Generating SystemC Implementations for Clock Constraints Specified in UML/MARTE CCSL

Judith Peters[1] Robert Wille[1,2] Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{jpeters,rwille,drechsler}@informatik.uni-bremen.de

***Abstract*—Due to the increasing complexity of today's embedded systems, the design on higher levels of abstraction becomes more and more important. In this context, modeling languages such as UML and its profile MARTE received significant attention in the recent past. They provide formal descriptions that can be exploited to automatically generate initial implementations of a system e. g. in SystemC. While corresponding approaches have been developed in the past, they often focused on functional specifications. Besides that, also non-functional behavior such as clocking constraints needs to be considered in this process. In this work, we propose an approach which addresses this gap. Given a formal specification of clocking constraints specified in the Clock Constraint Specification Language (CCSL; a MARTE accessory), we propose an automatic code generation scheme which enriches an existing SystemC implementation by a module triggering the desired clocks in the system.**

I. Introduction

Today's embedded systems are composed of a significant amount of components such as gates and signals. At the same time, they are enriched with additional sensors and actors eventually forming a so called cyber-physical system. As a consequence, such systems have reached an intrinsic complexity which led to serious challenges for their design and implementation. This development is also reflected in the design flow which accordingly evolved over the last decades.

While a textual specification always has been the starting point of the design process, the actual implementation was and still is conducted on several abstraction levels. The *Electronic System Level* (ESL) is considered the highest implementation level thus far. Here, languages such as SystemC are applied which provide description means for both software and hardware. This allows for the realization and simulation of respective components prior to the hardware/software partitioning. Afterwards, the resulting hardware components are precisely implemented as descriptions on the *Register Transfer Level* (RTL) using languages like VHDL or Verilog. From these descriptions, the respective hardware blocks are realized through the gate level, transistor level, etc., to an actual chip.

The design tasks in the lower abstraction levels have received significant improvements e. g. by automatic methods for synthesis, optimization, and verification. However, the initial implementation of an ESL description from a given (textual) specification still constitutes a serious gap in today's design flow and usually is performed manually.

In order to close this gap and inspired by corresponding developments in software engineering, researchers and designers started the consideration of modeling languages such as the *Unified Modeling Language* (UML, [1]). They provide a "bridge" between the given specification and its initial implementation by providing (formal) description means while, at the same time, hiding precise implementation details. In the design of embedded and cyber-physical systems, particularly the *Systems Modeling Language* (SysML, [2]) as well as the *Modeling and Analysis of Real-time and Embedded systems* profile (MARTE, [3]) finds considerable attention. They build the basis for an emerging abstraction level recently denoted as *Formal Specification Level* (FSL, [4]).

Having a formal specification e. g. by means of MARTE descriptions allows to get a better understanding of the intended design and to detect errors early in the design process (using methods as proposed e. g. in [5]). Moreover, such descriptions also enable an automatic generation of initial implementations and, by this, aid a design step which was mainly conducted manually thus far.

First approaches which translate FSL-descriptions into ESL-descriptions have been proposed in [6], where SysML diagrams are translated into SystemC modules, and in [7], where MARTE descriptions are linked to SystemC via a polyhedron and a loop model. Later, in [8] a strategy has been proposed which relied on sequence diagrams from MARTE to generate corresponding SystemC implementations. All these approaches focus on the functional translation of a formal specification.

But, in addition to the common modeling tools MARTE offers a sophisticated time modeling mechanism. It extends the primitive timing profile of the UML with mechanisms to model time (TimeStructure) and to access time (Clocks) [9]. This includes a formal language to describe the behavior of the clocks: the *Clock Constraint Specification Language* (CCSL, [10]).

The CCSL defines clocks and their relations to each other including ticking coincidences or order of ticks over several clocks. These specifications can be utilized during the design of a system whenever the question "When?" is asked. The semantics of the CCSL have been formally defined in [11].

Based on this, several approaches have been proposed in the past which aim for the simulation and verification of CCSL constraints. For example, the work presented in [12], [13] addressed VHDL generating observers to check the compliance of a CCSL specification, i. e. they simply skipped the ESL and directly proceeded to RTL. An alternative approach was to generate Promela code for verification, which was checked afterwards using the *Simple Promela Interpreter* (SPIN) [14]. An analysis of CCSL statements was also done in [15], [16], where a so-called polychrony clock calculus is generated and checked using the *Signali* language. Here, a disadvantage is that the handling of non-deterministic parts cannot be chosen in this approach. Polychrony can only be generated after eliminating the non-determinism.

These approaches are partially integrated in *TimeSquare* – the main framework to simulate and visualize clocks which is available as a plugin for eclipse [17]. TimeSquare consists of an editor for CCSL, a clock calculus to generate clock traces, and a verification part to check the correctness of clock traces. The user can choose several simulation policies to handle nondeterministic parts of CCSL. Waveforms can be generated and the Papyrus tool offers a graphical view of the simulation. However, TimeSquare offers traces, which – once completed – cannot interact with the rest of the system. This limits the applicability of CCSL within the entire design flow sketched above. More precisely, while formal specifications such as MARTE can be utilized in order to create initial SystemC implementations, corresponding clock constraints are not handled thus far.

In this work, we are addressing this issue. An approach is proposed which translates CCSL into SystemC and, by this, allows for the simulation of the specified clocks in SystemC. We assume thereby that a complete SystemC description of the desired system is already available (e. g. by means of the code generation approaches mentioned above). Then, our approach adds one further module, a *TimeController*, to this implementation which handles the clock behavior specified in the CCSL and triggers the available clocks as desired. The corresponding CCSL constraints are thereby accordingly considered. The resulting SystemC implementation can then be utilized to simulate not only the functional but also the clock behavior of the design according to the specification.

The proposed approach is described in the remainder of this paper as follows: Section II briefly reviews CCSL and SystemC while Section III introduces the general idea of our solution. Afterwards, Section IV describes the implementation of our approach in detail, while Section V illustrates its application by means of an example. Section VI concludes the paper.

II. Preliminaries

This section reviews the background on both, the *Clock Constraint Specification Language* (CCSL) as a part of the UML/MARTE specification [3] and SystemC [18] as the quasi-standard language for simulation at the ESL.

A. The Clock Constraint Specification Language (CCSL)

The *Modeling and Analysis of Real-time and Embedded systems* profile (MARTE) for UML provides a language for describing timing constraints: the *Clock Constraint Specification Language* (CCSL) [3]. Central part of the underlying time definition are *instants*, i. e. moments in the raw, unordered time, defined by clock ticks. The *clock* is an instrument to access a set of instants [11]:

Definition 1. *A* ***clock*** $\langle \mathcal{I}, \prec, \mathcal{D}, \lambda, u \rangle$ *consists of a set of instants $\mathcal{I}$, which owns a quasi-order relation $\prec$, a set of labels for the instants $\mathcal{D}$, a labeling function λ, and a unit u for the clock ticks. A finite clock has a finite number of ticks. If no ticks are left, the clock is empty.*

Example 1. *Consider a processor executing assembler commands. The computation cycles of the processor can be described using the clock* ***processor****. They form the set of instants of the clock, which are illustrated by dots at the line shown in Fig. 1. Every instant represents a clock tick. These ticks represent processor cycles. The order of the instants on the line is specified by $\prec_{processor}$.*

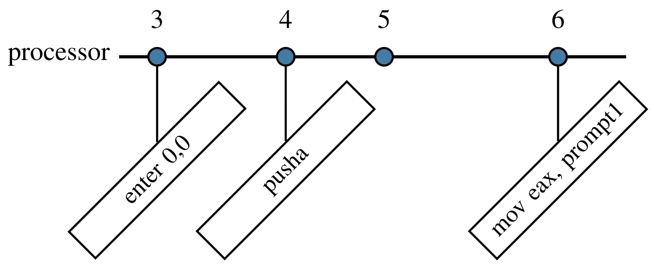


Fig. 1. Clock *processor* with instants (the dots), indices of the instants (the numbers above), and labels (the commands below [19])

Certain processor cycles at the beginning of the clock are reserved for computations which set up the system. Those are labeled by the commands which shall be executed in the respective instant. In Fig. 1, these labels are denoted below the dots representing the instants.

Clocks in general can be logical or chronometric [3]. The main difference between both is, that the intervals of logical clock ticks can vary concerning physical time. Logical clocks can refer to any event like processor cycles or sensor data, while chronometric clocks refer to physical time and can also be dense.

Concerning a simulation with SystemC, the target system is discrete and, because of that, provides only logical time. This directly follows from the clock of the processor as basic clock, which itself is a logical clock, counting processor cycles. For that reason, all clocks are simulated as logical discrete clocks and chronometric continuous clocks will be simulated in a discrete manner, too.

From the clocks, a time structure can be derived [11]:

Definition 2. *A* ***time structure*** *is a pair $\langle \mathcal{C}, \preccurlyeq \rangle$, where $\mathcal{C}$ is a set of clocks and $\preccurlyeq$ is a binary relation on $\cup_{c \in \mathcal{C}} \mathcal{I}_c$ named precedence (one clock tick takes place before or coincidently with the other). From $\preccurlyeq$ three further instant relations can be derived:*

- *Coincidence:*
$$\equiv \triangleq (\preccurlyeq \cap \succcurlyeq)$$
The two clock ticks take place coincidently.
- *Strict precedence:*
$$\prec \triangleq (\preccurlyeq \setminus \equiv)$$
One of the clock ticks takes place strictly before and not coincidently with the other.
- *Exclusion:*
$$\# \triangleq (\prec \cup \succ)$$
One clock tick takes place before or after, but not coincidentally with the other.

Some of these instant relations are also CCSL statements:

- "i_1 coincidentWith i_2" is $i_1 \equiv i_2$
- "i_1 (strictly) precedes i_2" is $i_1 \prec i_2$ resp. $i_1 \preccurlyeq i_2$

Example 2. *CCSL can be used to describe complex temporal relations. As an example, consider the clocking depicted in Fig. 2 which describes the determination of an Advent day: The fourth Sunday of Advent is the Sunday before or at Christmas. This means that Advent is a day, which is a Sunday. It is the last before Christmas or it can be the Christmas day itself.*

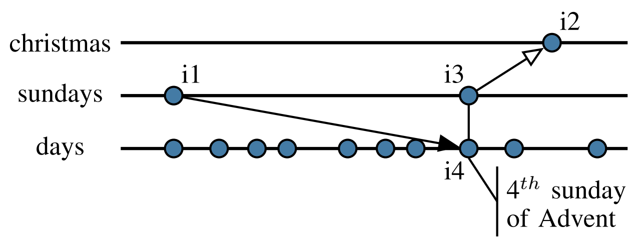


Fig. 2. An instant relation for determining the 4^{th} Sunday of Advent

This is specified by the CCSL statements given in the listing below. The clocks days, sundays, *and* christmas *are omitted here, but will behave as the names suggest.*

```
Instant i1 is sundays;
Instant i2 is christmas;
Instant i3 is sundays;
Instant i4 is days;
i1 strictly precedes i4;
i3 coincidentWith i4;
i3 precedes i2;
```

These instant relations affect the instants to which they are referring to, but not the rest of the instants of the clock. In other words, from the instants no behavior of the clocks can be derived. It not even is possible to force instants to appear. In fact, instants may just be specified in order to describe a forbidden behavior rather than a desired behavior. This is illustrated in Example 2, where not all instants of the clock *days* are supposed to be Advent days.

Hence, instants will be reported, when they appear, but their appearance has no further effect. In contrast, if instant relations are defined for all instants of the clock, they become clock relations (or clock definitions, in terms of CCSL). These clock relations constrain the complete behavior of the clocks, so they can be used to generate a specified behavior. A list of all possible expressions is given in Table I, which contains all statements and a short explanation of their semantics. CCSL offers various expressions to define clocks and to express relations between them. In Table I, they are sorted concerning their kind of clock expression (column *kind*), which can be a clock definition (def), a clock relation (rel), or a chrono-nonfunctional-property (cnfp). The category refers to the fashion in which they are translated to SystemC (this is discussed later in Section IV-B).

New clocks (from clock definitions) are always subclocks, i. e. their ticks are a subset of the ticks of their reference clock(s). In this paper, we will focus on the generation of clock behavior.

In principle, clock behavior can be defined in two fashions.

- According to the fact whether clocks are grouped (e. g. tick together or not) and
- according to the fact how a clock behaves with respect to other clocks (e. g. being periodic to another clock)

Formally, grouping clocks can be expressed as a binding.

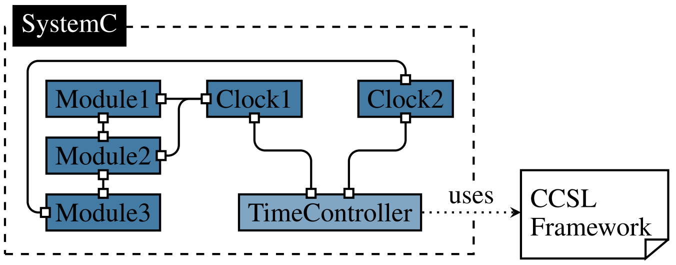


Fig. 3. Example SystemC implementation extended by the TimeController

Definition 3. *Clocks can be bound together using* ***bindings****. Consider the clocks c_1, c_2 and the event e. Then, two kinds of bindings exist:*

- $(c_2 \rightarrow c_1)$ *or* $(e \rightarrow c_1)$*, i. e.* c_1 *ticks, if* c_2 *or* e *ticks or*
- $(c_2 \leftrightarrow c_1)$ *or* $(c_1 \leftrightarrow e)$*, i. e. the clocks* c_1, c_2 *or the event* e *tick or tick not together.*

Clock behavior over time in relation to other clocks can be described by using several terms, which is considered later in Section IV-B.

B. SystemC

As a class library for C++, SystemC is used to implement and simulate systems at a high abstraction level, i. e. at the ESL. SystemC provides proper description means supporting hardware-specific properties [10] in addition to the software syntax which comes with C++ anyway.

A SystemC description of a system consists of modules, which are connected via ports and communicate using signals. The modules own specific processes, which can be sensitive on certain signals, i. e. they react on value changes on these signals.

Such descriptions are abstract enough so that they do not require a hardware/software partitioning, i. e. it is not necessary to decide whether a module shall be realized as hardware or software. At the same time, the description is precise enough to enable a complete simulation of the implemented system. SystemC can model specific hardware properties like inter-process communication, parallelism, and synchronization.

SystemC has been certified by the IEEE in 2005 [18] and is a de-facto-standard in the industry.

III. Problem Formulation and General Idea

Using modeling languages like UML enables the precise specification of the structure as well as the behavior of a system to be implemented. At the same time, additional description means provided by extensions such as MARTE enable the specification of non-functional properties such as timing. In this work, we focus on timing requirements specified in CCSL as introduced in the previous section.

The considered scenario is the following: Given an initial MARTE specification, all structural and functional aspects of the system have already been implemented in SystemC (e. g. using the approaches from [6], [7], [8]), i. e. a SystemC implementation composed of several modules, attributes, methods, but also clocks is already available (see Fig. 3). However, the respective timing behavior of the clocks has not been realized yet. Hence, in the next step an extension to the SystemC implementation needs to be created, which triggers all clocks according to the specification provided by the respective CCSL constraints. Thus far, this extension has been created manually. In this work, we propose an automatic approach.

TABLE I
CCSL CLOCK DEFINITIONS (DEF), RELATIONS (REL), AND CHRONO-NFPS (CNFP)

kind	category	new clock	statement	meaning
def	A	c_{new}	when e	$c_{new} \leftrightarrow e$
def	A	c_{new}	c_1 restrictedTo b	Boolean property b evaluates to true $\Rightarrow c_1 \leftrightarrow c_{new}$
def	A	c_{new}	c_1 filteredBy β	binary word β evaluates to 1 $\Rightarrow c_1 \leftrightarrow c_{new}$
def	C	c_{new}	c_1 discretizedBy r	c_{new} is the time-discretized equivalent of c_1
def	A	c_{new}	c_1 delayedBy n	index of c_1 $> n \Rightarrow c_1 \leftrightarrow c_{new}$
def	B	c_{new}	c_1 followedby c_2	c_1 is not empty $\Rightarrow c_1 \leftrightarrow c_{new}, c_1$ is empty $\Rightarrow c_2 \leftrightarrow c_{new}$
def	B	c_{new}	c_1 inter c_2	$c_{new} \leftrightarrow (c_1 \wedge c_2)$
def	B	c_{new}	c_1 minus c_2	$c_{new} \leftrightarrow (c_1 \ \wedge \ \neg c_2)$
def	F	c_{new}	c_1 sampledTo c_2	c_{new} ticks with c_2, if c_1 has ticked since the last tick of c_2
rel	C	–	c_1 isPeriodicOn c_2 period n	c_1 has a pause of n ticks on c_2 between its ticks, then $c_1 \rightarrow c_2$
rel	C	–	c_1 isSporadicOn c_2 gap n	c_1 has a pause of at least n ticks on c_2 between its ticks, then $c_1 \rightarrow c_2$
rel	A	–	c_1 isFinerThan c_1	$c_2 \rightarrow c_1$
rel	A	–	c_1 isCoarserThan c_2	$c_1 \rightarrow c_2$
rel	D	–	c_1 isFasterThan c_2	the n^{th} tick of c_1 is before them of c_2
rel	D	–	c_1 isSlowerThan c_2	the n^{th} tick of c_2 is before them of c_1
rel	D	–	c_1 haveMaxDrift c_2	the number of ticks of c_1, c_2 differs never more than n
rel	A	–	$c_1 = c_2$	$c_1 \leftrightarrow c_2$
rel	B	–	$c_1 \# c_2$	$c_1 \rightarrow \neg c_2 \wedge c_2 \rightarrow \neg c_1$
rel	F	–	c_1 alternatesWith c_2	tick n of c_1 $\preccurlyeq$ tick n of c_2 $\wedge$ tick n of c_2 $\prec$ tick n+1 of c_1
rel	E	–	c_1 hasSameRateThan c_2 offset n	equivalent to c_1, c_2 haveOffset n
cnfp	E	–	c_1 hasStability r	the distance between two ticks of a chronometic clock can vary by r
cnfp	E	–	c_1, c_2 haveOffset d	c_2 ticks like c_1, but d ticks on a reference clock later
cnfp	E	–	c_1, c_2 haveSkew r	the offset between c_1, c_2 gets higher in every step by r
cnfp	E	–	c_1, c_2 haveDrift r	the skewness between c_1, c_2 gets higher in every step by r

The general idea is to extend the existing SystemC implementation by a *TimeController* as shown in Fig. 3. This controller can access all clocks and is capable of triggering them. At the same time, it incorporates all CCSL constraints ensuring that the controller only triggers clocks when this accords to the respective specification. More precisely, for any considered system state during the execution of the SystemC implementation, the TimeController traverses all clocks and checks whether clocks *can* tick, *cannot* tick, or *must* tick according to events in former system states.

The semantics of the CCSL constraints can thereby be distinguished between two kinds of constraints: Some constraints are based on former system states, e. g. if one clock has ticked, two steps later another clock must tick. Other constraints are the bindings which contain information about the distributions of clocks between the sets of clocks which will tick or not. A binding does not state that a clock must or must not tick in general. Bindings group several clocks together, i. e. depending on the kind of binding, both clocks can tick or not tick together, but both clocks cannot tick alone.

Example 3. *Consider a MARTE specification with the following CCSL constraints:*

```
Clock days;
Clock sundays;
sundays isPeriodicOn days period 6.0;
```

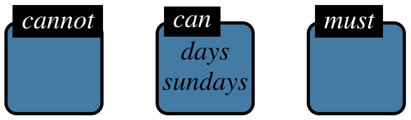
That means, ***sundays*** *is a subclock of* ***days****. When* ***sundays*** *ticked, it cannot tick until the clock* ***days*** *has ticked six times and then has to tick with* ***days****. In other words: If* ***sundays*** *ticks, it has to tick again coincidently with the seventh tick of* ***days****.*

Additionally, at the beginning of the simulation, the following must be defined: If ***sundays*** *ticks,* ***days*** *must tick as well, which is in terms of bindings:*

$$\textbf{\textit{sundays}} \rightarrow \textbf{\textit{days}}$$

This ensures that all instants of ***sundays*** *are also instants of* ***days****. Additionally, in the initialization of the simulation must be defined that the clock* ***sundays*** *has to tick no later than the* 7^{th} *tick of* ***days****. This ensures that the periodical ticking of* ***sundays*** *starts in time.*

Having these constraints, the TimeController first traverses all clocks and checks whether clocks can, cannot, or must tick based on the former system states or the initialization. In this simple example, both clocks, i. e. days *and* sundays *report that they can but are not forced to tick (there is no constraint which forces them nor prohibits them to tick), i. e. they are categorized as follows:*



Next, the TimeController checks for all bindings whether they provide any further information about the behavior. The only binding is ***sundays*** $\rightarrow$ ***days****, which causes no effects at this point.*

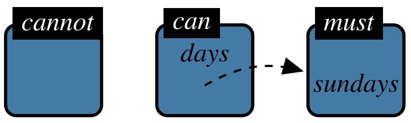
At this stage, no information about the ticking behavior of the clocks is stored in the *can* list. These clocks are not forced to tick but also not forbidden to tick. On the other side, it is also not possible to decide in general, whether they shall tick or not. If there are e. g. two clocks in the *can* list which exclude each other, it is not allowed to let both of them tick.

To prevent the system from remaining in such an idle state, a *policy* can be defined. In our solution, we already provide simple policies. However, further ones can easily be defined by the user. This mechanism is, to the best of our knowledge, not considered in any FSL language and especially not in UML or CCSL.

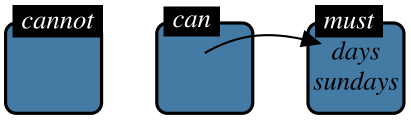
Policies choose a clock and assign it to the *must* or *cannot* list. Afterwards it is checked which implications can be concluded from this assignment. If the assignment leads to a contradiction, the clock is assigned to another list (e. g. if the first assignment was “let the clock not tick”, it is now tried whether an assignment "let the clock tick" works) and implications are checked again. If this leads to a contradiction, too, a contradiction in the specification might be detected. If the *can* list is not empty, this is repeated, until all clocks in it are assigned.

At the end of this procedure, a valid assignment for all clocks is determined.

Example 4. *Consider again the example from above. This system is now enriched with a policy which arbitrarily chooses the clock* ***sundays*** *to tick (adds it to* ***must****). This is valid, since sundays is initially assigned to the* ***can****-list. In order to implement this policy, sundays is now recategorized as* ***must****:*



Because of that and in order to avoid the violation of the binding ***sundays*** $\rightarrow$ ***days****, the TimeController additionally has to assign the clock* ***days*** *to* ***must****. Again this is possible without any violation, since, in the current system state,* ***days*** *was initially assigned to* ***can****.*



Finally, all clocks in ***must*** *are called to tick, i. e. the TimeController triggers these clocks, while all remaining clocks are not triggered. This whole process is repeated in the next system state.*

If in one state a situation like above occurs, but ***days*** *is in* ***cannot*** *instead of* ***can****, a contradiction would occur. At this point,* ***sundays*** *would be removed from* ***must*** *and added to* ***cannot****. Again, the bindings would check if this is still in accordance to their constraints. If not, a contradiction in the specification is unveiled.*

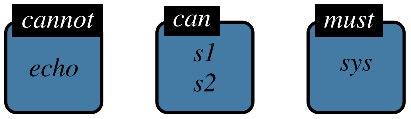
A contradiction in a simulation like mentioned above does not necessarily mean that the specification is erroneous, but the combination of the specification with the chosen ticking policy may cause problems. At some stages, an unwisely chosen policy may have effects to some system states, which cause problems in later states.

Example 5. *Consider a system with two sensors s1 and s2. Each of them has to report its values in every second tick of the system clock sys (here considered as a minimal clock, which ticks in every step). From the second report of sensor s2 on, this report will produce an echo, which will interfere with the report of s1. This echo and the report of s1 are not allowed to take place at the same time. This is represented by the following specification:*

```
Clock sys is minClock;
Clock s1;
Clock s2;
Clock echo is s2 delayedBy 1.0;

s1 isPeriodicOn sys period 1.0;
s2 isPeriodicOn sys period 1.0;
s1 # echo;
```

In the first step, the minimal clock ***sys*** *is categorized to* ***must****,* ***s1*** *and* ***s2*** *to* ***can*** *(since they could both tick in this or the next step), and* ***echo*** *to* ***cannot*** *(because* ***s2*** *has not ticked until now).*



At this stage nothing forbids to let both, ***s1*** *and* ***s2****, tick in this step. But if the policy chooses them both to tick, they must both tick again with* ***sys*** *in the* 3^{rd} *step. Now* ***echo*** *occurs with* ***s2****. This would mean that all clocks would have to tick in this step, while* ***s1*** *and* ***echo*** *are not allowed to tick together. This leads to a contradiction.*

However, if in the first step the policy chooses only one of ***s1*** *and* ***s2****, they will tick alternately in the following and by this, the conflict between* ***s1*** *and* ***echo*** *is avoided.*

While the evaluation of the respective CCSL constraints is trivial for simple examples as illustrated above, more elaborated constraints may occur. For this purpose, the TimeController needs to be enriched by a proper data structure as well as corresponding implementations. How they are generated for different CCSL constraints is described next.

IV. Implementation

This section first outlines the data structure used in the proposed solution in order to translate CCSL constraints into SystemC. Afterwards, the translation scheme is described. The application of the patterns presented here are illustrated in Section V.

A. Applied Data Structure

The TimeController is implemented in SystemC and, hence, based on C++. To properly represent and compute the timing behavior, three C++ classes are applied:

- The *TimeController* itself is the unique computing instance, controlling the interaction between the other objects. As depicted before in Section III, it has access to the lists of clocks which can, cannot, or must tick. Additionally it owns the bindings representing how clocks are bound together as well as the respectively applied policy.
- For each clock, an additional *ClockMonitor* is added which represents information on earlier events that might affect the current ticking behavior. From the information of the ClockMonitor, the first assignment of this clock to the *can/cannot/must*-lists is performed. Additionally, the ClockMonitor holds information on how the ticking of its clock does affect the behavior of the other clocks.
- Finally, *Bind* objects represent the bindings as reviewed in Section II-A. They can access the information stored in ClockMonitor objects of the respectively involved clocks. Based on this information, it is decided whether, according to the current state of the monitors, an assignment of a clock to the *can/cannot/must*-lists has to be changed to satisfy this binding. Alternatively, the existence of a contradiction can be detected.

Fig. 4 shows the resulting structure. For each clock to be simulated a ClockMonitor object is created which is owned by the TimeController. Depending on the given CCSL constraints, these ClockMonitor objects may be enriched with further Bind

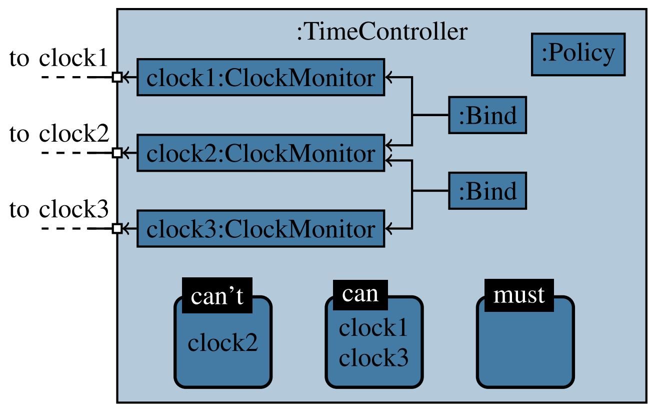


Fig. 4. TimeController with three ClockMonitor objects, two Bind objects, the policy, and the lists for clocks that can, cannot, or must tick

objects. Based on this data structure, CCSL statements can accordingly be expressed in SystemC.

B. Representation of CCSL in the Framework

CCSL contains a rich variety of expressions (see Table I) which require different realizations within the proposed SystemC scheme. Therefore, the total set of possible CCSL expressions is categorized with respect to their representation in terms of bindings, conditions, etc. In total, six different categories (A to F) are considered which are also respectively indicated in Table I. In the following, the respective handling of CCSL constraints from each category is described[1].

A The first category is composed of expressions which can be defined and translated using the bindings $\leftrightarrow$ and $\rightarrow$ from Definition 3 as well as conditions evaluating to Boolean values only. They can be realized in a straightforward fashion using the respective ClockMonitor object with no further restrictions. Only a respective Bind object is created which accordingly binds the considered clocks together.

B CCSL expressions stating clock bindings conditioned by other clocks ticks (e. g. $\#$) are summarized in this category. Here, a clock is only bound to another, if other clocks tick, tick not, or if they have certain clock-specific properties like emptiness. Again, these constraints are realized by creating simple Bind objects to the affected ClockMonitor objects. The respective conditions can easily be checked by those objects.

C The CCSL expressions in this category restrict the ticking of a clock in relation to the ticking defined by other clocks. This includes

- constraints stating that a clock must have ticked until a certain tick of a reference clock occurred (e. g. a clock c_1 must tick, until a clock c_2 has ticked n times, i. e. $c_1.tickIn(n, c_2)$),
- constraints stating that a clock must tick in an explicitly specified fashion with another clock (e. g. a clock c_1 may tick together with the n^{th} tick of a clock c_2, i. e. $c_1.tickExactlyIn(n, c_2)$), or
- constraints prohibiting a clock to tick until a reference clock has ticked a certain amount of times (e. g. clock c_1 may not tick, until a clock c_2 has ticked n times, i. e. $c_1.lockFor(n, c_2)$).

This kind of constraint requires that the ClockMonitor objects interact with each other. To this end, a *lock* and a *tickIn* method is provided for each ClockMonitor. They can be used to provide other clocks with the information that they are not allowed to tick anymore or that they, in fact, have to tick. By calling these operations, the future assignments of the *can/cannot/must*-lists are modified.
More precisely, each ClockMonitor owns lists of clocks it interacts with, i. e. a list of clocks to lock, a list of clocks to call ticking, and a list for the precise ticking calls. Each entry in these lists is composed of triples like

$$(c_{target}, c_{reference}, l)$$

where c_{target} is the target clock of the statement (e. g. the clock which shall be locked), $c_{reference}$ is the reference clock (e. g. the clock which defines the length of the lock or ticking call), and the length l of the lock/ticking call itself. Whenever a clock ticks, its corresponding ClockMonitor traverses its lists and e. g. locks all clocks c_{target} from the lock list for the defined time l on $c_{reference}$.
Corresponding to the lists of clocks to lock or call ticking, each clock owns the respective lists of the applied ticking calls and locks. In each step a clock is asked, if it can, cannot or must tick. The answer to this question depends amongst others on these lists. If there are locks left in the lock list, a clock cannot tick, if a ticking call expires, it must tick and so on. Here – besides other times in the simulation – also contradictions can be found, e. g. if a ticking call expires on a locked clock.
These lists are updated after every simulation step. Each clock has a further list with clocks, which use this clock as reference clock for any statement. If the clock ticks, it notifies all listening clocks, so they can update their locks and/or ticking calls.

D CCSL expressions from category D represent restrictions to the number of ticks of a clock. A clock restricted by a constraint from this category can tick without restrictions in time, but has only a certain contingent of ticks. If no ticks are left, it can not tick anymore. The actual number of ticks is related to another clock which may grant more ticks when it ticks by itself. For example, a constraint such as $c_1.addTick(c_2)$ belongs to this category stating that if c_2 ticks c_1 is granted another tick.
Constraints like these are realized in a similar fashion as constraints from category C. Again, ClockMonitor objects influence each other by adding ticks. This is also realized with lists. The clock which adds ticks holds a list with all clocks to which ticks shall be added. Triples are not needed here, because there is no reference clock and the number of ticks added is always 1. Instead, the clock with the ticking restriction handles the ticks by a list composed of tuples such as

$$(c_{granting}, i),$$

where $c_{granting}$ is the clock which grants the ticks and i is the number of ticks which are currently left. Only if all entries in this list are larger or equal one, the clock can (but must not) tick.

E CCSL expressions providing constraints for chronometric clocks are summarized in this category. They rely on calculating the next tick of the restricted clock and then enforce a respective ticking. Two kinds of restrictions are considered here: Stability lets a clock restrict itself by

[1]As mentioned above, examples illustrating the application are afterwards provided in Section V.

calculating the next tick, while offset-related constraints make the first clock restricting the following by calculating its next tick.
These constraints are realized in a similar fashion as constraints from category C. The only difference is that the value l from the triple $(c_{target}, c_{reference}, l)$ is replaced by a corresponding function which may depend on drift, skew, or stability of the clock.

F The last category is for expressions which have an own and more complicated behavior. They are implemented using an own implementation of the class Bind. The behavior can be represented by a simple automaton, whose different states affect the behavior of the implementation. It behaves like different bindings in the different states of the automaton. The state-changes are dependent on which clock of a given set of possible clocks ticks.

Besides that, CCSL statements may include intervals, tuples, and other vague statements as arguments of timing expressions. Those cause further non-determinisms in the execution. Theoretically, a policy could handle this, but would need more sophisticated policies compared to the ones described above. Hence, such vague statements are omitted at this stage. However, future contributions with more elaborate policies might include vague statements.

The next section gives an example, how a CCSL specification can be implemented in SystemC and how it behaves during the simulation.

V. Application

In this section, the generation of a SystemC implementation for a given set of CCSL constraints is illustrated by means of an example. To this end, the considered example is briefly introduced first. Afterwards, the code generation process and the execution of the resulting implementation is illustrated. All code generation has been conducted using an Xtext-based parser.

A. Considered Example

We illustrate the application of the proposed solution by means of a communication satellite scenario depicted in Fig. 5. This example displays special timing constraints due to the specific conditions in space, where a behavior timed by positions of celestial bodies in relation to each other is needed. Additionally this scenario is inspired by the works of the cooperation partner of our graduate school *System Design* (SyDe), the *German Aerospace Center* (DLR). Space missions have very strict timing constraints – logical as well as chronometric – and, hence, work with very different time dimensions: from milliseconds, when e. g. the rocket starts, to years or longer, when the travel time to the target is considered. This makes it very attractive to model these dimensions at the FSL.

The satellite orbits the earth at a certain height and needs 10 hours to circumnavigate the earth. Energy is gained using a solar panel. If the earth's shadow is passed and the panels do not gain enough energy, the power supply is switched to batteries.

To hold its orbit, the satellite has two rocket engines which are fired electrically. This firing needs a significant amount of energy and, because of that, is not allowed while passing the shadow. The height is checked at least once a day, but, due to the rapid temperature change, cannot be checked if an alteration between light and shadow occurs (i. e. during dusk and dawn moments).

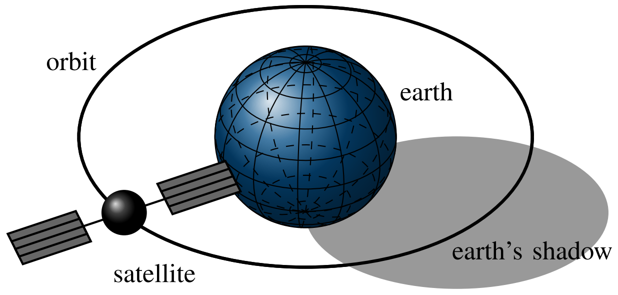


Fig. 5. The satellite on its orbit

To model this behavior, several events are described:

- Minutes, hours and days
  - all defined through one central clock
  - 60 minutes = 1 hour
  - 24 hours = 1 day
- Entering the earth's shadow and sunlight
  - one circumnavigation lasts 10 hours
  - flying through the shadow lasts 3 hours
- Times to adjust the antenna
  - every minute
- Times to check the orbit
  - every minute
  - not during dusk and dawn
  - not more often than once in five minutes

The resulting CCSL description is given in Fig. 6. This includes various CCSL expressions. The constraints *isPeriodicOn* and *isSporadicOn* are expressions from category C, i. e. they restrict when – in relation to their clocks – a clock can tick. Both state that the restricted clock is a subclock of the reference clock. But while *isPeriodicOn* lets the new clock tick with every *period*+1st tick of the reference clock, *isSporadicOn* just states that at least *gap* ticks of the reference clock must be between two ticks of the subclock. The # statement forbids two clocks to tick at the same time. It is a statement making the clock's behavior depending from other clocks and, hence, is a constraint from category B. The *isFasterThan*-expressions belongs to category E and, therefore, restricts the number of ticks, i. e. **checkOrbit** must tick more often than **days**. The last expression – *haveOffset* – states that every tick of **enterLight** is followed by a tick of **enterShadow** with the fifth tick on the reference clock **hours**.

B. Generating SystemC

As outlined before, we assume that all functional behavior of the system is already implemented in SystemC. Then, the CCSL description from Fig. 6 is translated to a SystemC module as described in the previous section. The respectively added code lines are provided in Fig. 7.

First, the module's basic structure is generated. This includes the surrounding structures in lines 1–2 and the concluding SC_CTOR-block in lines 48–54. Additionally, a method for the initialization of the module (lines 12–37) and for running the simulation (lines 39–46) is created. Both are mentioned in the SC_CTOR-block. The method *run* (see line 50) is called in every step of the simulation, while *initialize*

```
ClockConstraintSystem satellite {
  Clock minutes is chronometric 1 m;
  Clock hours;
  Clock days;
  Clock enterShadow;
  Clock enterLight;
  Clock checkOrbit;

  hours isPeriodicOn minutes period 59.0;    // define hours on minutes
  days isPeriodicOn hours period 23.0;       // define days on hours
  enterLight isPeriodicOn hours period 9.0;  // 10 hours between light entries
  enterLight, enterShadow
    haveOffset {7.0} on hours;               // light lasts 7 hours, rest is shadow
  checkOrbit isFasterThan days;              // check orbit at least once a day
  checkOrbit isSporadicOn minutes gap 4.0;   // not more than once in 5 minutes
  enterShadow # ckeckOrbit;                  // no orbit check during dusk/dawn
  enterLight # checkOrbit;
}
```

Fig. 6. The CCSL description of the satellite clock system

is only called once (see line 52) at the beginning in order to set up the simulation. The thread is sensitive to the internal clock *clk* (see lines 3 and 51) which directs the simulation. So far, all structures mentioned right now are equal for all SystemC modules, except the two methods which are only part of all TimeController modules.

Next, the model-specific CCSL parts are added: For all clocks from Fig. 6, corresponding ClockMonitor objects are generated (see lines 14–22). Since the clock *minutes* is chronometric, the period to the internal clock is accordingly adjusted (line 14). Furthermore, a port is added for every clock which is directed by the TimeController (see lines 5–10). This is needed in order to trigger the respective tickings.

Finally, the respective constraints are implemented (see lines 24–36). First, the *isPeriodicOn* constraints are realized using to the clocks *hours*, *days*, and *enterLight*. This is a constraint from category C. For a more detailed explanation, consider: Here, *hours* is the clock which is restricting itself by an

```
Controller::addIsPeriodicOn(hours,
    minutes, period_33);
```

exact ticking call. So, internally, the CCSL framework forms a triple of the parameters, where *hours* is the target clock to be restricted, *minutes* is the reference clock, and *period_33* is the duration of the restriction (which is set to 59.0 in line 24). This triple is added to the exact ticking call list of the ClockMonitor for *hours*. As *isPeriodicOn* is a subclocking statement where the restricted clock is a subclock of the reference clock, a Bind object is created, which prevents *hours* from ticking, if *minutes* is not ticking ($hours \rightarrow minutes$). The other *isPeriodicOn* statements are generated analogously.

Afterwards, *isFasterThan* (line 31) is added to *checkOrbit*. Internally *checkOrbit* restricts *days* by restricting the number of ticks for *days*. The ClockMonitor *checkOrbit* has to tick more often. Internally, *days* is added to the tick granting list of *checkOrbit* and the initial ticks of *days* are set to 0, i. e. *checkOrbit* has to tick first to grant more ticks to *days*.

The realization of *isSporadicOn* in line 33 is similar to the realization of *isPeriodicOn*. The difference is only that the triple is not added to the exact ticking call list, but to the lock list.

Both # statements in lines 34–35 are realized by creating an internal Bind object. This guarantees that they will not tick together. The last CCSL expression, *haveOffset*, is again very similar to *isPeriodicOn*. However, the target clock is not the restricting clock *enterLight*, but the offset clock *enterShadow*. The reference clock is *hours* and the duration is *offset_36* as defined in line 30. The offset clock is now a subclock of *hours*.

After automatically generating the complete module, the user only has to connect the ports manually to their respective clocks. Then, the simulation can be started.

C. Running the Simulation

Having the automatically generated SystemC realization of the CCSL constraints, simulations on them can be performed by simply invoking the method *run* in every simulation step. Corresponding policies can additionally be enforced as e. g. illustrated in line 42 defining a simple maximum policy. Each simulation step consists of three phases:

1) At the beginning of every simulation step, all ClockMonitor objects are asked by the TimeController whether they are allowed to tick, they have to tick, or they are not supposed to tick, and are assigned to the respective list (*can, cannot* or *must*). The ClockMonitors assume thereby logical clocks except for the clock *minutes* which is chronometric. This clock reports either that it must or cannot tick (always a definite answer) dependent on the system clock clk, which represents physical time.
   All other ClockMonitors decide their ticking abilities depending on earlier simulation steps. This may be influenced e. g. by the number of ticks which were granted by other clocks who are restricting their number of ticks or locks (which forbid ticking). In the first simulation step, all report that they can tick, except *enterShadow* which cannot tick until *enterLight has*.
2) For all ClockMonitor objects in the *can* list, the TimeController has to decide whether they shall tick or not. To do so, all Bind objects are checked, if there is

```
SC_MODULE("satellite_TimeController")
{
    sc_in<bool> clk;

    sc_out<bool> port_minutes;
    sc_out<bool> port_hours;
    sc_out<bool> port_days;
    sc_out<bool> port_enterShadow;
    sc_out<bool> port_enterLight;
    sc_out<bool> port_checkOrbit;

    void initialize()
    {
        int period_30 = 1 * 60000;
        ChronometricClockMonitor* minutes
            = new ChronometricClockMonitor("minutes", period_30, &port_minutes);
        ClockMonitor* hours = new ClockMonitor("hours", &port_hours);
        ClockMonitor* days = new ClockMonitor("days", &port_days);
        ClockMonitor* enterShadow
            = new ClockMonitor("enterShadow", &port_enterShadow);
        ClockMonitor* enterLight = new ClockMonitor("enterLight", &port_enterLight);
        ClockMonitor* checkOrbit = new ClockMonitor("checkOrbit", &port_checkOrbit);

        double period_33 = 59.0;
        Controller::addIsPeriodicOn(hours, minutes, period_33);
        double period_34 = 23.0;
        Controller::addIsPeriodicOn(days, hours, period_34);
        double period_35 = 9.0;
        Controller::addIsPeriodicOn(enterLight, hours, period_35);
        double offset_36 = (7.0);
        Controller::addIsFasterThan(checkOrbit, days);
        double gap_37 = 4.0;
        Controller::addIsSporadicOn(checkOrbit, minutes, gap_37);
        Controller::addExclude(enterShadow, ckeckOrbit);
        Controller::addExclude(enterLight, checkOrbit);
        Controller::addHaveOffset(enterLight, enterShadow, hours, offset_36);
    }

    void run()
    {
      try {
          Controller::run(new ccsl::clk::MaximumPolicy());
      } catch(ccsl::clk::ContradictionException* e) {
          cout << e.what();
      }
    }

    SC_CTOR(satellite_TimeController)
    {
        SC_METHOD(run);
        sensitive << clk.pos();
        initialize();
    }
};
```

Fig. 7. The automatically generated TimeController module in SystemC

already a contradiction or if some ClockMonitors can be assigned as a conclusion from the Bind objects. In this precise simulation, the first check of the Bind objects reports, that there is no information known about further behavior of clocks just from the bindings.
Then, the MaximumPolicy arbitrarily chooses one ClockMonitor (e. g. *days*) and assigns it to the ticking list. There exists a binding which states that if *days* ticks, *hours* has to tick as well (because as a periodic clock *days* is a subclock of *hours*). The same is valid for *hours* being periodic on *minutes*. Since *minutes* is already in the ticking list, no other implications are concluded at this point.
Next the Policy chooses *enterLight* to tick. This causes no further changes and finally, *checkOrbit* is assigned to tick, too. The final ticking list is

$$\{minutes, hours, days, enterLight, checkOrbit\}$$

3) If in the CCSL specification there are instants defined, it is now checked if one of them appeared in this step. After that, all ClockMonitors are called to tick and flush their ports with the trigger signal to the clock modules. Afterwards they iterate their lists of ClockMonitors to lock or call ticking and proceed these invocations. ClockMonitor *hours* calls itself to tick again in exactly 60 ticks on *minutes*, *days* calls itself ticking in exact 24 ticks of *hours*. Beginning of light phase, *enterLight*, calls itself to tick again in 10 ticks on *hours*, and *enterShadow* to tick in 7 ticks on *hours*. The ClockMonitor *checkOrbit* simply locks itself (forbids itself to tick) for 4 ticks on *minutes*. Additionally *days* is granted a tick by *checkOrbit*, because it is not allowed to tick faster (more often) than *checkOrbit*.
Finally, all ClockMonitors iterate the list of ClockMonitors which are listening to them and notify each of them of their ticking. The notified ClockMonitors iterate all their saved ticking calls and locks and decrease the respective duration until the lock is solved or a ticking call must be served.

In the further simulation, these three phases are repeated in every simulation step. Until *minutes* ticks again, no clock can tick. But with the 60^{th} tick of *minutes*, *hours* will tick again, with the 5^{th} tick *checkOrbit* can tick and so on. In every step this process is repeated until the simulation is terminated.

VI. Conclusions and Future Work

In this paper, we propose a translation of MARTE CCSL descriptions to SystemC. Assuming the rest of the system already existing in SystemC, we added the behavior described in CCSL, by creating a TimeController module. This triggers the ticking of the clock modules using our CCSL framework to compute, whether a clock ticks or not.

CCSL statements were translated using their characteristics to define a simple data structure fitting all CCSL statements. During the simulation in each cycle the set of ticking clocks is computed and the ticks in the respective modules are triggered.

Our further research will focus on improving the framework e. g. by improving the policies for better reasoning. The policies shortly touched in Section III are still quite simple. They neither take the relations between clocks in account nor their restrictions by their own ticking behavior. Furthermore there is a lack of specification means for policies for the user. At the moment, the user has to implement the policy interface manually, which is not desired during a specification process on FSL. To find better description means here is another target of our further work.

Acknowledgments

This work has been partially supported by the Graduate School SyDe, funded by the German Excellence Initiative within the University of Bremen's institutional strategy and by the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1.

References

[1] Object Management Group, *OMG Unified Modeling Language TM (OMG UML) Superstructure*. Object Management Group, 2011.
[2] ——, *OMG Systems Modeling Language (OMG SysML^{TM})*. Object Management Group, 2012.
[3] ——, *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems*. Object Management Group, 2011.
[4] R. Drechsler, M. Soeken, and R. Wille, "Formal Specification Level: Towards Verification-driven Design Based on Natural Language Processing," in *Forum on Specification and Design Languages*, 2012, pp. 53–58.
[5] M. Soeken, R. Wille, and R. Drechsler, "Verifying Dynamic Aspects of UML Models," in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, Dresden, 2011.
[6] W. Raslan and A. Sameh, "Mapping SysML to SystemC," in *Forum on Specification and Design Languages*, 2007, pp. 225–230.
[7] R. B. Atitallah, E. Piel, J. Taillard, S. Niar, and J. L. Dekeyser, "From high level MPSoC description to SystemC code generation," in *International ModEasy Workshop in conjunction with Forum on specification and Design Languages (FDL'07)*, 2007.
[8] E. Ebeid, D. Quaglia, and F. Fummi, "Generation of SystemC/TLM code from UML/MARTE sequence diagrams for verification," in *Symposium on Design and Diagnostics of Electronic Circuits and Systems*, 2012, pp. 187–190.
[9] C. André, F. Mallet, and R. de Simone, "Time modeling in MARTE," in *Forum on Specification and Design Languages*, 2007, pp. 268–273.
[10] T. Grötker, *System design with SystemC*. Springer, 2002.
[11] C. André and F. Mallet, *Clock Constraints in UML/MARTE CCSL*. Institut National de Recherche en Informatique et en Automatique, 2008.
[12] C. André, F. Mallet, and J. DeAntoni, "VHDL Observers for Clock Constraint Checking," in *International Symposium on Industrial Embedded Systems (SIES)*, 2010, pp. 98–107.
[13] F. Mallet, "Automatic Generation of Observers from MARTE / CCSL," in *IEEE International Workshop on Rapid System Prototyping*, 2012, pp. 86–92.
[14] L. Yin, F. Mallet, and J. Liu, "Verification of MARTE/CCSL Time Requirements in Promela/SPIN," in *International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2011, pp. 65–74.
[15] H. Yu, J.-P. Talpin, L. Besnard, T. Gautier, F. Mallet, C. André, R. de Simone, and R. D. Simone, "Polychronous Analysis of Timing Constraints in UML MARTE," in *International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, 2010, pp. 145–151.
[16] H. Yu, J.-P. Talpin, and L. Besnard, "Polychronous controller synthesis from MARTE CCSL timing specifications," in *ACM & IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, 2011, pp. 21–30.
[17] J. Deantoni, "TimeSquare: Treat Your Models with Logical Time," in *International Conference on Objects, Models, Components, Patterns (TOOLS)*, 2012, pp. 34–41.
[18] I. S. Association, *1666-2011 – IEEE Standard for Standard SystemC Language Reference Manual*. IEEE Standards Association, 2011.
[19] P. A. Carter, *PC Assembly Language*. Paul A. Carter, 2006.