Grant ANR-12-INSE-0011

ANR INS GEMOC

D3.4.2 - Experimental validation
Task 3.4

Version 1.1

DOCUMENT CONTROL

	–: 2015/11/26	A: 2015/11/26	B:	C:	D:
Written by Signature	Joel CHAMPEAU	Vincent LEILDE	Ciprian TEODOROV		
Approved by Signature					

Revision index	Modifications
–	version 1.0 — Revised version
A	
B	
C	
D	

Authors

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

Contents

1. Introduction

1.1 Purpose

One of the benefits provided by assigning an execution semantics onto a DSL is to pave the way for *exhaustive exploration*. *exhaustive exploration* is a good candidate technique in complex and safety system design to ensure the correct functioning or behavior of the system. An *exhaustive exploration* consists in building an exhaustive finite state space of a system which represents the whole set of relevant configurations your system may reach and onto which model checking techniques are applied for checking relevant properties. In this field CLOCKSystem is a framework that provides finite state-space generation capability for a given executable model.

Model checking technique [3] offers a rigorous and automated framework for formal verification applied on a finite model of a system. Even if its usability has been proved on some concurrent industrial systems, the more these systems are composed of interacting components the more the state-space growth. This issue is known as the state-space explosion [20] and even if significative efforts have been made to reduce its impact [19] most of the time system designers manually tunes their verification model to restrict its behavior. Instead of doing it manually which is error prone, approaches such as CDL [10] helps to capture the interactions between the system and its environment. Besides properties generally represent fomalized requirements in a language enabling their automatic verification on a given system. CDL also provides a language to define properties via assertions or observer automatons to later target model checking tools.

In GEMOC project Exhaustive Exploration feature facilitates the connection to model-checking tools such as [5, 13, 14]. 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 is an update of D3.4.1 and makes a particular focus on properties that can be checked on executable models within the GEMOC project. It validates by experimentation the approach chosen in D3.4.1 and presents the final tool chain available in GEMOC studio.

1.2 Perimeter

This document is the version v1 of the D3.4.2 deliverable. Its purpose is to describe based on a simple example the approach chosen in Gemoc to exhaustively explore executable models and verify relevant properties. It describes the complete tool chain integrated to the GEMOC STUDIO (ref. T 3.4 = V&V of the formalized semantics).

1.3 Definitions, Acronyms and Abbreviations

- **Model**: model which contributes to the convent of a View
- **Language Workbench**: a language workbench offers the facilities for designing and implementing modeling languages.
- **Language Designer**: a language designer is the user of the langage workbench.
- **Modeling Workbench**: a modeling workbench offers all the required facilities for editing and animating domain specific models according to a given modeling language.
- **Domain Engineer**: user of the modeling workbench.

- **GEMOC Studio**: Eclipse-based studio integrating both a language workbench and the corresponding modeling workbenches.
- **DSML**: Domain Specific Modeling Language.
- **xDSML**: Executable Domain Specific Modeling Language.
- **AS**: Abstract Syntax.
- **MOC**: Model of Computation.
- **RTD**: RunTime Data.
- **DSA**: Domain-Specific Action.
- **MSA**: Model-Specific Action, an instance of a Domain-Specific Action. While a DSA is specific to a language (to its Abstract Syntax), an MSA is specific to a model conforming to a language.
- **DSE**: Domain-Specific Event.
- **MSE**: Model-Specific Event, an instance of a Domain-Specific Event. While a DSE is specific to a language, an MSE is specific to a model conforming to a language.
- **GUI**: Graphical User Interface.
- **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.
- **API**: Application Programming Interface
- **CCSL**: Clock-Constraint Specification Language.
- **TESL**: Tagged Events Specification Language.
- **OBP**: Observer Based Prover
- **Static semantics:** Constraints on a model that cannot be expressed in the metamodel. For example, static semantics can expressed as OCL invariants.
- **Execution semantics:** Define when and how elements of a language will produce a model behavior.
- **Dynamic semantics:** see *Execution semantics*.
- **Behavioral semantics:** see *Execution semantics*.

1.4 Summary

Starting from a short motivating example in 2 to illustrate exploration and property verification, this document describes the exploration and verification tooling3 proposed in Gemoc. It then presents a categorization of properties and their relation in the whole Gemoc approach in 4. The following chapter 5 presents the integration strategy of the tooling up to property checking, and ends with chapter 6 presenting a quick tutorial.

2. Sigpml Motivation Example

The objective of this section is to provide an updated overview of D3.4.1 illustration example. It focuses on exhaustive exploration and property verification for this model of DSL constrained with MoCCML in the GEMOC STUDIO.

2.1 *SigPML* Motivation Example SIGPML

2.1.1 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 [15]).

In the *SigPML* syntax, an application is described as a set of *Block*s. Upon activation, each block 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 *Block*s are stored in *Connector*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 define explicitly the MoCC of the language and therefore automatically obtain the execution model of a given *SigPML* program.

2.1.2 The SigPML *MoCC*

Before explicitly introducing the MoCC constraints, the set of relevant *SigPML* events have to be identified. In the context of a *Block* the relevant event is *execute* and in the context of a *Connector* the relevant events are *write* and *read*.

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.

```
context Block
  def : execute : Event = self.execute()

context Connector
  def : write :Event = self.push()
  def : read :Event = self.pop()

 inv ConnectorComputing:
  let capacity : Integer = self.oclAsType(Connector).capacity in
  let inRate : Integer = self.itsInputPort.oclAsType(Port).rate in
  let outRate : Integer = self.itsOutputPort.oclAsType(Port).rate in
  let currentSize : Integer = self.oclAsType(Connector).currentSize in

 Relation ConnectorSDFPAM (
  self.itsOutputPort.oclAsType(OutputPort).owner.oclAsType(Block).execute,
  self.itsInputPort.oclAsType(InputPort).owner.oclAsType(Block).execute,
  capacity,
  inRate,
  outRate,
  currentSize
)

inv write_push:
  Relation Coincides(self.itsOutputPort.oclAsType(OutputPort).owner.oclAsType(Block).execute, self
      .write)

inv read_pop:
  Relation Coincides(self.read, self.itsInputPort.oclAsType(InputPort).owner.oclAsType(Block).
      execute)
```

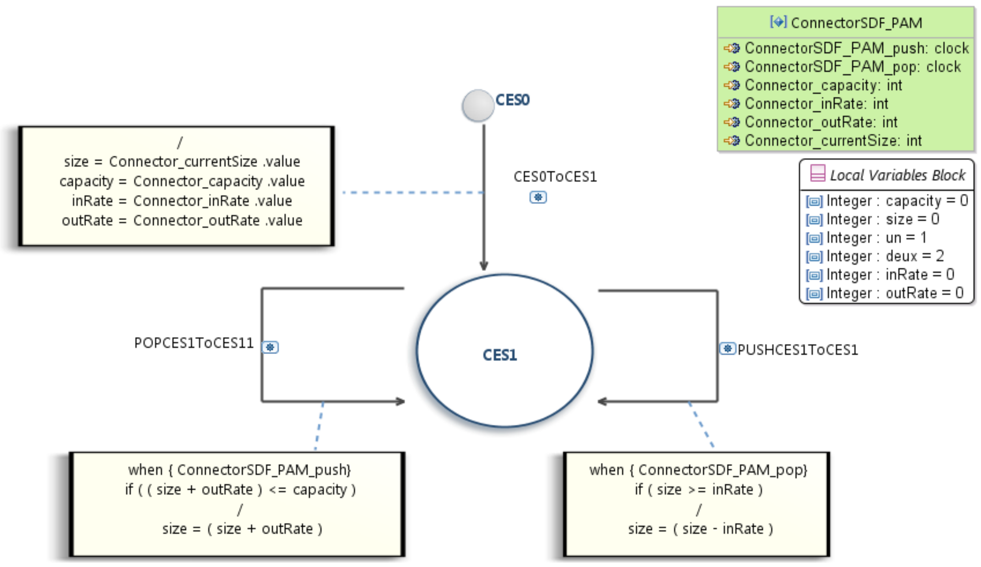

ConnectorSDF_PAM
ConnectorSDF_PAM_push: clock
ConnectorSDF_PAM_pop: clock
Connector_capacity: int
Connector_inRate: int
Connector_outRate: int
Connector_currentSize: int
Local Variables Block
Integer : capacity = 0
Integer : size = 0
Integer : un = 1
Integer : deux = 2
Integer : inRate = 0
Integer : outRate = 0
CES0
CES0ToCES1
/
size = Connector_currentSize .value
capacity = Connector_capacity .value
inRate = Connector_inRate .value
outRate = Connector_outRate .value
CES1
POPCES1ToCES11
PUSHCES1ToCES1
when { ConnectorSDF_PAM_push}
if ((size + outRate) <= capacity)
/
size = (size + outRate)
when { ConnectorSDF_PAM_pop}
if (size >= inRate)
/
size = (size - inRate)

Figure 2.1: The ConnectorSDFPAM automata

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 *Block* is associated with an execute event. This event has to be constrained to provide the adequate semantics. For instance in line 11 of listing 2.1 the events are used in a constraint (i.e. ConnectorSDFPAM) defined in the context of a *Connector*. The ConnectorSDFPAM automata is shown on Figure 2.1. It defines a constraint between the *execute*s of two *Block*s attached through a *Connector*. This automata imposes that reading does not occur if there is not enough data in the connector and writing does not occur if there is not enough room in the connector and thus reproduces the SDF semantics. This constraint is instantiated for each instance of *Connector* in a *SigPML* model.

The reader should note that these automata could be modified to provide variants of the semantics.

2.1.3 A SigPML *model*

A Passive Acoustic Monitoring (PAM) system is modeled by using the *SigPML* language previously defined. The PAM system is composed of six *Blocks* (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 version, no hardware platform deployment is defined which means that infinite resource possibilities are assumed.

2.2 Exhaustive exploration for property verification

2.2.1 exhaustive exploration *exploration*

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

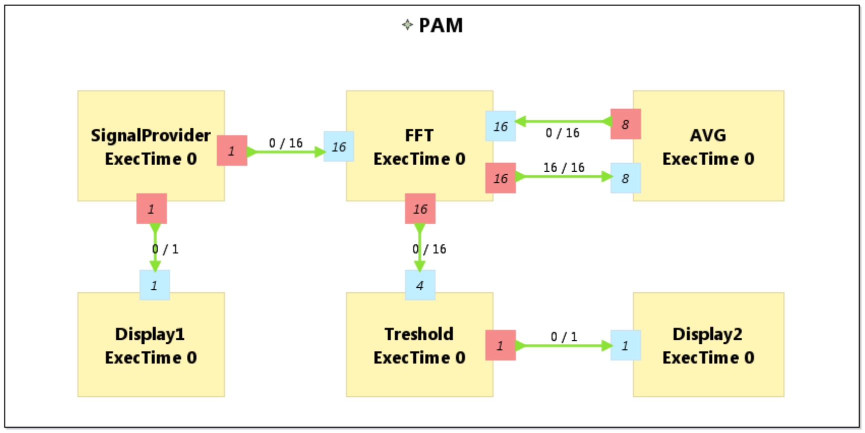


Figure 2.2: PAM Use Case Conceptual Model.

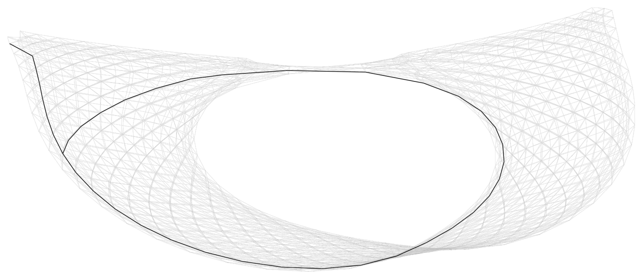


Figure 2.3: State space representation of PAM_0, encoding the set of valid schedules. Emphasis on the schedule obtained under ASAP policy (black line).

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 *Connector*s. 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.

2.2.2 Expressing Properties

Our model must respect a SDF-like semantics and therefore we are interested in verifying corresponding properties:

- If all the input ports of a *Block* haven't enought data to consume then the *Block* canno't be executed.
- The current size of a Connector canno't exceed its capacity. An internal moc variable called *currentsize* increments or decrements respectively if a data is *push* or *pop* within the *Connector*, and this variable must be always lower than the maximal capacity of the Connector.

Some other interesting properties could also be checked.

- An execution of *Display1* must always precede an execution of *Display2*. In our flow the *Display1* produces a data which has to be later processed by *Display2*.
- Deadlocks. Even if this could also be determinated statically we will benefits from the exhaustive exploration tooling in Gemoc to verify that our system model doesn't introduce any deadlocks.

This example shows that generating an execution model that configures a generic execution engine provides simulation and exploration for free and therefore facilitates property verification. 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 and verification(SDF). Such manipulations includes:

- the verification of temporal logic properties (safety and liveness) [17], 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 [12], 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 [16].

3. Exploration and Verfication Gemoc Tooling

This chapter describes the various tools involved in the exhaustive exploration chain within GEMOC STUDIO and beyond see figure 3.1. The chapter extends the deliverable D3.4.1 by explaining the functionalities that have been experimented in the scope of Gemoc.

3.1 Transformations

The mapping between DSML and MoCML is made at the ECL specification level. In this specification, we define events associated with the actions of the DSA and also events associated with the DSE events. On these event bindings we apply the MoCML relations of the MoC Library to schedule the events. To make an exhaustive exploration of the finite state space of a system using such scheduling constraints, the idea is to generate two transformations:

- (1) Generate a tranformation T1 from this ECL specification at language level. This new transformation provides a generic template to later target exhaustive exploration or simulation tools.
- (2) Generate a model to target exhaustive exploration tools at modeling level from the T1 template and a given model .

To address this issue we developped a plugin for the Transformation $T1$ producing the transformation $T2$ and a plugin using the Transformation $T2$ generating the input file for CLOCKSystem:

- org.gemoc.mocc.transformations.ecl2mtl
- org.gemoc.mocc.transformations.ecl2mtl.ui

These plugins are submitted in the GIT collaborative repository and are natively integrated in Gemoc studio.

The *Fiacre* models for *EF* and *ED* approach described in D3.4.1 is not investigated in Gemoc studio. The reason is that Clocksystem generates a LTS with sufficient informations(data values, transitions, states, ticked clocks) to enable property verifications. Fiacre tranformation also implies technical issues such as scalability (state-space explosion due to many processes created).

3.2 CLOCKSystem

3.2.1 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

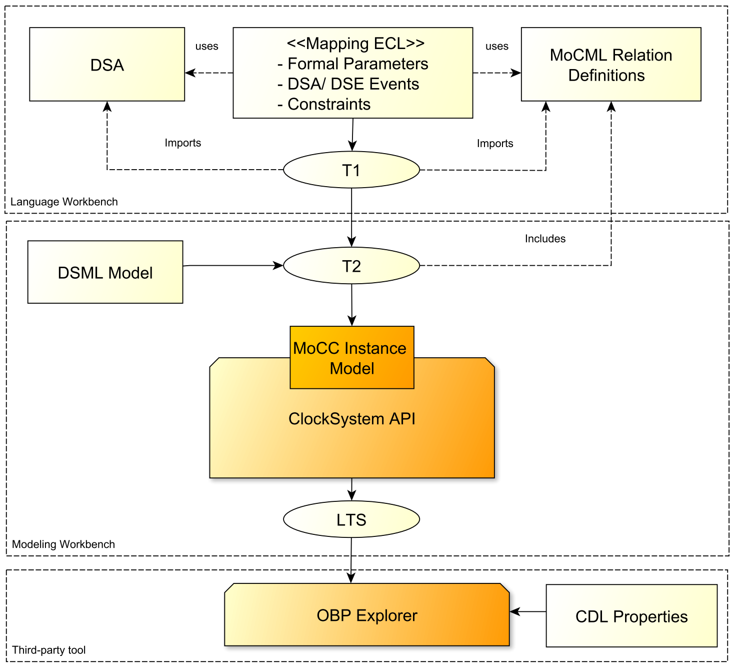


Figure 3.1: From DSL and MoCC definition to *exhaustive exploration* tool

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.

3.2.2 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 [7] 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 3.2. 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 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 3.2 simply composes the set of Clocks and ClockRelations defined in a given model.

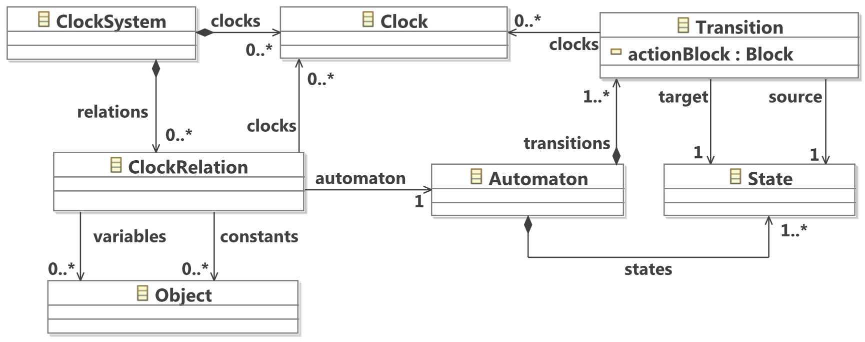


Figure 3.2: CLOCKSystem model (abstract syntax)

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) [8].

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.

3.2.3 CLOCKSystem Results

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 of properties by model-checking [11] using a tool like OBP. Relying on the exhaustive reachability results, an interface with the OBP model-checking toolkit [8] 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.

This feature is in the scope of Gemoc at least for the generation of the LTS. From several transformations a clocksystem file is generated for a given model of a DSL and a given execution semantics. Then a Clocksystem service to perform exhaustive exploration is invoked producing a LTS graph. The later can be reused in a reachability analysis scenario in conjunction with properties using OBP.

Clocksystem currently consists of an image and a Pharo VM which depends on the operating system. To avoid size issues with Pharo VM(Clocksystem represents approximatively 30Mo) it has been decided to separate the clocksystem plugin into plugin fragments each supporting an different OS's VM. Therefore only the required VM is installed in the user Gemoc distribution.

The action commands to execute CLOCKSystem services are embedded in the following plugins:

– org.gemoc.mocc.clocksystem.common

– org.gemoc.mocc.clocksystem.common.ui

The Clocksystem VMs are embedded in the following fragments:

– org.gemoc.mocc.clocksystem.win

– org.gemoc.mocc.clocksystem.linux

– org.gemoc.mocc.clocksystem.macosx

These plugins are submitted in the GIT collaborative repository and fully integrated in the CI process. Their integration in Gemoc studio can be realized via the discovery mecanism. Clocksystem generates files:

- *.lts* file stores labeled transition system (LTS) which represents all the possible configurations the system can reach.
- *obp.lts* file stores labeled transition system (LTS) in a format understandable by OBP.
- *.results* extract global information about the size of the explored graph(number of states, transitions and time of exploration).
- *full.gml* is the representation of the LTS graph stored in a Graph Modelling Language (GML) [1] format providing a simple syntax to represent graph.
- *fcr.gml* is the representation of the LTS graph with the coincidences flatten for Fiacre stored in GML.
- *.mtx* stores the representation of the LTS graph as a Matrix Market (MM) providing a simple and standardised way to exchange matrix data [2].

3.3 CDL

CDL is a language to both describe the environment and properties of a system.

As described in introduction model checking is often challenged with unmanageable large state-space also known as state-space explosion. In order to reduce the state-space of system during its exploration CDL offers to the designer the possibility to specify the system's environment and therefore set its constraints of use. The environment is composed of a set of *contexts* which has an acyclic behavior communicating asynchronously with the system. The interleaving of these contexts generates a labelled-transition system(LTS) representing all behaviors of the environment, which can be fed as input to traditional modelcheckers.

Besides CDL provides a property language for capturing the requirements. The CDL formalism provides 3 distincts constructs for expressing safety and bounded-liveness properties [18] predicates to express invariants over states, observers to express invariants over execution traces and property patterns, for simplifying the expression of complex properties.

Properties are described using CDL syntax and must be specified at instance level thus the name of the processes or variable used in properties reflects the names and variables of instances within the clocksystem model. CDL properties are written in a simple text file with the extension .cdl.

3.4 Observer Based Prover

As shown in Figure 3.3, 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 [4] 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).

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.

In the context of Gemoc OBP is used as a model checking engine that takes as input both the LTS file generated from Clocksystem and the set of CDL properties to be checked. This operation has to be done

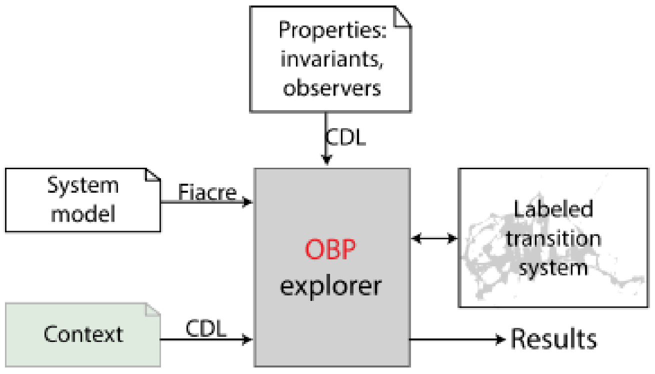


Figure 3.3: Overview of the OBP Tool Usage

manually and is not part of the Gemoc studio tooling since OBP is not integrated to the studio.

4. Properties for metamodeling Patterns

The goal of the GEMOC project is to provide an approach of separation of concerns at meta level to ease the definition of xDSML. Associated with Abstract Syntax (AS) and Concrete Syntax (Graphical or Textual CS), the executable semantics of the xDSML is also expressed and in the case of the concurrent executable semantics, we have defined the MoCCML metalanguage. The MoCCML language is used as constraint automata language on the events which abstract the operationnal actions of the AS. The explicit definition of the concurrent executable semantics is suitable to provide a precise semantics and also to ease analysis on the concurrent behavior of the resulting models, or on metamodel patterns or on the concurrent execution of the language itself.

To provide significant analysis on the MoCCML models, we experiment the use of a model checking approach to support the verification of properties with the OBP tooling. To target this tooling, first we have designed and developp a dedicated tool, CLOCKSystem, to create an exhaustive exploration state space from the MoCCML specification as we describe in the previous chapter. This state space is a Labelled Transition System (LTS) which is processed by the OBP tooling to verify safety properties.

The content of this section is to present the motivations of the verification of the properties regarding the models of the xDSML, the mapping of the MoCCML model to the AS and finally to the MoCC of the language definition itself.

4.1 Properties at model instances

The previous chapter exemplifies, explains and details this approach on the *SigPML* ‘s xDSML. The result of these experiments give us the identificationof properties of each model instance. The properties are expressed on the model instance based on variables and clocks of one (or several) the model element. The properties are expressed as assertion and observer automata with appropriate variables and clocks of the model instance, described in the section 2.2.2.

The expression of the properties are model dependent so on each instance you must rewrite the properties.

The next section, we experiment how can we express properties related to representative model instances which are based on a mapping between a generic AS, or metamodel pattern, and a mapped MoCC on this AS. In this approach, we are looking for a reducing number of instances to verify and so to increase the generality of the verification approach.

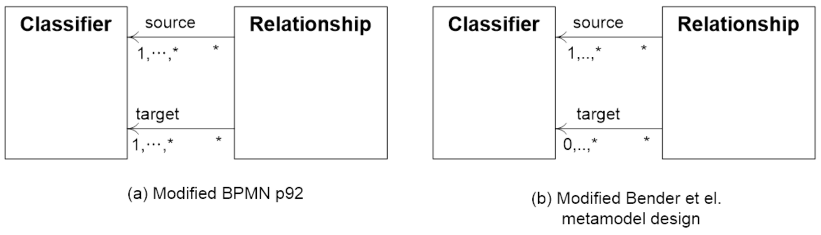


Figure 4.1: Metamodel examples from [6]

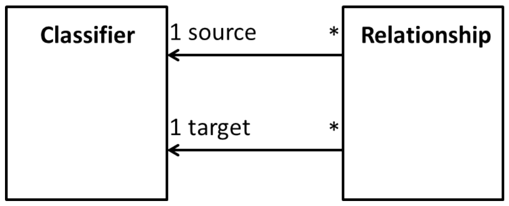


Figure 4.2: Classifier-Relationships metamodel pattern

4.2 Properties related to the metamodel pattern and MoCC definition

In the GEMOC approach, the separation of concerns is applied at meta level to define all the concerns of the language definition with a mapping definition between the AS and the MoCC definition, in our focus. To increase the reusability of the verification process in this context, we apply the property verification on model instance which are based on a generic approach at meta level for the metamodel itself and also for the MoCC expression.

First on the metamodel side, metamodels patterns [6] and the generalized approach with the model type formalization [9] provide capacities to define metamodels in a composable modular way. To illustrate the modularization of a metamodel, we consider metamodels to define entities and relationships between them.

The figure 4.1 presents two metamodel examples including this pattern. Based on these examples, we define the classifier-relationships pattern as the metamodel presented in the figure 4.2.

A metamodel pattern is a reusable solution that must be adapted regarding the situation to define a new metamodel. In our case the *Classifier* class is the *Block* class and the *Realtionship* is the *Connector*. But in the *SigPML* metamodel, the *Port* concept is an intermediate concept between the *Block* and *Connector* but the metamodel pattern is present as showed in the figure 4.3.

At this step, we define the MoCC via the definition of an *AutomataRelationDefinition*, see figure 2.1.

This MoCCML relation is mapped to the metamodel pattern by defining a *dse* program as presented in listing 2.1. This program binds the relation definition in the *Connector* context and also each *execute* event of the blocks from both side of the *Connector* are mapped to the *push* and *pop* clocks of the MoCCML relation.

The extended pattern is now composed of the $< AS, MoCCML, DSE >$ and if each of these elements change the complete pattern changes. Based on this pattern, we can create representative instances 4.4 where properties can be verified. A representative instance is a model that spreads a configuration of connectors and blocks that is structuraly relevant regarding the metamodel pattern.

On this representative model, we can verify properties tightly linked with the MoCCML semantics. These properties can be expressed as:

- (1)If all the input ports of a *Block* haven't enought data to consume then the *Block* canno't execute.
- (2)If the number of data of an output port is less than the capacity minus the current size then the *Block* can execute.
- (3)In any case, the current size of a Connector canno't exceed its capacity.

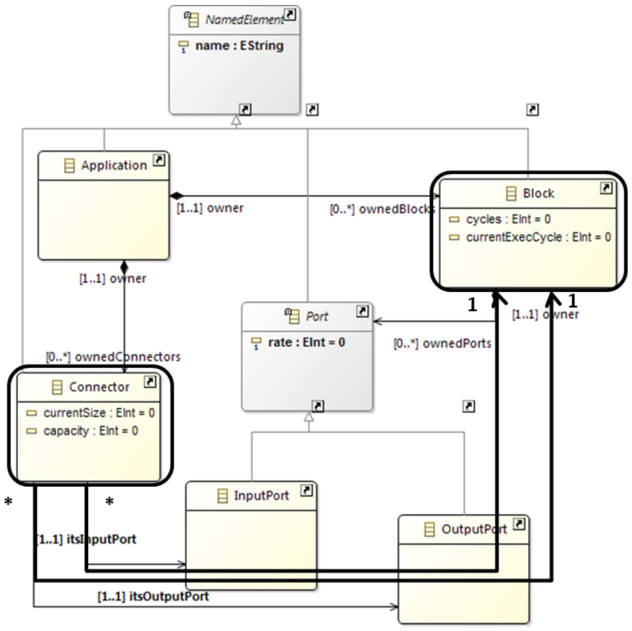


Figure 4.3: The metamodel pattern in *SigPML* metamodel

These properties are representatives of the MoCC and could be verified for every model. So we verify these properties on the representative model instance, to improve the trust on our extended pattern.

This approach based on the selection of a metamodel pattern completed by the MoCC definition can be applied on another metamodel pattern which is recurrent in modeling languages to define nested container of classifier, presented in the figure 4.5.

5. Exhaustive Exploration Gemoc Integration

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 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 and verify properties. 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 flow of figure 3.1 for exploration and verification .

5.1 Generating the input model formats for CLOCKSystem and OBP

There are several actions and model transformations that lead to the building of *exhaustive exploration* graph. The Figure 3.1 presents the first phase aiming at the generation of the input format for CLOCKSystem.

A first model transformation $T1$ takes as input the ECL mapping definition between the DSL and the MoCCML to generate a transformation ($T2$) 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. The transformation to Fiacre(named $T2$ in former D3.4.1 document) has been abandoned and replaced only with the former $T3$. Therefore processes modeling the behavior of a given system model are not part of the generated transformation. Now the current $T2$ instantiates the MoCCML

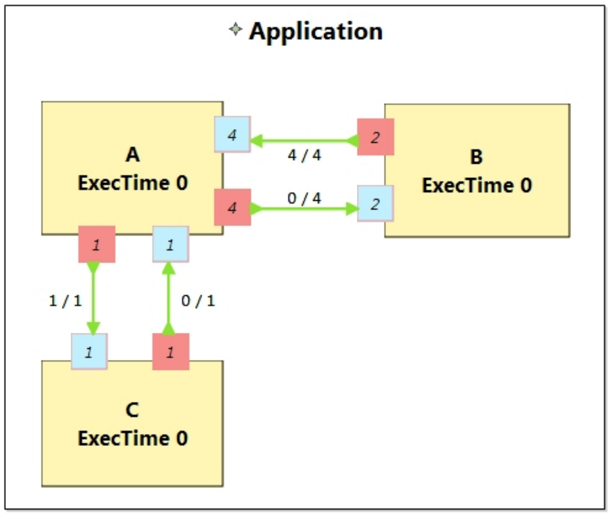


Figure 4.4: The representative model instance

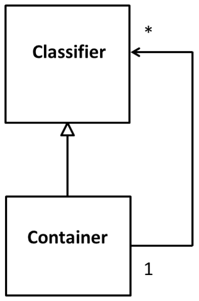


Figure 4.5: The nested container metamodel pattern

relation definitions described on the MoCC file and produces processes taking into account these MoCC properties applied on DSL instance model. $T2$ takes as input an system instance model of the system (DSL model). The models generated by $T2$ are taken as input in CLOCKSystem.

5.2 Executing CLOCKSystem

Execute ClockSystem action call is used to generate the LTS and the *exhaustive exploration*. It takes as input the MoCC instance model produced by $T2$. The file generated by $T2$ 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.

5.3 Validating properties

CDL can be used to express predicates which are invariants over states, observers to express invariants over execution traces and property patterns, for simplifying the expression of complex properties. Properties must rely on the instance model elements(states, processes etc). Once the property is defined the LTS used to build the *exhaustive exploration* can be composed with the property in OBP. The LTS corresponds to the input format for OBP (xobp.lts extension).

6. 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 and property verification. The tutorial goes through a simple significative example of three connected blocks implemented using the *SigPML* DSL as shown by Figure 6.1. 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 $T1$ should be called, then $T2$ is used to generate de CLOCKSystem input. The Screenshot in Figure 6.2 shows how to call the transformation $T1$ generating $T2$. Right-click on the ECL file $\rightarrow$ Exhaustive Exploration $\rightarrow$ Generate ClockSystem transformation from ECL model. The Transformation $T2$ is generated in the repository <clocksystem-gen>. The Screenshot in Figure 6.3 shows the content of the resulting $T1$, an acceleo template which requires a model of a DSL as input.

In the Modeling environment (*modeling workbench*) where the DSL instance model is realized, the transformation $T2$ takes as input the instance system model to generate the MoCC instance model described in CLOCKSystem specification format. Notice that currently you have to copy the <clocksystem-gen> repository previously generated in your project. As shown in 6.4 Right-click on the *SigPML* model (.sigpml) $\rightarrow$ Exhaustive Exploration $\rightarrow$ Generate ClockSystem file from DSL model. $T2$ generates the .clocksystem file corresponding to the MoCC-based specification model to take as input in CLOCKSystem.

The action for *exhaustive exploration* graph generation is called on this file. As illustrated in 6.5 right-click on the CLOCKSystem model (.clocksystem) $\rightarrow$ ClockSystem $\rightarrow$ Execute ClockSystem. The launched action generates a set of files representing the LTS outputs (.lts) of the MoCC instance model, but also several graphical format (.mtx , .gml) that can be visualized graphically and representing the *exhaustive exploration* for the instance model taking into account the MoCC relations. The picture 6.6 illustrates possible paths for the representative example.

The OBP tool requires two input files i.e the LTS generated from CLOCKSystem and the CDL model with properties.

In the figure **??** three properties are encoded in a CDL pseudo code(real names as been replaced by simple names to ease the understanding of the code).

- The size of channels between A and B canno't exceed the capacity of the connector. This is described with predicates p1 and p2 that check if the fifo size limit is reached.

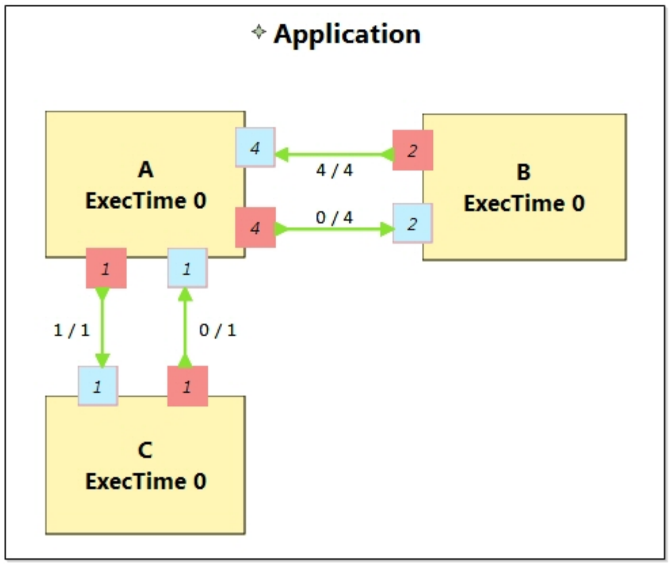


Figure 6.1: Example of communicating Blocks in *SigPML*

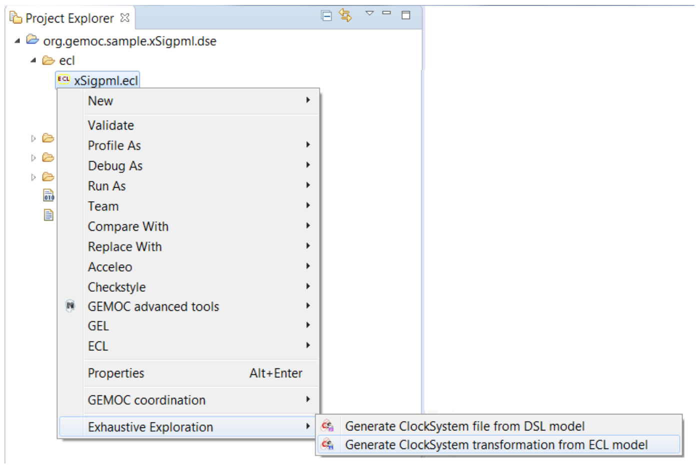
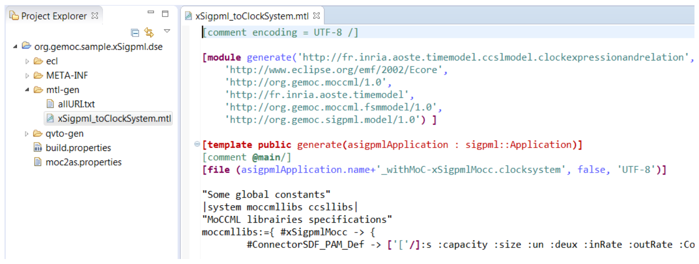


Figure 6.2: Generating $T1$



```
[comment encoding = UTF-8 /]

[module generate('http://fr.inria.aoste.timemodel.ccslmodel.clockexpressionandrelation',
    'http://www.eclipse.org/emf/2002/Ecore',
    'http://org.gemoc.moccml/1.0',
    'http://fr.inria.aoste.timemodel',
    'http://org.gemoc.moccml.fsmmodel/1.0',
    'http://org.gemoc.sigpml.model/1.0') ]

[template public generate(asigpmlApplication : sigpml::Application)]
[comment @main/]
[file (asigpmlApplication.name+'_withMoC-xSigpmlMocc.clocksystem', false, 'UTF-8')]

"Some global constants"
|system moccmllibs ccsllibs|
"MoCCML librairies specifications"
moccmllibs:={ #xSigpmlMocc -> {
        #ConnectorSDF_PAM_Def -> ['['/]:s :capacity :size :un :deux :inRate :outRate :Co
```

Figure 6.3: Result of $T1$

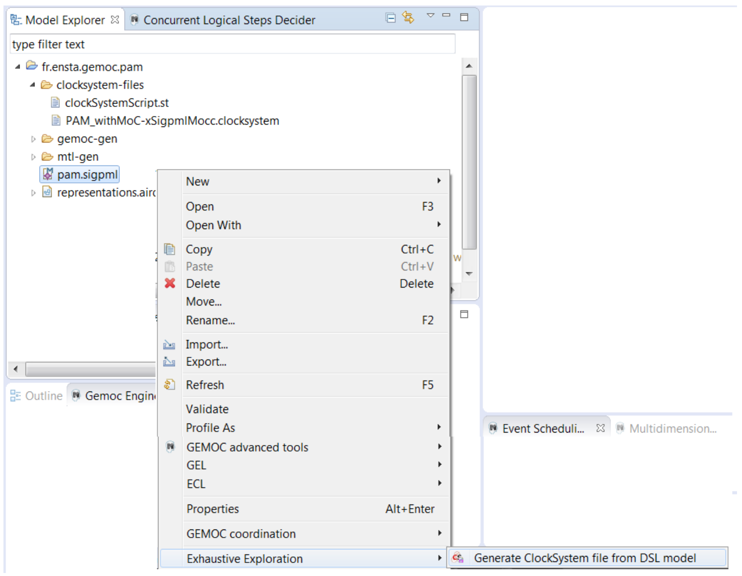


Figure 6.4: Generating $T2$

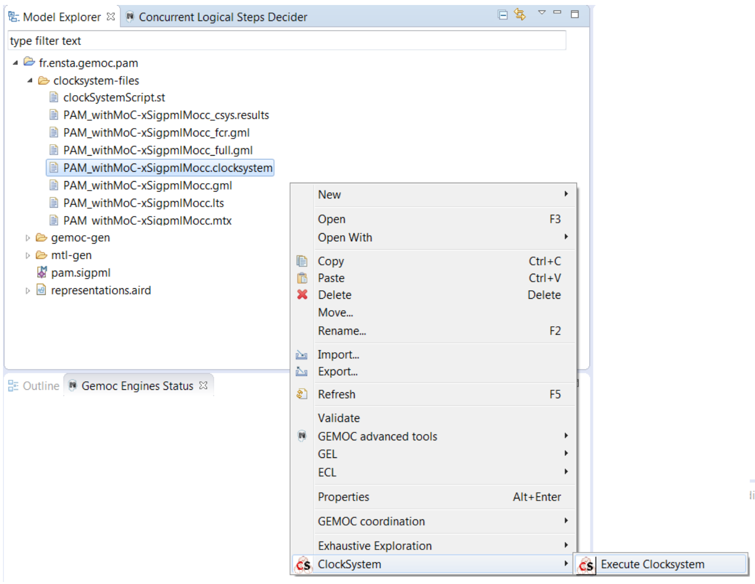


Figure 6.5: Execute CLOCKSystem

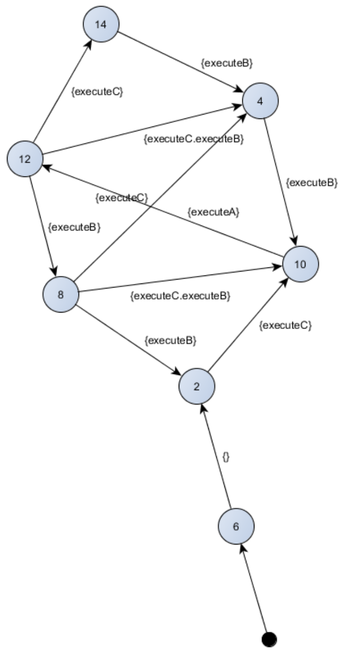


Figure 6.6: representative instance graph

- It is not possible to write in a *Connector* if it is full. This is checked via the observer automata o1 for one *Connector*. If the size of the connector plus the output rate exceed the size limit and if then Block execute(eB) the observer o1 goes to reject state.

7. Conclusion

This document described various developpements made in GEMOC STUDIO to enable *exhaustive exploration* in the context of modeling executable systems that are constrained with MoCCML relations. It also presented a reflexion on the kind of properties which can be relevant at different level of abstractions. An experimentation was made on a simple but representative model on which we have been able to verify properties related to SDF-semantics.

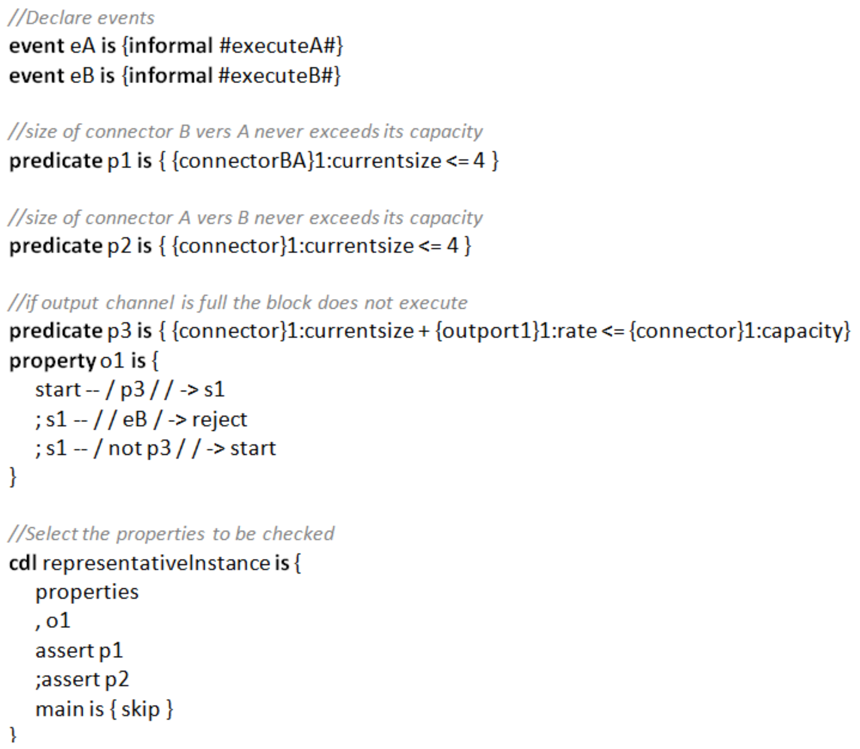
8. References

[1] Website. http://www.fim.uni-passau.de/fileadmin/files/lehrstuhl/brandenburg/projekte/gml/gml-technical-report.pdf.

[2] Matrix Market. Website. http://math.nist.gov/MatrixMarket/formats.html.

[3] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[4] 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.



```
//Declare events
event eA is {informal #executeA#}
event eB is {informal #executeB#}

//size of connector B vers A never exceeds its capacity
predicate p1 is { {connectorBA}1:currentsize <= 4 }

//size of connector A vers B never exceeds its capacity
predicate p2 is { {connector}1:currentsize <= 4 }

//if output channel is full the block does not execute
predicate p3 is { {connector}1:currentsize + {outport1}1:rate <= {connector}1:capacity}
property o1 is {
   start -- / p3 / / -> s1
   ; s1 -- / / eB / -> reject
   ; s1 -- / not p3 / / -> start
}

//Select the properties to be checked
cdl representativeInstance is {
   properties
   , o1
   assert p1
   ;assert p2
   main is { skip }
}
```

Figure 6.7: Pseudo code CDL for Sigpml

[5] 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.

[6] Hyun Cho and Jeff Gray. Design paterns for metamodels. In *Proceedings of workshop in SPLASH 2011*, pages 25–32, New York, NY, USA, 2011. ACM.

[7] Ciprian Teodorov. Embedding Multiform Time Constraints in Smalltalk, 2014.

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

[9] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean Marc Jezequel. Melange: A meta-language for modular and reusable devlopment of dsls. In *Proceedings of the 8th Software Language Engineering 2015*, New York, NY, USA, 2015. ACM.

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

[11] 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.

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

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

[14] 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.

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

[16] 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.

[17] 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.

[18] Ciprian Teodorov, Philippe Dhaussy, and Luka Le Roux. Environment-driven reachability for timed systems. *International Journal on Software Tools for Technology Transfer*, pages 1–17, 2015.

[19] Antti Valmari. Stubborn sets for reduced state space generation. In *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets: Advances in Petri Nets 1990*, pages 491–515, London, UK, UK, 1991. Springer-Verlag.

[20] Antti Valmari. The state explosion problem. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer Berlin Heidelberg, 1998.