Grant ANR-12-INSE-0011

ANR INS GEMOC

D3.4.1 - Encoding of the Formal model: Composition operators and MoCCs / xDSMLs

Task 3.4

Version 1.1

DOCUMENT CONTROL

	–: 2014/09/28	A: 2014/09/28	B:	C:	D:
Written by Signature	Joel Champeau	Papa Issa Diallo			
Approved by Signature					

Revision index	Modifications
–	version 1.0 — Revised version
A	version 1.1 — Revised with review comments
B	
C	
D	

Authors

Author	Partner	Role
Joel Champeau	LabStic - ENSTA Bretagne	Lead author
Papa Issa Diallo	LabStic - ENSTA Bretagne	Contributor
Ciprian TEODOROV	LabStic - ENSTA Bretagne	Contributor
Julien DeAntoni	I3S / INRIA AOSTE	Contributor

Contents

1. Introduction

1.1 Purpose

The phases of system verification and validation are becoming more integrated into the system design stages to ensure their proper functioning. For instance, the model-checking tools provide features for comprehensive and automated verification of system models. The model-checking techniques can be used to formally verify the requirements on a system model. Such techniques are based on the building of a finite state-space model for the system, on which properties expressed as purely temporal logic formula can be checked.

In the context of the GEMOC project, we want to provide a new feature that will ease the connection to model-checking tools such as [2, 9, 10]. The feature provides the first step towards model-checking by building the exhaustive finite state-space of a system model that integrates MoCC properties. The exhaustive state-space highlights all the possible schedules of a constrained system model (constraints expressed with MoCCML, see D2.2.1). The graph built from the *exhaustive exploration* of all the possible schedules can be used in a model-checking tool to verify behavioral properties of the MoCCML models.

This document presents the integration of the "CLOCKSystem *exhaustive exploration* builder" in the GEMOC STUDIO. The GEMOC STUDIO currently integrates several simulation and model animation tools for executable system models. However, the environment does not yet include a model-checking tool, nor a connector to model-checking tools. CLOCKSystem will provide this connector by building the finite state-space of a system model integrating the MoCC properties. The key ideas behind CLOCKSystem are: a) provide a minimal kernel for experimenting with logical time formalisms; b) offer a flexible and simple language for extending the kernel with user-defined event relations; c) enable the development of new analysis tools for CLOCKSystem specification, like exhaustive reachability analysis.

CLOCKSystem currently provides a connector to the OBP [5] model-checking tool that can be downloaded at [7].

1.2 Perimeter

This document is the version v1 of the D3.4.1 deliverable. Its purpose is to describe the plugins provided to the GEMOC STUDIO to support *exhaustive exploration* of system models (ref. T 3.4 = V&V of the formalized semantics). The document describes the provided $Software$s, their functionalities, but also how they can be integrated and used in the GEMOC STUDIO.

1.3 Definitions, Acronyms and Abbreviations

- **AS**: Abstract Syntax.
- **API**: Application Programming Interface.
- **Behavioral Semantics**: see *Execution semantics*.
- **CCSL**: Clock-Constraint Specification Language.
- **CS**: Concrete Syntax.
- **Domain Engineer**: user of the Modeling Workbench.
- **DSA**: Domain-Specific Action.
- **DSE**: Domain-Specific Event.

- **DSL**: Domain-Specific Language.
- **DSML**: Domain-Specific (Modeling) Language.
- **Dynamic Semantics**: see *Execution semantics*.
- **Eclipse Plugin**: an Eclipse plugin is a Java project with associated metadata that can be bundled and deployed as a contribution to an Eclipse-based IDE.
- **ED**: Execution Data (part of DSA).
- **EF**: Execution Function (part of DSA).
- **Execution Semantics**: Defines when and how elements of a language will produce a model behavior.
- **GEMOC Studio**: Eclipse-based studio integrating both a language workbench and the corresponding modeling workbenches.
- **GUI**: Graphical User Interface.
- **Language Workbench**: a language workbench offers the facilities for designing and implementing modeling languages.
- **Language Designer**: a language designer is the user of the language workbench.
- **MoCC**: Model of Concurrency and Communication.
- **Model**: model which contributes to the convent of a View.
- **Modeling Workbench**: a modeling workbench offers all the required facilities for editing and animating domain specific models according to a given modeling language.
- **MSA**: Model-Specific Action.
- **MSE**: Model-Specific Event.
- **RTD**: RunTime Data.
- **Static semantics**: Constraints on a model that cannot be expressed in the metamodel. For example, static semantics can be expressed as OCL invariants.
- **Operational semantics**: Constraints on a model that defines its step-by-step execution semantics (see **Execution Semantics**).
- **TESL**: Tagged Events Specification Language.
- **xDSML**: Executable Domain-Specific Modeling Language.
- **MoCCML**: MoCC Modeling Language.

1.4 Summary

CLOCKSystem will provide exhaustive system model exploration highlighting all possible schedules, which is the first step to perform properties verification and validation of executable models taking into account the MoCC properties, and the DSA Execution Functions (*EF*) and Execution Data (*ED*). The chapter 2 presents a motivating example showing *exhaustive exploration* results from a case study implemented in the GEMOC STUDIO; the chapter 3 presents the CLOCKSystem tool, its functionalities; the chapter 4 presents the integration strategy of CLOCKSystem and the implemented artifacts in the plugins; Finally the chapter 5 presents a quick tutorial to use this integration of CLOCKSystem[1].

[1] An extra chapter A briefly presents OBP.

2. Motivating example on the *SigPML* language

The objective of this section is to provide an overview of how exhaustive exploration can be built based on system models that conform to a DSL constrained with MoCCML in the GEMOC STUDIO. The considered DSL is a agent-based language called *SigPML*. The resulting exhaustive exploration shows all the acceptable schedules of a *SigPML* model taking into account the MoCC definitions.

2.1 *SigPML* overview

SigPML is a simple data flow language in which the application can be deployed on an hardware platform. The application behavioral semantics is inspired by the SDF (Synchronous Data Flow [11]).

In the *SigPML* syntax, an application is described as a set of *Agent*s. Upon activation, each agent executes N processing cycles and uses the data on its *Input Port*s to produce computed results on its *Output Port*s. Data in transition between *Agent*s are stored in *Place*s. The target architecture is described as a set of interconnected resources defined using 3 concepts: *Computing Resource*s, *Storage Resource*s and *Communication Resource*s.

We use MoCCML to show how to explicitly define the MoCC of the language to automatically obtain the execution model of a given *SigPML* program; and to show how to obtain the exhaustive exploration for the execution model.

2.2 The *SigPML* MoCC

Before explicitly introducing the MoCC constraints, the set of relevant *SigPML* events have to be identified. In the context of an *Agent* the relevant events are: *start*, *stop* and *isExecuting* (see Listing 2.1). Moreover, the ports have also the *read* (input port) and *write* (output ports) events.

The events are part of the mapping since they are defined in the *context* of a concept of the DSL and are used as parameters by MoCCML constraints (see line 7 listing 2.1).

```
1 context Agent
2   def : start : Event
3   def : stop : Event
4   def : isExecuting : Event
5 context Place
6  inv PlaceLimitation:
7  Relation PlaceConstraint(self.outputPort.write,self.inputPort.read, self.outputPort.rate, self.
       inputPort.rate, self.delay, self.capacity)
```

Listing 2.1: Part of the event and constraint mapping in the context of the *SigPML* concepts

With such a mapping, for a specific model, any instance of the meta class *Agent* is associated to the three events. These events have to be constrained to provide the adequate semantics. For instance in line 7 of Listing 2.1 the events are used in a constraint (i.e. *PlaceConstraint*) defined in the context of a *Place*. The *PlaceConstraint* automata is shown on Figure 2.1. It defines a constraint between the *read* of an input port and the *write* event of an output port linked by a place. This automata imposes that *read* does not occur if there is not enough data in the place and *write* does not occur if there is not enough room in the place. This constraint is instantiated for each instance of *Place* in a *SigPML* model.

The constraint automata defined in the context of an Agent implements the following behavior: *1) read* is simultaneous to *start*, *2) isExecuting* occurs only between *start* and *stop 3) stop* occurs at the N^{th} occurrences of *isExecuting* that follows *start*, and *4) stop* is simultaneous to a *write*. In the case where N equals 0 (*i.e.,*the SDF abstraction), then the *read*, the *start*, the *stop* and the *write* are simultaneous. However, N can be different from 0 meaning that an execution time can be specified between the *start* and *stop*, for example according to a deployment.

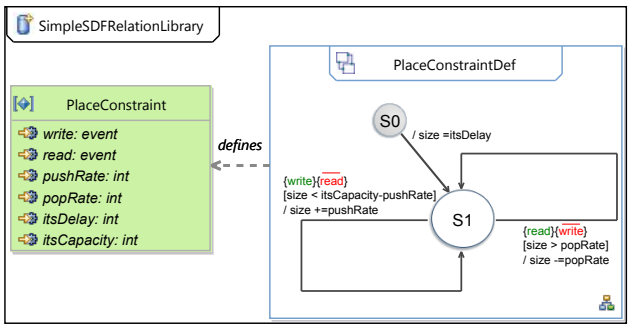

Figure 2.1: The *PlaceConstraint* automata

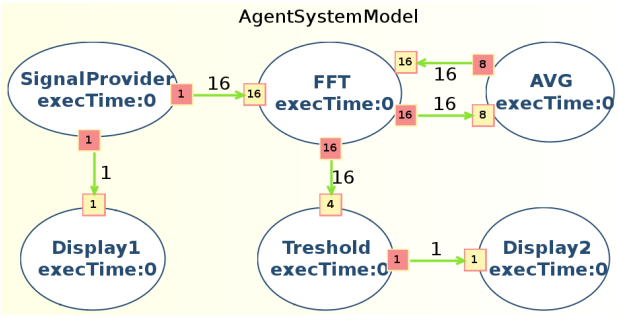

Figure 2.2: PAM Use Case Conceptual Model.

These two constraint automata reproduce the SDF semantics but explicitly defined on the *SigPML* concepts. The reader should note that these automata could be modified to provide variants of the semantics.

2.3 A *SigPML* model

A Passive Acoustic Monitoring (PAM) system is modeled by using the *SigPML* language previously defined. The goal of this section is to present how the MoCC of *SigPML* can be used for analysis of the scheduling state-space.

The PAM system is composed of six agents (Signal Provider, FFT, AVG, Treshold, Display1, Display2), as illustrated in Figure 2.2. When activated, the Signal Provider produces 1 data; the FFT consumes data by 16 and transforms them into a time-frequency representation processed by AVG and Treshold to overcome the variations of the data on a long time interval. AVG consumes the data by 8 and performs an average "high-pass filter" with a feedback effect to the FFT. Finally Treshold processes data by 4. In this first version, no hardware platform deployment is defined which means that infinite resource possibilities are assumed.

2.4 *exhaustive exploration* results

The PAM system, described above conforms to *SigPML*, hence its execution model is automatically generated from the MoCC and its mapping. Thus, the execution model can be used by a generic execution engine either to simulate the system or to explore exhaustively all acceptable schedules. The exploration of all schedules can be captured explicitly in a state space graph as presented in Figure 2.3. Any cyclic path in this graph (starting from the initial configuration) represents a valid schedule of the model. In Figure 2.3 there is an initialization phase before entering the periodic schedules. This initialization phase changes according to the initial number of data in the *Place*s (*i.e.,*according to $itsDelay$ in Figure 2.1). Figure 2.3 shows a graphical representation of the set of available schedules of the PAM application, under an infinite resource assumption (referred to as PAM_0 latter on). The black line in the figure emphasizes the schedule that is obtained under ASAP simulation policy.

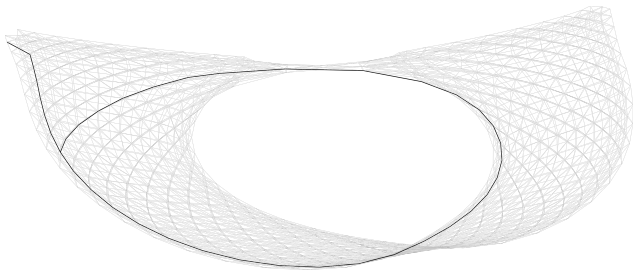

Figure 2.3: State space representation of PAM_0, encoding the set of valid schedules. Emphasis on the schedule obtained under ASAP policy (black line).

These results highlight the benefits of generating an execution model that configures a generic execution engine: simulation and analysis are obtained for free. This is quite different from existing approaches that usually hide (a part of) the semantics in a dedicated framework. Additionally, in our approach the execution model is an explicit entity that can be manipulated for reasoning. Such manipulations may include:

- the verification of temporal logic properties (safety and liveness) [13], on the state space graph structure;
- the mining of the graph to extract a schedule that optimizes specific objectives (like the extraction of minimal buffer requirements done in [8], but in our case, directly applied at the DSL level, without requiring the transformation towards another formalism);
- the extraction of the system properties by static analysis of an event-graph representation of the execution model, such as in [12].

The above *exhaustive exploration* was realized using CLOCKSystem. Our purpose is to provide CLOCKSystem as a plugin for use in the GEMOC STUDIO in order to build exhaustive state-space graph that can be used for properties verification in model-checking tools such as OBP [5]. The next chapters describe the CLOCKSystem tool and the different implemented plugins to be used in the GEMOC STUDIO.

3. CLOCKSystem Description and Functionalities

This chapter describes CLOCKSystem and its functionalities. The chapter is an excerpt of the paper [4] presenting the CLOCKSystem tool.

3.1 CLOCKSystem Description

CLOCKSystem is a meta-described clock-constraint engine, which embeds a formal model of logical time. It relies on the primitives provided by Clock Constraint Specification Language (CCSL) defining a simple yet powerful toolkit for logical time specifications. It also extends the CCSL language, through an automata-based approach, with domain-specific user-defined operators and provides an embedded DSL for writing executable specification in a language close to the abstract CCSL notation.

3.2 CLOCKSystem Functionalities

CLOCKSystem offers several functionalities that range from exhaustive state-space exploration to testing and monitoring.

3.2.1 Cyclic Trace Interpretation

The CLOCKSystem simulator implements a trace-based simulator. While executing a given specification, it constantly verifies the existence of loops back to an already seen system state, in which case it can either stop the simulation reporting an infinite trace (infinite due to the possibility to loop-back an arbitrary number of times) or it can continue, maybe choosing a different path.

3.2.2 Exhaustive Reachability Analysis and Model-Checking

The CLOCKSystem toolkit provides the possibility to perform exhaustive reachability analysis of relation specifications (e.g. MoCCML or CCSL specifications). The possibility to exhaustively explore the state-space of a given specification paves the way to verification by model-checking [6]. The scope of this verification can either be the MoCCML/CCSL specifications alone or their composition with the time-constrained-system, which however has to be expressed in a formal language (e.g. Fiacre [1]).

Relying on the exhaustive reachability results, an interface with the OBP model-checking toolkit [4] was developed. This approach enables the verification of safety and bounded liveness property on a system model conforming to a DSL and constrained using CLOCKSystem specifications.

3.2.3 Design-space Exploration

An important aspect during system design is creating a feedback-loop between a given system model and the analysis results. Conceptually simple, this process, known also as design-space exploration, states that the analysis results should be taken into account to improve the model.

Design-Space Exploration, is enabled by the complementarity between the MoCCML/ CCSL formalism, enabling the simple encoding of specifications, and the imperative nature of the CLOCKSystem implementation language, which offers the power of a imperative general-purpose programming language for "scripting" different analyses phases searching throughout the solution-space.

3.2.4 Testing and Monitoring

Testing and Monitoring are other usage scenarios in which the CLOCKSystem specifications are simply seen as the compact encoding of a test case which can, also, be deployed in production and which follows the system execution (through events) constantly checking the coherence between the specification and the real execution observed.

3.3 CLOCKSystem Specifications

The CLOCKSystem language is an extension of the CCSL domain-specific language (DSL). It represents time relations through the logical time formalism following a high-level domain-agnostic approach.

The CLOCKSystem operational semantics is implemented through an automata-based approach. This approach naturally represents the MoCCML automata-based relations, while being able to capture the semantics of the declarative relations (see [] for more details).

This approach proved very useful since it enabled from the beginning the possibility of using automata-theoretic analysis techniques, such as reachability analysis and model-checking, directly on a model without recurring to complex model-transformation approaches. Moreover, it helped reducing the number of language concepts to a minimum (all primitives operators are meta-described by automata), and offered the tools for experimenting with the MoCCML automata-based relations. As opposed to the MoCCML language, the CLOCKSystem toolkit offers a dynamically typed language for the specification of event relations.

CLOCKSystem relies on the implementation of the meta-model presented in Figure 3.1. In this meta-model, the two central concepts are the Clocks and the ClockRelations. The Clocks are instantiated and linked to problem-space objects representing the different events of interests. Each ClockRelations contains an automaton specification encoding its operational semantics. Conceptually, this automaton is just a set of transitions between discrete states. Each transition is just an association, between one source state and one target state, labelled by a vector of Clocks that tick when the transition is executed and an actionBlock that is

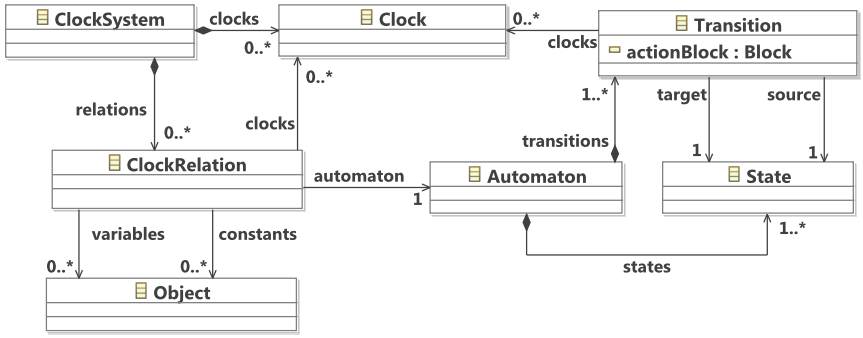

Figure 3.1: CLOCKSystem model (abstract syntax)

executed when the transition is fired. The purpose of this action block is to update either the state-variables of the automaton or the global variables in the system. Semantically, the execution of each transition is considered atomic.

In the CLOCKSystem context, the MoCCML/ CCSL expressions are nothing more than simple ClockRelation instances with an "internal clock" representing the clock produced by the expression. The CLOCKSystem class, in Figure 3.1 simply composes the set of Clocks and ClockRelations defined in a given model.

Traditionally, in automata-based approaches for ensuring theoretical properties (such as decidability, termination, etc.) the state-machines are constrained to be finite. This is done in CLOCKSystem using a dedicated relation-definition DSL (relDSL) [4].

The specification of the relations (or constraints) is now directly defined by an automata-based formalism that replaces the MoCCML/CCSL formalism in the GEMOC STUDIO. Consequently, to exploit the above functionalities of CLOCKSystem, the MoCCML/CCSL specifications should be transformed into such automata-based formalism.

3.4 CLOCKSystem Image

CLOCKSystem is a specialized image integrating basic CCSL constraints primitives, automata-based constraint specifications, and potentially user-defined specifications. An CLOCKSystem *image* is an image of the environment which has been saved at a given point in time. When this image is reloaded into the runtime system, everything remains as it was at the time the image was saved. Images usually contain components such as the debugger, compiler, editors etc.

A started CLOCKSystem VM loads the saved state of objects (including open file streams, windows, threads and more) from the *image* into its memory and resumes execution where it left when the image was saved. The *image*, VMs and the implemented actions to call the VMs are defined as plugins (see 4.3.4).

4. Towards *exhaustive exploration* in GEMOC STUDIO and model-checking with OBP

In GEMOC, the DSL executability is defined by steps to integrate properties related to their execution such as *ED*s and *EF*s, the models of concurrency MoCC associated to the DSL. The *EF*s are used to modify the *ED*s at runtime and the MoCC properties describe constraints and causal relationships between actions (*EF*s) of the concurrent entities in the DSL.

All the above elements are necessary in the process of building the *exhaustive exploration* graph in CLOCKSystem or OBP. In fact, the *exhaustive exploration* is built on the composition: in one side, of the atomic behaviors induced by the DSL concurrent entities, and on the other side the behavioral constraints (MoCC-based) that allows to build up the finite state-space of the system model.

This section presents the different modules which have been implemented with the aim of providing support for the *exhaustive exploration* of the state-space of constrained system models, but also to perform verification of properties on these finite state-spaces (otherwise called model-checking activity).

In this context, the first proposed solution (i.e. CLOCKSystem) allows *exhaustive exploration* of MoCCML constraint instances, given an instance of system model. Instances of constraints are combined together (as synchronized product of automata based on [14]) to produce the *exhaustive exploration* graph. This *exhaustive exploration* provides all the possible schedules for a given system depending on MoCCML concurrency models. For example, an execution trace can be extracted in such *exhaustive exploration* and used as simulation trace for the system model.

In order to perform model-checking, a dedicated tool for model-checking should be taken (CLOCKSystem is not a tool for model-checking) and more particularly the previous *exhaustive exploration* graph must be refined to take into account the system state variable changes. The state variable changes are caused by the execution of the *EF*s. Indeed, the *EF*s modify the *ED*s which are state variables. Such changes must be taken into account in the *exhaustive exploration*.

Therefore, to realize property verification on system models, we are using the OBP model-checking tool [5] and we combine the results from the *exhaustive exploration* of CLOCKSystem with a behavioral model describing the triggering of the actions related to the *EF*s. We have implemented a solution to generate concurrent processes based on automata. The processes represent all possible *EF*s triggering. These processes are composed in OBP with the *exhaustive exploration* model by CLOCKSystem. Indeed, OBP implements a composition algorithm to produce a complete *exhaustive exploration* graph on which properties can be verified.

In the remainder of this section, we will present: the input model required by CLOCKSystem to produce the *exhaustive exploration* and the expected output of CLOCKSystem, besides the exhaustive exploration; the specification of the *Fiacre* process templates that we generate and which must be composed with the result of the exhaustive exploration of CLOCKSystem. The CLOCKSystem input model, the model resulting from *exhaustive exploration* and the concurrent *Fiacre* process models are achieved through different plugin which have been implemented for code generation and action calls to trigger CLOCKSystem functionalities.

All CLOCKSystem modules can be integrated in GEMOC STUDIO via the plugins update site allowing to reach the *exhaustive exploration* and *Fiacre* generation. Besides, the use of OBP requires to download the tool [7].

4.1 The CLOCKSystem input and output models

In CLOCKSystem, building the *exhaustive exploration* (from an input model) is based on the instantiation of the MoCC constraints for a given system model. Each element of the system model declares its own control events that are synchronized via instances of MoCC constraints. The relationship between control events and the MoCC constraints are provided in the ECL file defining the contexts of DSLs (the declared control events for each instance in system model depend on the control events declared in its context). CLOCKSystem input model then contains: the declaration of the control events related to the elements instantiated in the system, but it also contains the set of instances of MoCC constraints linking these control events. In comparison, the input model is equivalent to the instance model extendedCCSL produced for the simulation in TimeSquare, but implemented in smalltalk. CLOCKSystem implements basic CCSL primitives (e.g. precedes, coincides). When the constraints used in the constraints instance model are not defined in the CLOCKSystem image, then they must be described also in the input model. Thus all constraints used between declared control events. From the input model, CLOCKSystem builds a state-transition (LTS) system used to generate the *exhaustive exploration* graph. There are several output format for the *exhaustive exploration* graph that we describe in section 5. For model-checking, the complete *exhaustive exploration* of OBP is produced from the above LTS and its composition with instances of concurrent *Fiacre* processes which we describe in the next section 4.2.

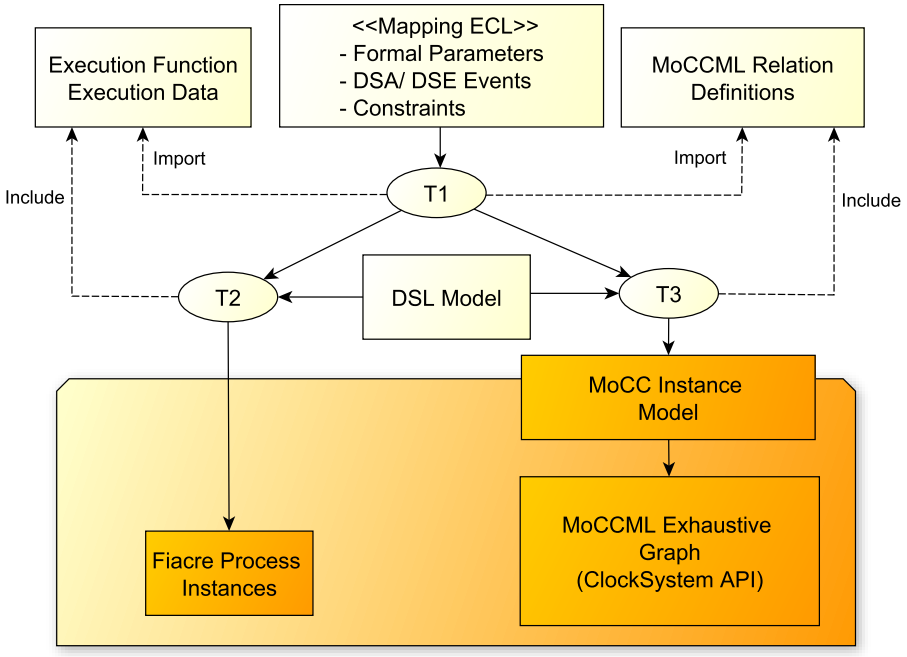

Figure 4.1: From DSL and MoCC definition to *exhaustive exploration* tool

4.2 The *Fiacre* models for *EF* and *ED*

One of the solutions that have been explored to create the *Fiacre* processes is to use the DSL contexts (ECL context). These *Fiacre* processes define state-transition behaviors on which each transition triggers an action *EF* declared in a DSL context. The described automata are not constrained knowing that the LTS will provide the constraints. The *EF* actions are described in the form of $function$. The $function$ can have parameters (*ED*) that they can modify. The described $function$ relies on the *EF* types $Modifier$ and $Query$ (see D 3.3.1). To facilitate the automatic generation of the *Fiacre* process descriptions, annotations on the *EF*s will be further defined to simplify the parsing of the *EF*s thus improving the current code generation rules. The context-specific *Fiacre* processes are instantiated as many times as they are instances of the context element in a system model. Automata defined by these *Fiacre* processes are then composed with the LTS in OBP.

4.3 Implemented Plugins for GEMOC STUDIO

This section describes the different parts that are defined as plugins to integrate in the GEMOC STUDIO in order to uses the CLOCKSystem for *exhaustive exploration* and further use OBP for model-checking. The integration of CLOCKSystem is compatible with windows, mac and unix operating systems. It uses the Pharo Virtual Machine (VM) [3] implemented in these different OS. The VM are used with an image of the CLOCKSystem environment.

There are several actions and model transformations that lead to the building of *exhaustive exploration* graph, or to the generation of the input format of OBP for model-checking. Each action call corresponds to a specific task as described in the next paragraphs. Some of the tasks are used to reach *exhaustive exploration* in CLOCKSystem, some others to produce the input format for OBP. We will first focus on the tasks that are important for CLOCKSystem. The Figure 4.1 presents the first phase aiming at the generation of the input format for CLOCKSystem and the generation of *Fiacre* processes. The transformations implement the ideas described in the section 4.1 and 4.2.

4.3.1 TASK: generating the input model formats for CLOCKSystem and OBP

A first model transformation $T1$ takes as input the ECL mapping definition between the DSL and the MoCCML to generate two transformations ($T2$, $T3$) describing transformation rules that will be used to produce the processes for the DSL related functions and data (DSA) and the behavioral processes corresponding to the MoCCML constraints. $T2$ generates *Fiacre* processes modeling the behavior of a given system model - the *Fiacre* processes includes the *ED* and *EF* of the DSL used to realize the models. $T3$ instantiates the MoCCML relation definitions described on the MoCC file and produces processes taking into account these MoCC properties applied on DSL instance model. Both $T2$ and $T3$ take as input an system instance model of the system (DSL model). The models generated by $T2$ and $T3$ are taken as input in OBP and CLOCKSystem.

4.3.2 TASK:generating an LTS and building exhaustive exploration *graph*

The *Launch Exhaustive Exploration* action call is used to generate the LTS and the *exhaustive exploration*. It takes as input the MoCC instance model produced by $T3$. The file generated by $T3$ is identified with a (.clocksystem) extension. The part of the reachability graph related to the MoCC instance model is generated, thus highlighting the set of all possible schedules.

The LTS used to build the *exhaustive exploration* can be later composed with the concurrent entities behavioral processes in OBP. The LTS corresponds to the input format for OBP (.lts extension). The next section provides description of the steps producing the *Fiacre* Processes.

4.3.3 TASK: generating the Fiacre *processes*

The $T2$ generates the *Fiacre* models (.fcr file extension). The *Fiacre* behavioral processes represent the DSL concurrent entities and their behavior. The behavior of a concurrent entity in *Fiacre* is described as a simple automaton that defines transitions corresponding to all the event triggers defined for the control of a given entity. This set of events is described in ECL as events related to DSA or DSE. There are as many transitions in the automaton as there are control events. The *EF*s and the *ED*s are also described in the file generated from $T2$.

To reach the description of the *ED*s and *EF*s, $T1$ and $T2$ parse the <extended> metamodel associated to each DSL and defining the DSA of the DSL. $T1$ retrieves the declared *ED*s and *EF*s, while $T2$ retrieves the core computation of the actions described in the functions (some core computations can be retrieved directly by $T1$ if they don't change during instance model modeling). OBP Explorer takes as input the *Fiacre* processes from $T2$ and can run compilation and reachability graph building.

There are several scenarios for generating the Executions Functions translated into *Fiacre*. Indeed, the parameters of executions functions may be described differently in the context of model-checking. In a first approach, we can assume that the parameters of the functions are fixed, or taken from varying intervals of values. The three identified scenarios for parameters description are summarized below. Each of these choices impacts the generation phase for *Fiacre* models.

- The parameters of all actions are described as fixed values.
- The parameters are described as variables taking values in intervals.
- In the last case, the action parameters combines the two first cases.

4.3.4 Currently available plugins for GEMOC STUDIO

The plugins implementing the various transformations $T1$, $T2$ and $T3$ and their related launching actions are provided in the following projects:

- Plugin for the Transformation $T1$ producing the transformation $T2$, $T3$
  - org.gemoc.mocc.transformations.ecl2mtl
  - org.gemoc.mocc.transformations.ecl2mtl.ui

- Plugin using the Transformation $T2$, $T3$ generating the *Fiacre* files and CLOCKSystem files for OBP Explorer and CLOCKSystem
  - org.gemoc.mocc.transformations.instance2clocksystem
  - org.gemoc.mocc.transformations.instance2clocksystem.ui

The action commands to use CLOCKSystem and its different VMs are embedded in the following plugins:

– org.gemoc.mocc.clocksystem.core

– org.gemoc.mocc.clocksystem.core.ui

These plugins are submitted in the GIT collaborative repository.

5. Quick Tutorial using CLOCKSystem in GEMOC STUDIO

The objective of the tutorial is to show the different sequences of actions leading to the generation of the *exhaustive exploration* graph, but also to show excerpts of the expected outputs of CLOCKSystem for OBP. The tutorial goes through a simple example of two connected agents implemented using the *SigPML* DSL as shown by Figure 5.1. The starting point is the ECL mapping file for *SigPML*. The Listing 2.1 already gave an excerpt of an ECL mapping file associated to *SigPML*.

Before calling CLOCKSystem specific actions, the model transformations generating $T2$ and $T3$ should be called, then $T3$ is used to generate de CLOCKSystem input. The Screenshot in Figure 5.2 shows how to call the transformation $T1$ generating $T2$, $T3$. Right-click on the ECL file → GEMOC → *Generate Rules [Fiacre/ ClockSystem]*. The Transformations $T2$ and $T3$ are generated in the repository <clocksystem-gen>.

```
system := ClockSystem named: 'Ex0'.
  system addClocks: #(
    A_start A_isExecuting A_stop
    B_start B_isExecuting B_stop
    A2B_push A2B_pop).

  system library: #SDFStateBasedLib
    relation: #actorRelationDef
    clocks: #(A_start A_isExecuting A_stop)
    constants: { 1 }
    variables: { 0 }.

  system library: #SDFStateBasedLib
    relation: #actorRelationDef
    clocks: #(B_start B_isExecuting B_stop)
    constants: { 1 }
    variables: { 0 }.

  system library: #SDFStateBasedLib
```

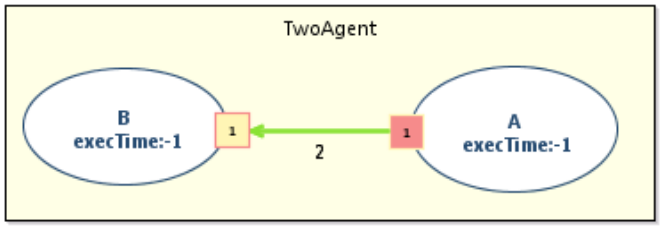

Figure 5.1: Example of two communicating Agents in *SigPML*

Figure 5.2: Generating $T2$ and $T3$

```
    relation: #edgeRelationDef
    clocks: #(A2B_pop A2B_push)
    constants: { 1 }
    variables: { 0 }.

  "stop causes push"
  system library: #Kernel relation: #<= clocks: #(A_stop A2B_push).
  "pop causes start"
  system library: #Kernel relation: #<= clocks: #(A2B_pop B_start).
```

Listing 5.1: Generating an Specification file for CLOCKSystem

In the Modeling environment (*modeling workbench*) where the DSL instance model is realized, the transformations $T2$ and $T3$ take as input the instance system model to generate the Fiacre processes and the MoCC instance model described in CLOCKSystem specification format. The actions launching the generation process are accessed in the same way as in Figure 5.2. For instance, Right-click on the *SigPML* model (.sigpml) → GEMOC → *Generate ClockSystem Instance model*. $T3$ generates the .clocksystem file corresponding to the MoCC-based specification model to take as input in CLOCKSystem. Listing 5.1 shows an excerpt of such file in which we can see the specification of the instance model using MoCCML relation definitions such as *#actorRelationDef*, *#edgeRelationDef*. and primitive CCSL relations such as *#coincides*.

The action for *exhaustive exploration* graph generation is called on this file. Right-click on the CLOCKSystem model (.clocksystem) → GEMOC → *Launch Exhaustive Exploration*. The launched action generates a set of files representing the output LTS (.lts) of the MoCC instance model, but also several graphical format (.mtx, .dot, etc) that can be visualized graphically and representing the *exhaustive exploration* for the instance model taking into account the MoCC relations.

The OBP tool requires two input files. The LTS generated from CLOCKSystem and the Fiacre model produced for OBP. The LTS file is built by the action *Launch Exhaustive Exploration*. The file does not contain the actual computations induced by the *EF*s of the DSA and the modifications of *ED*s. Such functions are defined as Fiacre processes representing the system computations. The Fiacre process generation is called as follows : Right-click on the *SigPML* model (.sigpml) → GEMOC → *Generate Fiacre Instance model*.

```
const SIZE_LocalMemory     : int is 2    //the size of local memmory buffers
const MAX_PortsInProcess   : int is 1    //the maximum number of input or output ports

const NUMBER_Ports         : int is 2    //the number of ports in the system
const NUMBER_Agents        : int is 2    //the number of agents in the system
```



```
const NUMBER_Channels      : int is 1    //the number of channels in the system

const CAPACITY_Channel     : int is 2    //the maximum channel capacity

const pA1_rate : int is 1
const pB1_rate : int is 1
```

Listing 5.2: Example of global variable declaration

```
function Agent_isExecuting_A(&global : SystemData, _self : read AgentData) : SystemData is
  begin
    global.ports[_self.outputPortIds[0]].data[0] := 2;
    return global
  end

function Agent_isExecuting_B(&global : SystemData, _self : read AgentData) : SystemData is
  begin
    //nothing here -- B is sync
    return global
  end
```

Listing 5.3: Example of function declaration

```
process A[start, isExecuting, stop : in none](&global : SystemData, id : read int) is
  states s0
  from s0 start; /*do nothing*/ to s0
  from s0 isExecuting; global := Agent_isExecuting_A(global, global.agents[id]); to s0
  from s0 stop; /*do nothing*/ to s0

process B[start, isExecuting, stop : in none](&global : SystemData, id : read int) is
  states s0
  var sum : int := 0
  from s0 start; /*do nothing*/ to s0
  from s0
    isExecuting;
    global := Agent_isExecuting_B(global, global.agents[id]);
    sum := (sum + global.ports[global.agents[id].inputPortIds[0]].data[0])%100;
  to s0
  from s0 stop; /*do nothing*/ to s0
```

Listing 5.4: Example of process declaration

There are different parts in this generation: Listing 5.2 relates to the declaration of global variables such as the number of ports, channels, the rates of the agents on their input/ output ports, etc; Listing 5.3 shows the capture of the implemented *EF*s, for instance the *EF* of the agent *A* i.e. *Agent_isExecuting_A*; Listing 5.4 shows the definition of each agent process transitions and *EF*s calls. For instance for each agent we have three transitions corresponding to the action triggers and their related actions. Finally, Listing 5.5 illustrate the parallelization of all the different instance processes which represents the composition of all the actions defined in the instance model.

```
component sys is
  var
    global : SystemData := INITIAL_Configuration

  port
    A_start, A_isExecuting, A_stop : none,
    B_start, B_isExecuting, B_stop : none,
    A2B_push, A2B_pop : none
  par
    A[A_start, A_isExecuting, A_stop](&global, 0)
  ||  B[B_start, B_isExecuting, B_stop](&global, 1)

  ||  Channel[A2B_push, A2B_pop](&global, 0)

  ||  Semantics[
```

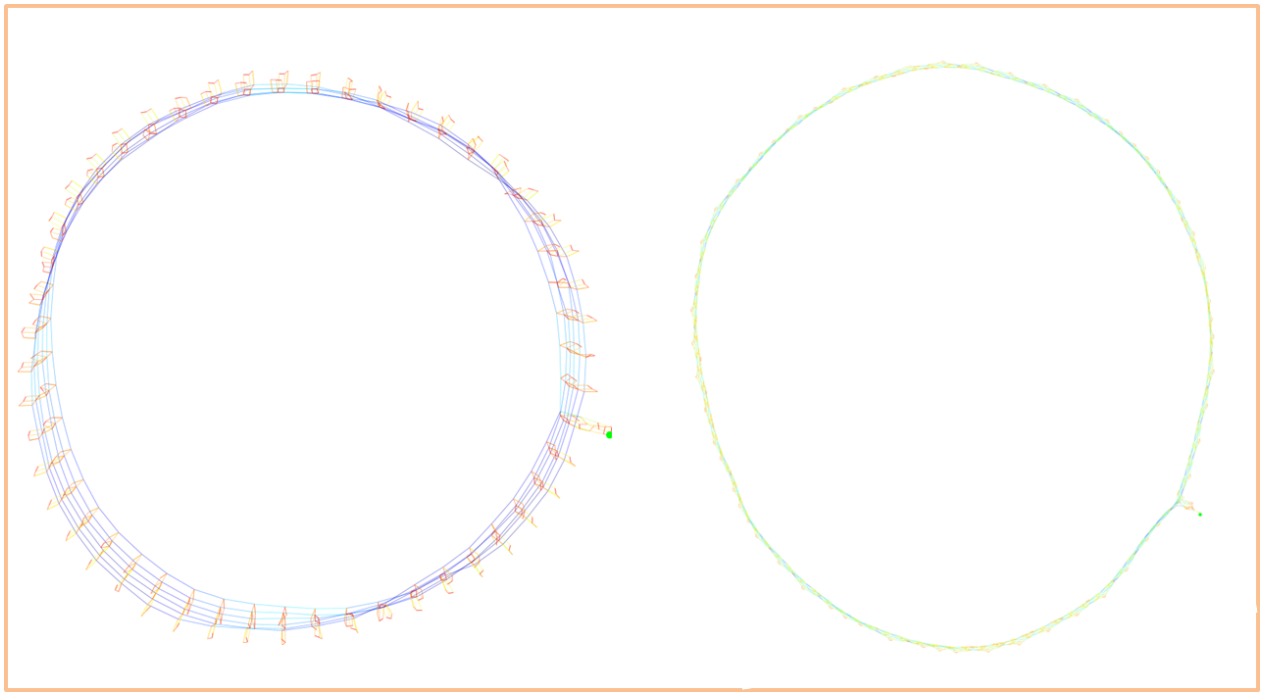

Figure 5.3: Excerpt of the Exhaustive graph generated from the example in 5.1

```
16      A_start, A_isExecuting, A_stop,
17      B_start, B_isExecuting, B_stop,
18      A2B_push, A2B_pop]
19    end
20  sys
```

Listing 5.5: Parallelized Fiacre Processes for the 5.1 example

Once the two files are generated, the user can download a model-checking tool that takes as input such LTS and Fiacre processes like for instance OBP. The tutorial and download of OBP can be obtained in [7].

The Figure 5.3 shows excerpts of the generated *exhaustive exploration* graph. The Figure on the left shows the *exhaustive exploration* for an unconstrained instance model; while the Figure on the right shows the *exhaustive exploration* when the CLOCKSystemMoCCML relations (from Listing 5.1) are taken into account. The differences are due to the constraints applied to the process leading to a new state-space.

The section A gives a brief description of OBP and its functionalities.

6. Conclusion

This document describes the integration of the CLOCKSystem tool in the GEMOC STUDIO to enable *exhaustive exploration* in the context of modeling executable systems that are constrained with MoCCML relations. The document presents the different inputs, the steps to using these inputs and finally their location in the collaborative GIT repository.

exhaustive exploration paves the way to connect with model-checking tools such as OBP in order to verify properties on the system model, based on its finite state-space execution model.

Appendices

A. The Observer-Based Prover (OBP) and Functionalities

As shown in Figure A.1, OBP verifies properties on finished system models taking into account their environment interacting with the system. The system models are described in the form of Fiacre programs [1] describing the behavior interactions and temporal constraints through timed-automata based approach. Besides, the system environment and its requirements (to be checked) are specified using the Context Description Language (CDL). CDL defines the environment in the form of an independent set of contexts that interact asynchronously with the system model. In addition, CDL allows the specification of properties (requirements) to check using the OBP Observation Engine. The properties expressed through patterns of properties are based on perceived events such as value change of a variable. They are also based on predicates that are composed to express invariants or observers.

The OBP Observation Engine checks a set of properties using reachability strategy (breath-first-search algorithm) on the graph induced by the parallel composition of the system, with its contexts. During the exploration, the OBP Engine captures the occurrences of events and evaluates the predicates after each atomic execution of each transition. The invariants and status of observers are then updated which allows an exhaustive state-space analysis. The Labeled-Transition System (LTS) resulting from the composition can also be used to find the state-system invalidating a given invariant, or to generate a (cons-example) based on the success/reject states of an observer thus serving as a guide for the system developer.

7. References

[1] Bernard Berthomieu, Jean-Paul Bodeveix, Patrick Farail, Mamoun Filali, Hubert Garavel, Pierre Gaufillet, Frederic Lang, François Vernadat, et al. Fiacre: an intermediate language for model verification in the topcased environment. In *ERTS 2008*, 2008.

[2] Bernard Berthomieu*, P-O Ribet, and François Vernadat. The tool tina–construction of abstract state spaces for petri nets and time petri nets. *International Journal of Production Research*, 42(14):2741–2756, 2004.

[3] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, Marcus Denker, et al. *Pharo by example*. 2009.

[4] Ciprian Teodorov. Embedding Multiform Time Constraints in Smalltalk. In *IWST*, 2014.

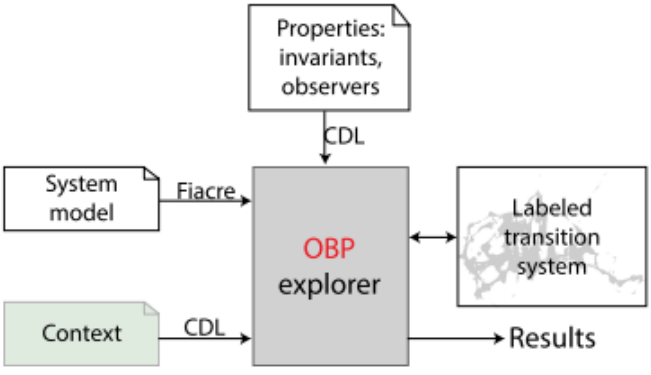

Figure A.1: Overview of the OBP Tool Usage

[5] Philippe Dhaussy, J Roger, and Frederic Boniol. A language : Context Description Language (CDL) and A toolset : Observer Based Prover (OBP).

[6] Philippe Dhaussy, J Roger, and Frederic Boniol. Reducing state explosion with context modeling for model-checking. In *High-Assurance Systems Engineering (HASE), 2011 IEEE 13th International Symposium on*, pages 130–137. IEEE, 2011.

[7] Dhaussy, Philippe. Observer Based Prover (OBP). Website. http://www.obpcdl.org/doku.php.

[8] Marc Geilen, Twan Basten, and Sander Stuijk. Minimising buffer requirements of synchronous dataflow graphs with model checking. In *DAC*, pages 819–824. ACM, 2005.

[9] Gerard J Holzmann. The model checker spin. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[10] Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, 1997.

[11] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proc. of the IEEE*, 75(9):1235–1245, 1987.

[12] V. Papailiopoulou, D. Potop-Butucaru, Y. Sorel, R. De Simone, L. Besnard, and J. Talpin. From design-time concurrency to effective implementation parallelism: The multi-clock reactive case. In *ESLsyn*, pages 1–6, 2011.

[13] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *5th Colloquium on International Symposium on Programming*, pages 337–351. Springer, 1982.

[14] MichaelW. Shields. The nivat/arnold process model. In *Semantics of Parallelism*, pages 342–367. Springer London, 1997.

