

A Framework to Specify System Requirements using Natural Interpretation of UML/MARTE Diagrams

Aamir M. Khan · Frédéric Mallet ·
Muhammad Rashid

Received: date / Accepted: date

Abstract The ever increasing design complexity of embedded systems is constantly pressing the demand for more abstract design levels and possible methods for automatic verification and synthesis. Transforming a text-based user requirements document into semantically sound models is always difficult and error prone as mostly these requirements are vague and improperly documented. This paper presents a framework to specify textual requirements graphically in standard modeling formalisms like UML and MARTE in the form of temporal and logical patterns. The underlying formal semantics of these graphical models allow to eliminate ambiguity in specifications and automatic design verification at different abstraction levels using these patterns. The semantics of these operators/patterns are presented formally as state automata and a comparison is made to the existing CCSL relational operators. To reap the benefits of MDE, a software plugin *TemLoPAC* is presented as part of the framework to transform the graphical patterns into CCSL and Verilog-based observers.

Keywords FSL · Graphical Properties · UML · MARTE · CCSL · Modeling · Embedded Systems

Partially funded by the National Science, Technology, and Innovation Plan (NSTIP) Saudi Arabia.

A. M. Khan (✉)
University of Buraimi, Buraimi, Oman
E-mail: Aamir.m@uob.edu.om

F. Mallet
Université Nice Sophia Antipolis, INRIA Méditerranée, Sophia Antipolis, France
E-mail: Frederic.Mallet@inria.fr

1 Introduction

Conventionally, the design of an embedded system starts with the system requirements specified by the requirements engineers. These requirements are usually in the form of natural language sentences mentioning the different design components, parameters, and constraints. For various kinds of embedded systems, like safety critical systems or Electronic Design Automation (EDA) domain, verification and validation are immensely important. This not only saves the cost of design/prototyping but is critical sometimes in its operations (like avionics or life-saving medical devices). So the next step for such systems is to build an executable model to implement the requirements and perform early validation. For instance, languages like SystemC [44] are often used to build models at the Electronic System Level (ESL) [50] in the early design phases. While the steps after ESL down to transaction level (TLM) or register transfer level (RTL) models have been well covered in the literature, there is a big gap between the early informal natural language requirements and ESL models. Intermediate levels like the Formal Specification Level (FSL) [22,27,54] have been proposed to fill this gap with models that are both close enough to requirements engineer concerns, and formal enough to allow further phases of automatic or semi-automatic generation and verification.

This paper contributes to this effort at FSL. Our solution attempts to reuse as much as possible the Unified Modeling Language (UML) [43] and some of its extensions. Indeed, the UML is a well-accepted modeling language which provides facilities to develop models at different abstraction levels. As a general

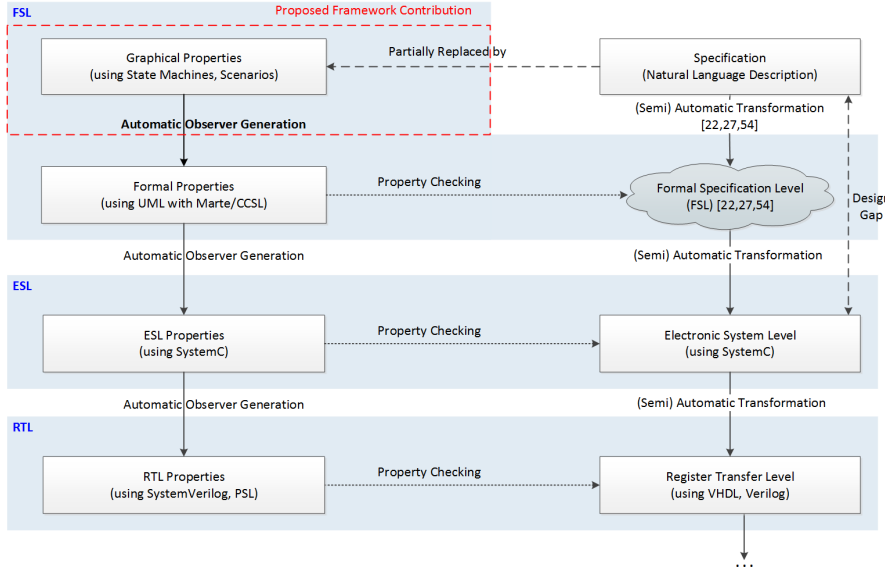


Fig. 1: Proposed Design flow

purpose language, UML needs to be tailored when addressing specific domains. Selic et al. [52] recommend a joint use of MARTE [42] and SysML [62] to build timed and untimed requirements for real-time and embedded systems. Several researchers have put efforts in this area [48]. We follow the same recommendation with special focus on timing (logical) aspects. Additionally, MARTE proposes as an annex, the Clock Constraint Specification Language (CCSL) [6] to complement UML/MARTE modeling elements with timed or causal extensions. CCSL is also used to encode the semantics of UML/MARTE models and resolve potential semantic variation points [7]. Our proposal targets both timed and untimed requirements where CCSL is important to keep those requirements formal and executable. We promote the use of UML-based models over the use of temporal logics formulations, since they have a wider acceptance in industry. Indeed, while temporal logics, like LTL/CTL [47], are widely used in the latter stages in conjunction with model-checkers [18], they are not suitable to directly express high-level requirements at early stages and are commonly rejected by requirement engineers [63, 17] despite the various attempts to alleviate the syntax with higher-level constructs like in PSL (Property Specification Language) [29].

Closing the gap between the requirements level and ESL is even more important for safety-critical systems for which we must ensure that what is verified is actually what was intended by the requirements. Indeed, making a formal verification of some model that represents neither the deployed code (or circuit) nor the original requirements would be useless. A seamless methodology from the requirements to the code synthesis is only possible if (1) the requirements language has a formal semantics that can be maintained through all the refinement steps, we propose to rely on MARTE/CCSL for that, (2) the syntax is simple enough to be widely accepted by all the engineers involved in the process, that is why we propose to rely on UML and extensions rather than on ad-hoc formalism or structured grammars.

Our approach (see Fig. 1) contributes to the trend to build a Formal Specification Level as an intermediate level from natural-language requirements to code synthesis. However it finds its specificity by three characteristics that are not, to the best of our knowledge, used jointly in previous approaches. (1) A set of pre-defined primitive domain-specific property patterns, (2) A graphical UML/ MARTE formalism to capture the properties. Rather than having to rely on natural-language, the semantics of these graphical properties is given by a MARTE/CCSL specification, (3) Logical polychronous time [25] as a central powerful abstraction to capture both causal and temporal constraints. While CCSL gives the syntax to build these specifications, TimeSquare [20] can be used to execute the model, generate code and conduct verification [40]. Having executable requirement models helps reduce the risks of misinterpretation. To summarize, there are four major contributions of this research work, namely

1. A framework based on UML, MARTE and our custom introduced *Observation profile* to specify causal and temporal patterns graphically both as annotated UML models and semantic MARTE models.

2. A library of various graphical patterns addressing the frequently occurring system behaviors and the application of those patterns on the case study of traffic light controller.
3. Formal semantics of these temporal/logical patterns in the form of CCSL syntax and state automaton.
4. Development of TemLOPAC (Temporal and Logical Pattern Analyzer and Code-generator) plugin as a means to reap MDE approach benefits and convert the modeled framework formalisms into CCSL and Verilog observers.

Our initial attempts were on the natural representation of UML diagrams to address the system requirements [32]. In that proposal, the work was only focused on the representation of patterns graphically. Patterns were presented in the form of state transition events like *turnStateA* or *turnNotStateA* etc. Feedback from that work enhanced our understanding leading to the current approach. Here in this paper, we propose a complete framework to interpret the UML diagrams in a natural way and provide tools to transform those graphical representations into observers. Section 2 discusses the related work and their differences with the proposed approach. Popular LTL based property specification approaches like Property Sequence Charts (PSCs), TILCO-X, Metric Temporal Logic (MTL), Drag and Drop PSL (DDPSL) are discussed and compared to our proposed framework. Research work from the domain of runtime verification community is discussed next which mainly targets LTL, MTL and generation of efficient observers. A comparative analysis of the latest work done related to graphical patterns is presented next. Moreover, a comparison is made to the Formal Specification Level (FSL) in terms of their area of focus, utility, similarities and differences. Section 3 provides the state of the art about UML artifacts and its profiles. The section first introduces UML state machine and sequence diagrams mentioning the features they lack. UML itself lacks the notion of time which is essentially required in modeling temporal as well as causal patterns. So we introduce next the MARTE profile which provides desired timings concepts. In the last, CCSL semantics are defined which facilitates later as a means to formally define framework artifacts.

Section 4 presents the proposed framework by specifying and formally defining graphical patterns. This section explains the formal semantics of those patterns and how they are created. The combined features of UML and MARTE provide a working ground for developing timed-models but they still lack the specialized features to model graphical patterns. Two types of patterns are introduced. State-state patterns are graphically represented using state machine diagrams annotated with MARTE profile and a specialized *Observation profile* which is designed to target the expressiveness of graphical properties. This profile resides on top of the UML/MARTE to provide a structure for building some predefined patterns. State-event patterns are presented next using the sequence diagrams annotated with MARTE profile. They are used to show the relation of various systems states and related events.

Section 5 presents a library of formally defined temporal and logical patterns. Various other works related to timed/untimed patterns and behavior

representation of state-based systems are discussed. Then these patterns are mathematically defined along with the presentation of their graphical representation. Section 6 applies these patterns into use on a traffic light controller case study. The result of simulating observers in the design is shared at the end of the section.

Once these patterns are established, we present their formal semantics in section 7. Selected patterns from this framework are presented in CCSL specification. These CCSL relations can be converted using TimeSquare tool into VHDL observers for direct hardware validation of the modules. Further, this section presents a more direct approach in the form of automaton diagram of graphical patterns. These automata are then used later to generate the desired Verilog code of the graphical patterns. Section 8 discusses the tool implementation for the presented framework. It introduces the Eclipse plugin TemLOPAC (*Temporal and Logical Pattern Analyzer and Code Generator*) developed in Java EMF. It operates on the UML state machine and sequence diagrams to generate the CCSL or Verilog code from the modeled graphical patterns. Finally section 9 concludes the paper with a glimpse over the future possibilities.

2 Related Work

Various efforts have been made over the past two decades to bridge the gap between natural language and temporal/causal logic. Initially property specification patterns [23] were proposed in the form of a library of predefined LTL/CTL formulae from where the users can pick their desired pattern to express the behavior. Later other works proposed the specification of properties through graphical formalism [53], [13], [4], and [34]. There have also been several attempts to encode temporal logics formula as UML diagrams [65,66].

Property Sequence Charts (PSC) [11], [12] are presented as an extension to UML sequence diagrams to model well-known property specification patterns. Originally PSCs focused on the representation of order of events and lacked the support for the timed properties. But the later extension in the form of Timed PSC [65], [66] support the specification of timing requirements. PSCs mainly target LTL properties. The domain of expressiveness of CCSL is different from LTL and LTL-based languages, like PSL. CCSL can express different kind of properties than LTL [24]. Here we are interested to express properties on logical clocks for which LTL is not an obvious choice. Also LTL is not meant to express physical time properties, for which, this framework prefer the use of CCSL. Moreover, PSCs do not benefit from the popular embedded systems modeling and analysis profiles like MARTE. Rather than encoding formula with UML extensions, we propose to reuse UML/MARTE constructs to build a set of pre-defined property patterns pertinent for the domain addressed.

The work presented by Bellini and his colleagues [15] is a review of the state of the art for real-time specification patterns, organizing them in a unified way. It presents a logic language, TILCO-X, which can be used to specify temporal

constraints on intervals, again based on LTL. The work of Konrad et al. [33] is to express real-time properties with a facility to denote time-based constraints in real-time temporal logics MTL (Metric Temporal Logic) extending the LTL. Finally, another research work, DDPSL (Drag and Drop PSL) [21], presents a template library defining PSL formal properties using logical and temporal operators.

The research domain of *runtime verification* [35,51] relies on lightweight formal verification techniques to check the correctness of the behavior of a system. Most works on such online monitoring algorithms focus on the LTL, CTL, MTL for expressing the temporal constraints [14,61] where high expertise is required to correctly capture the properties to be verified. Moreover, specialized specification formalisms that fit the desired application domains for runtime verification are usually based on live sequence charts (LSCs) or on MSCs [17]. Another research work to mention here is about the generation of controller synthesis from CCSL specifications [64]. Mostly the main focus of runtime verification community is on the generation of efficient observers for online monitoring whereas our proposed framework targets the integration of observers in a complete work-flow. The focus here is on presenting a natural way to model temporal/causal behavior where UML models represent the traditional meaning of UML formalisms and can easily be traced back to the requirements from those diagrams.

Some very recent research work by Rettberg and his colleagues [60,49] focus on similar issues that we address here. One approach [60] targets the model-based methodology to formalize natural language like expressions describing non-functional requirements. It is used to specify communication behavior in systems using SysML annotated with Requirement Specification Language (RSL) patterns. In contrast to our work, this approach focuses on event-based relations where as our approach targets state-based and state-event based relations. Moreover, this particular approach addresses lower abstraction level requirement which is of less use to ease the design and early validation efforts. For example, concrete time delays used in this work (like 100ns or 10ms) are rarely available in early design stages. The other research work [49] presents strategies to model requirements and design of embedded systems by representing their time and event characteristics. It mainly uses MARTE and SysML UML profiles targeting real-time embedded systems. Both chronometric and logical clocks are present to model different systems abstraction models managed through the semaphores. Again in this work, the focus of the authors is on low abstraction level models using chronometric clocks and no use of logical clocks is provided. Moreover, the use of semaphores makes the approach complex and confusing.

Another aspect of the related work is the advent of Formal Specification Level (FSL), still an informal level of representation. The focus of the FSL approach is to transform natural language descriptions directly into models bridging the design gap (shown on the right in Figure 1). On the other hand, our proposed framework targets the graphical representation of timed/untimed properties (red box on the left of Figure 1) targeting the verification and val-

idation of system/specification. *Natural language front-end* is a general trend to allow for a syntax-directed translation of concrete pattern instances to formulae of a temporal logic of choice, also used in [33] and [10]. Our framework approach is different from the FSL approach as we target the verification and validation of a subset of behavior rather than the complete system. Design engineers are usually well acquainted to UML diagrams (both state machine and sequence) and any graphical alternative to complex CCSL or LTL/CTL notations is more likely to get wider acceptance. Moreover, the use of MARTE profile allows to reuse the concepts of time and clocks to model functional/non-functional requirements of embedded systems.

3 UML, MARTE and CCSL Semantics

This paper proposes a framework to interpret the UML diagrams in a natural way. By natural we mean the diagrams use the UML formalisms (like state machines) in their traditional way and the end-user can easily trace those diagrams back to the requirements. Hence the first sub-section introduces state of the art about the UML artifacts we consider: state machines and sequence diagrams. UML itself lacks the notion of time/clocks which is essentially required in modeling temporal/causal patterns. So selected features from the MARTE time model are explained in the second sub-section which are used in the framework to facilitate semantically sound representation of models.

3.1 UML State of the Art

UML *state machine diagrams* [43] provide a standardized way to model functional behavior of state-based systems. They provide behavior to an instance of a class (or object). Each state machine diagram basically consists of states an object can occupy and the transitions which make the object change from one state to another according to a set of well defined rules. Transitions are marked by guard conditions and an optional action. Formally a UML state machine can be defined by the 5-tuple:

$$StateMachine = \langle S, R, top, container, \mathcal{T} \rangle$$

where

- S is a finite set of states consisting of simple states S_{simple} , composite states $S_{composite}$, final states S_{final} , initial pseudo-states $S_{initial}$ and choice pseudo-states S_{choice} .
- R is a finite set of regions (disjoint from S).
- $top \in R$ is the unique top region.
- $container : (S \cup R \setminus \{top\}) \rightarrow (S \cup R)$ describes the state hierarchy of the state machine, and
- \mathcal{T} is a finite set of transitions

Among the set of state machine elements defined, only a small subset is used by the presented framework to represent graphical properties by giving specific semantics to that chosen subset. The framework specifies S as a set of finite states consisting of simple states S_{simple} while the other states (like initial pseudo-states $S_{initial}$, final states S_{final} and choice pseudo-states S_{choice}) are not used at all. From the standard set of state machine elements, only the top region is used while all other set of regions like in state hierarchy are not considered. The framework considers $top \in R$ as the unique top region and does not consider any other region in R . Finally, \mathcal{T} is considered as a finite set of valid transitions.

UML *sequence diagrams* [43] allow describing interactions between system objects and actors of the environment. A sequence diagram describes a specific interaction in terms of the set of participating objects and a sequence of messages they exchange as they unfold over time to effect the desired operation. Sequence diagrams represent a popular notation to specify scenarios of the activities in the form of intuitive graphical layout. They show the objects, their lifelines, and messages exchanged between the senders and the receivers.

A sequence diagram specifies only a fragment of system behavior and the complete system behavior can be expressed by a set of sequence diagrams to specify all possible interactions during the object life cycle. It is useful especially for specifying systems with time-dependent functions such as real-time applications, and for modeling complex scenarios where time dependency plays an important role. Sequence diagrams consist of objects, events, messages and operations. *Objects* represent observable properties of their class(es). Object existence is depicted by an object box and its ‘life-line’. A life-line is a vertical line that shows the existence of an object over a given period of time. An *event* is a specification of a significant occurrence having time and space existence. A *message* is a specification of a communication that conveys information among objects, or an object and its environment. The formal abstract syntax of sequence diagram is available online [36], which can be used to construct well-formed sequence diagrams. Excerpt from this article is presented next.

$$\begin{aligned}
 &SequenceDiagram ::= \\
 &\quad sname \stackrel{\text{def}}{=} CombinedFragment \\
 &CombinedFragment ::= Interaction \mid \\
 &\quad CombinedFragment; CombinedFragment \mid \\
 &\quad \mathbf{opt}(Cond, CombinedFragment) \mid \\
 &\quad \mathbf{alt}(Cond, CombinedFragment, CombinedFragment) \mid \\
 &\quad \mathbf{loop}(Cond, CombinedFragment) \mid \\
 &\quad \mathbf{loop}(1, n, CombinedFragment) \\
 &Interaction ::= skip \mid \mathbf{ref}(sname) \mid Message \\
 &\quad \mid Message \stackrel{\text{exe}}{=} \{CombinedFragment\}
 \end{aligned}$$

$Message ::= (Sender, A, Receiver, MethodCall)$
 $Cond ::= booleanexpression$
 $Sender ::= objectname : CN$
 $Receiver ::= objectname : CN$
 $CN ::= classname$
 $A ::= associationname$
 $MethodCall ::= method(para)$

where $\stackrel{exe}{=}$ expresses the left side message method call will invoke the messages inside of the brace body $\{.\}$. This abstract syntax helps us define the message as a tuple:

$msg = (s : S, A, r : R, m(para)),$

where s is the sender object of the message with class type S , r is the receiver object of the message with class type R , A is an association between classes S and R , and $m(para)$ is a method call from sender object s to receiver object r with the specified parameter $para$. From the abstract syntax, the sequence diagram can also be defined as:

$\Delta = (ObjectSet, MessageSet)$

in which *ObjectSet* is the set of objects which participate in the sequence diagram and *MessageSet* consists of all the messages in the diagram numbered in a sequence demonstrating the possible execution order as well as the implementation relationships among messages.

Just like the state machine diagrams, the proposed framework focuses on a subset of sequence diagram elements. Amongst the combined fragment elements, this work only uses the *consider* fragment. Consider contains the collection of all the events that are relevant to the modeled scenario. It is just like the sensitivity list in SystemC, Verilog or VHDL. If the list of relevant events is large, then the list of events that are not relevant maybe modeled using the Ignore. In this research work, sequence diagrams are not used in a hierarchical fashion. Moreover, *StateInvariants* are used in the sequence diagrams to show the occurring of specific state.

3.2 MARTE Profile and CCSL

The proposed framework uses concepts of clocks and time for which MARTE time model [42, 9] is utilized. MARTE time model provides a sufficiently expressive structure to represent time requirements of embedded systems. In MARTE, time can be physical viewed as dense or discretized, but it can also be logical related to user-defined clocks. Time may even be multiform, allowing different times to progress in a non-uniform fashion, and possibly independently to any (direct) reference to physical time. MARTE time model is a set of logical clocks and each clock can be represented by $\langle \mathcal{I}, < \rangle$, where \mathcal{I} represents the set of instants and $<$ is the binary relation on \mathcal{I} .

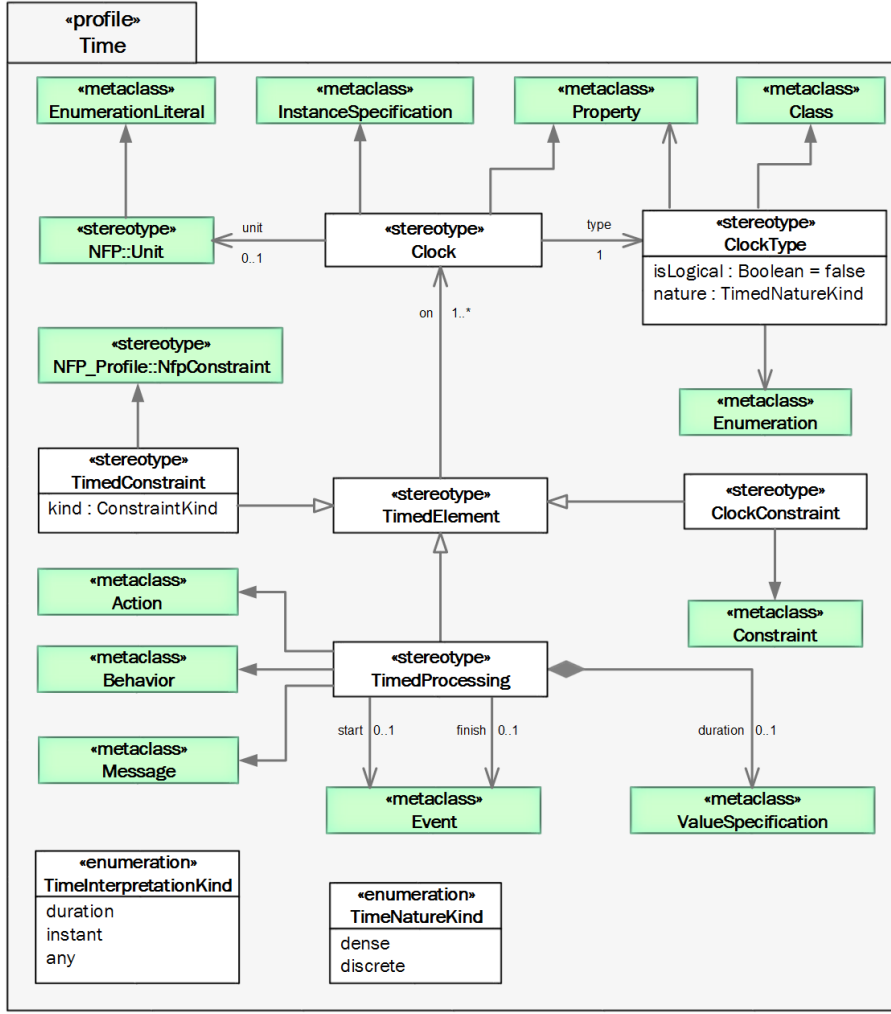


Fig. 2: Excerpt of MARTE Time Sub-profile

Figure 2 presents a simplified view of MARTE Time sub-profile. The green elements are not part of the time sub-profile. At the heart of the profile, the stereotype *ClockType* extends the metaclass *Class* while the stereotype *Clock* extends metaclasses *InstanceSpecification* and *Property*. Clocks can appear in structural diagrams (like SysML block definition, internal block definition, or UML composite structure) to represent a family of possible behaviors. This clock association gives the ability to the model elements identifying precisely instants or duration. MARTE introduces *ClockConstraint* stereotype extending the metaclass *Constraint* through which a MARTE timed system can be specified. *TimedConstraint* is a constraint imposed on the occurrence of an

event or on the duration of some execution, or even on the temporal distance between two events. *TimedProcessing* extends UML Action to make explicit starting and finishing of those events. When those events are clocks, then a *TimedConstraint* can constrain the underlying action to start or finish its execution as defined by the specification. The presented framework uses the *TimedConstraint* on both the state machine and sequence diagram constraint elements, discussed further in the text later. In the state machine diagrams, *TimedProcessing* stereotype is applied to *Do Action* behavior of states providing the start/finish triggers for states. As it extends *TimedElement*, it has the attribute *on* that corresponds to all the clock events that are associated with the current state.

The MARTE *Generic Quantitative Analysis Modeling* sub-profile is intended to support accurate and trustworthy evaluations of models using formal quantitative analyses based on sound mathematical models. It may supplement designer intuition and feeling. Model analysis can detect problems early in the development life cycle and reduce cost and risk. Figure 3 shows the top level stereotypes of the GQAM sub-profile. In this framework, the *GaAnalysisContext* stereotype is applied to our observation patterns. The *workload* attribute of the stereotype corresponds to the scenarios that activate the behavior of the system. The *platform* attribute corresponds to the resource (structural model) on which this state machine (or interaction) is running.

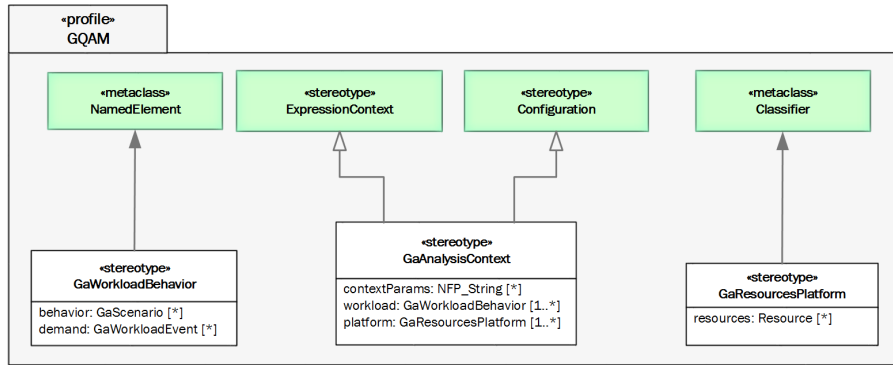


Fig. 3: Top-level Stereotypes of GQAM Sub-profile

The CCSL (Clock Constraint Specification Language) [6] is a declarative language annexed to the specification of the MARTE UML profile. It is used to specify constraints imposed on the clocks consisting of at least one clock relation. A clock relation relates two clock specifications. A clock specification can be either a simple reference to a clock or a clock expression. A clock expression refers to one or more clock specifications and possibly to additional operands. The clock relations can be classified as synchronous, asynchronous, or a combination of both. There are three basic clock relations in CCSL: prece-

dence (\sqsubseteq), coincidence (\sqequiv), and exclusion ($\#$). Two more relations, the strict precedence (\prec) is derived from the precedence relation while the subclocking (\sqsubset) relation is a one-to-one coincidence between one clock and a subset of events of another clock. Subclocking constraint (\sqsubset) is an example of synchronous clock constraint based-on coincidence. Each instant of the subclock must coincide with one instant of the superclock in an order-preserving fashion. The exclusion constraint ($\#$) states that the instants of the two clocks never occur at the same time. Non-strict precede constraint (\sqless) is an example of asynchronous clock constraint based-on precedence. Given the relation $a \sqless b$, for all natural numbers k , the k^{th} instant of ‘a’ precedes or is coincident with the k^{th} instant of ‘b’ ($\forall \in \mathbb{N}, a[k] \sqless b[k]$). Mixed clock constraints combine coincidence and precedence relations. An example is defer constraint (\rightsquigarrow) which enforces delayed coincidences. The expression $c = a(ns) \rightsquigarrow b$ (read as *a deferred b for ns*) imposes c to tick synchronously with the n^{th} tick of b following a tick of a . Another mixed clock constraint is the strict sampling $a \searrow b$ which defines a subclock of ‘b’ that ticks whenever clock ‘a’ has ticked at least once since the previous tick of b .

4 Proposed Framework and Observation Profile

The combined features of UML and MARTE provide a working ground for developing timed-models but they still lack the specialized features to model graphical temporal and logical patterns. This section serves as the first major contribution of this research work by presenting a framework that provides set of reusable generic graphical patterns. Design flow of the proposed framework is shown in Figure 4. It clearly distinguishes three major design flow phases: requirements engineering, systems modeling, and system verification and validation. *Requirements engineering phase* is the one when design engineers collect system structural, functional and non-functional requirements using traditional software engineering techniques. In the *systems modeling phase*, these requirements are then used to manually build the UML model using the custom-defined Observation profile. These annotated UML models are automatically transformed into equivalent but semantically rich MARTE models. In the *verification and validation phase*, these semantic MARTE models are used to generate the CCSL constraints and Verilog observers code. There are numerous advantages to this generated information. The generated Verilog observers can be used along with system-under-verification implementation to perform simulation and verification in tools like QuestaSim [41]. On the other side, the generated CCSL code can be used in TimeSquare tool [20] for simulation, early design validation, and verification analyses. Moreover, these CCSL constraints can also be used to generate the VHDL-based observers code [8].

Here concerning the core part of the framework, two types of behavioral constraints are extracted from the system functional and non-functional requirements: state-state relations and state-event relations. The focus of the

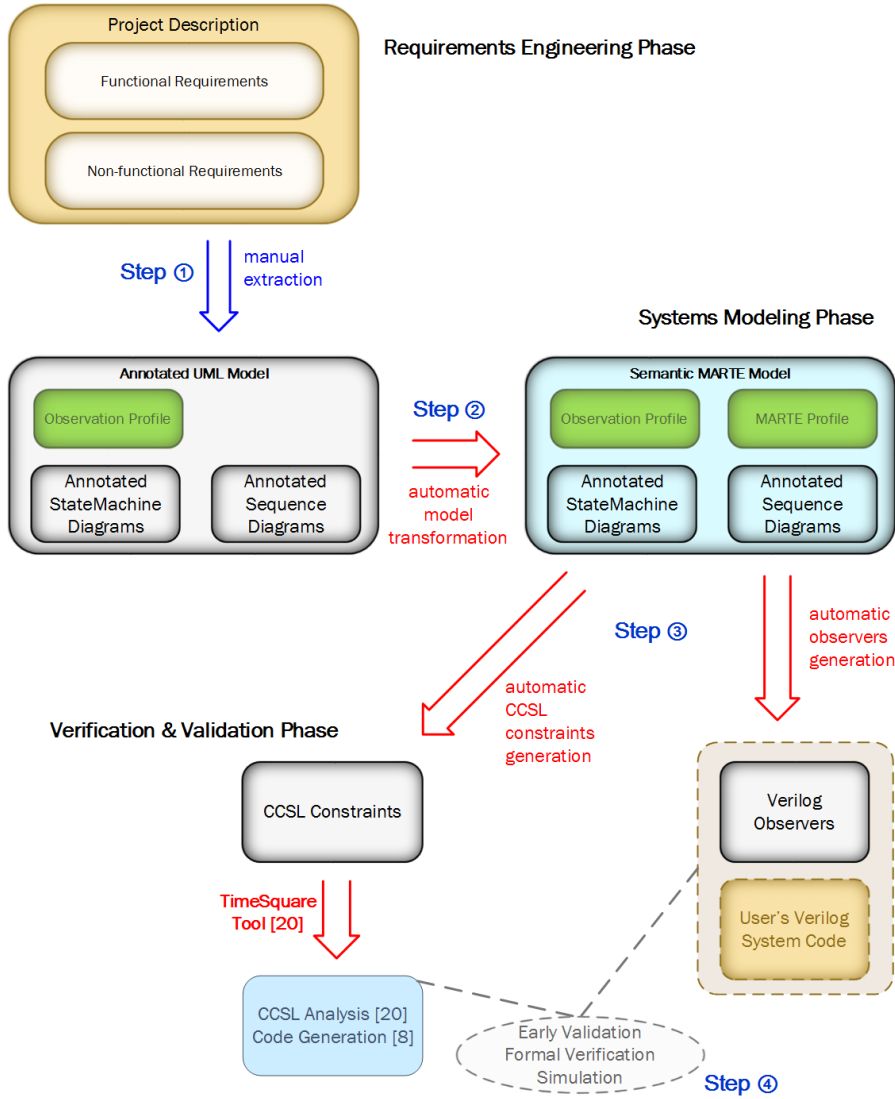


Fig. 4: Overall Framework Design Flow

presented work is not the minimalist approach but rather the expressiveness of the property from the system designer's point-of-view. So when a property specifies occurrence of something at some point in time, then it is an event and it seems natural to represent such properties as logical clocks. But when the properties specify duration or interval (not a particular point in time), then the obvious choice of representation is state relations.

Extended state machine diagrams are mainly used to provide a more natural and syntactically sound interpretation of graphical behavioral patterns

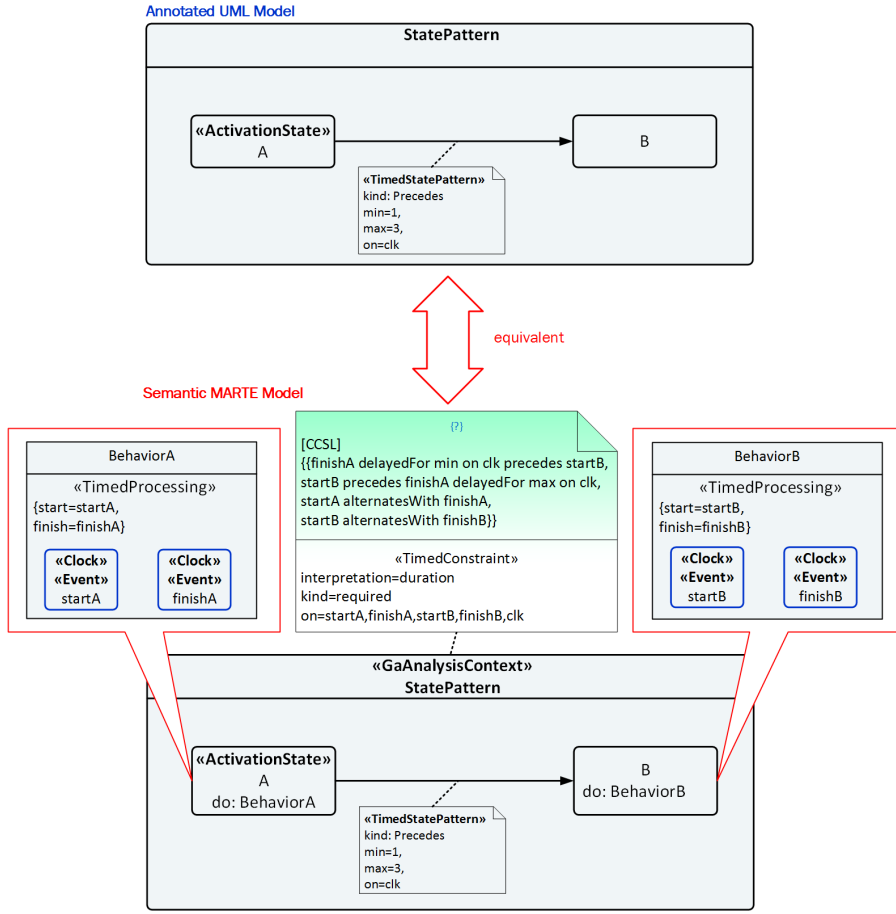


Fig. 5: Sample State-State Timed Pattern

of system states. Figure 5 shows the two representations of a sample state-state temporal pattern of precedence relation (A precedes B by $[1,3]$ on clk). It is achieved by altering the semantics of states and transitions from their traditional meanings. These features are beyond the existing capabilities of both UML and MARTE and for this a custom profile (named *Observation* profile) is introduced, as shown in Figure 6. It is designed to target the expressiveness of graphical properties and resides on top of the UML to provide a structure for building some predefined patterns. This profile presents *ActivationState* stereotype, extending the state metaclass, as a means to identify the antecedent state in relations. It is used to identify the state that activates this particular temporal/logical pattern. It is active whenever the system is in a specified condition (like in relation A precedes B , A is the activation state). Stereotypes *TimedStatePattern* and *UntimedStatePattern* extend the transition metaclass. They are used to specify that the given state transition represents a predefined

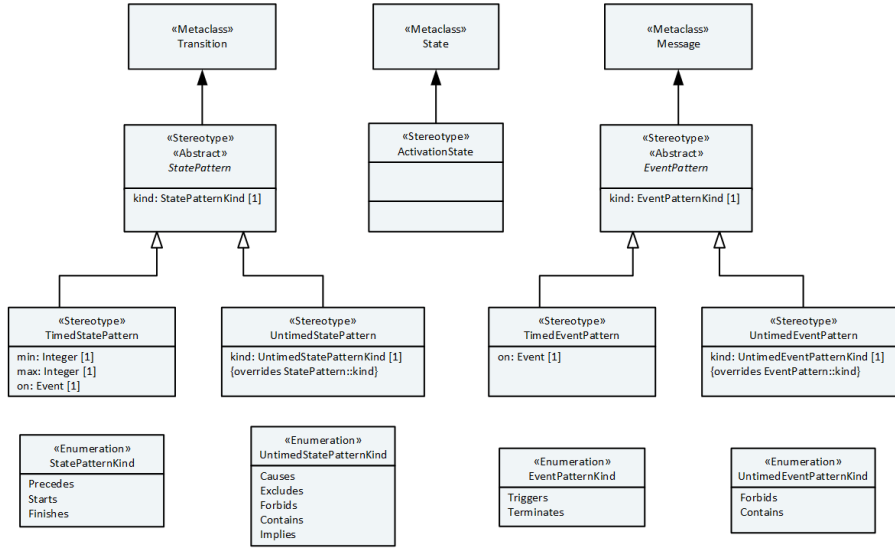


Fig. 6: Proposed Observation Profile

pattern. A number of predefined state-state relation patterns are identified by the *kind* attribute of these stereotypes. The timed graphical patterns have additional attributes (min, max, on) to specify time delay on some base clock.

The semantically rich graphical pattern (lower part of Figure 5) uses the MARTE profile along with Observation profile. As the state machine is used as an analysis formalism, it utilizes the existing analysis context stereotype (Ga-AnalysisContext) from the *Generic Quantitative Analysis Modeling* (GQAM) package of MARTE-AnalysisModel. State machine states (like A,B) are behavioral entities which can be associated with an inner behavior (*OpaqueBehavior*) using the *Do Activity* attribute. The framework proposes to use the MARTE *TimedProcessing* stereotype from the *Time* package of the MARTE-Foundations. This MARTE specialization facilitates to provide the start and finish events for the state (like A.start and A.finish events). Further, a UML constraint is introduced to the state machine specialized by MARTE *TimedConstraint* stereotype, from the *Time* package of the MARTE-Foundations. It provides semantics to the pattern in the form of CCSL specification with the use of *TimedConstraint* that provides the set of logical clocks used for semantic modeling. In short, the user specifies a pattern looking simple and easy-to-understand (upper part of Figure 5), that is transformed into a model packed with all the annotations and technicalities required to make it semantically correct and at the same time being able to generate accurate CCSL or Verilog observers.

Extended sequence diagrams are used to provide the interpretation of graphical patterns relating system events and states. Figure 7 shows the two representations of a sample state-event logical pattern of forbiddance relation (*A forbids B*). In a typical state-event relation, the event (like e) is

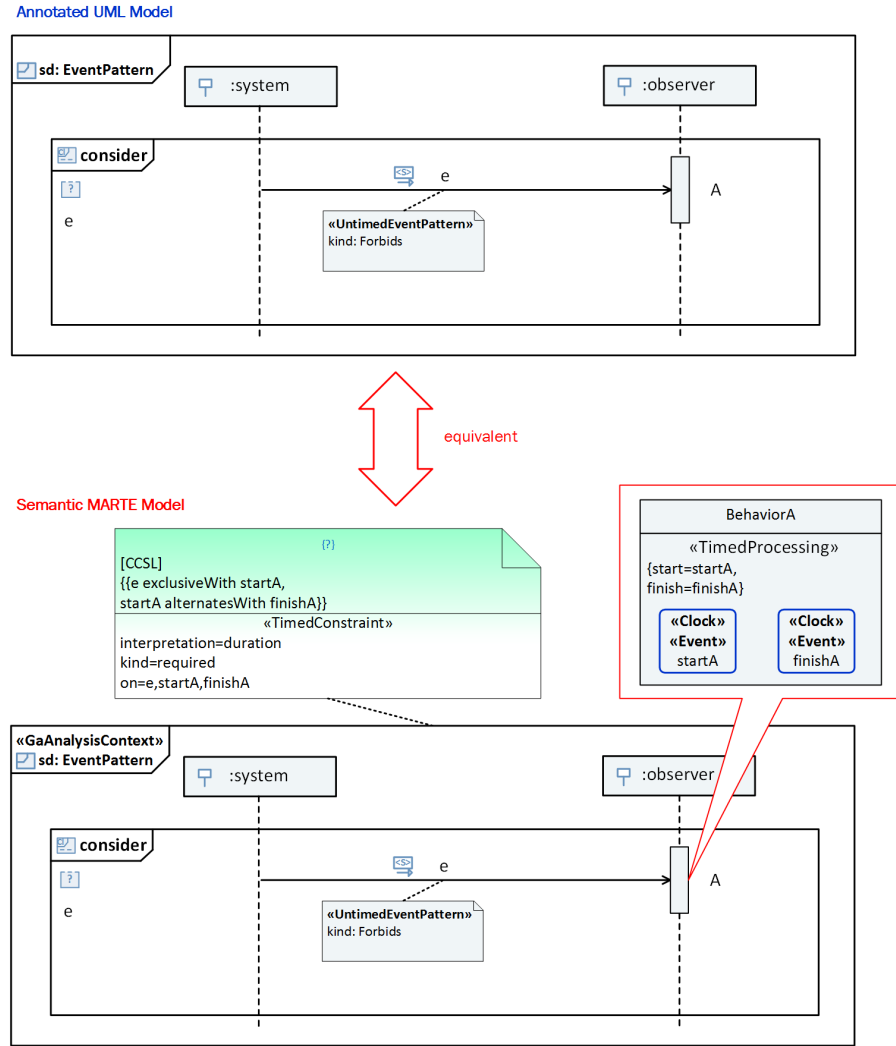


Fig. 7: Sample State-Event Untimed Pattern

represented by the asynchronous message, while the *Behavior Execution Specification* represents the state behavior (like state A). The pattern relation is specified by extending the semantics of interaction messages. The Observation profile contains the stereotypes *TimedEventPattern* and *UntimedEventPattern* that extend the message metaclass. It specifies that the asynchronous message passing between lifelines represents a predefined behavior identified by the *kind* attribute of these stereotypes. For timed patterns base clock event is also specified in the *TimedEventPattern* stereotype while the time duration is specified graphically using UML *Duration Constraint* between the occurrence

specification of message reception and start execution occurrence specification of the behavior.

The semantically rich graphical pattern (lower part of Figure 7) uses the MARTE profile along with Observation profile. Just like the state machines, the *GaAnalysisContext* stereotype from the MARTE_AnalysisModel is used for the sequence diagram. As the state is associated behavior execution specification, its *Behavior attribute* specifies the inner state behavior (*OpaqueBehavior*) just as we did earlier for state machines. The state opaque behavior is stereotyped by MARTE *TimedProcessing* from the *Time* package of the MARTE_Foundations. This MARTE specialization facilitates to provide the start and finish logical events for the state (like A_start and A_finish events). Moreover, a constraint is created having the context of sequence diagram to specify the CCSL semantics of the given graphical pattern. Further, it is specialized by MARTE *TimedConstraint* stereotype, from the MARTE_Foundations, to provide all the logical clocks used in the CCSL specification.

From the end-user perspective different steps involved in using the framework, from high-level requirements to the point where verification results are achieved, are listed below. These steps are also marked on the Figure 4.

1. **Annotated UML Modeling** - The first step is the manual modeling of the system properties as plain annotated UML models from the requirements specification. For this purpose the state machine and sequence diagrams are used for the state-state and state-event properties respectively, annotated with Observation profile features. Requirements can be initially specified in textual form using specialized formalisms like SysML requirements diagrams which are then satisfied by graphical patterns.
2. **Semantically Annotated MARTE Modeling** - The second step is to automatically annotate (using software) these generic UML models with MARTE formalisms from its time and GQAM sub-profiles. This step provides semantically sound models with the desired capabilities to model temporal/logical patterns. There is semantics application of these diagrams by producing UML/MARTE/CCSL information. So the textual requirements are converted into semantically sound patterns that can then further be used for verification and validation purposes.
3. **CCSL Constraints and Verilog Observers Generation** - Once the desired graphical patterns are complete and semantically sound, they can be used to generate CCSL constraints and Verilog Observers from our TemLOPAC plugin [31]. CCSL constraints can further be used to automatically generate VHDL code [8] using the TimeSquare tool [20].
4. **Simulation and Verification** - Generated CCSL code can be simulated directly in TimeSquare for an early design validation. Moreover, the generated Verilog observers can be simulated with the actual system HDL code (in softwares like QuestaSim [41]) for design verification. This system HDL code is usually available to the design engineers during the requirements engineering phase.

5 Proposed Graphical Patterns

This section narrates the second major contribution of the presented framework. For identifying temporal/logical properties of systems, we started by considering several examples like the famous stream boiler case study [1,3], railway interlocking system [28] and the traffic light controller case study [19]. Working on these diverse examples to model behavioral properties in UML, we noted that several patterns were repeated across different examples. These patterns collected across various examples were refined with some inspiration taken from the Dwyer's patterns [23] and Allen's algebra targeting intervals [5]. This practice gave us a valuable collection of generic patterns divided into three major categories of behavioral relations that may exist in a system: state-state relations, state-event relations, and event-event relations. Researchers have already shown that constraints specified in CCSL are capable of modeling logical event-event relations [9,56]. These CCSL constraints can be represented graphically using SysML/MARTE models [37–39]. Temporal/logical patterns for the other two categories of relations are:

State-State Relations: precedes, causes, contains, starts, finishes, implies, forbids, and excludes.

State-Event Relations: forbids, contains, triggers, and terminates.


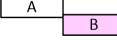
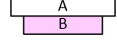
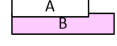
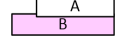
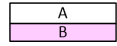
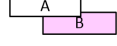
Next subsections discuss selected few of these patterns. A detailed list of these patterns with their syntax and semantics is available online [30].

5.1 State-State Relations

Presented framework can be used to formally model relations between two different states of a system. Almost two decades ago, Dwyer [23] identified two broader categories of patterns: occurrence patterns (absence, existence, universality) and ordering patterns (precedence and response). Moreover, Allen's algebra [5] for intervals provides a base defining thirteen distinct, exhaustive and qualitative relations of two time intervals, depicted in Table 1. From the comparative analysis of these relations, six primitive relations are extracted that can be applied to the state-based systems. Further two 'negation' relations are added, based on their usage and importance in the examples, to complete the set. The overlapping of states is not particularly interesting relation to dedicate a pattern for. But it can easily be modeled indirectly using the state-event relation 'A contains b_s ' where b_s is the start event of state B.

Semantically a state can be considered similar to an interval. We use the nomenclature of using capital letters (A,B,...) to denote states and small letters (a,b,...) for events/clocks. Notations like a_s and a_f are used throughout the text to represent the state start and end events. Note that these events are introduced as part of the framework methodology and the system under verification only provides us relevant states. Formally defining, given a strict partial ordering $\mathbb{S} = \langle S, < \rangle$, a state in \mathbb{S} is a pair $[a_s, a_f]$ such that $a_s, a_f \in S$ and $a_s < a_f$ where a_s is the start and a_f is the end of the state interval. An

Table 1: Allen's Algebra and Proposed Relations

No.	Graphic	Allen's Algebra for Intervals	State-State Relation	Symbol
1		A precedes B	A precedes B	\leq
2		B preceded-by A		
3		A meets B	A causes B	\models
4		B met-by A		
5		A contains B	A contains B	\supseteq
6		B during A		
7		A starts B	A starts B	\vdash
8		B started-by A		
9		A finishes B	A finishes B	\dashv
10		B finished-by A		
11		A equals B	A implies B	\Rightarrow
12		A overlaps B	See the text	
13		B overlapped-by A		
14			A forbids B	\neg
15			A excludes B	#

event or point e belongs to a state interval $[a_s, a_f]$ if $a_s \leq e \leq a_f$ (both ends included).

Precedence is an important state property where the relation ‘A precedes B’ means the state A comes before the state B. It includes a delay/deadline clause to explicitly specify the duration between the termination of state A and the start of state B. The unit of the duration is dependent on the level of abstraction being target of the graphical specification, that is, it can be physical clock, loosely timed clock, or logical clock. Deadline defers the evaluation until some number of ticks of clk (or any other event) occur. The number of ticks of clk considered are dependent on the two parameter natural numbers min and max evaluated as:

- $[0, n]$ means ‘before n ’ ticks of clk
- $[m, 0]$ means ‘after m ’ ticks of clk

- $[m, m]$ means ‘exactly m ’ ticks of clk

Mathematically, given a partial ordering S having the states $A ([a_s, a_f])$ and $B ([b_s, b_f])$, constants m, n and a clock clk , the equation

$$A \sqsubseteq B \text{ by } [m, n] \text{ on } clk$$

means $a_f \leq b_s$ and b_s occurs within the duration $a_f + \Delta$, where Δ is between m and n ticks of event clk . The last tick of clk coincides with the start of the state B (i.e., b_s). Graphically, precedence is shown as the sample pattern earlier in Figure 5. The first state (A) is an activation state as shown by the stereotype. If we need to specify duration, *TimedStatePattern* stereotype is applied. Time interval is specified using min and max attributes along with the base clock for the delay (using ‘on’). Semantics of the states are defined using *TimedProcessing* stereotype applied to the behavior of the states. Semantics of the pattern are given in the constraint using CCSL which works with the logical clocks provided by the MARTE *TimedConstraint* stereotype. These CCSL semantics are further discussed in the section 7 related to state automata.

Forbiddance is a negation property. Relation ‘ A forbids B ’ bars B to occur after state A occurs. Hence mathematically, given a partial ordering S having the states $A ([a_s, a_f])$ and $B ([b_s, b_f])$, the relation $A \sqsupset B$ means $b_s \neq a_f$. Its graphical logical/causal pattern is shown in Figure 8. Scenario activates whenever state A triggers.

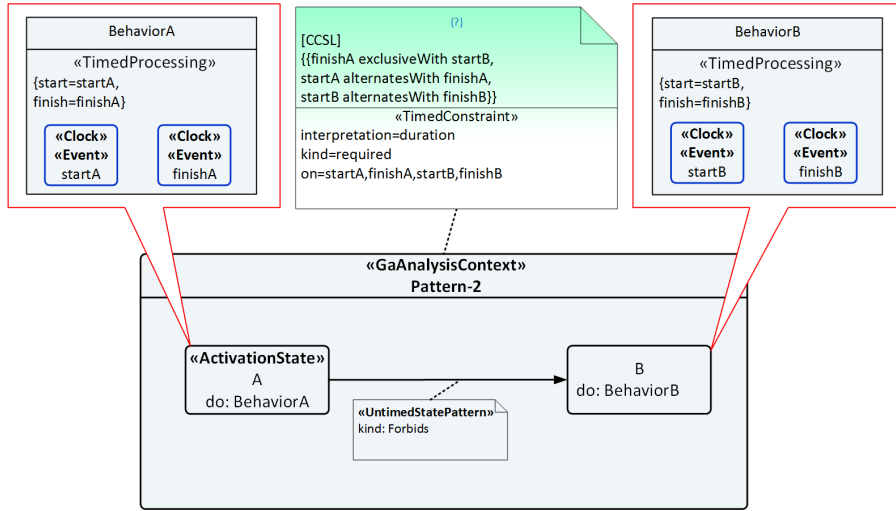


Fig. 8: A forbids B

Exclusion relation between two states (‘ A excludes B ’) restricts them to occur at the same time. Mathematically, given a partial ordering S having the states $A ([a_s, a_f])$ and $B ([b_s, b_f])$, the relation $A \# B$ means that $b_f < a_s$ or

$a_f < b_s$ for all instances of A and B. Graphically, this logical/causal pattern is shown in Figure 9. So the state A activates the pattern and the behavior specified in CCSL constraint bars state B to exist concurrently.

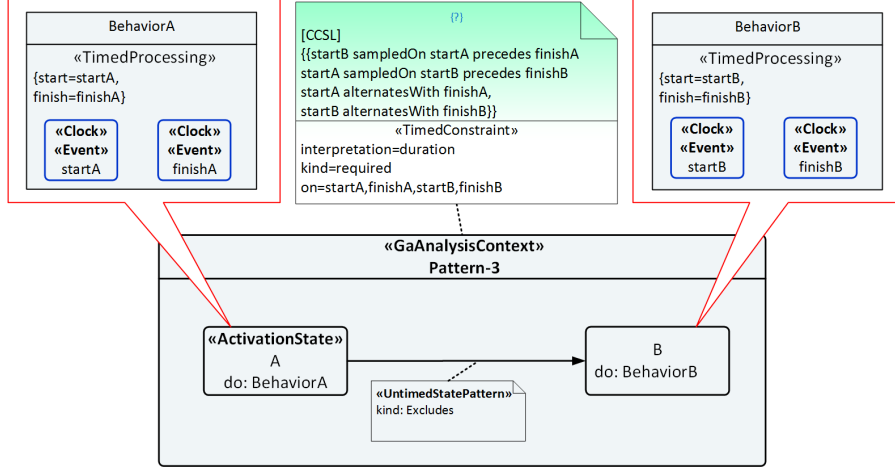


Fig. 9: A **excludes** B

5.2 State-Event Relations

The relations between the system states and events can only be modeled using the UML sequence diagrams which suits modeling flow of events. The concept of behavior execution specification is used to represent the system states. Moreover the sequence diagrams have the consider/ignore combined fragments which allow us to maintain the list of events that are relevant to the current pattern scenario.

The **forbids** relation is similar to forbids relation for states which is implemented using state machines but this forbids relation for events is implemented using sequence diagrams (it only has one state to consider). Relation e *forbids* A implies A must not occur when the event e triggers. Hence given a partial ordering \mathbb{S} having the state A $([a_s, a_f])$, an event e , the relation is expressed mathematically as $e \dashv A$ which means $e < a_s$ or $a_f < e$. As this relation involves an event, the graphical pattern is best expressed using sequence diagram, as shown earlier in Figure 7.

The **triggers** relation is similar to starts relation for states. The relation e *triggers* A, given a partial ordering \mathbb{S} having the state A $([a_s, a_f])$, an event e , constants m, n and a clock clk , can be expressed mathematically as

$$e \models A \text{ after } [m, n] \text{ on } clk$$

It means a_s occurs within the duration $e + \Delta$, where Δ is between m and n ticks of event clk . Graphically, sequence diagram is used to model such relations as shown in Figure 10. *Duration constraint* element is used to specify the required delay limits.

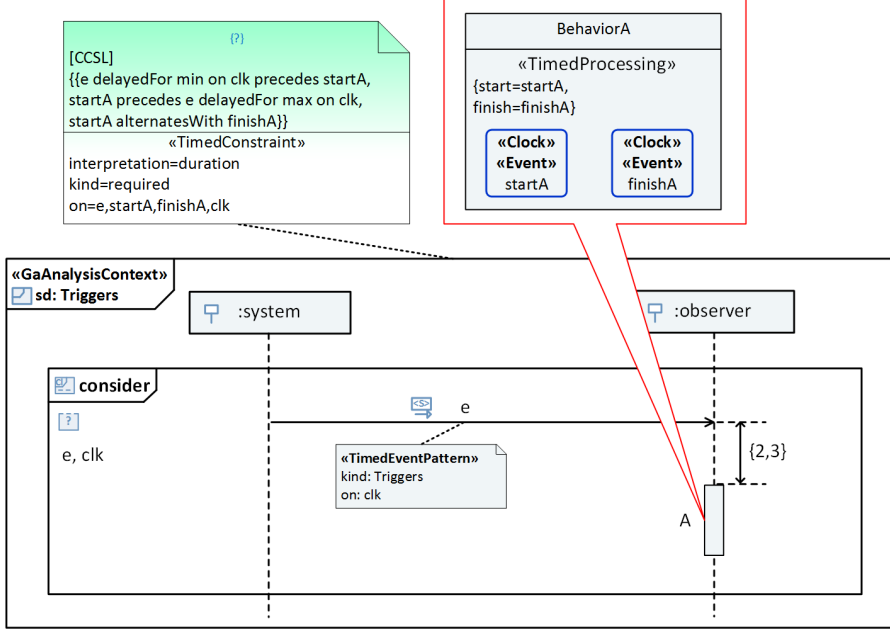


Fig. 10: e triggers A after $[2,3]$ on clk

The **terminates** relation is similar to finishes relation for states. The relation e terminates A can be expressed mathematically, given a partial ordering S having the state A ($[a_s, a_f]$), an event e , constants m, n and a clock clk , as

$$e \Rightarrow A \text{ after } [m, n] \text{ on } clk$$

It means a_f occurs within the duration $e + \Delta$, where Δ is between m and n ticks of event clk . Graphically, it is implemented similar to the triggers pattern, as shown in Figure 11.

6 Application of Graphical Patterns

The case study considered to demonstrate the approach is of the traffic light controller taken from the *SystemVerilog Assertions Handbook* [19]. It consists of a cross-road over North-South highway and the East-West farm road. There are sensors installed for the emergency vehicles and for the farm road traffic.

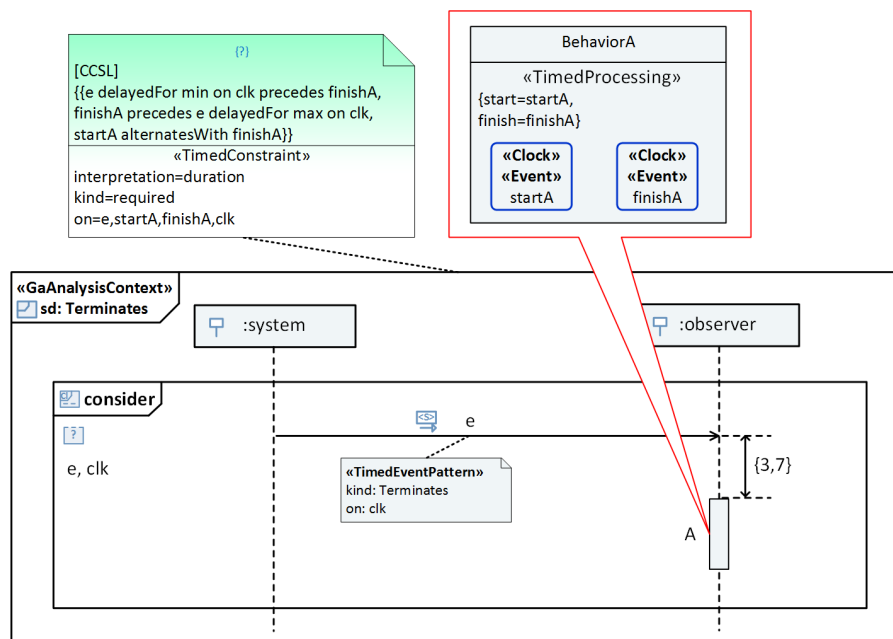


Fig. 11: e terminates A after [3,7] on clk

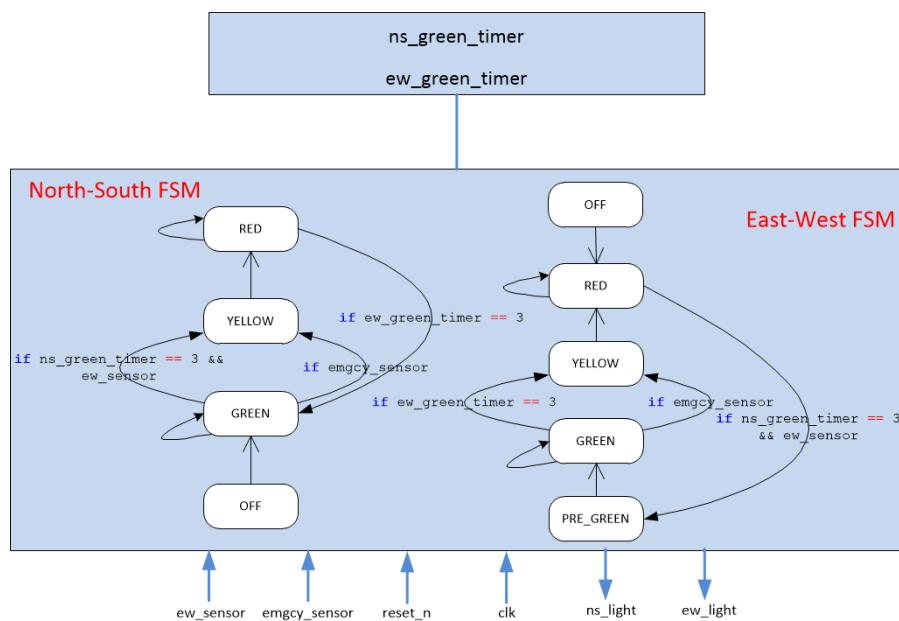


Fig. 12: Traffic Intersection Controller Module

Highway traffic is only interrupted if there is a vehicle detected by the farm road sensor. The architecture for the traffic light controller consists of two state machines, interface signals of the module and the timers, as shown in Figure 12. A few temporal/logical verification properties of the design are discussed next.

Safety, never NS/EW lights both GREEN simultaneously. This property is the exclusion of two states `ns_light=GREEN` and `ew_light=GREEN`. From our library of graphical patterns, we consider the state-state excludes pattern, as shown in Figure 13. Here if state `ns_light=GREEN` is activated, then the pattern ensures the other state is not true (`ew_light=GREEN`).

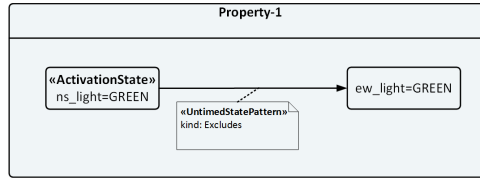


Fig. 13: `ns_light=GREEN` **excludes** `ew_light=GREEN`

State of lights at reset. This constraint requires that whenever reset occurs, the `ns_light` turns off. This property shows that `ns_light=OFF` is the consequence of reset. Remember that the reset input over here is not an event but a state as it can stay high or low for multiple clock cycles. Reset is an active low input so when `reset=0` is the state, the implication relation ensures `ns_light=OFF`. From our library of graphical properties, we use the implies operator for the relation, shown in Figure 14.

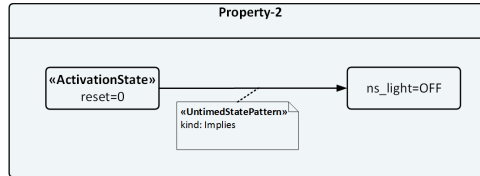


Fig. 14: `reset=0` **implies** `ns_light=OFF`

State of lights during emergency. This constraint requires that whenever the emergency sensor triggers, `ns_light` switches from GREEN to YELLOW to RED. The book (chapter 7, SystemVerilog assertions handbook [19]) uses the timing diagram to explain the intended temporal relation of the constraint. Text further specifies at another place in the book;

The design also takes into account emergency vehicles that can activate an emergency sensor. When the emergency sensor is activated, then the North-

South and East-West lights will turn RED, and will stay RED for a minimum period of 3 cycles.

Yet the SystemVerilog assertion for the constraint tests the `ns.light` equals RED after two cycles of the emergency. The YELLOW state is never tested. This vindicates our statement that the textual requirement specifications are usually ambiguous, not precise, bulky and the information is scattered. We can implement this property using the triggers relation for the event `emgcy_sensor` and the state `ns_light=RED`, as shown in Figure 15. A delay of exactly one clock cycle is shown using the duration constraint.

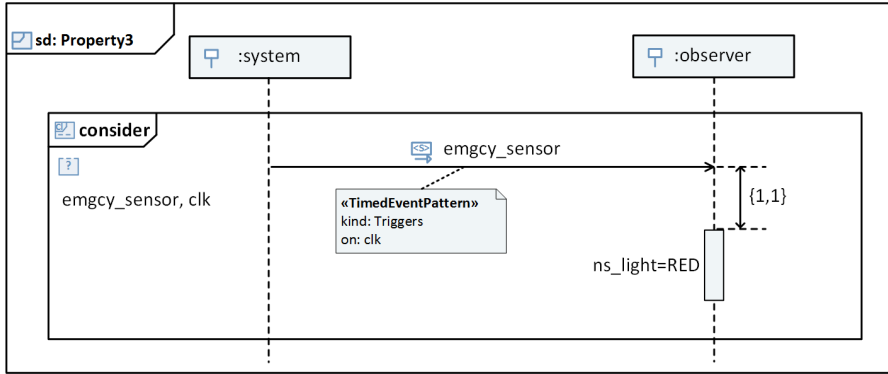


Fig. 15: `emgcy_sensor` triggers `ns_light=RED` after [1,1] on `clk`

Safety, GREEN to RED is illegal. Need YELLOW. This constraint is another example of the difference between the textual specification and the constraint implemented as assertion. Though the YELLOW state is specified in the text but it is never tested in the assertion. Here the graphical approach is clear and precise in using the *causes* relation for states, shown in Figure 16. This property here ensures state YELLOW leads to state RED. A little varying intent can also be implemented using the forbids relation `ns_light=GREEN forbids ns_light=RED` which seems to be the desired one for the requirement ‘green to red is illegal’.

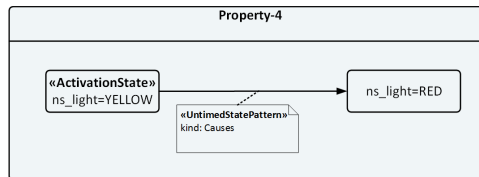


Fig. 16: `ns_light=YELLOW` causes `ns_light=RED`

Safety, GREEN and YELLOW at the same time are illegal. This constraint specifies when the highway light is GREEN, farm-road light must not be YELLOW and vice-versa. This specification requires two similar patterns to verify the behavior. For simplicity, here we only consider one pattern. Graphically it is modeled using the *excludes* relation for states, shown in Figure 17.

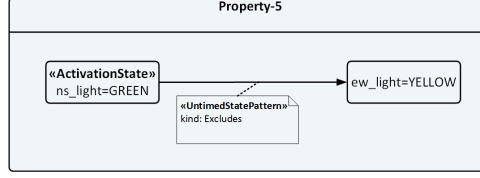


Fig. 17: ns_light=GREEN **excludes** ew_light=YELLOW

6.1 Observers

Graphical properties are implemented in the code as observers. Verification by observers is a technique widely used in property checking [26, 2, 16]. They check the programs for the property to hold or to detect anomalies. In the presented framework, each graphical pattern is finally transformed into a unique observer code for a specific abstraction level (like TLM, RTL). It proposes to create a library of verification components for each graphical pattern. An observer provides implementation to the semantically sound graphical patterns

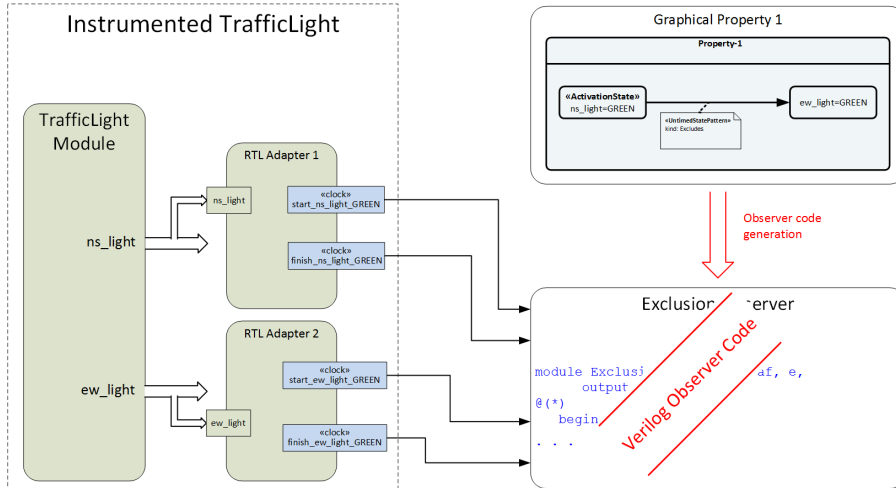


Fig. 18: Integration of Observer in the Verification Environment

(established through semantic MARTE models). An observer consists of a set of input parameters, usually state transition events and a base clock. A special violation output is there to flag any anomaly in the behavior.

One important thing to note here is that there is a gap between the way property is captured in Verilog or other low-level HDLs and what the system specification actually requires. Design requirements are not implemented exactly as per specification but as per the capabilities of the implementation language. That is what we saw in the case of property 4 earlier in this section. In our framework, to make the design process less ambiguous we propose to use a straight forward mapping of requirements to pattern elements (states and events). Hence when the requirements specification speaks about state information, patterns should model states and when we have event information in the specification, patterns should model events.

The way these patterns work is by relying on *adapters* as a glue logic. These adapters convert the signal or group of signals from the system to state events. The property patterns implemented in the framework use these state events. So adapters come in-between the module-under-verification and the observers. They receive inputs in the form of design module interface signals and state values. From this they generate the appropriate logical output clocks to be consumed by the observers. Figure 18 shows the integration of observer code in the verification environment. Every design language should have its own set of adapters e.g., if the design module is in VHDL, adapters written in VHDL should be used that will interact with the design and provide the appropriate inputs to the observer. For example, *start_ns.light.GREEN* is an event that triggers whenever the traffic light output *ns.light* turns GREEN. In SystemVerilog, we can implement this logic using the *\$rose* or the *\$past* operators. The adapter code for the state is given as:

```
start_nslight_GREEN = $past(ns_light!=GREEN)
                      && ns_light==GREEN;
```

Timed RTL observers are cycle-accurate and need system clocks to operate. However the observers for untimed patterns (like excludes) do not need system clock. Observers in other abstraction levels may have different requirements.

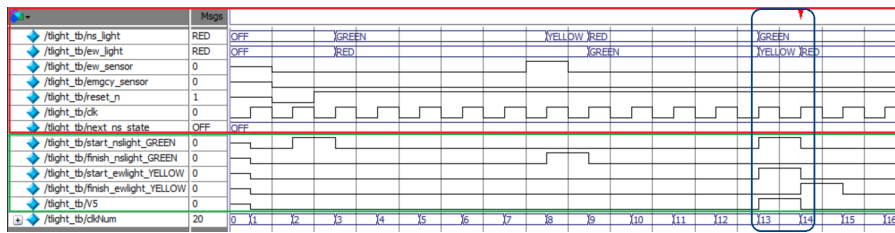


Fig. 19: Simulation Results of an Observer for the Traffic Light Controller

6.2 Results

Observer-based verification is the technique in which observer code is bound to the design module and is simulated with different input scenarios. For the verification of the traffic light controller, various observers (implementing temporal/logical patterns) from a predefined library were instantiated next to the design module in QuestaSim verification software [41]. Simulation trace in Figure 19 shows the design signals in the upper half (outlined red) and logical clocks from the adapter in the lower half (outlined green). To keep the figure simple, we only show the property 5 signals in which a design anomaly occurs. Though the first four constraints satisfy this particular execution scenario, the last exclusion relation between the `ns_light=GREEN` and `ew_light=YELLOW` fails, as shown by the red marker in the execution trace (encircled in Figure 19). This is exactly the case with this faulty FSM as presented in the book (chapter 7 case study) [19]. To summarize, some of the observations made from this work are:

- In this particular case study, we gathered fourteen different requirements specified in the book scattered all over the chapter. We managed to model all the requirements using nineteen graphical patterns, as some of the requirements needed more than one pattern. Of all these nineteen patterns, fourteen were implemented using the *causes relation*, three were the *excludes relation* while two *precedes relation* were used. We noted that in general state-based systems, triggering/causality is a very common behavior relating the states.
- There were two types of anomalies that we detected using graphical patterns: specification and design anomalies. While there was only one *design anomaly* in the case-study that was introduced intentionally by the book authors, the *specification anomalies* were more common and dangerous. As the book presents SystemVerilog verification properties for the requirements specification, we detected at least three occurrences where the design intent was partially or completely misinterpreted in the SystemVerilog assertions (SVAs). More importantly, we also traced two conflicting specifications targeting the same requirement.

7 Expressing Graphical Patterns Formally

This section highlights the third major contribution of this proposed framework. As discussed earlier, all the state-state and state-event relations presented previously can be encoded in CCSL, which in turn can then be used to generate HDL code for the intended relation. But a more direct approach to expressing these operators is to formally encode them as automata directly. Then these automata can easily be encoded in HDLs like Verilog. This framework advocates the direct approach as a fast, less complex alternative to two-step approach using TemLOPAC and TimeSquare tool. Next we discuss the automa-

ton of some of the state/event relations and the derived Verilog code. A comprehensive list of these operator automata is given on the project website [30].

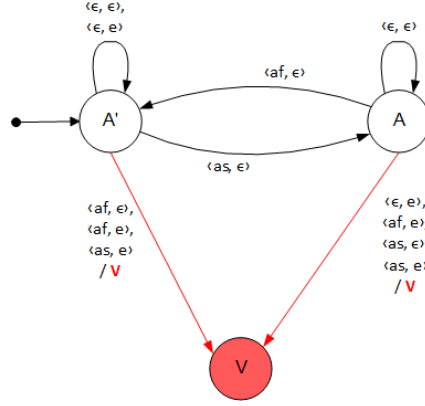


Fig. 20: Automaton of e forbids A

To discuss the automata for the state relations, we start with the less complex **state-event relations**. The *forbids* relation $e \neg A$ can be viewed as a collection of three distinct interacting events e , a_s and a_f . Figure 20 describes the automaton of exclusion relation of a state and an event. Here the trigger $\langle \epsilon, \epsilon \rangle$ represents a no operation transition which occurs at each state when nothing else is happening. As the state events a_s and a_f are mutually exclusive, the first transition value in $\langle \epsilon, \epsilon \rangle$ represents the state A events, while the second one represents event e . Here the transition $\langle a_f, e \rangle$ is the coincident occurrence of events a_f and e . The automaton given in the figure consists of two states A' and A representing the current status of state A and a violation state V . Any transition causing violation state is irreversible and in any such scenario system is considered violated and invalid. Similar exclusion behavior can be represented in CCSL using the two relations, as

$$\begin{aligned}
 e &\searrow a_s \boxed{\leq} a_f \\
 a_s &\boxed{\sim} a_f
 \end{aligned}$$

The first rule samples e on the event a_s and the result is tested to precede the a_f . If e comes in between the a_s and a_f , then the sampling will give wrong results. Here it is important to note that the second CCSL relation is what we call as the ‘*state integrity rule*’ while the first one establishes the desired exclusion relation. These two CCSL relations are equivalent to our proposed automaton and apparently our direct approach is much effective with regards to code size and ease of use.

The trigger operator in its basic form consists of two state events a_s, a_f and an external trigger e . The desired behavior automaton is simple, easy to

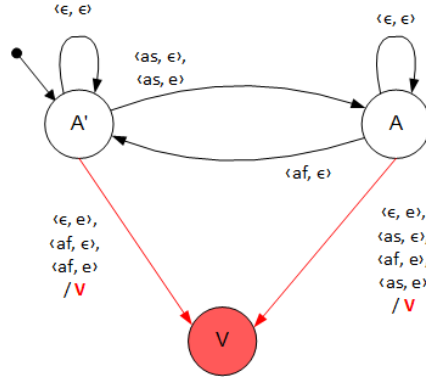


Fig. 21: Automaton of e triggers A

follow and quite similar to the excludes relation, as shown in Figure 21. But when the flexible delayed triggering option is added, the desired automaton explodes quickly (though the number of states are still two) and no more easy to trace just on paper (shown in Figure 22). Several additional variables (ds, df, i, array d[] etc) are added for event count and time keeping. Remember to improve the state machine readability the violation transitions are not shown. All the transitions not shown explicitly are violations. Using CCSL the desired pattern can be achieved using,

$$e(min) \rightsquigarrow clk \boxed{\leq} as \boxed{\leq} e(max) \rightsquigarrow clk$$

$$as \boxed{\sim} af$$

Here again the second equation is the integrity rule for the states while the first one defines the triggering. In this equation, the event e delayed for 'min' clock events on clk is expected to precede the event a_s . Also the event a_s is expected to precede the event e delayed for 'max' number of clock cycles on clk. So we see that a single operator (trigger) of our framework is represented by five operators in CCSL to get the desired behavior.

For the **state-state relations**, the automaton of the most complex state relation *A precedes B by [m,n] on clk* is given in Figure 23. The introduction of the duration constrained between m and n ticks of clock clk raises many corner cases and hence FSM becomes complex. The automaton in the Figure 23 shows the level of complexity and is not meant to be read. However, high-resolution automaton diagrams are available on project website [30] along with the much detailed version of these automata and resulting CCSL and Verilog code. There are four states in the FSM corresponding to the two state variables A and B. When modeling the same behavior in CCSL we would require several relations,

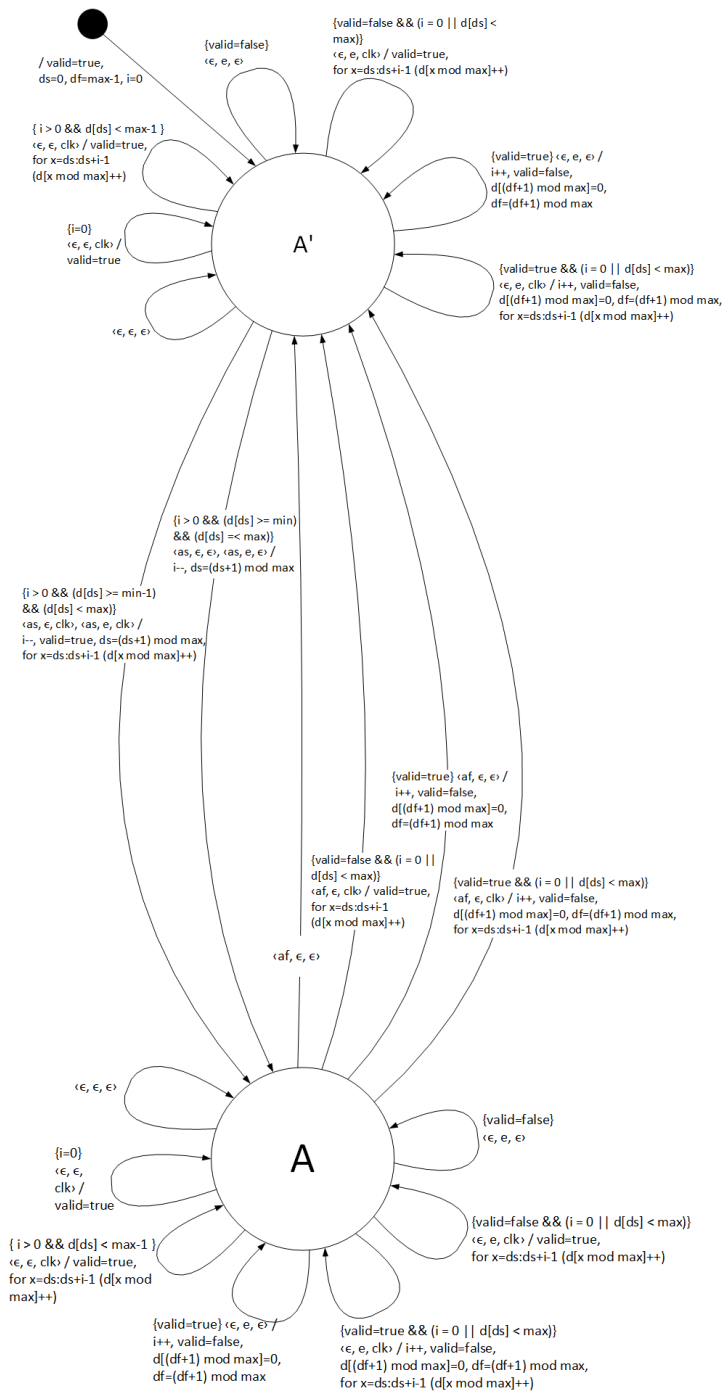


Fig. 22: Automaton of e triggers A after [m,n] on clk

$$\begin{aligned}
af(min) \rightsquigarrow clk &\sqsubseteq bs \sqsubseteq af(max) \rightsquigarrow clk \\
as &\sim af \\
bs &\sim bf
\end{aligned}$$

Here again second and third CCSL relations are the state integrity rules for A and B while the first one is the precedence relation with the delay operator to enforce the limits min and max. If we observe, this relation is quite similar to the one for trigger, as both the operators with limit constraints behave quite similar. The equation tells us that a_f delayed for some min clock cycles should precede b_s and then in the second part b_s must precede the same a_f signal delayed this time for max cycles of clock.

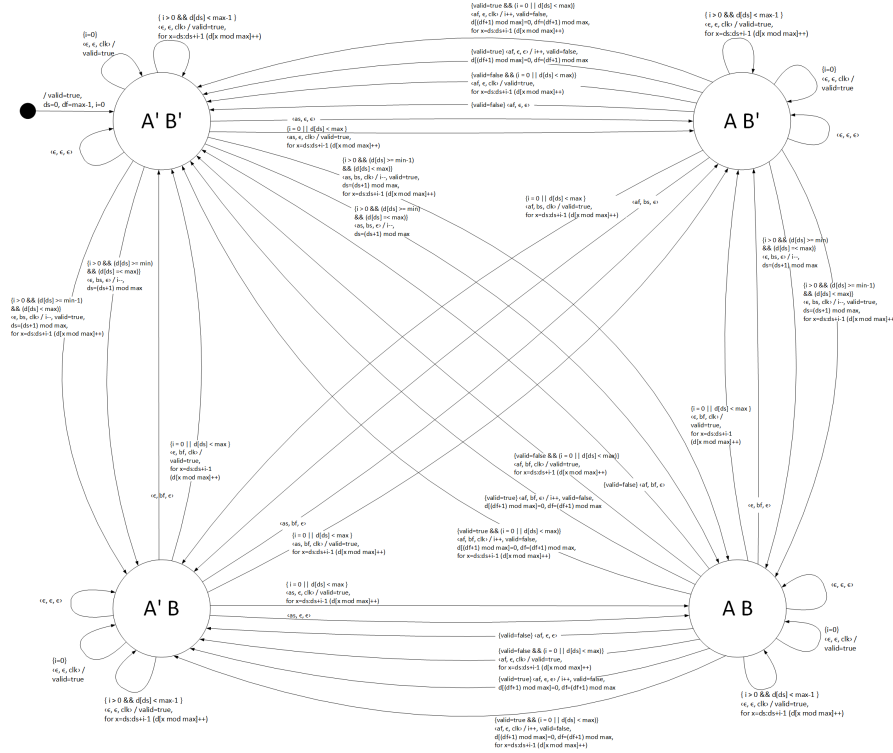


Fig. 23: Automaton of A precedes B by $[m,n]$ on clk

8 Implementation

Figure 24 shows the implementation approach used for the proposed framework. The system modeling can be done in any UML tool but we suggest to use

Eclipse-based tools (such as Papyrus UML) as it well incorporates our model transformation plugin. Modeling patterns using state machine diagrams requires Observation profile.

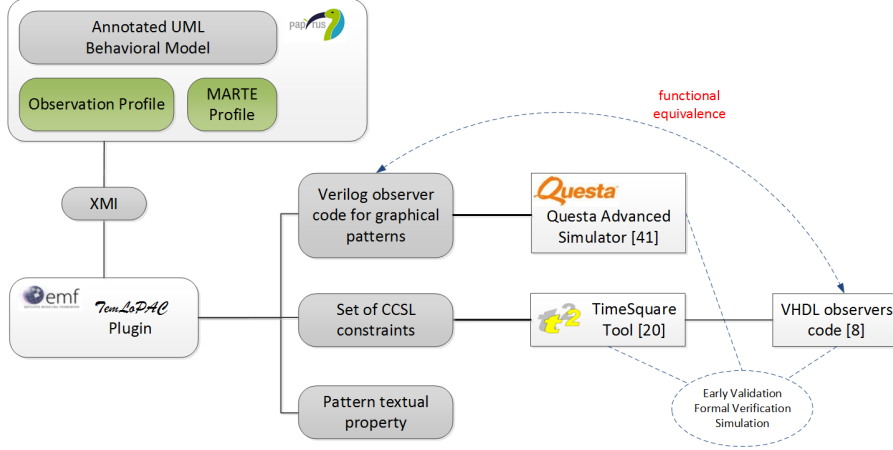


Fig. 24: Framework Work-flow and Model transformation

We have developed a model transformation plugin named TemLOPAC (Temporal and Logical Pattern Analyzer and Code Generator). It is implemented in Java based on the Eclipse Modeling Framework (EMF) [55,59]. The model transformation plugin is available online from the project website [31]. The plugin generates three types of code output files: pattern textual property, its observer as a Verilog code implementation, and the equivalent CCSL constraints (as shown in Figure 24). The graphical pattern textual code is based on the state relation operators presented in the earlier sections. These operators are the direct textual description of the graphical properties. The generated Verilog code is the observer for these relational operators derived directly from the state operator automata described earlier. Lastly, the transformation plugin also generates the equivalent code in CCSL in the form of CCSL constraints.

Representation of state-based behavior as CCSL constraints provides numerous advantages. CCSL is effectively being used in collaborative industrial projects [57,45,58] for verification and validation purposes. The generated CCSL constraints from our TemLOPAC plugin can be used in TimeSquare tool [20] for simulation to find a possible bug in the specification. As the CCSL constraints are available in the design long before the actual implementation, they provide an insight for the key performance metrics. Hence TimeSquare provides an opportunity for early validation of the system behavior and simulation of various possible test cases. Other than simulation, TimeSquare also provides the facility to generate VHDL code [8]. This generated code can be used as observers in physical design implementations. Moreover, researchers

have also shown that CCSL specifications can be used to partially generate SystemC code [46].

This two-step code generation (from UML to CCSL to VHDL) is equivalent to our EMF-based code generation directly from the model. As an example, we model the forbids state-event logical/causal pattern between the event *e* and state *A* (shown earlier in Figure 7). The state property generated from the annotated UML model is straightforward textual representation.

e forbids A

The generated Verilog code is the observer for the forbids operator implemented from the automaton directly.

```

module Forbids2 (
    input as ,
    input af ,
    input e ,
    output violation );

    reg [1:0] FSM = 0;
    reg v = 0;

    always @ (as or af or e)
    begin
        case (FSM)
            0 :
                if (as==1'b0 && af==1'b0 && e==1'b0) FSM=0;
                else if (as==1'b0 && af==1'b0 && e==1'b1) FSM=0;
                else if (as==1'b1 && af==1'b0 && e==1'b0) FSM=1;
                else
                    begin
                        FSM=2;
                        v=1'b1;
                    end
            1 :
                if (as==1'b0 && af==1'b0 && e==1'b0) FSM=1;
                else if (as==1'b0 && af==1'b1 && e==1'b0) FSM=0;
                else
                    begin
                        FSM=2;
                        v=1'b1;
                    end
            default :
                FSM=2;
                v=1'b1;
        endcase
    end

```

```
assign violation = v;
endmodule
```

Lastly the CCSL code generated from the UML using EMF plugin is given next.

```
e exclusiveWith startA
startA alternatesWith finishA
```

where the logical clocks *startA* and *finishA* represents the start and end boundaries of state A respectively. This generated code contains two parts, the relation enforcing temporal/logical property and the relation ensuring state integrity. The state integrity rule ensures the state start event alternates with the occurrence of state terminate event (i.e., two state start events back-to-back are illegal).

8.1 Discussion

Regarding the proposed framework, some implementation and performance issues need to be addressed. One such issue is the *scalability and maintenance of property models* created under this framework. The algorithms to compute CCSL constraints take exponential time regarding the number of clocks. So it can be expensive if we have lots of clocks. But for observation, we treat the clock properties as independent entities with no dependence on each other. We don't address the system properties as a whole. Normally a typical property only covers two or three clocks. So as such this approach does not have any issues regarding scalability and maintenance of property models no matter how large the system grows. We suggest the development of a library of these logical and temporal properties which can be used/reused across multiple projects. With the passage of time, new property patterns can be added to the UML model library. A detailed discussion of adding and maintaining these properties in UML is shown in the TemLOPAC installation guide and users' manual [31].

The *context of use for the graphical patterns* is various kinds of embedded systems, like safety critical systems or Electronic Design Automation (EDA) domain, where verification and validation are immensely important. This not only saves the cost of design/prototyping but is critical sometimes in its operations (like avionics or life-saving medical devices). The proposed framework helps to validate such system models early in the design cycle. Moreover, it helps to generate observers that facilitate the design verification by bridging the gap between the requirements and the verification code.

This work has several *limitations* too. This framework is not meant for modeling to generate the structural or behavioral code. It is only dealing with the verification aspects of the system. This framework does not provide any alternative for requirements, it only provides a way to verify or early validate them. Also this framework, does not automate the design effort to create UML models. It is still done manually and is the job of the system designer to

correctly extract requirements from specification and turn them into useful graphical patterns.

9 Conclusion and Future Work

This paper presents a natural way to interpret UML diagrams annotated with features from MARTE to specify system requirements. The framework proposed a UML based approach to capture properties in a bid to replace temporal logic properties like in LTL. It also proposed a way to extend the existing capabilities of CCSL which can though represent state relations but is not practically meant for that task. The framework identified two major categories of patterns, state-based and mixed state/event relations. Semantics of the states in both types of properties have been expressed as state-start and state-end events and can be expressed in the form of CCSL specification. This CCSL specification can then be analyzed to detect bugs in the system specification. An exhaustive set of state relations have been proposed based on Allen's work and Dwyer's patterns. Later these relations are modeled graphically using a subset of state machine diagrams and sequence diagrams coupled with features from MARTE time model. This framework has presented a tool plugin for model transformation of such graphical patterns directly into CCSL and observers based-on Verilog HDL.

The presented work provided a comprehensive outlook on the implementation and use of graphical patterns. However it would be interesting to see application of this approach from the examples of different domains like safety critical systems or automotive embedded systems. Moreover serious attempts can be made to formally incorporate CCSL in the proposed framework. Presently CCSL is used to represent event-event relations in a system. We propose a new unified language CCSL+, an extension of CCSL, that can handle all sorts of state-based, event-based or mixed relations.

Acknowledgements This project is partially funded by NSTIP (National Science Technology and Innovative Plan), Saudi Arabia under the Track "Software Engineering and Innovated Systems" bearing the project code "13-INF761-10".

References

1. Abrial, J.R., Börger, E., Langmaack, H.: The Stream Boiler Case Study: Competition of Formal Program Specification and Development Methods. In: Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control (the Book Grew out of a Dagstuhl Seminar, June 1995)., pp. 1–12. Springer-Verlag, London, UK, UK (1996). URL <http://dl.acm.org/citation.cfm?id=647370.723887>
2. Aceto, L., Burgueño, A., Larsen, K.: Model Checking via Reachability Testing for Timed Automata. In: B. Steffen (ed.) Tools and Algorithms for the Construction and Analysis of Systems, *Lecture Notes in Computer Science*, vol. 1384, pp. 263–280. Springer Berlin Heidelberg (1998). DOI 10.1007/BFb0054177. URL <http://dx.doi.org/10.1007/BFb0054177>

3. Al-Lail, M., Sun, W., France, R.B.: Analyzing Behavioral Aspects of UML Design Class Models against Temporal Properties. In: *Quality Software (QSIC)*, 2014 14th International Conference on, pp. 196–201 (2014). DOI 10.1109/QSIC.2014.56
4. Alfonso, A., Braberman, V., Kicillof, N., Olivero, A.: Visual Timed Event Scenarios. In: *Proceedings of the 26th International Conference on Software Engineering, ICSE '04*, pp. 168–177. IEEE Computer Society, Washington, DC, USA (2004). URL <http://dl.acm.org/citation.cfm?id=998675.999423>
5. Allen, J.F.: Maintaining Knowledge About Temporal Intervals. *Commun. ACM* **26**(11), 832–843 (1983). DOI 10.1145/182.358434. URL <http://doi.acm.org/10.1145/182.358434>
6. André, C.: Syntax and Semantics of the Clock Constraint Specification Language (CCSL). Research Report RR-6925, INRIA (2009). URL <https://hal.inria.fr/inria-00384077>
7. André, C., Deantoni, J., Mallet, F., de Simone, R.: The Time Model of Logical Clocks available in the OMG MARTE profile. In: S.K. Shukla, J.P. Talpin (eds.) *Synthesis of Embedded Software: Frameworks and Methodologies for Correctness by Construction*, p. 28. Springer Science+Business Media, LLC 2010 (2010). URL <https://hal.inria.fr/inria-00495664>. Chapter 7
8. André, C., Mallet, F., DeAntoni, J.: VHDL Observers for Clock Constraint Checking. In: *Industrial Embedded Systems (SIES)*, 2010 International Symposium on, pp. 98–107 (2010). DOI 10.1109/SIES.2010.5551372
9. André, C., Mallet, F., de Simone, R.: Modeling Time(s). In: G. Engels, B. Opdyke, D. Schmidt, F. Weil (eds.) *Model Driven Engineering Languages and Systems, Lecture Notes in Computer Science*, vol. 4735, pp. 559–573. Springer Berlin Heidelberg (2007). DOI 10.1007/978-3-540-75209-7_38. URL http://dx.doi.org/10.1007/978-3-540-75209-7_38
10. Autili, M., Grunske, L., Lumpe, M., Pelliccione, P., Tang, A.: Aligning Qualitative, Real-Time, and Probabilistic Property Specification Patterns Using a Structured English Grammar. *IEEE Transactions on Software Engineering (TSE)* **41**(7), 620–638 (2015). DOI 10.1109/TSE.2015.2398877
11. Autili, M., Inverardi, P., Pelliccione, P.: Graphical Scenarios for Specifying Temporal Properties: An Automated Approach. *Automated Software Engg.* **14**(3), 293–340 (2007). DOI 10.1007/s10515-007-0012-6. URL <http://dx.doi.org/10.1007/s10515-007-0012-6>
12. Autili, M., Pelliccione, P.: Towards a Graphical Tool for Refining User to System Requirements. *Electron. Notes Theor. Comput. Sci.* **211**, 147–157 (2008). DOI 10.1016/j.entcs.2008.04.037. URL <http://dx.doi.org/10.1016/j.entcs.2008.04.037>
13. Baresi, L., Ghezzi, C., Zanolin, L.: Modeling and Validation of Publish/Subscribe Architectures, pp. 273–291. Springer Berlin Heidelberg, Berlin, Heidelberg (2005). DOI 10.1007/3-540-27071-X_13. URL http://dx.doi.org/10.1007/3-540-27071-X_13
14. Bauer, A., Leucker, M., Schallhart, C.: Runtime Verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.* **20**(4), 14:1–14:64 (2011). DOI 10.1145/2000799.2000800. URL <http://doi.acm.org/10.1145/2000799.2000800>
15. Bellini, P., Nesi, P., Rogai, D.: Expressing and Organizing Real-time Specification Patterns via Temporal Logics. *J. Syst. Softw.* **82**(2), 183–196 (2009). DOI 10.1016/j.jss.2008.06.041. URL <http://dx.doi.org/10.1016/j.jss.2008.06.041>
16. Bensalem, S., Bozga, M., Krichen, M., Tripakis, S.: Testing Conformance of Real-Time Applications by Automatic Generation of Observers. *Electronic Notes in Theoretical Computer Science* **113**, 23–43 (2005). DOI <http://dx.doi.org/10.1016/j.entcs.2004.01.036>. URL <http://www.sciencedirect.com/science/article/pii/S157106610405251X>. *Proceedings of the 4th Workshop on Runtime Verification (RV 2004)*
17. Chai, M., Schlingloff, B.H.: Runtime Verification: 5th International Conference, RV 2014, Toronto, ON, Canada, September 22–25, 2014. *Proceedings*, chap. Monitoring Systems with Extended Live Sequence Charts, pp. 48–63. Springer International Publishing, Cham (2014). DOI 10.1007/978-3-319-11164-3_5. URL http://dx.doi.org/10.1007/978-3-319-11164-3_5
18. Clarke, E., Grumberg, O., Peled, D.: *Model Checking*. MIT Press (1999). URL <https://books.google.com.om/books?id=Nmc4wEaLXFEC>

19. Cohen, B., Venkataramanan, S., Kumari, A., Piper, L.: *SystemVerilog Assertions Handbook: for Dynamic and Formal Verification*, 2nd edn. VhdlCohen Publishing, Palos Verdes Peninsula, CA, USA (2010)
20. Deantoni, J.: TimeSquare: Logical Time Matters. URL <http://timesquare.inria.fr/>
21. Di Guglielmo, L., Fummi, F., Orlandi, N., Pravadelli, G.: DDPSL: An Easy Way of Defining Properties. In: *Computer Design (ICCD)*, 2010 IEEE International Conference on, pp. 468–473 (2010). DOI 10.1109/ICCD.2010.5647654
22. Drechsler, R., Soeken, M., Wille, R.: Formal Specification Level: Towards Verification-driven Design based on Natural Language Processing. In: *Specification and Design Languages (FDL)*, 2012 Forum on, pp. 53–58 (2012)
23. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in Property Specifications for Finite-state Verification. In: *Proceedings of the 21st International Conference on Software Engineering, ICSE '99*, pp. 411–420. ACM, New York, NY, USA (1999). DOI 10.1145/302405.302672. URL <http://doi.acm.org/10.1145/302405.302672>
24. Gascon, R., Mallet, F., Deantoni, J.: Logical Time and Temporal Logics: Comparing UML MARTE/CCSL and PSL. In: *Proceedings of the 2011 18th International Symposium on Temporal Representation and Reasoning, TIME '11*, pp. 141–148. IEEE Computer Society, Washington, DC, USA (2011). DOI 10.1109/TIME.2011.10. URL <http://dx.doi.org/10.1109/TIME.2011.10>
25. Guernic, P.L., Gautier, T., Talpin, J., Besnard, L.: Polychronous Automata. In: *2015 International Symposium on Theoretical Aspects of Software Engineering, TASE 2015*, pp. 95–102. IEEE Computer Society (2015). DOI 10.1109/TASE.2015.21. URL <http://dx.doi.org/10.1109/TASE.2015.21>
26. Halbwachs, N., Lagnier, F., Raymond, P.: Synchronous Observers and the Verification of Reactive Systems. In: M. Nivat, C. Rattray, T. Rus, G. Scollo (eds.) *Algebraic Methodology and Software Technology (AMAST93), Workshops in Computing*, pp. 83–96. Springer London (1994). DOI 10.1007/978-1-4471-3227-1_8. URL http://dx.doi.org/10.1007/978-1-4471-3227-1_8
27. Harris, I.: Extracting Design Information from Natural Language Specifications. In: *Design Automation Conference (DAC)*, 2012 49th ACM/EDAC/IEEE, pp. 1252–1253 (2012). DOI 10.1145/2228360.2228591
28. Haxthausen, A.E.: Automated Generation of Formal Safety Conditions from Railway Interlocking Tables. *International Journal on Software Tools for Technology Transfer (STTT)* **16**(6), 713–726 (2014). DOI 10.1007/s10009-013-0295-9. URL <http://dx.doi.org/10.1007/s10009-013-0295-9>
29. IEEE: Standard for Property Specification Language (PSL) Std 1850-2010 (2010). URL <http://standards.ieee.org/findstds/standard/1850-2010.html>
30. Khan, A.M.: Semantics of Graphical Patterns using UML/MARTE. Research report, NSTIP. URL <http://www.modeves.com/patterns.html>
31. Khan, A.M.: TemLoPAC: Temporal and Logical Pattern Analyzer and Code-generator EMF Plugin. URL <http://www.modeves.com/temlopac.html>
32. Khan, A.M., Mallet, F., Rashid, M.: Natural Interpretation of UML/MARTE Diagrams for System Requirements Specification. In: *Industrial Embedded Systems (SIES)*, 2016 11th IEEE International Symposium on (2016)
33. Konrad, S., Cheng, B.H.C.: Real-time Specification Patterns. In: *Proceedings of the 27th International Conference on Software Engineering, ICSE '05*, pp. 372–381. ACM, New York, NY, USA (2005). DOI 10.1145/1062455.1062526. URL <http://doi.acm.org/10.1145/1062455.1062526>
34. Kugler, H., Harel, D., Pnueli, A., Lu, Y., Bontemps, Y.: Temporal Logic for Scenario-based Specifications. In: *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'05*, pp. 445–460. Springer-Verlag, Berlin, Heidelberg (2005). DOI 10.1007/978-3-540-31980-1_29. URL http://dx.doi.org/10.1007/978-3-540-31980-1_29
35. Leucker, M., Schallhart, C.: A Brief Account of Runtime Verification. *The Journal of Logic and Algebraic Programming* **78**(5), 293 – 303 (2009). DOI <http://dx.doi.org/10.1016/j.jlap.2008.08.004>. URL <http://www.sciencedirect.com/science/article/pii/S1567832608000775>. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLA-COS07)

36. Li, X.: A Characterization of UML Diagrams and their Consistency. In: 11th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'06), pp. 10 pp.– (2006). DOI 10.1109/ICECCS.2006.1690356
37. Mallet, F.: Clock Constraint Specification Language: Specifying Clock Constraints with UML/MARTE. *Innovations in Systems and Software Engineering* 4(3), 309–314 (2008). DOI 10.1007/s11334-008-0055-2. URL <http://dx.doi.org/10.1007/s11334-008-0055-2>
38. Mallet, F.: Formal Modeling and Verification of Cyber-Physical Systems: 1st International Summer School on Methods and Tools for the Design of Digital Systems, Bremen, Germany, September 2015, chap. MARTE/CCSL for Modeling Cyber-Physical Systems, pp. 26–49. Springer Fachmedien Wiesbaden, Wiesbaden (2015). DOI 10.1007/978-3-658-09994-7_2. URL http://dx.doi.org/10.1007/978-3-658-09994-7_2
39. Mallet, F., André, C.: UML/MARTE CCSL, Signal and Petri nets. Research Report RR-6545, INRIA (2008). URL <https://hal.inria.fr/inria-00283077v4>
40. Mallet, F., de Simone, R.: Correctness Issues on MARTE/CCSL Constraints. *Sci. Comput. Program.* **106**, 78–92 (2015). DOI 10.1016/j.scico.2015.03.001. URL <http://dx.doi.org/10.1016/j.scico.2015.03.001>
41. Mentor Graphics: Questa Advanced Simulator. URL <https://www.mentor.com/products/fv/questa/>
42. Object Management Group (OMG): UML Profile for MARTE: Modeling and Analysis of Real-time Embedded Systems (2011). URL <http://www.omg.org/spec/MARTE/1.1/PDF/>
43. Object Management Group (OMG): Unified Modeling Language (UML), Superstructure Specification, Version 2.4 (2011). URL <http://www.omg.org/spec/UML/2.4/>
44. Panda, P.R.: SystemC: A Modeling Platform Supporting Multiple Design Abstractions. In: Proceedings of the 14th International Symposium on Systems Synthesis, ISSS '01, pp. 75–80. ACM, New York, NY, USA (2001). DOI 10.1145/500001.500018. URL <http://doi.acm.org/10.1145/500001.500018>
45. Peraldi-Frati, M.A., DeAntoni, J.: Scheduling Multi-Clock Real-time Systems: From Requirements to Implementation. In: 2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, pp. 50–57 (2011). DOI 10.1109/ISORC.2011.16
46. Peters, J., Wille, R., Drechsler, R.: Generating SystemC Implementations for Clock Constraints Specified in UML/MARTE CCSL. In: Engineering of Complex Computer Systems (ICECCS), 2014 19th International Conference on, pp. 116–125 (2014). DOI 10.1109/ICECCS.2014.24
47. Pnueli, A.: The Temporal Logic of Programs. In: 18th Annual Symposium on Foundations of Computer Science, pp. 46–57. IEEE Computer Society (1977). DOI 10.1109/SFCS.1977.32. URL <http://dx.doi.org/10.1109/SFCS.1977.32>
48. Ribeiro, F., Pereira, C., Rettberg, A., Soares, M.: Model-based Requirements Specification of Real-time Systems with UML, SysML and MARTE. *Software & Systems Modeling* pp. 1–19 (2016). DOI 10.1007/s10270-016-0525-1. URL <http://dx.doi.org/10.1007/s10270-016-0525-1>
49. Ribeiro, F., Rettberg, A., Pereira, C., Soares, M.: Annotating SysML Models with MARTE Time Stereotypes for Requirements Specification and Design of Real-Time Systems. In: 2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC), pp. 36–43 (2016). DOI 10.1109/ISORW.2016.15
50. Rigo, S., Azevedo, R., Santos, L.: Electronic System Level Design: An Open-Source Approach, 1st edn. Springer Publishing Company, Incorporated (2011)
51. Runtime Verification Community: Runtime Verification Events 2001-16. URL <http://runtime-verification.org/>
52. Selic, B., Gérard, S.: Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2013)
53. Smith, R.L., Avrunin, G.S., Clarke, L.A., Osterweil, L.J.: PROPEL: An Approach Supporting Property Elucidation. In: Proceedings of the 24th International Conference on Software Engineering, ICSE '02, pp. 11–21. ACM, New York, NY, USA (2002). DOI 10.1145/581339.581345. URL <http://doi.acm.org/10.1145/581339.581345>

54. Soeken, M., Drechsler, R.: *Formal Specification Level - Concepts, Methods, and Algorithms*. Springer (2015). DOI 10.1007/978-3-319-08699-6. URL <http://dx.doi.org/10.1007/978-3-319-08699-6>
55. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: *EMF: Eclipse Modeling Framework 2.0*, 2nd edn. Addison-Wesley Professional (2009)
56. Suryadevara, J.: *Model Based Development of Embedded Systems using Logical Clock Constraints and Timed Automata*. Ph.D. thesis, Malardalen University, Sweden (2013)
57. Suryadevara, J., Sapienza, G., Seceseanu, C., Seceseanu, T., Ellevseth, S.E., Pettersson, P.: Wind Turbine System: An Industrial Case Study in Formal Modeling and Verification, pp. 229–245. Springer International Publishing, Cham (2014). DOI 10.1007/978-3-319-05416-2_15. URL http://dx.doi.org/10.1007/978-3-319-05416-2_15
58. Suryadevara, J., Seceseanu, C., Mallet, F., Pettersson, P.: Verifying MARTE/CCSL Mode Behaviors Using UPPAAL, pp. 1–15. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). DOI 10.1007/978-3-642-40561-7_1. URL http://dx.doi.org/10.1007/978-3-642-40561-7_1
59. The Eclipse Foundation: Eclipse Modeling Framework (EMF). URL <http://www.eclipse.org/modeling/emf/>
60. Walter, S., Rettberg, A.: Formal Requirements for Specification of Timing Constraints in Distributed Real-time Systems. In: 2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC), pp. 28–35 (2016). DOI 10.1109/ISORW.2016.14
61. Watterson, C., Heffernan, D.: Runtime Verification and Monitoring of Embedded Systems. *IET Software* **1**(5), 172–179 (2007). DOI 10.1049/iet-sen:20060076
62. Weikiens, T.: *Systems Engineering with SysML/UML: Modeling, Analysis, Design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2008)
63. Wile, B., Goss, J., Roesner, W.: *Comprehensive Functional Verification: The Complete Industry Cycle (Systems on Silicon)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2005)
64. Yu, H., Talpin, J.P., Besnard, L., Gautier, T., Marchand, H., Guernic, P.L.: Polychronous Controller Synthesis from Marte CCSL Timing Specifications. In: *Formal Methods and Models for Codesign (MEMOCODE)*, 2011 9th IEEE/ACM International Conference on, pp. 21–30 (2011). DOI 10.1109/MEMCOD.2011.5970507
65. Zhang, P., Grunske, L., Tang, A., Li, B.: A Formal Syntax for Probabilistic Timed Property Sequence Charts. In: *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, ASE '09*, pp. 500–504. IEEE Computer Society, Washington, DC, USA (2009). DOI 10.1109/ASE.2009.56. URL <http://dx.doi.org/10.1109/ASE.2009.56>
66. Zhang, P., Li, B., Grunske, L.: Timed Property Sequence Chart. *Journal of Systems and Software* **83**(3), 371–390 (2010). DOI 10.1016/j.jss.2009.09.013. URL <http://dx.doi.org/10.1016/j.jss.2009.09.013>