



1
2

Grant Agreement: 287829

3

Comprehensive Modelling for Advanced Systems of Systems

4

C O M P A S S

5

CML Interpreter Design Document

6

Technical Note Number: DXX

7

Version: 0.2

8

Date: Month Year

9

Public Document

10

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

¹¹ **Contributors:**

¹² Anders Kaelz Malmos, AU

¹³ **Editors:**

¹⁴ Peter Gorm Larsen, AU

¹⁵ **Reviewers:**

¹⁶ **Document History**

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

¹⁷

¹⁸ **Abstract**

¹⁹ This document describes the overall design of the CML simulator/animator and pro-
²⁰ vides an overview of the code structure targeting developers.

21 **Contents**

22	1 Introduction	6
23	1.1 Problem Domain	6
24	1.2 Definitions	8
25	2 Software Layers	8
26	2.1 The Core Layer	9
27	2.2 The IDE Layer	10
28	3 Layer design and Implementation	11
29	3.1 Core Layer	11
30	3.2 The IDE Layer	16

³¹ 1 Introduction

³² This document is targeted at developers and describes the overall design of the CML
³³ simulator, it is not a detailed description of each component. This kind of documenta-
³⁴ tion is done in Javadoc and can be generated automatically from the code. It is assumed
³⁵ that common design patterns are known like ??.

³⁶ 1.1 Problem Domain

³⁷ The goal of the interpreter is to enable simulation/animation of a given CML ?? model
³⁸ and be able to visualize this in the Eclipse IDE Debugger. CML has a UTP semantics
³⁹ defined in ?? which dictates how the interpretation progresses. Therefore, the overall
⁴⁰ goal of the CML interpreter is to adhere to the semantic rules defined in those docu-
⁴¹ ments and to somehow visualize this in the Eclipse Debugger.

⁴² In order to get a high level understanding of how CML is interpreted without knowing
⁴³ all the details of the semantics and the implementation of it. A short illustration of how
⁴⁴ the interpreter represents and progresses a CML model is given below.

⁴⁵ In listing 1 a CML model consisting of three CML processes is given. It has a R
⁴⁶ (Reader) process which reads a value from the inp channel and writes it on the out
⁴⁷ channel. The W (Writer) process writes the value 1 to the inp channel and finishes.
⁴⁸ The S (System) process is a parallel composition of these two processes where they
⁴⁹ must synchronize all events on the inp channel.

```

50 channels
51   inp : int
52   out : int
53
54 process W =
55   begin
56     @ inp!1 -> Skip
57   end
58
59 process R =
60   begin
61     @ inp?x -> out!x -> Skip
62   end
63
64 process S = W [ | {inp} | ] R

```

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

⁶⁵ The interpretation of a CML model is done through a series of steps/transitions starting
⁶⁶ from a given entry point. In figure 1 the first step in the interpretation of the model
⁶⁷ is shown, it is assumed that the S process is given as the starting point. Process are
⁶⁸ represented as a circle along with its current position in the model. Each step of the
⁶⁹ execution is split up in two phases, the inspection phase and the execution phase. The
⁷⁰ dashed lines represents the environment (another actor that invokes the operation e.g a
⁷¹ human user or another process) initiating the phase.

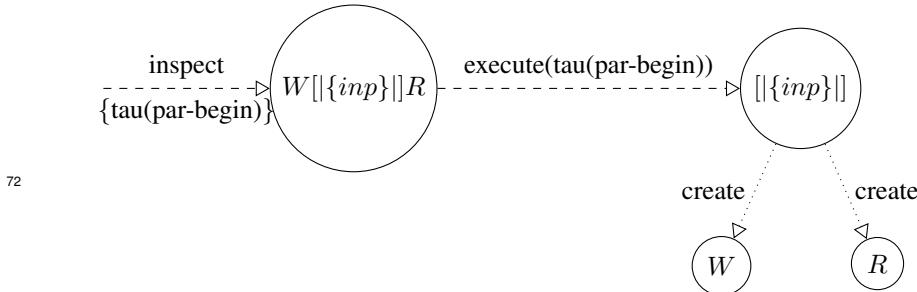


Figure 1: Initial step of Listing 1 with process S as entry point.

73 The inspection phase determines the possible transitions that are available in the next
 74 step of execution. The result of the inspection is shown as a set of transitions below
 75 “inspect”. As seen on figure Figure 1 process P starts out by pointing to the parallel
 76 composition constructs, this construct has a semantic begin rule which does the initial-
 77 ization needed. In the figure that rule is named tau(par-begin) and is therefore returned
 78 from the inspection. The reason for the name tau(..) is that transitions can be ob-
 79 servable or silent, so in principle any tau transition is not seen from the outside of the
 80 process. In the interpreter however all transitions flows out of the inspection. When
 81 the inspection phase has completed, the execution phase begins. The execution phase
 82 executes one of the transitions returned from the inspection phase. In this case, only
 83 a single transition is available so the tau(par-begin)) is executed which creates the two
 84 child processes. The result of each of the shown steps are the first configuration shown
 85 in the next step. So in this case the resulting process configuration is shown in figure
 86 Figure 2.

87 The second step shown in Figure 2 has a little more interesting inspection phase. Ac-
 88 cording to the parallel composition rule for this construct, we have that any event on the
 89 inp channel must be synchronized, meaning that W and R must perform any transition
 90 that involves event on the inp channel synchronously.

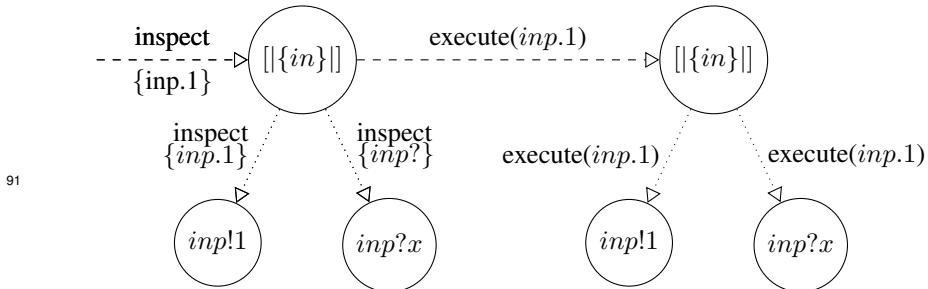


Figure 2: Second step of Listing 1 with S as entry point.

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

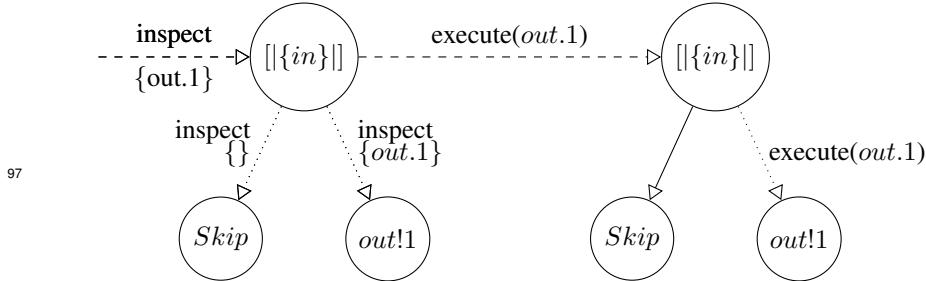


Figure 3: Third step of Listing 1 with S as entry point

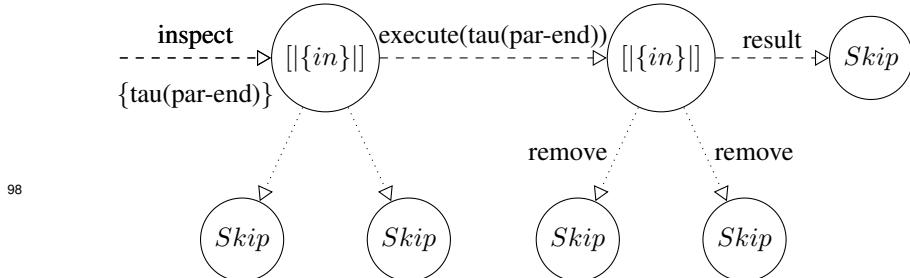


Figure 4: Final step of Listing 1 where the parallel composition collapses onto a Skip process

99 1.2 Definitions

100 **CML** Compass Modelling Language

101 **UTP** Unified Theory of Programming, a semantic framework.

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

104 **Animation** Animation is when the user are involved in taking the decisions when interpreting the CML model
105

106 2 Software Layers

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

109 **Core layer** Has the responsibility of interpreting a CML model as described in the
110 operational semantics that are defined in [?] and is located in the java package
111 named *eu.compassresearch.core.interpreter*

112 **IDE layer** Has the responsibility of visualizing the outputs of a running interpretation
113 a CML model in the Eclipse Debugger. It is located in the *eu.compassresearch.ide.cml.interpreter.plugin*
114 package.

115 Each of these components will be described in further detail in the following sections.
116

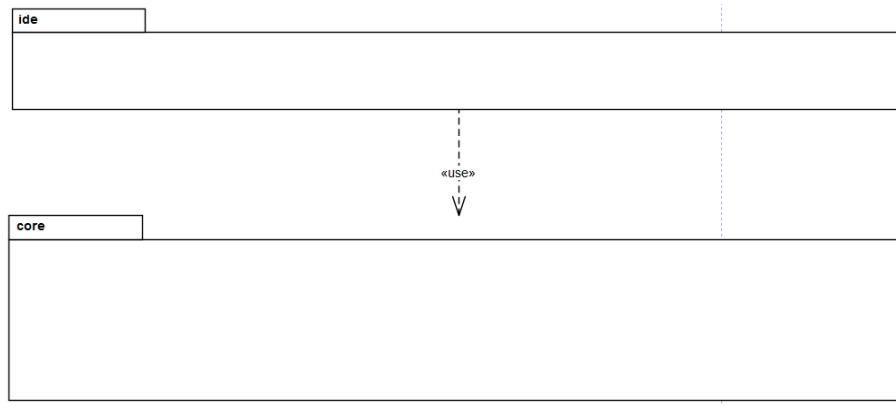


Figure 5: The layers of the CML Interpreter

117 2.1 The Core Layer

118 The design philosophy of the top-level structure is to encapsulate all the classes and
 119 interfaces that makes up the implementation of the core functionality and only expose
 120 those that are needed to utilize the interpreter. This provides a clean separation be-
 121 tween the implementation and interface and makes it clear for both the users, which
 122 not necessarily wants to know about the implementation details, and developers which
 123 parts they need to work with.

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

125 **eu.compassresearch.core.interpreter.api** This package and sub-packages contains all
 126 the public classes and interfaces that defines the API of the interpreter. This
 127 package includes the main interpreter interface **CmlInterpreter** along with ad-
 128 ditional interfaces. The api sub-packages groups the rest of the API classes and
 129 interfaces according to the responsibility they have.

130 **eu.compassresearch.core.interpreter.api.behaviour** This package contains all the com-
 131 ponents that define any CML behavior. A CML behaviour is either an observable
 132 event like a channel synchronization or a internal event like a change of state.
 133 The main interface is **CmlBehaviour**.

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

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

141 **eu.compassresearch.core.interpreter.api.values** This package contains all the values
 142 used in the CML interpreter. Values are used to represent the the result of an
 143 expression or the current state of a variable.

144 **eu.compassresearch.core.interpreter.debug** TBD

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

147 **eu.compassresearch.core.interpreter.utility.events** This package contains components
 148 helps to implement the Observer pattern.

149 **eu.compassresearch.core.interpreter.utility.messaging** This package contains general
 150 components to pass message along a stream.

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

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

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

161 **factories** This folder contains all the factories in the package, both the public **Vanil-**
 162 **laInterpreteFactory** that creates the interpreter and package internal ones.

163 **utility**

164 ...

165 2.2 The IDE Layer

166 The IDE part is integrating the interpreter into Eclipse, enabling CML models to be
 167 debugged/simulated/animated through the Eclipse interface. In Figure 6 a deployment
 168 diagram of the debugging structure is shown.

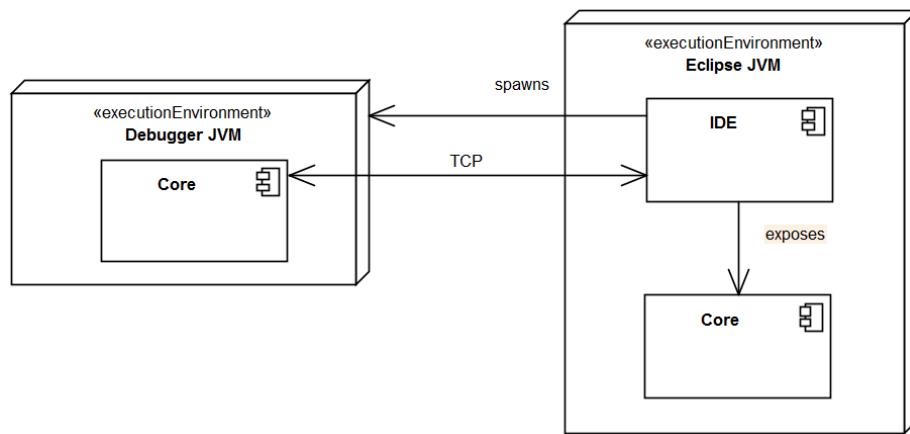


Figure 6: Deployment diagram of the debugger

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

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

174 • **CmlInterpreterController**: This is responsible for controlling the CmlInter-
175 preter execution in the debugger JVM. All communications to and from the in-
176 terpreter handled in this class.

177 • **CmlDebugTarget**: This class is part of the Eclipse debugging model. It has the
178 responsibility of representing a running interpreter on the Eclipse JVM side. All
179 communications to and from the Eclipse debugger are handled in this class.

180 A debugging session has the following steps:

181 1. The user launches a debug session
182 2. On the Eclipse JVM a **CmlDebugTarget** instance is created, which listens for
183 an incoming TCP connection.

184 3. A Debugger JVM is spawned and a **CmlInterpreterController** instance is cre-
185 ated.

186 4. The **CmlInterpreterController** tries to connect to the created connection.

187 5. When the connection is established, the **CmlInterpreterController** instance
188 will send a STARTING status message along with additional details

189 6. The **CmlDebugTarget** updates the GUI accordingly.

190 7. When the interpreter is running, status messages will be sent from **CmlInter-
191 preterController** and commands and request messages are sent from **CmlDe-
192 bugTarget**

193 8. This continues until **CmlInterpreterController** sends the STOPPED message

194 TBD...

195 3 Layer design and Implementation

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

198 3.1 Core Layer

199 3.1.1 Static Model

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

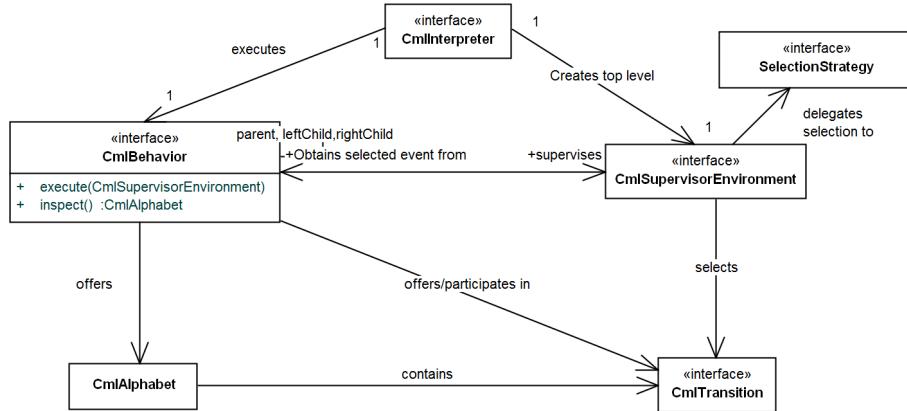


Figure 7: The high level classes and interfaces of the interpreter core component

- 202 **CmIInterpreter** The main interface exposed by the interpreter component. This inter-
203 face has the overall responsibility of interpreting. It exposes methods to execute,
204 listen on interpreter events and get the current state of the interpreter. It is imple-
205 mented by the **VanillaCmIInterpreter** class.
- 206 **CmIBehaviour** Interface that represents a behaviour specified by either a CML pro-
207 cess or action. It exposes two methods: *inspect* which calculates the immediate
208 set of possible transitions that the current behaviour allows and *execute* which
209 takes one of the possible transitions determined by the supervisor. A specific
210 behaviour can for instance be the prefix action “a -;_i P”, where the only possible
211 transition is to interact in the a event. in any
- 212 **CmISupervisorEnvironment** Interface with the responsibility of acting as the super-
213 visor environment for CML processes and actions. A supervisor environment
214 selects and exposes the next transition/event that should occur to its pupils (All
215 the CmlBehaviors under its supervision). It also resolves possible backtracking
216 issues which may occur in the internal choice operator.
- 217 **SelectionStrategy** This interface has the responsibility of choosing an event from a
218 given CmlAlphabet. This responsibility is delegated by the CmlSupervisorEnvi-
219 ronment interface.
- 220 **CmITransition** Interface that represents any kind of transition that a CmlBehavior can
221 make. This structure will be described in more detail in section ??.
- 222 **CmlAlphabet** This class is a set of CmlTransitions. It exposes convenient methods
223 for manipulating the set.
- 224 To gain a better understanding of figure 7 a few things needs mentioning. First of all
225 any CML model (at least for now) has a top level Process. Because of this, the inter-
226 preter need only to interact with the top level CmlBehaviour instance. This explains
227 the one-to-one correspondence between the CmIInterpreter and the CMLBehaviour.
228 However, the behavior of top level CmlBehaviour is determined by the binary tree of
229 CmlBehaviour instances that itself and it's child behaviours defines. So in effect, the
230 CmIInterpreter controls every transition that any CmlBehaviour makes through the top

231 level behaviour.

232 3.1.2 Transition Model

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

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

240 An observable transition represents either that time passes or that a communication/syn-
 241 chronization event takes place on a given channel. All of these transitions are captured
 242 in the ObservableTransition interface. A silent transitions is captured by the TauTrans-
 243 sition and HiddenTransition class and can respectively marks the occurrence of a an
 244 internal transition of a behavior or a hidden channel transition.

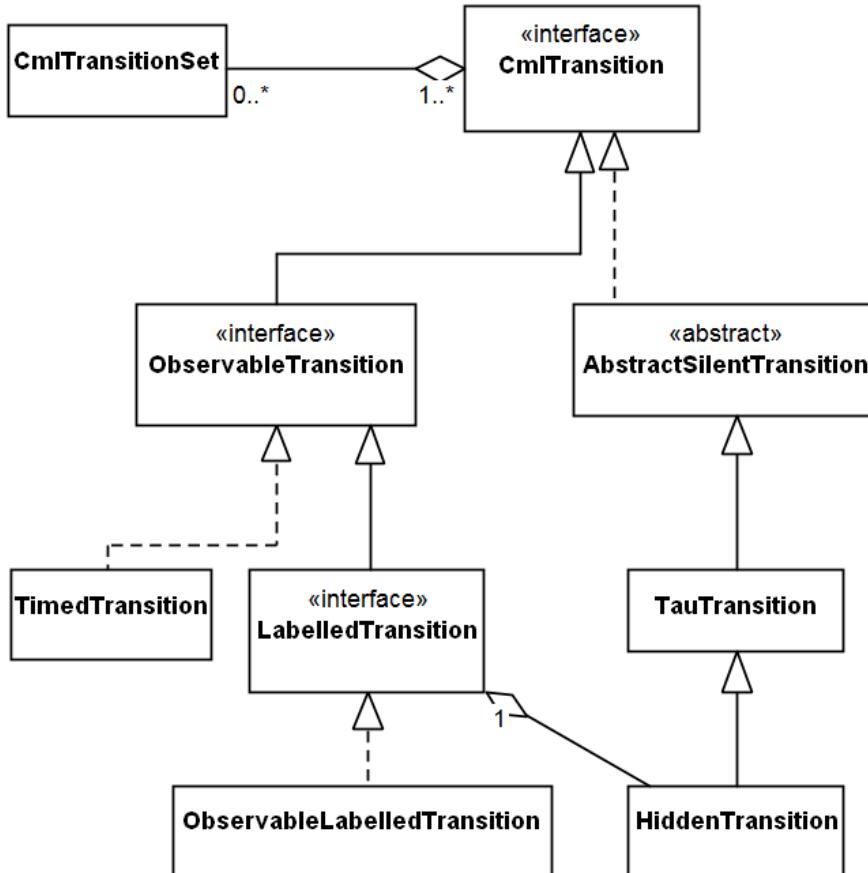


Figure 8: The classes and interfaces that defines transitions/events

- 245 **CmlTransition** Represents any possible transition.
- 246 **CmlTransitionSet** Represents a set of CmlTransition objects.
- 247 **ObservableTransition** This represents any observable transition.
- 248 **LabelledTransition** This represents any transition that results in a observable channel event
- 249 **TimedTransition** This represents a tock event marking the passage of a time unit.
- 250 **ObservableLabelledTransition** This represents the occurrence of a observable channel event which can be either a communication event or a synchronization event.
- 251 **TauTransition** This represents any non-observable transitions that can be taken in a behavior.
- 252 **HiddenEvent** This represents the occurrence of a hidden channel event in the form of a tau transition.

257 **3.1.3 Action/Process Structure**

258 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

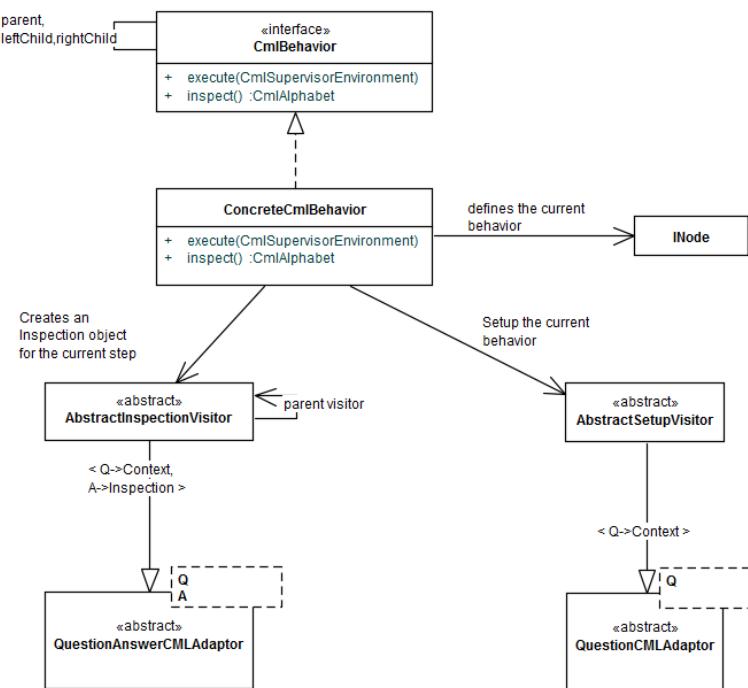


Figure 9: The implementing classes of the CmlBehavior interface

261 As shown the **ConcreteCmlBehavior** is the implementing class of the CmlBehavior
 262 interface. However, it delegates a large part of its responsibility to other classes. The
 263 actual behavior of a ConcreteCmlBehavior instance is decided by its current instance
 264 of the INode interface, so when a ConcreteCmlBehavior instance is created a INode
 265 instance must be given. The INode interface is implemented by all the CML AST
 266 nodes and can therefore be any CML process or action. The actual implementation
 267 of the behavior of any process/action is delegated to three different kinds of visitors
 268 all extending a generated abstract visitor that have the infrastructure to visit any CML
 269 AST node.

270 The following three visitors are used:

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

274 **AbstractEvaluationVisitor** This has the responsibility of performing the actual be-
 275 havior and is invoked inside the **execute** method. This involves taking one of the
 276 possible transitions.

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

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

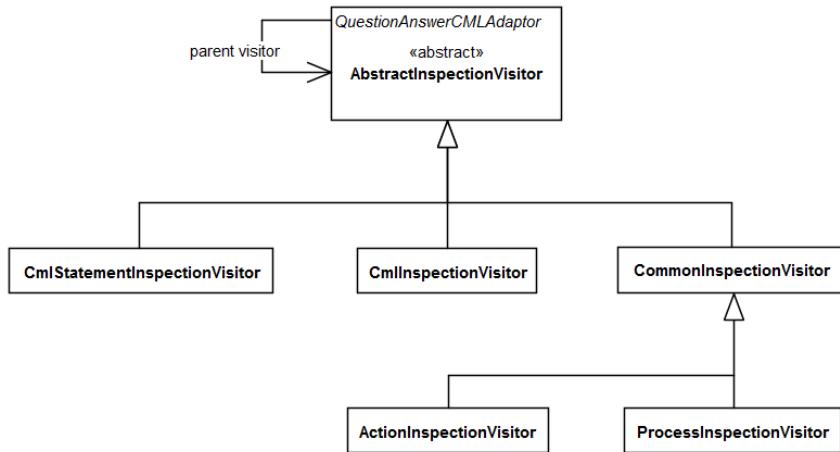


Figure 10: Visitor structure

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

285 **3.1.4 Dynamic Model**

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

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

296 To start a simulation/animation of a CML model, you first of all need an instance of the
297 **CmInterpreter** interface. This is created through the **VanillaInterpreterFactory** by
298 invoking the **newInterpreter** method with a typechecked AST of the CML model. The
299 currently returned implementation is the **VanillaCmInterpreter** class. Once a **Cm-
300 Interpreter** is instantiated the interpretation of the CML model is started by invoking
301 the **execute** method given a **CmlSupervisorEnvironment**.

302 In figure 11 a high level sequence diagram of the **execute** method on the **VanillaCm-
303 Interpreter** class is depicted.

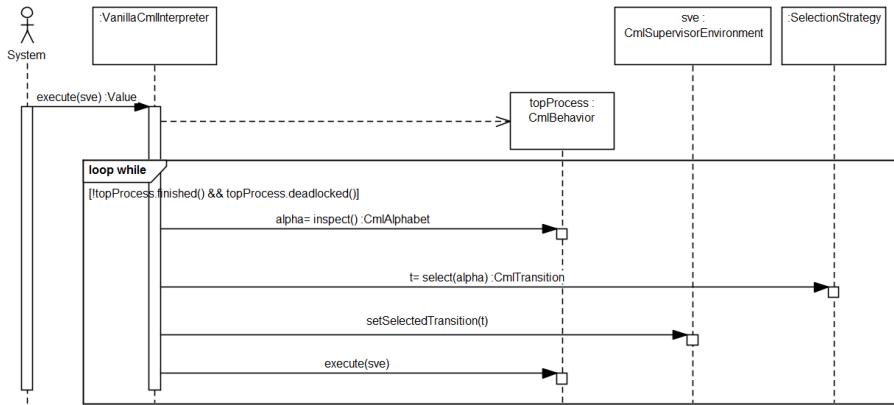


Figure 11: The top level dynamics

304 As seen in the figure the model is executed until the top level process is either success-
305 fully terminated or deadlocked. For each

306 **3.1.5 CmlBehaviors**

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

309 **3.2 The IDE Layer**