Computing and Informatics, Vol. 32, 2014, ??–??, V 2016-Feb-29

TOWARDS A TRANSFORMATION APPROACH OF TIMED UML MARTE SPECIFICATIONS FOR OBSERVER-BASED FORMAL VERIFICATION

Nadia Menad[1], Philippe Dhaussy[2], Rachida Mekki[1], Zoé Drey[2]

[1] *University of Science and Technology of Oran Mohamed Boudiaf*
Department of Computer Science
Faculty of Mathematics and Computer Science, Algeria
[2] *UEB, Lab-STICC Laboratory UMR CNRS 6285*
ENSTA Bretagne, France
e-mail: `nadia.menad@univ-usto.dz, firstname.name@ensta-bretagne.fr`

Abstract.

Modeling timing constraints of distributed systems and multi-clock electronic systems aims to describe different time requirements aspects at a higher abstraction level. An important aspect is the logical time of the behavior of these systems. To model these time requirements, a specification language with multiple clock domains called *Clock Constraint Specification Language* (CCSL) has been introduced, in order to enrich the formalisms of existing modeling tools and also to facilitate the description and analysis of temporal constraints.

Once the software has been modeled, the difficulty lies in both expressing the relevant properties and in verifying them formally. For that, formal transformation techniques must be introduced. However, it remains difficult to exploit initial models as such, and to integrate them into a formal verification process.

This paper introduces a methodology and an original tool chain for exploiting UML MARTE models enriched with CCSL specification. These will be integrated together with a range of tools for expressing and verifying time constraints. We propose a more general translation approach that verifies not only CCSL constraints implementations but also properties of the complete model including all the functional components. We evaluate our approach with a case study.

Keywords: Formal verification; model-checking; CCSL time constraints; observer automata.

1 INTRODUCTION

In the field of modeling software architectures of control-command systems, distributed systems or multi-clock electronic systems, the specification of systems is often associated with temporal constraint specifications. These systems are often critical and the requirements to be respected during the modeling step, concern not only the determinism but also temporal constraints at a functional level. In the system development process, the designers look for methods and languages that allow them to describe their architectures, throughout the cycle and at various levels of abstraction. At each level, the modeling of such systems should allow the expression and the manipulation of time requirements, and the evaluation of the accuracy and efficiency of applications in terms of temporal and measurable requirements.

For this purpose, the concept of abstract modeling of logical clocks has been introduced with the *Clock Constraint Specification Language* (CCSL) [1] within MARTE [2] and has been adopted by the *OMG* [3]. CCSL is a language to define causal, chronological and temporal relationships. It aims to complement the existing formalisms and to provide models that can be analyzed so as to assess their accuracy with regard to requirements expressed by the designer. It is therefore essential to adopt temporal analysis approaches by integrating verification and validation processes based on robust formal notions, in order to meet current quality requirements of these systems.

To address this issue, several studies have proposed an engineering approach founded on models, and the use of semi-formal notations such as UML, enriched with formal notations. For example, the UML MARTE profile aims to express temporal constraints on UML models. The models that are built must not only be simulated but also formally analyzed so as to check the temporal requirements defined by the designer. However, few approaches integrate formal verification to UML MARTE models. In this study, we use *model-checking* verification techniques [4, 5] to enable formal analysis of requirements on UML MARTE models. These techniques have become highly popular due to their ability to automatically confirm software model properties . A range of model-checking tools have been developed for this purpose [6, 7, 8, 9, 10]. However, these techniques are difficult to exploit for both expressing and verifying the functional properties and the time properties of a software model under development.

This paper is an extension of a former publication [11] which studies the association of CCSL constraint specification and a model-checking tool named *Observer-Based Prover* (OBP)[1] [12]. The verifications carried out by OBP are based on the exploration of programs expressed in a high-level language dedicated to the specification of distributed systems called FIACRE [13] [2].

We have defined a transformation process to verify that UML MARTE mod-

[1] http://www.obpcdl.org

[2] For a detailed description the reader can refer to the FIACRE documentationhttp ://projects.laas.fr/fiacre/papers.php

els extended with CCSL specifications guarantee functional and time properties on their behavior. These properties are expressed as invariants and observer automata in a language called *Context Description Language* (CDL). The automatic transformation of UML MARTE models to FIACRE code is described in an other paper [14] and not detailed here. Only the transformation of CCSL properties is described in this article. Our contributions are as follow:

- We provide a verification methodology and tool-chain to facilitate the verification of distributed system models.
- We define a transformation approach to generate a FIACRE model from a CCSL specification integrated in UML MARTE models.
- We show how to specify time properties in the form of CDL observer automata, given as input of the OBP tool together with a specification of MARTE models.
- We provide a full description of a case study and we show a quantitative experiment of our verification process on this case study.

In this article, we integrate our transformation process into a general verification methodology that verifies not only CCSL constraints implementations, but also properties on the complete model including all the functional components. We provide a detailed description of a case study defined in MARTE.

This paper is organized as follows: Section 2 presents related work in formal verification of CCSL constraints. We present the context of our approach and the CCSL language in Section 3. An illustrative case study is presented in Section 4 and the principles of transformation of CCSL constraints into FIACRE are introduced in Section 5. Section 6 describes the CDL specification of properties based on observers and invariants (scenarios). In Section 7, we introduce and discuss some results of property proofs. We conclude in Section 8.

2 RELATED WORK

This section is conceptually divided into three parts. We have separated each part according to two criteria, which are : (1) the input specifications of the transformation process for formal verification, (2) taking into consideration the time specifications in the transformation process. The first part 2.1 concerns transformation approaches that deal with transforming a general specifications (UML, UML MARTE.etc) into an input language for formal verification purpose. The second part 2.2 deals with approaches dedicated to extending a formal language with logical time for verification. Finally, the last part 2.3 concerns approaches which concern only the transformation of CCSL specifications into a formal representation to be verified by a model checker tool.

2.1 Transforming a specification language into the input language for model checking tool

A number of model checking based techniques have been proposed for specifying and analyzing temporal constraints in several behavioral models, such as activity diagrams and state machines (e.g., [15, 16, 17, 18, 19, 20, 21]). In order to apply such techniques, the semi-formal specification models must be transformed to the tool-specific input languages. Many approaches to transform a specification language into an input language of adequate model-checking tool have been proposed. For example, *Ge et al.* present a verification-driven approach in [15] consisting in translating UML MARTE Activity Diagrams into *Time Transition System (TTS)* in order to guarantee the correctness of time properties. The authors use a formal semantics of *Time Petri Nets* (TPN) to avoid the state space explosion on the *TIme petri Net Analyzer (TINA)* model-checking tool. This approach is limited as it only verifies a particular type of properties (i.e., synchronization and schedulability).

In [16], *Ge et al.* present a framework dedicated to time property verification for UML MARTE. This work relies on a property-driven transformation from both UML architecture and behavior models to executable and verifiable *TPN* models, by translating time properties into a set of property patterns corresponding to *TPN* observers. Another approach to formalizing and validating temporal requirements is proposed by *Cimatti et al.* in [22]. A formalism for representing and analyzing requirements using the *NuSMV* [10] model checker is proposed, where temporal constraints are expressed by means of temporal operators, resulting in a fragment of first order temporal logic. The formalism builds on class diagrams, and combines fragments of first order logic with temporal operators to describe the evolution of scenarios. The drawback of the approach is that only a part of the scenario is considered to be controllable.

2.2 Extending a formal language with discrete time

Bosnacki et al. proposed a discrete time extension of *Promela* for concurrent systems in [23]. A variable named "*timer*" is defined and corresponds to the discrete time countdown *timer*. However, the proposed extension would be difficult to use in order to express the coincidence clock tickings. *Bosnacki et al.* also proposed another work, which can can be found in [24], which models time specifications from *Algebra of Communicating Processes, (ACP)* by macro definitions on the level of *Promela*. This work presents an extension of *Promela* and the *SPIN* model checker [6] with discrete time that provides an opportunity to model systems the correct functioning of which crucially depends on timing parameters.

2.3 CCSL language transformation for verification

Several approaches based on model transformation have been published to enable the formal verification of CCSL time requirements. This includes work by *Mail-*

let [25], who presented a technique for standard VHDL based design environment and synchronous languages (*Esterel*). This work addressed VHDL generating observers to check the compliance of a CCSL specification. The paper defines a state-based semantics for some CCSL operators based on labeled transition systems.

Peters et al. presented a work in [26] in which they propose to translate CCSL into *SystemC* which allows the simulation of the specified clocks in *SystemC.* The approach adds a TimeController, which corresponds to one further module for handling the clock behavior. The disadvantage of this approach is that the user has to implement the policy interface manually, which is not desired during the process of specification.

Yu et al. [27] propose a framework for translating CCSL specifications into dynamic systems, which are handled using the SIGALI [28] model-checker to verify specified constraint relations. This approach only focuses on the implementation of CCSL constraints with SIGNAL programs. Additionally, the handling of the non-deterministic parts cannot be chosen in this work, as polychrony can only be generated after eliminating the non-determinism. The approach is too restrictive, as it does not take into account mixed clock expressions, that deal with multiple future scheduled ticks. So the approach does not resolve all existing CCSL constraints. Moreover, the generated executable controller only enforces the satisfaction of the specified timing constraints without considering functional properties.

André [29] proposed an approach for implementing observers [30] for the formal verification of CCSL specifications. Observers, encoding CCSL constraints are translated into *Esterel* code. Mallet *et al.* [25] describe a technique to generate observers from a CCSL specification. In this approach, a reachability analysis allows the determination of whether or not an observer has reached an error state. The *Times Square Environment* [31], dedicated to solving CCSL constraints and computing solutions, implements a code generator in *Esterel.* The tool allows the detection of inconsistencies in CCSL specifications such as deadlocks. It provides the user with chronograms showing the different temporal evolutions of executions to facilitate the development of those specifications. However, the environment is a simulator that allows the analysis of execution traces but it does not verify the most general properties on execution graphs as do model-checkers.

Gascon *et al.* [32] contribute to the comparison of CCSL and *Property Specification Language, PSL* [33] expressiveness. They identify CCSL constructs that cannot be expressed in *PSL* temporal logic. The article also designates the class of *PSL* formulas that can be encoded in CCSL. It defines a translation between fragments of CCSL and *PSL* using the automata formalism as an intermediate representation. However, this approach does not take into account the CCSL constructs that cannot be expressed in *PSL.*

In [34], Yin *et al.* propose a translation of CCSL specifications into a *Promela* model to formally verify the CCSL constraints by the *SPIN* model checker. We have been inspired by this work to design the automatic translation of CCSL constraints into FIACRE automata. Also, in this approach the properties to be checked are expressed in *Linear Temporal Logic* (LTL) logic. The vUML [35] tool automatically converts UML state machines to *Promela* specifications and then invokes the *SPIN* model checker to verify the desired properties. Although the system is modeled as UML state machines, the temporal properties are specified in LTL. UML and OCL analysis tools, such as OCLE [36] and USE [37] provide support for validating structural properties. However, it is difficult to express specific time constraints with OCLE and USE, so these latters are limited when analyzing temporal properties. We propose to express properties with the CDL language with observer automata which allow a better expressiveness. For example, in our paper, we show a property (illustrated in Fig. 12) that would be tedious to express in *LTL*.

Another work by Romenska *et al.* [38] deals with CCSL unbounded operators. More precisely, it defines a state-based representation of CCSL operators. In this approach, a lazy evaluation is used to represent intentionally infinite transition systems, by providing an algorithm to make the synchronized product of such transition systems with studying its complexity.

Recent work has been proposed by Suryadevara *et al.* in [39], that presents a technique for transforming MARTE CCSL mode behavior into Timed automata for model-checking using the UPPAAL tool, which enables verification of both logical and chronometric properties of the system. This work only considers time specification and analysis. In contrast, we have proposed a more general translation approach that verifies not only CCSL constraints, but also functional properties. Furthermore, in our approach, these properties are separated from the application model thanks to a high level language (CDL) which facilitates the separation of the concerns. These properties, whether time-related or functional, are expressed in the CDL language [40]. In addition, we use the notion of contexts, also expressed in CDL , which in some cases allows to reduce the combinatorial explosion during the exploration of models.

3 PRINCIPLES OF THE APPROACH

The goal of our work is to integrate formal verification techniques with real-time systems described with semi-formal modeling languages such as UML, with respect to behavior and time requirements that must be guaranteed by such systems. To do so, we define an approach that is based on a tool-chain (Fig. 1) that bridges the gap between UML MARTE and formal specifications. In this Section, we first present our approach. We then provide a definition of time constraints specified by the CCSL language.

3.1 A tool-chain for the verification of models

The entry point of our tool-chain (Fig. 1) is a semi-formal specification model of a real-time system architecture (the structural model) and its behavioral model using the UML MARTE profile, a language that is dedicated to the specification of real-time systems. The UML MARTE models can be enriched with *time constraints* that describe causality (i.e., relations between the input/output states of a system), chronological and timing properties on the models. These constraints are described in a dedicated language called CCSL, which has been defined as an annex of the UML MARTE profile.

An UML MARTE model aims to describe the requirements that are written by the user for whom the real-time system model is designed. These requirements informally describe behavioral properties that should be guaranteed by the real-time system model.

Tooling of our approach

To be verified, the UML MARTE models need to be transformed into a formalism for which verification tools exist. To do so, our approach consists of a tool-chain (Fig. 1) that

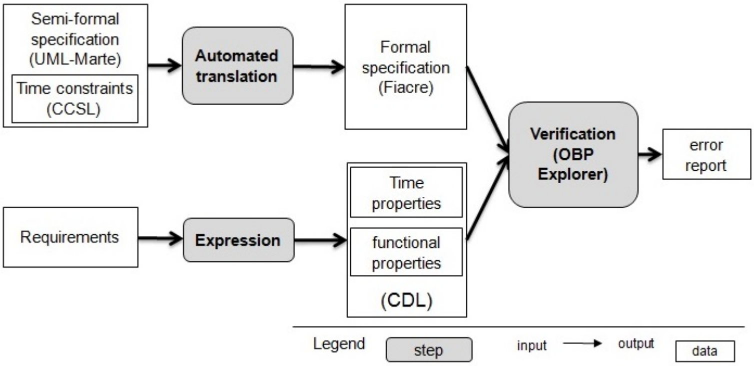

Fig. 1. Overview of the proposed model transformation.

leverages three elements: (1) *CDL*, a language to formally express requirements of the system, (2) *FIACRE*, a tool to model the system as a set of automata and (3) *OBP*, a tool to explore a system with respect to *CDL* properties.

- CDL (Context Description Language) is a language to formally specify properties on the behavior of a system. CDL defines high-level syntactic constructs that ease the expression of properties in terms of input/output states, representing scenarios of execution that must be verified by a FIACRE specification. In so doing, CDL contributes to bridging the gap between the user requirements and the formal model of a system.
- FIACRE is a language for defining state-machine based representations of real-time systems, aimed at being used as inputs for formal verification and simulation purposes. The choice of FIACRE for our approach is driven by (1) the need to formally describe real-time systems with a formal semantics, (2) the need to make models (in the sense of model-driven engineering) of systems amenable to verification.
- The OBP tool[3] is based on model-checking techniques and is aimed at the exploration of the execution states of a model. It takes CDL and FIACRE specifications as its inputs. OBP generates an exploration graph representing all the possible behaviors of a system, given some input data representing the environment with which the system should interact, and returns an error report (e.g., by means of counter-examples) when CDL properties are violated.

The UML MARTE specification is automatically translated into FIACRE together with the CCSL constraints with which it is associated. The translation from UML MARTE (without CCSL) to FIACRE is described in [14, 41] and is not the subject of this paper. The requirements are formalized into CDL properties. These properties are either functional (i.e., they express a functionality that the system must achieve), or time-related (i.e., they express timing constraints that the system should take into account when achieving a functionality).

[3] http://www.obpcdl.org

Finally both the generated FIACRE specification and the CDL properties are given as inputs to the OBP tool, which verifies the FIACRE specifications against the CDL properties. We use the OBP tool to conduct our experiments and verify that a system (initially described in MARTE) containing time constraints (expressed in CCSL) guarantees the expected requirements (expressed in CDL).

3.2 Time constraints and the CCSL Language

CCSL [1], introduced as an annex of MARTE, is a declarative language used to specify time constraints by means of binary relations between events based on logical clock concepts. CCSL is based on a mathematical model giving a formal semantics to time. In a MARTE model, any event (for example a communication, transmission or reception action, as computing start) may be used to define a time base, by means of logical clocks (*clock*). A *clock* represents a set of occurrences of discrete events, called *instants* .These instants are strictly ordered and provide a temporal reference.

For modeling distributed applications (distributed systems or digital circuits) it is necessary to identify a set of temporal or clock repositories and causal relations between events. Logical clocks can be referenced in the expression of temporal constraints to express causal relations such as synchronous or precedence constraints. These clocks can also be used to provide sampled clocks (sub-clocks) or filtering (see [29] for the specification of a set of relations). This vision of time allows the manipulation of the *simultaneity* notion in a succession of discrete instants [42]. In a given instant, events can be executed; these events are causally inter-dependent and considered to be simultaneous just like the *instant reaction* concept, an abstraction exploited in synchronous languages [43]. We recall briefly the formal bases of CCSL language in the remainder of this Section.

3.2.1 Formal foundations of CCSL language

Taking the formalization described in [42], a clock is considered to be a quintuplet $\langle I, \preceq, \mathcal{D}, \lambda, \mu\rangle$ where I is a set of instants, $\preceq$ is a binary, transitive ordered relation on I, $\mathcal{D}$ is a set of labels, $\lambda : \mathcal{I} \rightarrow \mathcal{D}$ is a labeling function, μ is a symbol, standing for a unit *unit*.

A *Time Structure* is a quadruplet $\langle \mathcal{C}, \mathcal{R}, \mathcal{D}, \lambda\rangle$ where $\mathcal{C}$ is a set of clocks, $\mathcal{R}$ is a relation in $\bigcup_{a,b\in\mathcal{C},a\neq b} (\mathcal{I}_a \times \mathcal{I}_b)$, with $\mathcal{I}$ an instant. $\mathcal{D}$ is a set of labels, $\lambda : \mathcal{I}_\mathcal{C} \rightarrow \mathcal{D}$ is a labeling function. $\mathcal{I}_\mathcal{C}$ a set of instants of the time structure, quotient of this set by the coincidence relation induced by the relation $\mathcal{R}$.

Relations between clocks are based on the precedence relation $\preceq$ which derives new relations from instants: Coincidence ($\equiv \overset{def}{=} \preceq \bigcap \preceq^{-1}$), Strict precedence ($\prec \overset{def}{=} \preceq \setminus \equiv$), Independence ($|| \overset{def}{=} \overline{\preceq \bigcup \preceq^{-1}}$), and Exclusion ($\# \overset{def}{=} \prec \bigcup \prec^{-1}$).

A CCSL specification consists of both a declaration of a set of clocks and a set of relations between clocks. These relations are applied to both clocks and the expressions referencing clocks. Each execution phase represents a possible evolution of clocks, according to the expressed relations. At each execution step (*step*), a set of clocks (or events) occurs (clock *tic*). At each execution step, the operational semantics of CCSL allows the evaluation of the conditions for which a clock can *tic*.

A relation between clocks is the generalization of relations between every instant of these clocks. The set of instants I is indexed by natural numbers in order to respect the order on I. Let $N^* = N - \{0\}$. $idx : I \to N^*$. $\forall i \in I$, idx(i) = k if and only if i is the k^{th} instant in I. For any discrete time clock $c = \langle I_c, \prec_c, \mathcal{D}_c, \lambda_c, \mu_c \rangle$, $c[k]$ denotes the k^{th} instant in I_c *i.e* $k = idx_c(c[k])$.

4 AN ILLUSTRATING CASE STUDY

We consider a data acquisition circuit (C), with two channels, consisting of acquisition components ($Sensor_i$ and Acq_i) ($i \in \{1, 2\}$), an acquired data processing component (*Comput*) and a filter (*Filter*) sampling the calculated values. Each acquisition channel i is associated with a pair of components $Sensor_i$ and Acq_i. We assume that, for each channel i, the component $Sensor_i$ receives data from the environment (from a device Dev_i outside the circuit) and transmits the value to Acq_i through a shared memory M_i. Each Dev_i sends N data $data_{ik}, k \in [0 \dots N-1]$. Acq_i provides *Comput* with each datum $data_{ik}$ via a synchronous communication [4] port $portAcq_i$.

Comput applies the addition of $data_{1k}$ and $data_{2k}$ respectively received from Dev_1 and Dev_2 and provides the *Filter* with the sum via a $fifo$. *Filter* provides the sampled data (one in every three values) to Dev_{out}, external to the circuit.

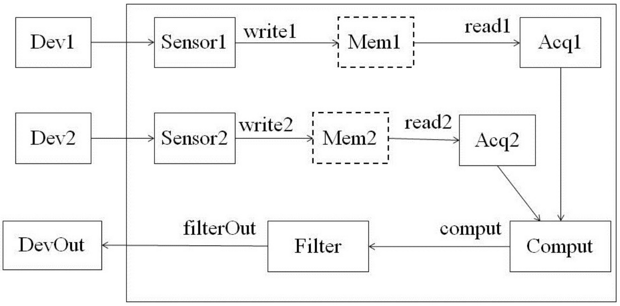

Fig. 2. Circuit architecture C.

The temporal requirements associated with this circuit are:

- *Req*1*a*: Each acquired datum $data_1$ is written in the memory M_1 before being read by Acq_1.
- *Req*1*b*: Each acquired datum $data_2$ is written in the memory M_2 before being read by Acq_2.
- *Req*2: *Comput* starts the calculation of a sum after two receptions of $data_ik$ from each Acq_i (with $i \in \{1, 2\}$).
- *Req*3: *Filter* provides the environment with a sampled value from a sequence of one in every three values calculated by *Comput*.

In summary, all the timing requirements for our case study, are specified using the CCSL language as follows:

[4] Synchronous communication uses passive port, it needs synchronization with other ports to initiate an interaction.

$$\begin{array}{ll}
write_1\ alternatesWith\ read_1 & (Req1a)\\
write_2\ alternatesWith\ read_2 & (Req1b)\\
read_1\ strictPrec\ comput & (Req2a)\\
read_2\ strictPrec\ comput & (Req2b)\\
filterOut\ =\ comput\ filteredBy\ (001)^w & (Req3)
\end{array}$$

In addition to the above time constraints, we express the requirements that are specifically associated with the expected behavior of the circuit. For example, we can express the following requirement:

- *Req*4 : the data $result_j, j \in [0 \ldots (N-1)/3]$ provided to the environment after the sampling operation (one value in 3) must have the values $data_{1k} + data_{2k}\ with\ k = (3*j)+2$.

4.1 The MARTE model

Fig. 3 illustrates the model of this circuit using the concepts of the MARTE profile. A package named *CircuitApplication* contains the description of the application circuit. We consider *Sensor1*, *Sensor2*, *Acq1*, *Acq2*. *Comput*, *Filter*, *Dev1*, *Dev2* and *DevOut* as active objects (stereotyped with *RtUnit*). Each of these units has a possibility to invoke other real-time actions, such as sending and receiving data.

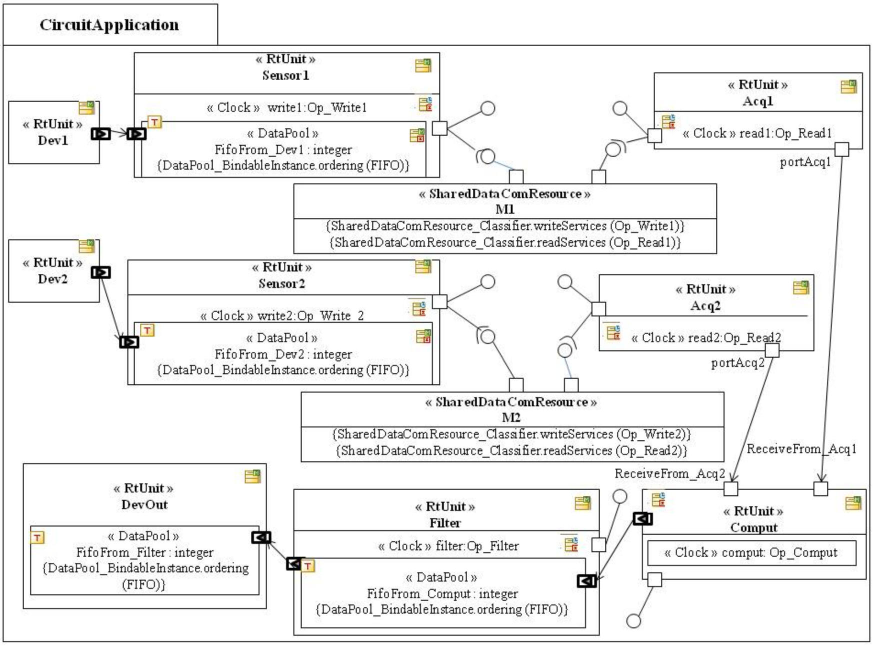
CircuitApplication
« RtUnit » Dev1
« RtUnit » Sensor1
« Clock » write1:Op_Write1
« DataPool » FifoFrom_Dev1 : integer {DataPool_BindableInstance.ordering (FIFO)}
« RtUnit » Acq1
« Clock » read1:Op_Read1
portAcq1
« SharedDataComResource » M1
{SharedDataComResource_Classifier.writeServices (Op_Write1)}
{SharedDataComResource_Classifier.readServices (Op_Read1)}
« RtUnit » Dev2
« RtUnit » Sensor2
« Clock » write2:Op_Write_2
« DataPool » FifoFrom_Dev2 : integer {DataPool_BindableInstance.ordering (FIFO)}
« RtUnit » Acq2
« Clock » read2:Op_Read2
portAcq2
« SharedDataComResource » M2
{SharedDataComResource_Classifier.writeServices (Op_Write2)}
{SharedDataComResource_Classifier.readServices (Op_Read2)}
ReceiveFrom_Acq1
ReceiveFrom_Acq2
« RtUnit » DevOut
« DataPool » FifoFrom_Filter : integer {DataPool_BindableInstance.ordering (FIFO)}
« RtUnit » Filter
« Clock » filter:Op_Filter
« DataPool » FifoFrom_Comput : integer {DataPool_BindableInstance.ordering (FIFO)}
« RtUnit » Comput
« Clock » comput: Op_Comput

Fig. 3. Circuit architecture

The *CircuitApplication* package also introduces two shared resources called *M1* and *M2*, stereotyped by *SharedDataComResource*. We define two read and write services that read/write the shared data as the objects *Op_Read1* and *Op_Write1*. *Comput* and *Acq1* (resp. *Acq2*) exchange data with synchronous communication. To specify this communication, we associate *Comput* with two ports *ReceivedFrom_Acq1* and *ReceivedFrom_Acq2*, and we associate *Acq1* (resp. *Acq2*) with the *portAcq1* (resp. *portAcq2*) port. We then connect *portAcq1* and *portAcq2* to *ReceivedFrom_Acq1* and *ReceivedFrom_Acq2*), respectively. When a computation is completed, the *Comput* object generates a datum that is transmitted to the *Filter* object through a dedicated port which is connected to a *dataPool* (contained in *Filter*) named *FifoFrom_Comput*. This *dataPool* is characterized by the attribute *ordering* which is set to *FIFO* value to specify the kind of the asynchronous communication.

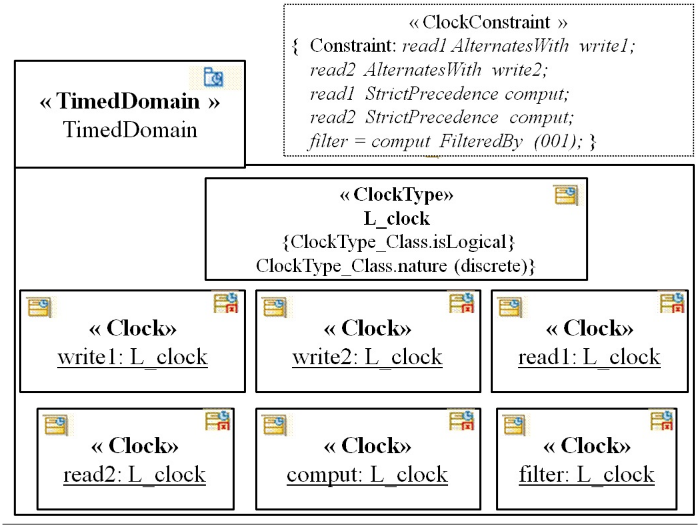
« ClockConstraint »
{ Constraint: read1 AlternatesWith write1;
read2 AlternatesWith write2;
read1 StrictPrecedence comput;
read2 StrictPrecedence comput;
filter = comput FilteredBy (001); }
« TimedDomain »
TimedDomain
« ClockType»
L_clock
{ClockType_Class.isLogical}
ClockType_Class.nature (discrete)}
« Clock»
write1: L_clock
« Clock»
write2: L_clock
« Clock»
read1: L_clock
« Clock»
read2: L_clock
« Clock»
comput: L_clock
« Clock»
filter: L_clock

Fig. 4. Timed Domain and CCSL specifications.

All the active objects provide real-time actions through their interface which carry operations stereotyped by *rtAction*[5] or *rtService*[6] (not shown in Fig. 3). These operations are defined in the *Op_Write1*, *Op_Write2*, *Op_Read1*, *Op_Read2*, *Op_Comput* and *Op_Filter* interfaces, bounded to dedicated ports. These ports also carry a *clock* stereotype (typed by clock specifications), indicating that the actions/services of the provided interface operations are considered as events which are invoked by those clocks. For instance, *Acq1* accesses the shared resource object (*M1*) by a reader service, invoking the reading real-time action. A boundary port can be connected to a port owning an interface so as to

[5] A real-time action is an action that specifies real-time characteristics. It defines a synchronization kind related to the execution of the action

[6] A real-time service is a service specialized with the additional pre-defined attributes of real-time constraints, which are applied for all the invocations of the rtService.

relay an action/service invocation or a data flow to a real-time unit. In that case, port directions are relayed as well.

4.2 The CCSL constraints

Once the application model has been described, we need to define a set of clock relations between the the different events of the system involved. These relations describe the desired access policies, e.g., preventing readers and writers to concurrently access the same resource. To do so and as illustrated in Fig. 4, a package named *TimedDomain* specifies a clock for each operation. Specifically, each clock is an instance of the *L_Clock* class, expressing a logical clock, which we defined with the MARTE *ClockType* stereotype. In this example, only one time domain is considered.

CCSL constraints are defined within the specific *ClockConstraint* MARTE stereotype. Each constraint defines a relation between the clocks as defined in the *TimeDomain* package. For example, an alternance between read and write operations is required in our application model. Such dependency between the read and write operations of *Sensor1* (resp. *Sensor2*) and *Acq1* (resp. *Acq2*), is specified by the *AlternatesWith* constraint. Another constraint (*FilteredBy*) indicates that *filter* is derived from *Comput*, with a pattern value equal to 001.

5 TRANSLATION OF MARTE-CCSL MODELS INTO FIACRE PROGRAMS

This Section presents the FIACRE language and the principles of the translation of MARTE models enriched with CCSL constraints into FIACRE programs.

5.1 Overview of the translation process

We present here the principles of the translation of MARTE models enriched with CCSL constraints (Fig. 5.a) into FIACRE code (Fig. 5.b). We illustrate these principles on the circuit architecture introduced in Section 4. Specifically, we detail how the CCSL constraints (*Req*1*a*, *Req*1*b*, *Req*2*a*, *Req*2*b* and *Req*3) are transformed into FIACRE processes.

The translation consists in generating the following FIACRE elements (Fig. 5.b): (1) a set of processes corresponding to the active objects of the MARTE model, (2) processes corresponding to the CCSL constraints, (3) a *Scheduler* process for synchronizing the execution of the generated processes and (4) a FIACRE component describing the architecture containing all the generated processes.

The translation method of CCSL constraints into FIACRE processes and the generation of the *Scheduler* process are inspired by the work described in [34]. In the next Sections, we detail how FIACRE code is generated.

5.2 Mapping a MARTE model into FIACRE

The transformation of the UML MARTE concepts into FIACRE constructs is summarized in Table 1. Note that we do not explain the complete transformation principle, which has

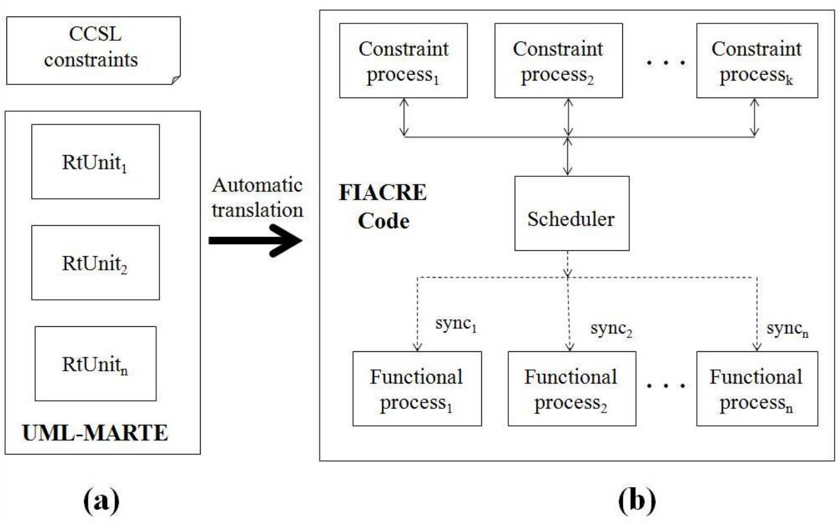

Fig. 5. Global view of the translation principles.

been the subject of another work (see [14, 41]). The active objects of the UML MARTE model (i.e., the *RtUnit* elements) correspond to the functional parts of the model. They are generated into FIACRE processes that we call *functional processes*. In our case study, the translation is applied to the following *RtUnit* elements: *Sensor1*, *Sensor2*, *Acq1*, *Acq2*, *Comput*, *Filter*.

The CCSL constraints that are attached to the MARTE model are also translated into FIACRE processes, named *constraint processes*. The *DataPool* elements (for the asynchronous communication) are translated into the FIACRE predefined queue data structure, and the Shared resource becomes a shared variable associated to the FIACRE processes corresponding to the involved *RtUnit* elements. The two ports used for the synchronous communication of two *RtUnit* elements are translated into the FIACRE predefined port constructs. Finally, the ports associated with an interface are translated into a FIACRE conditional statement that we detail in Section 5.4.

Table 1. Mapping from MARTE to FIACRE concepts.

MARTE	**FIACRE**
RtUnit	functional process
Clock Constraint	constraint process
DataPool	`queue` structure
SharedDataComResource	Shared variable
Synchronous port □⟶□	Synchronous communication port
Port with interface ○)⟶□	Triggering port

In our case study, the objects Dev_1, Dev_2 and Dev_{out} represent *actors* executed in the environment of the circuit. Their behavior is expressed in the CDL language, as we will see in Section 6.2.

5.3 Mapping a CCSL Time Constraint into FIACRE

We translate each CCSL constraint into a FIACRE process that implements the corresponding automaton. We call this process a *constraint process*.

These constraint processes are synchronized by a generated specific process, the *Scheduler*, which is described in Section 5.4. The *Scheduler* synchronizes the constraint processes via three ports, *start*, *update* and *end*, for the activation of the transitions in the constraint automaton. For example, in our case study, the transitions of *AlternatesWith* process are synchronized with the *Scheduler* via the ports *startA*, *updateA* and *endA*. *AlternatesWith* automaton updates the values of *clock_state* that allow the triggering of process $Sensor1$ and $Acq1$ (respectively $Sensor2$ and $Acq2$) by ports $sync\_pw1$ and $sync\_pr1$ (respectively $sync\_pw2$ and $sync\_pr2$) (cf Section 5.4.1).

The automaton corresponding to *AlternatesWith* constraint is shown in Fig. 6. The encoding principle for the two other constraints, *strict precedence* and *filtering* is similar.

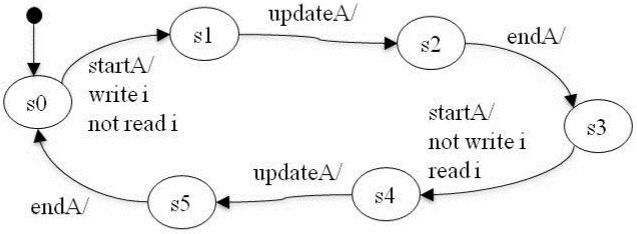

Fig. 6. Automaton for the constraint $write_i$ $alternatesWith$ $read_i$.

5.4 Interpreting Time Constraints with Scheduler

The role of the *Scheduler* process is to determine the order of execution of functional processes based on the constraint process state. The interpretation of time constraints is done through a Scheduler. It is in charge of triggering the functional processes according to the constraint states.

5.4.1 The scheduler and connection with processes

Fig. 7 is an excerpt of the FIACRE program generated for two functional processes (*Sensor1* and *Acq1*) and a constraint process (*alternatesWith*). Dash lines represent synchronization links. For example, *Sensor1* is synchronized with the *Scheduler* via the port $sync\_pw1$ to execute a writing operation of a given datum *data* in memory $M1$ shared between *Sensor1* and *Acq1* processes.

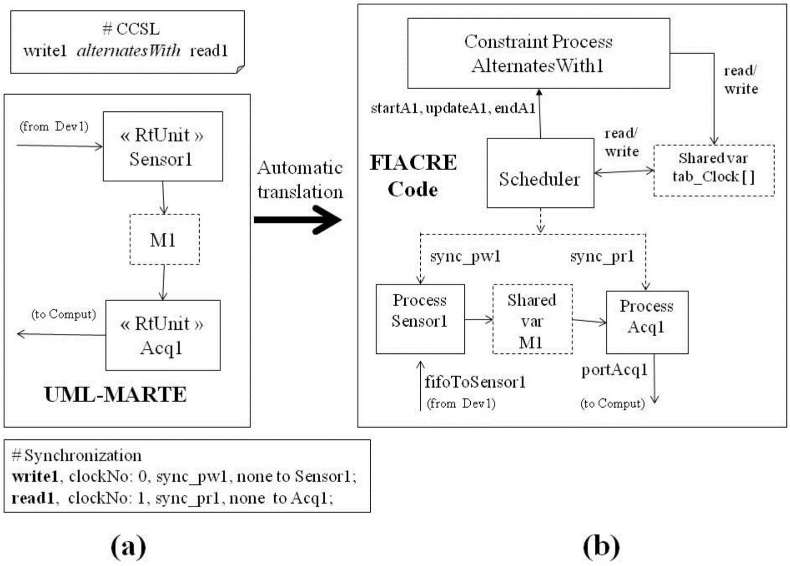

Fig. 7. Illustration of a part of the generated FIACRE program.

Acq1 and *Comput* communicate through the *portAcq*1 port with an integer value. *Comput* and *Filter* communicate through a shared variable *fifoFromComput* of the *fifo* type. *Filter* is synchronized with the *Scheduler* via *sync_filter* for filtering operation and *sync_filter* carries a boolean value required by the *Filter* behavior. The *Scheduler* process and constraint processes share logical clocks (*Table tab_Clocks* structure, cf Section 5.4.2) that correspond to events occurring in the circuit computation (*write*1, *write*2, *read*1, *read*2, *comput*, *filterOut*).

5.4.2 The scheduler behavior

The *Scheduler* process consists in an infinite loop. For each iteration of the loop, it executes four steps as shown in Fig. 8.(a): (1) the *Start* step for the declared clocks initialization and the activation of constraint processes. (2) the *End* step for the synchronization at the end of the constraint processes. (3) An active phase during which the *Scheduler* synchronizes with each functional process so that each process runs. (4) An intermediate phase *Update* is interposed between the *Start* steps and *End* steps to synchronize some constraints if required. An iteration is called an execution period and corresponds to the time between two *Start* steps. The algorithm executed by the *Scheduler* is repeated to simulate the coincident moment sequence (an *instant*). Interleaving or simultaneous execution of functional processes is simulated by synchronization between the *Scheduler* and the functional processes involved, at every temporally bounded instants. For example, Fig. 8.b shows two clocks *ck*1 and *ck*2 that are activated in each case at the same time. *ck*3 alternates with *ck*1 or *ck*2.

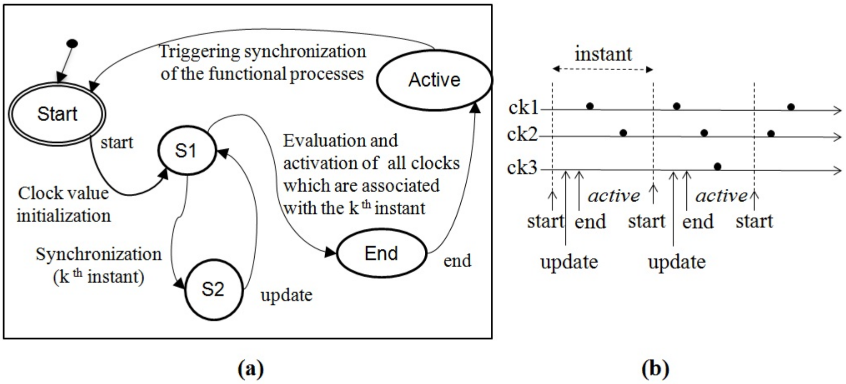

Fig. 8. Scheduler process automaton.

Each event in the model gives rise to a *clock* which is located by a FIACRE structure *tab_Clocks*. This structure is declared as follows (Listing 3):

```
type T_CLOCK is record clock_state:nat, enable_tick, dead: bool end
type T_ARRAY_CLOCK is array 7 of T_CLOCK
tab_Clocks: T_ARRAY_CLOCK
```

Listing 3. FIACRE declaration of structure *T_CLOCK*.

In each iteration of the *Scheduler*, each constraint process updates its variable *clock_state* which takes the integer values 0, 1 or 2, in accordance with the execution of the automaton it encodes. Once the process has executed a constraint, the *Scheduler* evaluates this variable and sets another variable *enable_tic* to either *true* or *false*. If *enable_tic* is evaluated as *true*, the functional process associated with the event is synchronized with the *Scheduler*, which triggers an execution step in the functional process (for example with *sync_pw*1 for triggering *Sensor1* as shown in Fig. 7). The assessment of the value *enable_tic* is set to *true* only if the *clock_state* value is equal to 2. In other cases, *enable_tic* are set to *false*. The value *dead* is set at *true* when the associated clock should not be active in the rest of the execution.

The *Scheduler* process that is generated thus includes the following code (Listing 4) which is executed during the *Active* step:

```
... if (tab_Clocks [0].enable_tick) then sync_pw1
elsif (tab_Clocks [1].enable_tick) then sync_pr1
elsif (tab_Clocks [2].enable_tick) then sync_pw2
elsif (tab_Clocks [3].enable_tick) then sync_pr2
elsif (tab_Clocks [4].enable_tick) then sync_comput
elsif (tab_Clocks [5].enable_tick) then sync_filter (true)
elsif (tab_Clocks [6].enable_tick) then sync_filter (false)
end ...
```

Listing 4. Excerpt of FIACRE code of the *Scheduler* for functional process synchronization.

5.5 Generated FIACRE architecture

Fig. 9 illustrates the FIACRE architecture of our case study, resulting from the translation of the MARTE /CCSL source model.

In our case study, the code generator produces 12 processes: the *Scheduler*, 5 constraint processes (2 for *alternatesWith*, 2 for *strictPrec*, 1 for *filterBy*) and 6 functional processes (*Sensor1*, *Sensor2*, *Acq1*, *Acq2*, *Comput* and *Filter*). Fig. 9 shows that the Scheduler process is connected to each functional process via synchronous communication ports called *triggering communications*. The *Scheduler* controls the execution of the connected functional processes and gives an *explicit rhythm* of execution of the different processes. The functional processes *Sensor1*, *Sensor2*, *Acq1*, *Acq2*, *Comput* and *Filter* are respectively synchronized with the *Scheduler* via *sync_pw1*, *sync_pr1*, *sync_pw2*, *sync_pr2*, *sync_comput* and *sync_filter* ports. For example, *Sensor1* and *Acq1* are respectively synchronized by the *Scheduler* for writing and reading operations in $M1$. Likewise, *Filter* is synchronized with the *Scheduler* via *sync_filter* for the filtering operation, where *sync_filter* carries a boolean value.

In addition, the *Scheduler* is connected to all the declared constraint processes via the shared *Table tab_clock*, in order to update the clock state values according to the constraints states to make a decision if such clock can tic or not in each specific instant.

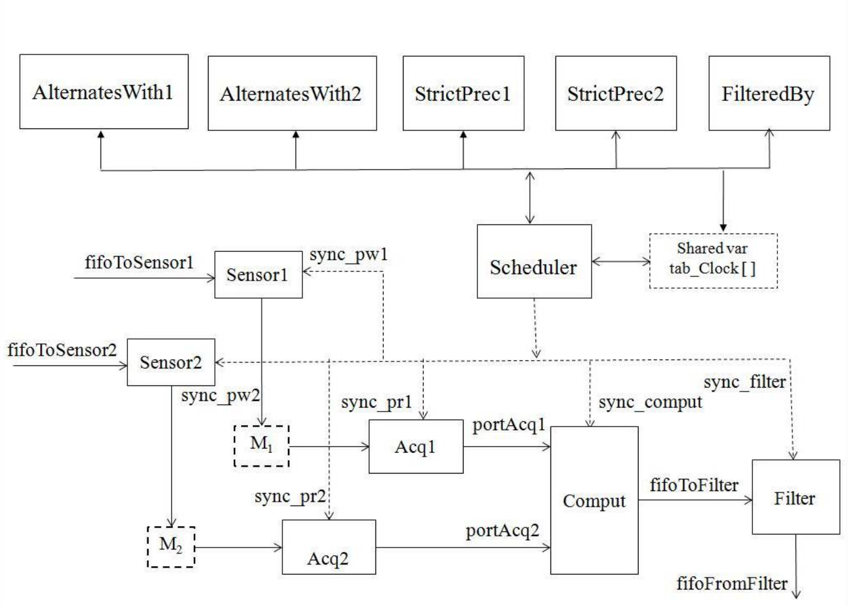

Fig. 9. Structure of the generated FIACRE code.

Generation of top level program

The processes representing MARTE elements, the CCSL constraints, and the interpretation of these constraints are finally instantiated in a FIACRE component called *C*, and specified as independent running entities through the || operator. The scheduler, the constraint processes and the functional processes are all synchronized through their communication ports.

As a result of our generation algorithm, the codes of functional processes, constraint processes and the *Scheduler* are built within *C*.

To enable automatic code generation, we must explicitly declare clock numbers (clockNo) and links between clocks and synchronization triggers provided by the *Scheduler*. For example, the *read*1 clock is associated with the *sync_pr*1 synchronization port to synchronize the first instance (*Acq:1*) of the *Acq* process. The $filter$ clock is associated with *sync_*$filter$ synchronization port which carries a boolean value. For this last clock, two clock numbers are declared, one for each boolean value. These attributes are specified as follows (Listing 5) [7].:

```
# Synchronization
write1: clockNo: 0, synchro: sync_pw1 none  to:  Sensor:1;
read1:  clockNo: 1, synchro: sync_pr1 none  to:  Acq:1;
write2: clockNo: 2, synchro: sync_pw2 none  to:  Sensor:2;
read2:  clockNo: 3, synchro: sync_pr2 none  to:  Acq:2;
comput: clockNo: 4, synchro: sync_comput none  to: Comput:1;
filterOut: clockNo: 5, synchro: sync_filter bool:true,
           clockNo: 6, synchro: sync_filter bool:false to: Filter:1;
```

Listing 5. Explicit declaration of clock numbers and links between clocks and synchronization triggers.

6 SPECIFICATION OF PROPERTIES AND CONTEXTS USING CDL

In order to check the requirements expressed for a real-time system model, it is necessary to specify them as formal properties that can be interpreted by the targeted model checker. Additionally, to validate the requirements, the environment in which the system is aimed to evolve may need to be described. In our approach, we use the CDL language (1) to express the properties of the system model as *observer automata*, and (2) to specify the interaction between the environment and the system model. In doing so, we are able to achieve two complementary objectives: one to verify that the implementation of CCSL constraints is correct, the other to ensure that the functional parts of the circuit (*Sensor1*, *Sensor2*, *Acq1*, *Acq2*, *Comput*, *Filter*) are properly implemented.

In this Section, we show how CDL properties are expressed for CCSL, and how to specify the interaction between the system model and its environment using contexts.

[7] The complete code of the case study can be found on the site http://www.obpcdl.org.

6.1 Properties associated with CCSL constraints

Here we illustrate the specifications of some properties associated with CCSL constraints included in our system model. The goal is to prove the correct FIACRE implementation of the Scheduler and constraint automata.

Alternance properties:

To verify the alternation requirements $Req1a$ and $Req1b$ described in Section 4, we declare the CDL events evt_write1, evt_read1, evt_write2 and evt_read2 (Fig. 10).

```
event evt_write1 is {sync sync_pw1 from {Scheduler}1 to {Sensor}1}
event evt_write2 is {sync sync_pw2 from {Scheduler}1 to {Sensor}2}
event evt_read1  is {sync sync_pr1 from {Scheduler}1 to {Acq}1}
event evt_read2  is {sync sync_pr2 from {Scheduler}1 to {Acq}2}
```

Listing 7. Declaration of CDL events.

With these events, we specify two properties $P1a$ and $P1b$: $P1a$ (resp. $P1b$) satisfies the alternating synchronization $write1$ and $read1$ (resp. $write2$ and $read2$). The CDL code of properties $P1a$ and $P1b$ is as follows (Listing 8):

```
property  P1a is {
start    --  / / evt_write1 / -> Sw;     // writing into M1
Sw       --  / / evt_read1  / -> start;  // reading
//----- errors ------
start    --  / / evt_read1  / -> reject;
Sw       --  / / evt_write1 / -> reject
}

property  P1b is {
start    --  / / evt_write2 / -> Sw;     // writing into M2
Sw       --  / / evt_read2 /  -> start;  // reading
//----- errors ------
start    --  / / evt_read2  / -> reject;
Sw       --  / / evt_write2 / -> reject
}
```

Listing 8. Declaration of CDL properties $P1a$ and $P1b$.

Both properties correspond to observers, as illustrated in Fig. 10. The initial state of the observer $P1a$ (resp. $P1b$) is the *Start* state and has an error state (*Reject*). Each transition of the observer is triggered by the occurrence of an event evt_write1 or evt_read1 (resp. evt_write2 or evt_read2).

The CDL language also allows the specification of predicates that can be verified during the exploration of the model. For example, if we want to check that clocks, in an instant, $write1$ and $read1$ (resp. $write2$ and $read2$) do not "tick" at the same instant, we first define the following predicates:

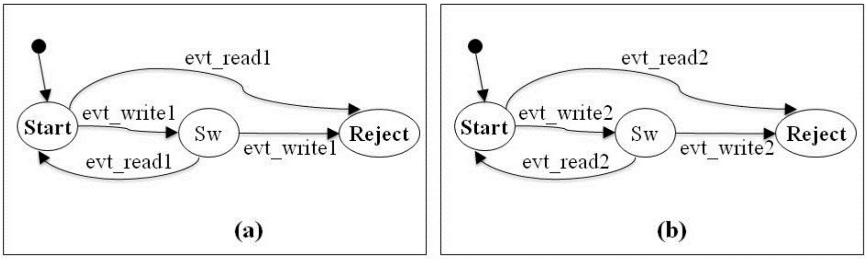

Fig. 10. Observer automata corresponding to properties P1a and P1b.

```
predicate enable_tick_pw1_true is
               {{C}1:tab_Clocks [0].enable_tick = true}
predicate enable_tick_pr1_true is
               {{C}1:tab_Clocks [1].enable_tick = true}
predicate enable_tick_rw1_together is
               {enable_tick_pw1_true and enable_tick_pr1_true}
predicate enable_tick_pw2_true is
               {{C}1:tab_Clocks [2].enable_tick = true}
predicate enable_tick_pr2_true is
               {{C}1:tab_Clocks [3].enable_tick = true}
predicate enable_tick_rw2_together is
               {enable_tick_pw2_true and enable_tick_pr2_true}
```

Listing 9. Declaration of CDL predicates.

We then declare the following invariants, using the *assert* operator[8]:

```
assert not act_tick_rw1_together
assert not act_tick_rw2_together
```

Listing 10. Declaration of CDL invariants.

During the exploration of the model, the OBP tool checks that the invariants are not violated.

Precedence properties:

In a similar way, we can specify observers to verify properties of the requirements *Req*2*a* and *Req*2*b* by declaring the event *evt_comput*:

```
event evt_comput is  {sync sync_comput from {Scheduler}1 to {Comput}1}
```

The CDL code of their corresponding properties *P*2*a* and *P*2*b* is as follows (Listing 11):

[8] See detailed syntax of the CDL language available at http://www.obpcdl.org.

```
property  P2a is {
start        --  / / evt_read1   / -> Sr;
Sr           --  / / evt_comput  / -> start;
    //------- error --------
start        --  / / evt_comput  / -> reject;
Sr           --  / / evt_read1   / -> reject
}
property  P2b is {
start        --  / / evt_read2   / -> Sr;
Sr           --  / / evt_comput  / -> start;
    //------- error --------
start        --  / / evt_comput  / -> reject;
Sr           --  / / evt_read2   / -> reject
}
```

Listing 11. Declaration of CDL properties $P2a$ and $P2b$.

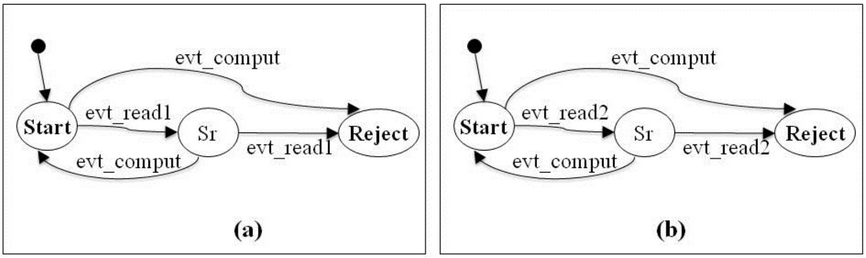

Fig. 11. Observer automata corresponding to properties P2a and P2b.

Filtering property:

The CDL predicates can also facilitate the writing of more complex observers when they refer to a large number of events. For example, the requirement *Req*3 associated with the generation of *data* by *Comput* and the filtering constraint is expressed by the CCSL term: $filterOut = comput\ filteredBy\ (001)^w$. During the exploration, we need to verify that the sequence of data generated from *Filter* is the sequence generated by *Comput* with a sampling of every third value. In the current version of the model, the filter word (001) is stored in an array variable $tabFilter$ of the constraint process *FilteredBy*. The $(i\ modulo\ 3)^{th}$ datum of the sequence generated by *Comput* will be copied in the sequence derived from *Filter* if the value $tabFilter[i\ modulo\ 3]$ is equal to 1. Otherwise, it is not copied into the sequence of data supplied to the environment. To verify this constraint, we therefore declare the following predicates (for $x \in \{0, 1, 2\}$):

```
predicate bitx_true  is {{FilteredBy}1:tabFilter[x] = 1}
predicate bitx_false is {{FilteredBy}1:tabFilter[x] = 0}
```

The transitions of an observer can be decorated with one of the predicates together with the events *evt_comput*, *evt_filterTrue* and *evt_filterFalse* which trigger the transitions; they are declared as follows:

```
event evt_filterTrue  is
                  {sync filter (true)  from {Scheduler}1 to {Filter}1}
event evt_filterFalse is
                  {sync filter (false) from {Scheduler}1 to {Filter}1}
```

Fig. 12 illustrates the observer encoding property $P3$ for requirement $Req3$ and referencing the above predicates and events.

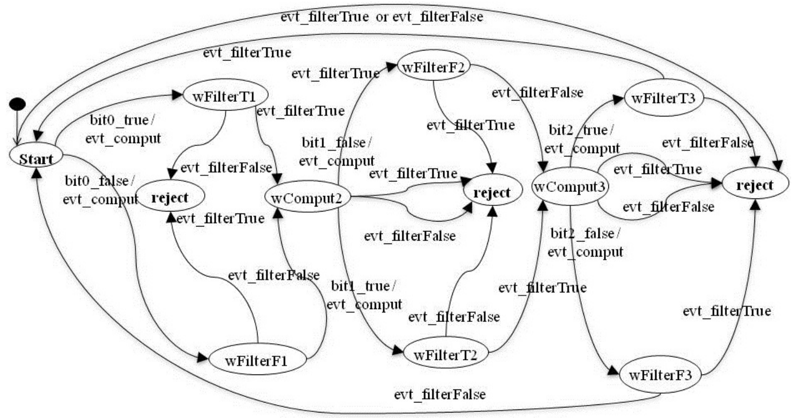

Fig. 12. Observer automaton corresponding to property $P3$ for requirement $Req3$.

A range of properties can be further specified on the behavior of our model. For example, the $Req4$ requirement, expressed in Section 4, can be expressed by an observer automaton using predicates and appropriate events.

6.2 CDL Context specification for case study

The core of the CDL language is based on the concept of **context**, which has an acyclic behavior communicating asynchronously with the system. The environment is specified through a number of such contexts. To illustrate CDL scenarios, we suppose that two devices Dev1, Dev2 each emit 3 values. Then we describe these interactions with CDL by (*event*) as follows:

```
  event evt_send_data1_sensor1 is {send DATA1 to {Sensor}1};
  event evt_send_data2_sensor1 is {send DATA2 to {Sensor}1};
  event evt_send_data3_sensor1 is {send DATA3 to {Sensor}1};
  event evt_send_data1_sensor2 is {send DATA1 to {Sensor}2};
  event evt_send_data2_sensor2 is {send DATA2 to {Sensor}2};
  event evt_send_data3_sensor2 is {send DATA3 to {Sensor}2};
```

Listing 13. Declaration of CDL events for scenarios.

These events allow the behavior of devices *Dev1, Dev2* to be specified with *activities* as follows:

```
activity Dev1 is
{
  event evt_send_data1_sensor1;
  event evt_send_data2_sensor1;
  event evt_send_data3_sensor1
}
activity Dev2 is
{
  event evt_send_data1_sensor2;
  event evt_send_data2_sensor2;
  event evt_send_data3_sensor2
}
```

Listing 14. Declaration of CDL activities Dev1 and Dev2.

The behavior of *DevOut* that receives three values of the *Filter* process is specified as follows:

```
event evt_recv_dataOut is {receive ANY from {Filter}1 to {env}1}
activity DevOut is { loop 3 {event evt_recv_dataOut} }
```

Listing 15. Declaration of CDL activity DevOut.

The context is finally described as follows:

```
cdl cdl_2dev is
{
 properties P1a, P1b, P2a, P2b, pty_FilteredBy
 assert  not act_tick_rw1_together;
 assert  not act_tick_rw2_together
 main is { Dev1 || Dev2 || DevOut }
}
```

Listing 16. Declaration of CDL context *cdl_2dev*.

cdl_2dev specifies that the environment is composed of 3 devices Dev1, Dev2 and DevOut. During the exploration by OBP , the properties $P1a$, $P1b$, $P2a$, $P2b$, *pty_FilteredBy*, *not act_tick_rw*1*_together* and *not act_tick_rw*2*_together* will be checked.

7 EXPERIMENTS AND DISCUSSION

To conduct the experiments for verifying properties of our case study, we use the OBP tool which has been developed in our group. The system model and the CDL specifications that we defined in the previous Section are fed to the OBP tool that generates a *Labeled Transition System (LTS)*. It is a state-transition graph that represents all the behaviors of the model, given input data representing the environment in which the model is intended to evolve. On this LTS, the verification of the properties is carried out by applying a reachability analysis of the reject/success states of the observers.

OBP is structured in three modules. The *front end* OBP imports FIACRE models corresponding to a translation of UML MARTE models including CCSL specifications. In addition, it imports CDL programs which describe the properties and context scenarios if required. *OBP Explorer* explores the model, and after each transition model run, it hands over to the *Observation Engine*. It captures the occurrences of events and evaluates the value of predicates and the status of all involved observers, at each step of the running model. A verification of all invariants and reachability analysis of error state observers is thus conducted.

At the end of exploration, a report is generated by OBP, revealing the list of properties evaluated to true or false. In addition, OBP provides counter examples when reaching a state of *reject* or when the invariant has been violated.

These indications may refer the user to the scenario having the defeated properties.

7.1 Result verification using the OBP tool

With CDL, we specified observers and invariants to express and verify a range of properties corresponding to requirements ($Req1a$ to $Req4$) as expressed in Section 4. In addition, we specified CDL scenario diagrams as shown in Section 6.2. From these diagrams, OBP generates a acyclic graph (called context graph) that represents all the possible interactions between the model and the environment. To verify properties, OBP composes the Circuit model with the context graph. Each property referenced in the CDL is checked on the result of this composition.

Table 2 shows the result of the OBP explorations. For example, we show the results if we consider three devices and 16 values issued by each device. It shows the number of LTS configurations and transitions that are generated during exploration by OBP. For this execution, we vary the fifo size shared between the environment and the Sensor components[9]. For example, for the case where the size of fifo is equal to 1, the number of explored configurations is then 744 592 and the number of transitions is 3 295 261.

With 16 values and a fifo size of 3, we notice a state explosion because of the explored configurations number and the limited memory of our computer. In other experiments, if we increase the number of devices (> 3), we notice an explosion in the number of configurations and transitions. In this case, the analysis of the properties cannot be brought to completion.

[9] All tests are run with a machine such as Windows 32-bit - 10 GB RAM with OBP vers.1.4.5.

Table 2. Complexity with 3 devices and 16 values received from the environment.

Fifo size	Number of configurations	Number of transitions
1	744 592	3 295 261
2	3 328 269	17 797 040
3	Explosion	Explosion

7.2 Handling the complexity with automatic splitting

The principal challenges regarding software verification of real-time systems deal with devising solutions that scale up to the increasing complexity of these systems. Actually, resources to verify these systems are limited, both in terms of time and memory size. We treat in this section how our approach attempts to take into account the larger models corresponding to the sizes of industrial models.

The context-aware Verification (CaV) has been proposed [44, 45, 46] and offers a solution for addressing some of these issues. With CaV, independent contexts are exploited by the verification tools by using new algorithms for fighting the state space explosion problem. In this section, we present a context driven reachability algorithm for automatically partitioning contexts [44, 45].

OBP integrates a powerful context-guided state-space reduction technique which relies on the automated recursive partitioning (splitting) of a given context in independent sub-contexts [12]. This technique is systematically applied by OBP when a given reachability analysis fails due to lack of memory resources to store the state-space.

The idea is to automatically split each identified context into a set of k smaller sub-contexts $context_i$. After splitting, each sub-contexts $context_i$, is composed with the model for exploration and the properties associated with $context_i$ are checked on the resulting global system. If the exploration fails with a $context_i$, it is automatically and recursively partitioned into a set of sub-contexts. Actually, we transform the global verification problem into k smaller verification sub problems. In our approach, the complete context model can be split into pieces that have to be composed separately with the system model (see the details in [12]). This technique allows models with a greater number of ranged states to be explored.

For example, Table 3 shows results in the case of three channels with a transmission of 16 different values. Without context splitting, the exploration does not end. Then applying the partitioning of contexts, we obtain the following results: The behavior of the environment is partitioned into four sub-contexts. The number of explored configurations (cumulative) is then 27 564 280 and 159 993 196 transitions.

These results show us that we can contain the combinatorial explosion with this specific technique of model exploration with splitting. OBP tool can provide an assessment of the validity of the property, even with a machine having a limited memory size (10 GB). If the model has a number of behaviors compatible with an exhaustive search, the verification results can be obtained without using the partitioning technique implemented in OBP. If the circuit is connected with an interacting environment, partitioning may be

Table 3. Complexity with 3 devices and 16 values received from the environment with Splitting.

Splitting	Fifo size	number of configurations	number of transitions	number of sub-contexts
yes	3	27 564 280	159 993 196	4

used. However, this is not always the case if the circuit has a large number of behaviors that do not interact with an environment. In this case, the method cannot be applied. Therefore it is necessary to have a machine with a larger memory, or to focus on parts of the model.

Table 4 shows exploration results in another case: 6 devices, fifo size equal to 3 and 3 values received from the devices. Without splitting, there is a state explosion because of the limited memory of our computer. With splitting and 7 sub-contexts, the checking is possible.

Table 4. Complexity with 6 devices, fifo size equal to 3, and 3 values received from the environment.

Splitting	LTS configurations	LTS transitions	sub-contexts
no	Explosion	-	-
yes	77 225 206	607 639 474	7

We will not go into more detail here as this has already been published [12] [46]. The necessity of clear methodologies (e.g., the splitting process) has also to be identified, since the context partitioning is still not trivial, i.e., it requires the formalization of the context of the subset of functions under study (out of scope of this paper). Therefore, an associated methodology must be defined to help users for modeling contexts.

7.3 Discussion

Now, the question is: Is the proof still relevant (applicable) to a number greater than 16 values ? In the above part we argue that the correctness of the properties is always verified for all natural numbers sent from the given devices. To demonstrate the correctness of the generated FIACRE model, we take the following reasoning: The behavior of the FIACRE processes does not depend on the acquired input values by Sensor processes. To achieve this goal, we introduce a stand alone code version in order to point out that the behavior of the circuit model involved is not exchangeable whatever the values sent from the environment. We transform our model by integrating two actors Dev1 and Dev2 in the model as FIACRE processes. Both processes iterate on the sending of the same value. The complexity of the exploration of the model is then 45 328 configuration and 168 664 transitions. So, we can check properties with observers and invariants described above.

If we increase the number of values to over 16, the complexity increases but the same configurations will result. We show here that if the sent values are constants, and in infinite time, the configuration of the exploration graph still finite. Moreover, the behavior of the circuit is independent from the input data values. This proves that we can verify all the expressed properties in this graph independently from the data values.

8 CONCLUSION

In this work, we have defined an automatic translation approach to generate FIACRE programs from UML MARTE models enriched with CCSL constraints. This approach allows the formal verification of the implementation of CCSL constraints and functional requirements.

We carried out a verification technique of properties by model-checking using the CDL language and the OBP tool. Functional as well as temporal properties can be easily expressed in CDL as predicates and observers which are checked during the exhaustive model exploration by OBP. We have shown that this language facilitates the expression of properties. They can be expressed with a very fine granularity, referencing variables and process states.

Our CDL language can be compared with the *Property Specification Language* (PSL) [32]. In future work, we aim to compare CDL expressiveness with PSL and the discussion in [32] is very interesting on this topic. Additionally, we are currently working to facilitate the interpretation of data provided by OBP and to display understandable data in the user's models, allowing ease of diagnosis.

We can take advantage of the CCSL automata encoding. These automata are reusable inputs to apply the verification. Our translation approach can be an important step towards the formal verification process of both MARTE models and CCSL specifications. Once the translation of CCSL constraints into FIACRE is completed, the operation requires only a single verification as it does not depend on the modeled application. Even though the model may change, the FIACRE code is reusable as this translation principle is independent of the application. We think that our approach contributes to clarifying its role when addressing this domain by expressing temporal properties dedicated to CCSL relation constraints.

REFERENCES

[1] C. André, "Syntax and semantics of the Clock Constraint Specification Language CCSL," Tech. Rep. 6925, INRIA, 2009.

[2] F. Mallet, C. André, and R. D. Simone, "CCSL: Specifying clock constraints with UML/MARTE," in *ISSE*, vol. 4, pp. 309–314, 2008.

[3] OMG, "Uml profile for marte, v1.1," in *Object Managment Group*, (Document number: PTC/10-08-32), August 2010.

[4] J.-P. Queille and J. Sifakis, "Specification and verification of concurrent systems in cesar," in *Proceedings of the 5th Colloquium on International Symposium on Programming*, (London, UK), pp. 337–351, Springer-Verlag, 1982.

[5] E. Clarke, E. Emerson, and A. Sistla, “Automatic verification of finite-state concurrent systems using temporal logic specifications,” *ACM Trans. Program. Lang. Syst.*, vol. 8, no. 2, pp. 244–263, 1986.

[6] G. Holzmann, “The model checker SPIN,” *Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[7] K. G. Larsen, P. Pettersson, and W. Yi, “UPPAAL in a Nutshell,” *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1-2, pp. 134–152, 1997.

[8] B. Berthomieu, P.-O. Ribet, and F. Verdanat, “The tool TINA - Construction of Abstract State Spaces for Petri Nets and Time Petri Nets,” *International Journal of Production Research*, vol. 42, pp. 2741–2756, July 2004.

[9] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu, “Cadp: A protocol validation and verification toolbox,” in *CAV ’96: Proceedings of the 8th International Conference on Computer Aided Verification*, (London, UK), pp. 437–440, Springer-Verlag, 1996.

[10] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, “Nusmv: a new symbolic model checker,” *Int. J. on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 410–425, 2000.

[11] N. Menad and P. Dhaussy, “A transformation approach for multiform time requirements,” in *11th International Conference on Software Engineering and Formal Methods (SEFM’13), Madrid, Spain*, vol. 8137, pp. 16–30, Lecture Notes in Computer Science, September 2013.

[12] P. Dhaussy, F. Boniol, J.-C. Roger, and L. Leroux, “Improving model checking with context modelling,” *Advances in Software Engineering*, vol. ID 547157, p. 13 pages, 2012.

[13] P. Farail, P. Gaufillet, F. Peres, J.-P. Bodeveix, M. Filali, B. Berthomieu, S. Rodrigo, F. Vernadat, H. Garavel, and F. Lang, “FIACRE: an intermediate language for model verification in the TOPCASED environment,” in *European Congress on Embedded Real-Time Software (ERTS)*, (Toulouse), SEE, january 2008.

[14] F. Jouault, C. Teodorov, J. Delatour, L. L. Roux, and P. Dhaussy, “Transformation de modeles UML vers FIACRE, via les langages intermediaires tUML et ABCD,” in *Revue Genie Logiciel*, no. 109, June 2014.

[15] M. P. Ning Ge and X. Crégut, “Time properties dedicated transformation from UML-MARTE activity to time transition system.,” pp. 37(4):1–8, 2012.

[16] N. G. Marc Pantel and X. Cregut, “A framework dedicated to time properties verification for UML-MARTE specifications.,” 2012.

[17] L. Ribeiro, O. M. dos Santos, F. L. Dotti, and L. Foss, “Correct transformation: From object-based graph grammars to PROMELA,” *Sci. Comput. Program.*, vol. 77, no. 3, pp. 214–246, 2012.

[18] R. K. Poddar and P. Bhaduri, “Verification of giotto based embedded control systems,” *Nord. J. Comput.*, vol. 13, no. 4, pp. 266–293, 2006.

[19] V. Bertin, E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine, “Taxys = esterel + kronos – a tool for verifying real-time properties of embedded

systems, proceedings of the 40th ieee conference on decision and control," 2001.

[20] A. G. R. Mauricio Gonçalves, Vieira Ferreira, "An approach to model-driven architecture applied to space real-time software.," 2012.

[21] P.-F. Yves Sorel, Marie-Agnes, "From high-level modelling of time in MARTE to real-time scheduling analysis.," pp. 129–144, 2008.

[22] A. S. Alessandro Ciçmatti, Marco Roveri and S. Tonetta., "Formalizing requirements with object models and temporal constraints," pp. 10(2):147–160, 2011.

[23] D. Bosnacki and D. Dams, "Integrating real time into spin: A prototype implementation.," in *Formal Description Techniques and Protocol Specification, Testing and Verification, FORTE XI / PSTV XVIII'98, IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XI) and Protocol Specification, Testing and Verification (PSTV XVIII)*, pp. 3–6 November, 423–438. 1998, Paris, France.

[24] D. Bosnacki and D. Dams, "Discrete-time Promela and Spin," in *Formal Techniques in Real-Time and Fault-Tolerant Systems, 5th International Symposium, FTRTFT'98, Lyngby, Denmark, September 14-18, 1998, Proceedings*, pp. 307–310, 1998.

[25] F. Mallet, "Automatic generation of observers from MARTE/CCSL," in *RSP 2012 - International Symposium on Rapid System Prototyping*, (Tampere, Finlande), pp. 86–92, IEEE, october 2012.

[26] J. Peters, R. Wille, and R. Drechsler, "Generating SystemC implementations for clock constraints specified in UML/MARTE CCSL," in *2014 19th International Conference on Engineering of Complex Computer Systems, Tianjin, China, August 4-7, 2014*, pp. 116–125, 2014.

[27] H. Yu, J.-P. Talpin, L. Besnard, T. Gautier, H. Marchand, and P. L. Guernic, "Polychronous controller synthesis from MARTE CCSL timing specifications," in *Memocode*, 2011.

[28] B. Dutertre, *Specification et preuves de systemes dynamiques.* PhD thesis, 1992.

[29] C. André, "Verification of clock constraints: CCSL observers in Esterel," Tech. Rep. 7211, INRIA, 2010.

[30] N. Halbwachs, F. Lagnier, and P. Raymond, "Synchronous observers and the verification of reactive systems," in *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93* (M. Nivat, C. Rattray, T. Rus, and G. Scollo, eds.), (Twente), pp. 83–96, Workshops in Computing, Springer Verlag, June 1993.

[31] J. DeAntoni, F. Mallet, and C. André, "Timesquare: on the formal execution of UML and DSL models," in *Tool session of the 4th Model driven development for distributed real time systems*, 2008.

[32] R. Gascon, F. Mallet, and J. Deantoni, "Logical time and temporal logics: Comparing UML MARTE/CCSL and PSL," Tech. Rep. ISBN: 978-0-7695-4508-0, INRIA, 2012.

[33] IEEE, "IEEE Standard for property Specification Language (PSL)," Tech. Rep. 1850, 2005.

[34] L. Yin and F. Mallet, “Correct transformation from CCSL to PROMELA for verification,” Tech. Rep. 7491, INRIA, 2011.

[35] J. Lilius and I. Paltor, “vuml: A tool for verifying UML models,” in *ASE*, pp. 255–258, 1999.

[36] D. Chiorean, M. Pasca, A. Cârcu, C. Botiza, and S. Moldovan, “Ensuring UML models consistency using the OCL environment,” *Electr. Notes Theor. Comput. Sci.*, vol. 102, pp. 99–110, 2004.

[37] M. Gogolla, F. Büttner, and M. Richters, “USE: A UML-based specification environment for validating UML and OCL,” *Sci. Comput. Program.*, vol. 69, no. 1-3, pp. 27–34, 2007.

[38] Y. Romenska and F. Mallet, “Lazy parallel synchronous composition of infinite transition systems,” in *ICTERI*, pp. 130–145, 2013.

[39] J. Suryadevara, C. C. Seceleanu, F. Mallet, and P. Pettersson, “Verifying MARTE/CCSL mode behaviors using UPPAAL,” in *SEFM*, pp. 1–15, 2013.

[40] P. Dhaussy and J.-C. Roger, “Cdl (context description language) : Syntax and semantics,” tech. rep., ENSTA-Bretagne, 2011.

[41] F. Jouault and J. Delatour, “tUML : syntax and semantic,” tech. rep., ESEO, 2014.

[42] C. André, “Le temps dans le profil UML MARTE,” Tech. Rep. ISRN I3S/RR-2007-19-FR, Laboratoire I3S, 2007.

[43] A. Benveniste and G. Berry, “The sysnchronous approach to reactive and real-time systems,” Tech. Rep. RR-1445, INRIA, 1991.

[44] P. Dhaussy, P.-Y. Pillain, S. Creff, A. Raji, Y. L. Traon, and B. Baudry, “Evaluating context descriptions and property definition patterns for software formal validation,” in *12th IEEE/ACM conf. Model Driven Engineering Languages and Systems, Models* (B. S. Andy Schuerr, ed.), vol. 5795, pp. 438–452, Springer-Verlag, LNCS, 2009.

[45] P. Dhaussy, F. Boniol, and J.-C. Roger, “Reducing state explosion with context modeling for model-checking,” in *13th IEEE International High Assurance Systems Engineering Symposium, Hase*, (Boca Raton, USA), 2011.

[46] L. L. Teodorov Ciprian and D. Philippe, “Context-aware verification of a cruise-control system,” in *h International Conference on Model and Data Engineering (MEDI)*, (Larnaca, Cyprus), September 2014.