML to the DelegateExpressionEvaluator overture class, which can evaluate VDM expressions. The reason for doing this is of course reuse.

### 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.

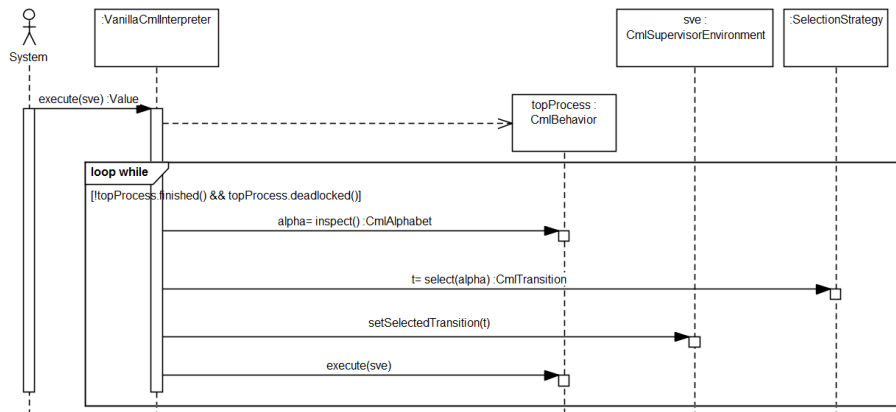


Figure 11: The top level dynamics

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

### 3.1.5 CmlBehaviors

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

336 **3.2 The IDE Layer**