Grant Agreement: 287829

Comprehensive Modelling for Advanced Systems of Systems

CML Interpreter Design Document

Technical Note Number: DXX

Version: 0.3

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
0.3	26-03-2014	Anders Kaels Malmos	Added draft of static and dynamic structure for the core interpreter

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 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 evovles 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.

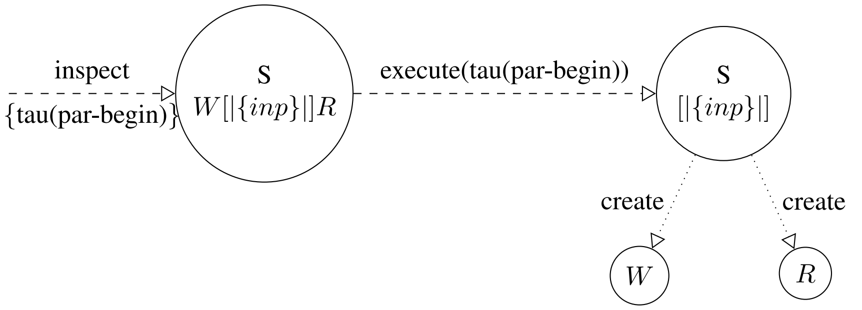

Figure 1: Initial step of Listing 1 with process S as entry point.

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.

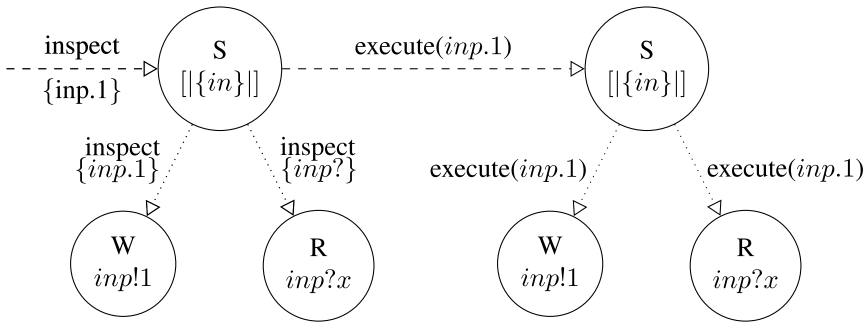

Figure 2: Second step of Listing 1 with S as entry point.

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.

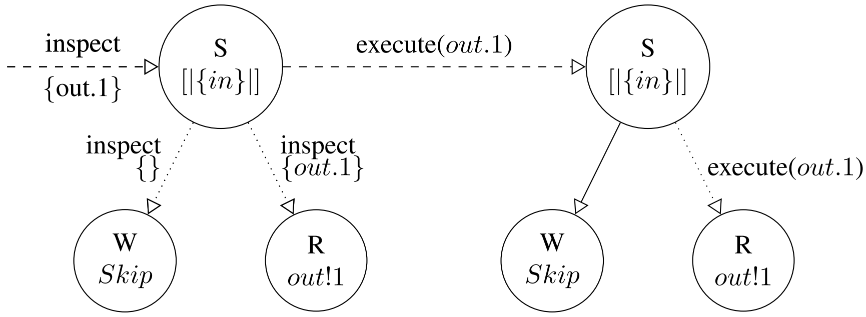

Figure 3: Third step of Listing 1 with S as entry point

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.

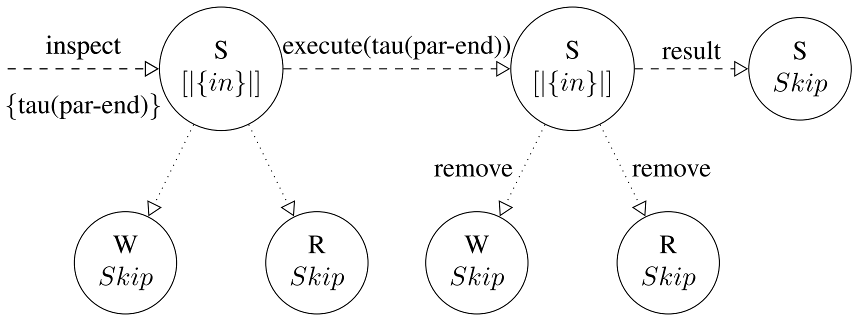

Figure 4: Final step of Listing 1 where the parallel composition collapses unto a Skip process

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

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.

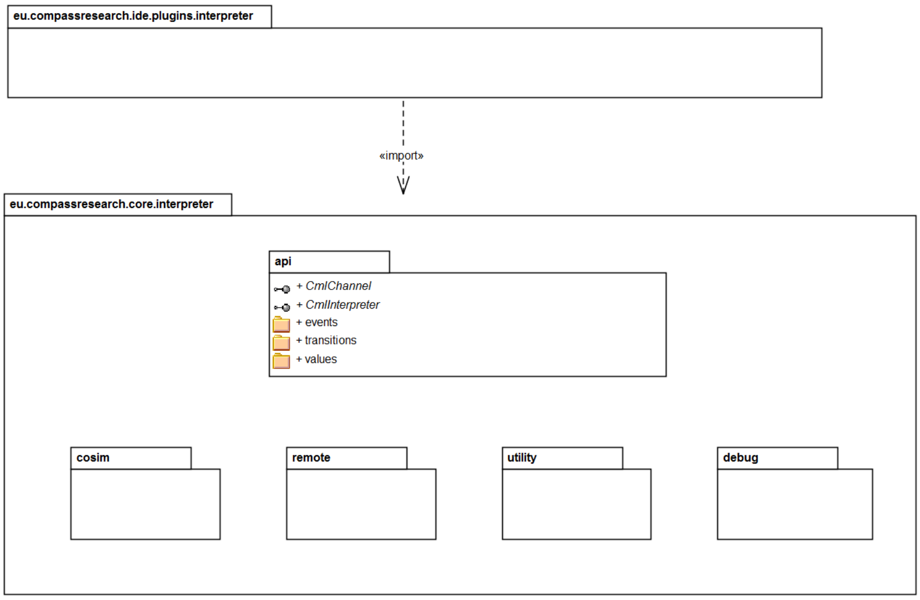

Figure 5: The layers of the CML Interpreter

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.

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*

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 through the Eclipse debugger.

3 Layer design and Implementation

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

3.1 The Core Layer

The core layer is responsible for the overall interpretation of a given CML model. To understand some of the choice made, the design philosophy needs a short word. The design philosophy of the top-level structure is to encapsulate all the classes and interfaces (hence make elements package accessible only when appropriate) 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.

In the following section both the static and dynamic model will be described in more details.

3.1.1 The Static Model

Packages

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. Furthermore, this package is split into two seperate 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 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 3.1.1.

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 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, 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.

The Top Level Elements

The top level interfaces and classes of the interpreter structure is depicted in Figure 6, followed by a short description of each the depicted components.

Before going into detals with each element on figure 6 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 6 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

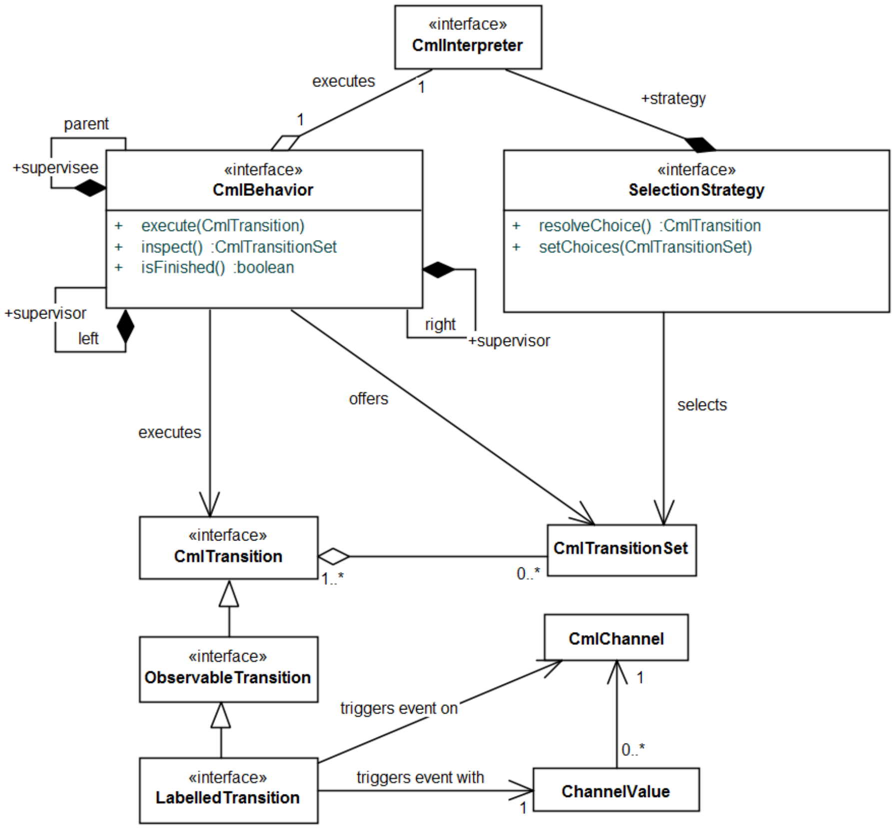

Figure 6: The high level classes and interfaces of the interpreter core component

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 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.

The 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 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.

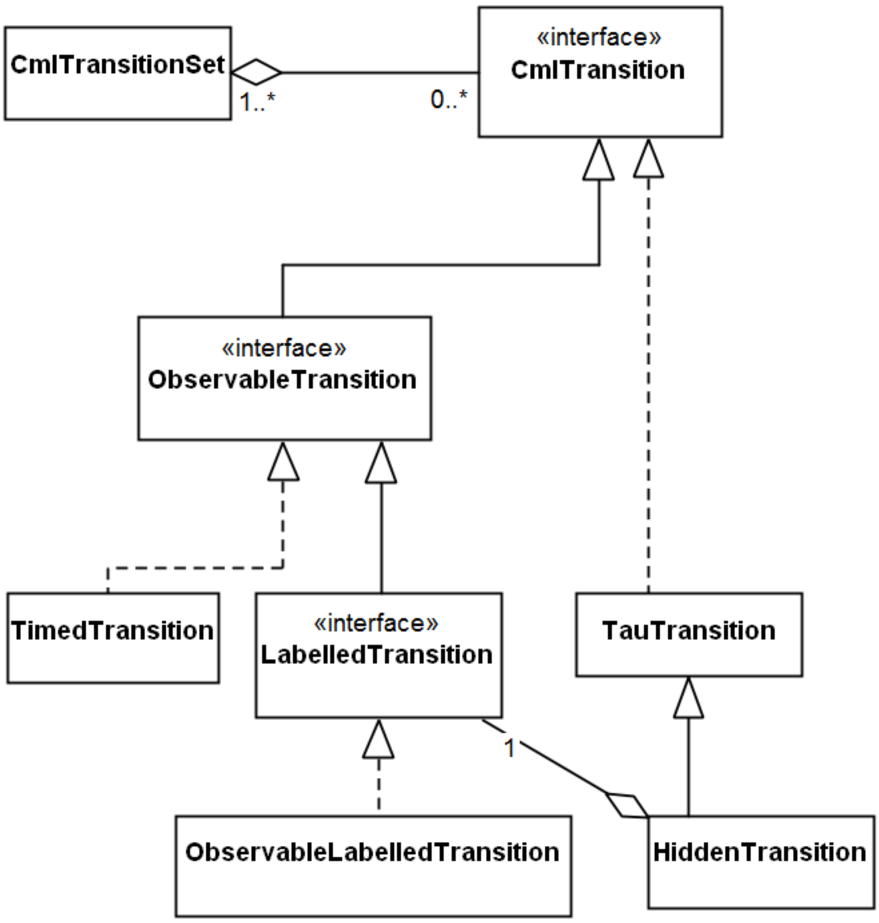

Figure 7: The classes and interfaces that defines transitions

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.

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.

The Default CmlBehavior Implementation

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 8 When

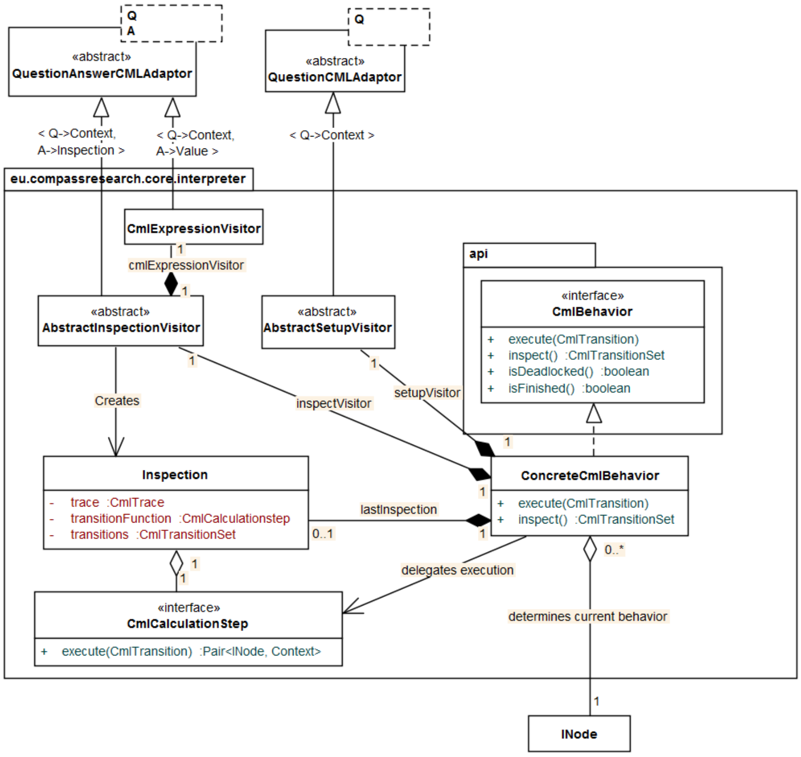

Figure 8: The classes and interfaces making up the default implementation the CmlBehavior interface

the interpreter runs in the default operation mode, meaning where only a single interpreter instance runs (opposed to the co-simulation modes where multiple instances of the interpreter might run or connected to an externally running system). Then all CmlBehavior instances will be in the form of the ConcreteCmlBehavior class. As described above a CmlBehavior has the responsibility to behave as a given action or process. However, as shown in Figure 8 the ConcreteCmlBehavior class delegates a large part of its responsibility to other classes. The actual behavior of a ConcreteCmlBehavior instance is decided by its current INode instance, 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 internal visitor classes as depicted in Figure 8. The used visitors are all extending generated abstract visitors that have the infrastructure to visit any CML AST node. The reason for this structure is to be able to utilize the already generated visitors by the AST-creator [] that enables traversing of CML AST's.

Here a brief description of each new element depicted in Figure 8:

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

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

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

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

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

The Visitors

In figure 9 a more detailed look at the inspection visitor structure is given.

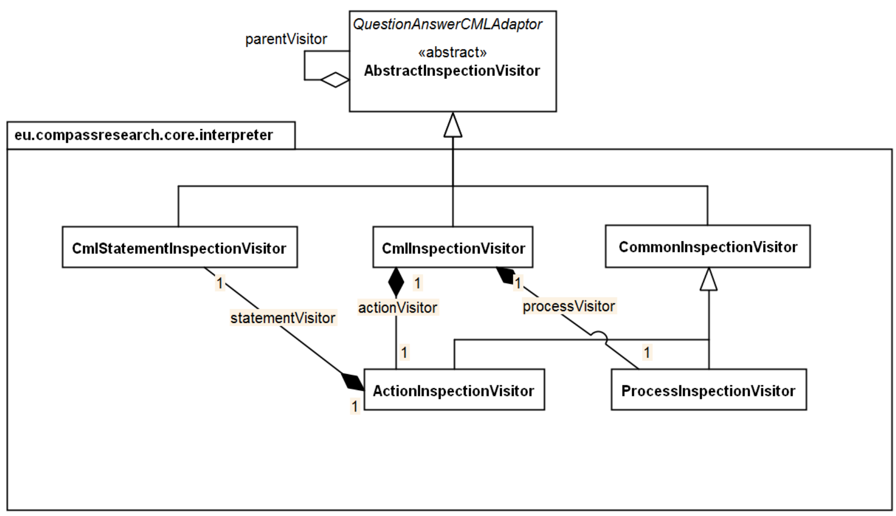

Figure 9: Visitor structure

As depicted the visitors are split into several visitors that handle different parts of the CML language. 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. For the inspection the top most visitor is the CmlInspectionVisitor which then delegates to either ActionInspectionVisitor or ProcessEvaluationVisitor depending on the given INode. This structures resembles the structure of the setup visitors.

The CmlExpressionVisitor is however a little different from the others. It takes care 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.

3.2 The Dynamic Model

This section will describe the high-level dynamic model. First of all, in the default operation mode (as mentioned above a single running instance of the interpreter) 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, but for matters such as the testing and deterministic behaviour a single threaded interpreter is much easier to handle and comprehend.

The Top Execution Loop

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 default returned instance is the VanillaCmlInterpreter class. Once a CmlInterpreter is instantiated the interpretation of the CML model is started by invoking the execute method.

In figure 10 a sequence diagram of the execute method on the VanillaCmlInterpreter class is depicted.

As seen in the figure the execution continues until the top level process is either successfully terminated or deadlocked. Each round taken in this loop is one step taken in the model, where the meaning of a step is explained in Subsection 1.1 with an inspection and execution phase. The actual decision of which transition to be taken next is decided by the given SelectionStrategy instance to the execute method. This decision is delegated to the two methods setChoices and resolveChoice.

Dynamics of the ConcreteCmlBehavior

As mentioned multiple times the ConcreteCmlBehavior class is the default realization of the CmlBehavior interface and is the only one of them explained in details in this report. To understand the dynamic model we need to see what happens in the inspect and execute methods, as these together determines the possible transitions at the top level shown in the last section.

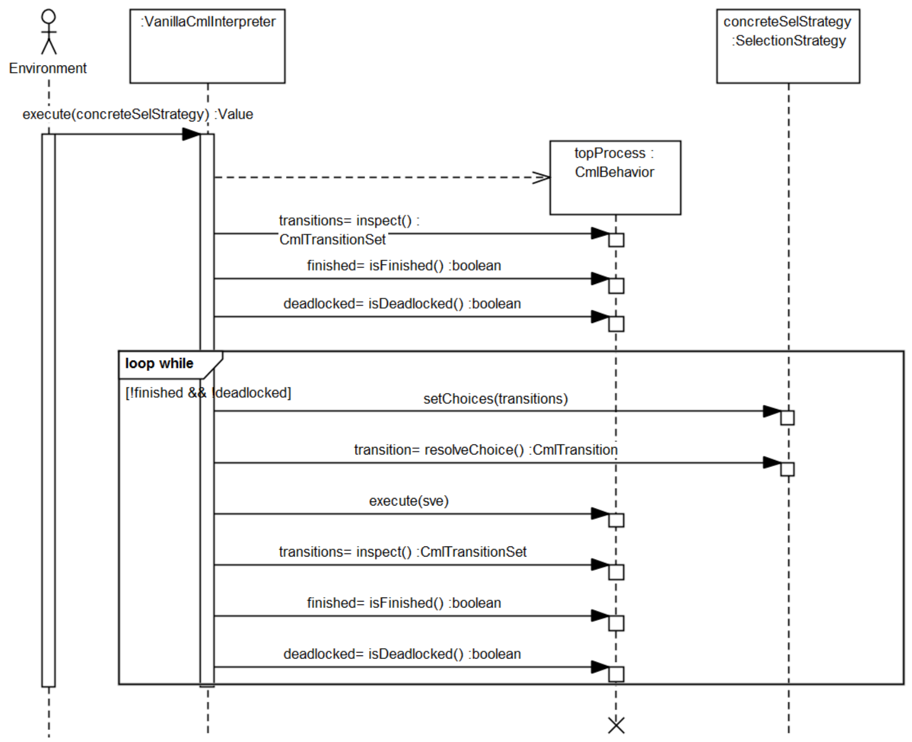

Figure 10: The top level dynamics

In Figure 11 the general inspect dynamics is depicted. When the inspect method is called on a ConcreteCmlBehavior it uses its nextNode (in the java source nextNode and nextContext is actually a Pair¡INode, Context¿) to delegate the actual inspection to the CmlInspectionVisitor. The CmlInspectionVisitor contains a method case¡INode instance name¿ for every CML AST node. So e.g. if nextNode is a AInterleavingAction then the visitor method caseAInterleavingAction(..) method is called with the nextContext. The called case method will return a Inspection which contains the next

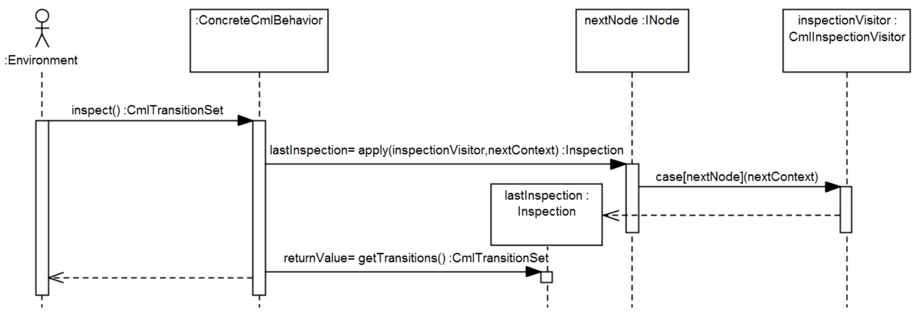

Figure 11: The general dynamics of the inspect method

possible transtions and a transition function to be called if the execute method is to be invoked. The last call in Figure 11 just grabs the CmlTransitionSet from the returned

Inspection object and return this as the result of the inspect call.

The execute method, shown in Figure 12, will execute the given transition and must only be called if one the returned transitions from the inspect method has been chosen for execution. The actual execution is delegated to the CmlCalculationStep instance contained in the last calculated Inspection object (lastInspection in the figure) in the inspect method. The instance of a CmlCalculationStep is an anonymous class created

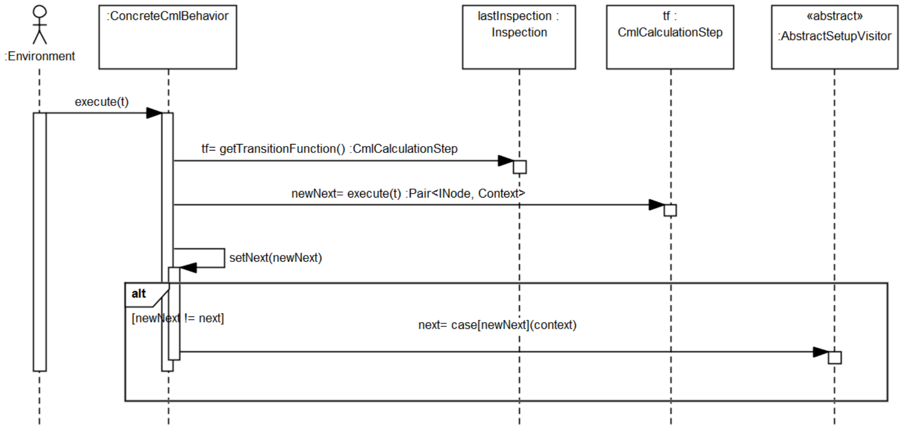

Figure 12: The general dynamics of the execute method

in a inspection visitor case, so the behavior of the execute method is entirely dependent on the current node contained in the next pair. The result of the execute method is the next node and context. As seen in Figure 12 the setup visitor is called if the newly returned pair is different from the current one. This enables any case specific setup behavior to be implemented here. E.g in some cases the context needs to be updated before the inspection phase is commenced.

3.3 The IDE Layer

3.3.1 Deployment Model

In order to get the big picture of how the IDE layer works together with the Core, a deployment view of the IDE is shown in Figure 13.

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

The JSON protocol need to be defined

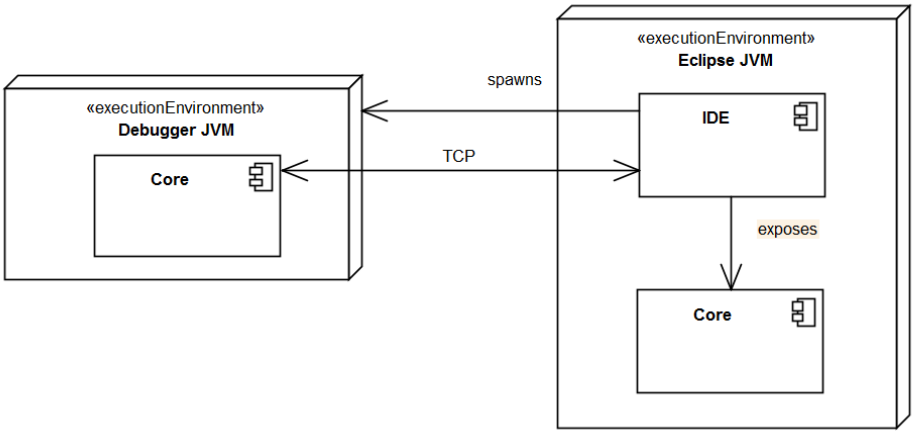

Figure 13: Deployment diagram of the debugger

3.3.2 Static Model

Packages

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

eu.compassresearch.ide.interpreter.model Contains all the classes that implements the Eclipse debug model [?]

eu.compassresearch.ide.interpreter.launching Classes that deals with launching a debugging session.

eu.compassresearch.ide.interpreter.protocol Classes that deals with the communication between the Eclipe CML debugger and a CML interpreter instance.

eu.compassresearch.ide.interpreter.view Contains the custom views of the Eclipse CML debugger.

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

CmlDebugger Interface with the responsibility of controlling the CmlInterpreter execution in a debugging session.

SocketServerDebugger Realization of the CmlDebugger interface that enables controlling the debugging session over a tcp connection.

DebugMain Class that contains the main method that initializes the core component on the debugger JVM side. This involves

CmlDebugTarget This class is part of the Eclipse debugging model. It has the responsiblity of representing a running interpreter on the Eclipse 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 with the main method in the DebugMain class as starting point.
4. A SocketServerDebugger instance is created and tries to connect to the created connection from step 2.
5. When the connection is established, the SocketServerDebugger 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 SocketServerDebugger and commands and request messages are sent from **CmlDebugTarget**
8. This continues until either the CML model successfully terminates or the user stops.