A Model-driven Framework for Design and Verification of Embedded Systems through SystemVerilog

Muhammad Waseem Anwar[1], Muhammad Rashid[2], Farooque Azam[1], Muhammad Kashif[3] and Wasi Haider Butt[1]

[1]Department of Computer & Software Engineering, CEME, National University of Sciences & Technology (NUST), Islamabad, Pakistan
[2]Computer Engineering Department, Umm Al-Qura University, Makkah, Saudi Arabia
[3] Department of Electronics and Computer Engineering, Istanbul Sehir University, Istanbul Turkey

waseemanwar@ceme.nust.edu.pk , mfelahi@uqu.edu.sa , farooq@ceme.nust.edu.pk , muhammadkashif@std.sehir.edu.tr , wasi@ceme.nust.edu.pk

Abstract

The demands of system complexity and design productivity for embedded systems can be managed by simplifying and reusing the design. Furthermore, these systems should be verified as early as possible in the development process to reduce the cost and effort. The rationale of the proposed framework in this article is to simplify the design and verification process of embedded systems in the context of Model Based System Engineering (MBSE). To achieve this, UMLSV (UML profile for SystemVerilog) is proposed to model the design and verification requirements. Particularly, we introduce various UMLSV stereotypes to model the system design (structure and behavior). Furthermore, a temporal extension of OCL (Object Constraint Language) is used to capture the verification requirements (properties / constraints) in UMLSV. Consequently, the proposed framework allows the modeling of system design (structure and behavior) along with the verification aspects at higher abstraction level. Following the MBSE process, the high-level models and the verification constraints are transformed into synthesizable SystemVerilog RTL (Register Transfer Level) and SystemVerilog Assertions code. This leads to perform the Assertions Based Verification (ABV) of system design in the early development phases by using state-of-the-art simulators. The effectiveness of the proposed framework is demonstrated with the help of multiple case studies including Traffic Lights Controller, Unmanned Aerial Vehicle, Elevator and Car Collision Avoidance System.

Keywords: Model-driven framework; UMLSV; Embedded systems; SystemVerilog

1. Introduction

The increasing use of embedded systems in multiple disciplines such as aerospace, environmental control, critical infrastructure, industrial automation and health care needs a simplified and reusable design at higher abstraction level [1]. Design verification of embedded systems involves certain technical hitches due to the complexity of temporal design characteristics. This urges to perform design verification in the early development stages in order to avoid the higher cost and larger development time. Model Based System Engineering (MBSE) is a promising approach that provides built-in features to verify the system design in the early development phases. Consequently, MBSE has been frequently applied and researched in the domain of embedded systems [2-3].

The MBSE process for embedded systems comprises three main phases: i.e. modeling (requirement specification), model transformation (model-to-model and / or model-to-text) and design verification (formal and /or dynamic). Modeling phase refers to the representation of structural, behavioral and verification requirements at higher abstraction level. The Unified Modeling Language (UML) [14] and its associated profiles have been commonly used to model the system requirements [4][53]. Moreover, several properties specification techniques have been introduced to represent the system constraints at higher abstraction level [15]. Once the modeling phase is completed, the model transformation phase is employed to generate the target output model for design verification. For example, in the case of formal verification, the target model can be timed automata [6] and Z notation [7] etc. Similarly, in the case of dynamic verification, the target model can be the lower level hardware languages like VHDL, SystemC etc.

While MBSE is the paradigm at higher abstraction level, SystemVerilog [9] on the other hand is a renowned hardware design and verification language at Register Transfer Level (RTL). Although SystemVerilog is based on the concepts of traditional hardware languages like Verilog and VHDL, it provides several advanced features like improved data types, interfaces etc. to simplify the development of system design. Moreover, it provides a complete support for Assertion Based Verification (ABV) by means of SystemVerilog Assertions (SVA's). Furthermore, it fully supports object oriented programming concepts [5] like inheritance, polymorphism etc. for the development of complex and large test benches. Additionally, it provides integration capabilities with other hardware languages like SystemC through Direct Programming Interface (DPI) [8]. Finally, SystemVerilog is fully compliant with the Universal Verification Methodology (UVM) [24] standard, therefore, it is not required to develop a customized simulation solution and design verification can be performed directly by utilizing existing simulators like QuestaSIM [10]. Due to the aforementioned advanced design and verification features of SystemVerilog, it is now frequently utilized for the development of highly safety critical systems [11]. Furthermore, the innovative features of SystemVerilog ensures its potential utilization to meet the growing demands of upcoming embedded systems with complex verification requirements.

Although SystemVerilog provides some sophisticated features to simplify the development of complex embedded systems, it operates at lower implementation level. Consequently, the design and verification aspects of modern embedded systems cannot be achieved completely due to the complexity of low level implementation specifics. In this regard, there is a strong need to develop a higher abstraction layer for SystemVerilog and integrate it in the MBSE process, so that, the powerful design and verification features of SystemVerilog can be utilized with simplicity. There is a clear evidence of literature (Section **5**) that shows the existence of such higher abstraction layer for different languages in MBSE. A typical example is ModelicaML [46] which provides higher abstraction layer for Modelica. Furthermore, there exist several studies where higher abstraction layer is proposed for different formalisms to simplify the design and verification activities e.g. Z-MARTE [41] provides higher abstraction layer for Z notation etc. Similarly, there are several attempts (e.g. [51-52] etc.) to bring the low level semantics of SystemC at higher abstraction level. However, to the best of our knowledge (Section **5**), SystemVerilog is rarely utilized in MBSE. Therefore, the industry and academia can greatly benefit by a solution that provides higher abstraction layer for SystemVerilog in MBSE to deliver significant design and verification simplicity. This leads to unify the power of MBSE and SVA's that reduces the gap between design and its verification.

This article presents a novel model-driven framework where a higher abstraction layer for SystemVerilog is introduced to specify both design and its verification requirements. Particularly, we propose UMLSV (**UML** profile for **S**ystem**V**erilog) to represent the structural and behavioral requirements at higher abstraction level. Furthermore, a temporal OCL extension named SVOCL (SystemVerilog in Object Constraint Language) [21] is used to capture the verification requirements in UMLSV. This provides the basis to generate the synthesizable RTL code as well as the SVA's code. To the best of our knowledge (Section **5**), no state-of-the-art model-driven framework exists that transforms the models into both synthesizable SystemVerilog RTL and assertions code for early design verification. The overview of research is shown in **Figure 1**.

This research has been carried out as a part of MODEVES[1] project that deals with a set of tools and techniques for model-based design verification (both formal and dynamic) of safety critical embedded systems. The major contributions of this paper are as follows:

1. The development of UMLSV profile (Section **2**) which comprises several stereotypes to model structural, behavioral and verification requirements of embedded systems hardware at higher abstraction level.
2. The implementation of UMLSV transformation engine (Section **3**) which enables the designer to generate synthesizable SystemVerilog RTL code along with the assertions code from the source UMLSV models.
3. The proposed framework is validated through four benchmark case studies i.e. Traffic Lights Controller, Unmanned Aerial Vehicle, Elevator and Car Collision Avoidance System.

[1] www.modeves.com

It is important to mention that the OCL temporal extension (SVOCL), used to represent the verification requirements, is already developed and published [21]. However, the previously published work only describes that how the OCL can be extended to represent verification constraints at higher abstraction level without: (1) discussing the behavior modeling which is an important part of system design (2) generating the corresponding RTL code which is the ultimate requirement in system design (3) providing the stereotypes for system design and verification. Consequently, this article presents a model driven framework for design and verification requirements of embedded systems. To achieve this, UMLSV (UML profile for SystemVerilog) is proposed to represent the design and verification requirements at higher abstraction level.

The proposed model-driven framework is capable of representing the structural, behavioral and verification requirements at higher abstraction level, as shown in **Figure 1**. Particularly, the SYSML (Systems Modeling Language) [13] Block Definition Diagram (BDD) and the UMLSV stereotypes (Section **2.1**) are proposed to model the system structure. Moreover, the State Machine Diagram (SMD) and various UMLSV stereotypes are proposed (Section **2.2**) to model simple as well as complex behavior of embedded systems. Furthermore, an OCL temporal extension (SVOCL) is used [21] to express the design verification requirements in UMLSV through a particular stereotype (Section **2.3**). As a part of research, the UMLSV transformation engine is developed (Section **3**). Its architecture (Section **3.1**) is divided into four main components (i.e. Application launcher, Java Services, UML-to-SystemVerilog Parser and Code Generator), as shown in **Figure 1**. The UMLSV transformation engine generates SystemVerilog synthesizable RTL code along with the SVA's code from the source UMLSV models. This allows to perform ABV in the UVM-compliant simulator of choice. After the successful design verification, RTL code can be directly deployed into the target device. The application of proposed framework is demonstrated through four case studies (Section **4**). The comparative analysis of proposed model-driven framework with state-of-the-art is presented in Section **5**. Finally, the article is concluded in Section **6**.

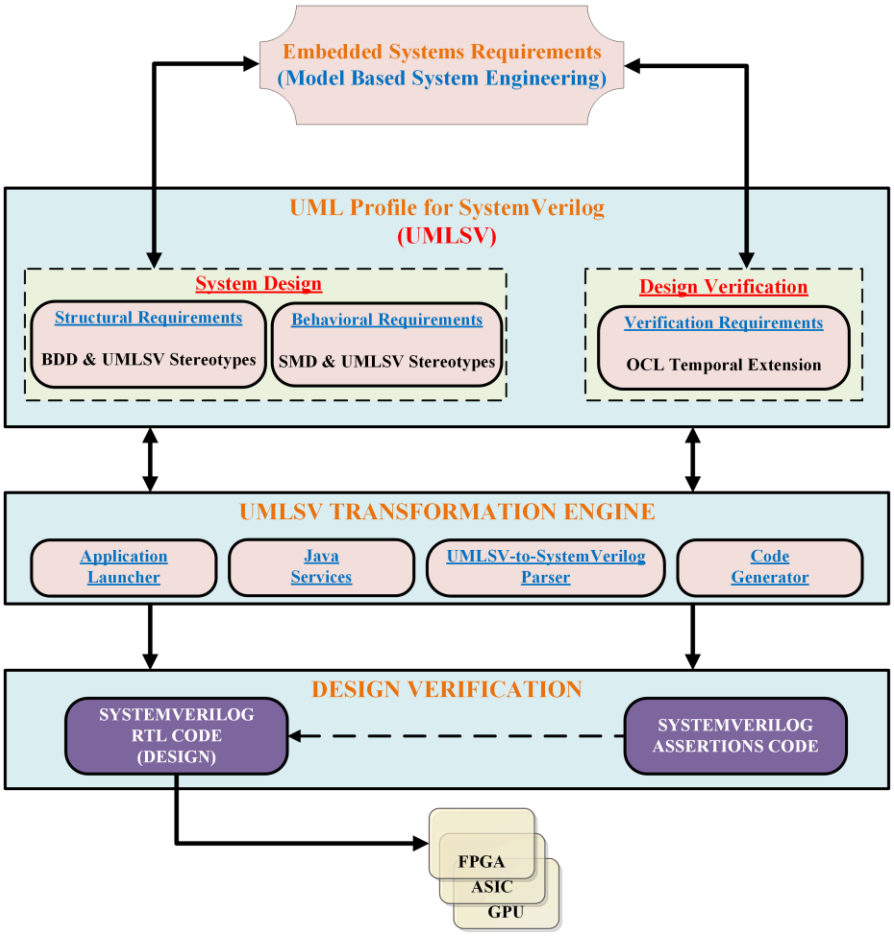

Figure 1: Proposed Model-driven Framework for Design and Verification of Embedded Systems

2. UML profile for SystemVerilog (UMLSV)

Unified Modeling Language (UML) is an Object Management Group (OMG) standard [14] to model system requirements at higher abstraction level. There are three main categories of UML models i.e. Classifiers, Events and Behaviors. The abstract syntax of UML is defined through a meta-model where different classes are expressed through meta-classes. In other words, UML defines a complete meta-model [14] for the development of customized profiles in order to achieve particular objectives. In this regard, papyrus modeling editor [12] fully supports the concepts of UML profiling e.g. the facility to extend UML meta-classes for the creation of required stereotype etc. Therefore, we have used papyrus modeling editor for the implementation of UMLSV. In this section, the description of stereotypes for structural (Section **2.1**) and behavioral (Section **2.2**) requirements is presented. Furthermore, a brief description of SVOCL functions, used to represent the verification constraints, is presented in Section **2.3**.

2.1 Structural Requirements in UMLSV

We utilize the concepts of SYSML Block Definition Diagram (BDD) to model the structural requirements. Particularly, the flow ports are utilized to represent different ports / registers of the system. Data types of the system variables are represented through UML primitive data types like Boolean, Integer, Real etc. Furthermore, the enumeration type can be used to develop a particular data type. Another important aspect, while modeling the embedded systems requirements, is the representation of clock and timer. In this regard, an easy to use solution is provided in UMLSV by utilizing activity diagram. Particularly, the clock and timer are implemented in two separate activity diagrams and provided as a built-in feature of UMLSV profile [27]. However, to create and call the clock and timer activity diagrams in UMLSV, the corresponding stereotypes are developed, as shown in **Figure 2.**

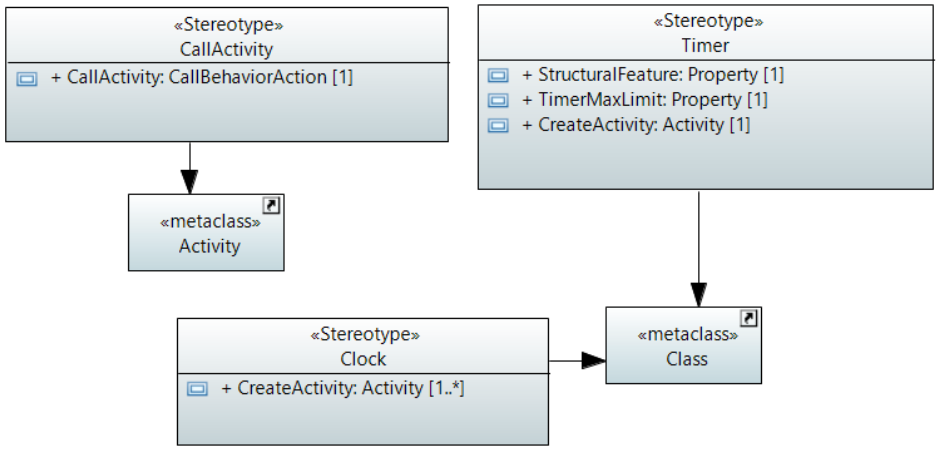

Figure 2: Stereotypes for Clock and Timer Activity Diagrams

The description of Timer, Clock and CallActivity stereotypes, shown in **Figure 2**, are given below:

Timer Stereotype: It is used to set the attributes of timer before calling the built-in timer activity in a particular block. The initial value and the maximum timer value can be set through the *StructuralFeature* attribute and the *TimerMaxLimit* attribute respectively. Maximum timer value specifies the time at which the timer gets reset to its initial value. Finally, the *CreateActivity* attribute is used to create an activity for calling a built-in timer activity.

Clock Stereotype: It is used to create an activity for calling a built-in clock activity. Unlike *Timer* stereotype, it doesn't have any initial and maximum value attributes. Consequently, it is only required to create an activity through the *CreateActivity* attribute. It is important to note that SystemVerilog supports multiple clocks within a single block. Therefore, the multiplicity of *CreateActivity* attribute is 1…*, so that, the desired number of activity diagrams can be created to call the built-in clock activity multiple times (if required) within a single block.

CallActivity Stereotype: The objective of this stereotype is to call the built-in timer and clock activities, created in the timer and clock stereotypes respectively. Essentially, an activity diagram can only be called within another activity diagram. Therefore, this stereotype extends the *Activity* meta-class, as shown in **Figure 2**.

2.2 Behavioral Requirements in UMLSV

There are two main diagrams in UML (i.e. State Machine and Activity) to model the system behavior. The meta-models of both diagrams are given in UML standard [14]. We extend the UML State Machine Diagram (SMD) meta-model and propose various stereotypes in UMLSV to model the diverse behavioral requirements of embedded systems. Since the States and Transitions are the main elements of SMD, the corresponding UMLSV stereotypes are categorized into two types i.e. transition level stereotypes and state level stereotypes. **Figure 3** shows the transition level stereotypes.

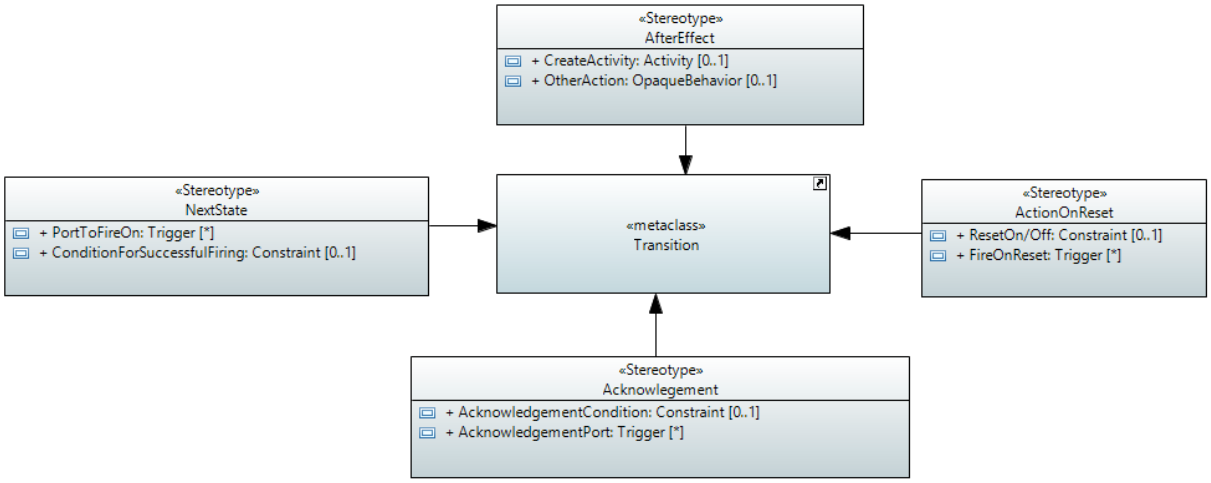

Figure 3: Transition Level Stereotypes

It can be observed from **Figure 3** that all the four stereotypes extend the *Transition* meta-class, thus, they are applicable only to SMD transitions. The description of each stereotype is given below:

NextState Stereotype: It is used to transition/move from one state to another state on the success of a particular constraint condition. Consequently, the constraint condition is specified through the *ConditionForSuccessfulFiring* attribute. There are rare situations where it is not required to specify any constraint condition to move from one state to another state. Therefore, the multiplicity of ConditionForSuccessfulFiring attribute is 0...1. On the other hand, the *PortToFireOn* attribute ensures that any number of ports can be triggered while specifying the constraint condition through *ConditionForSuccessfulFiring* attribute.

AfterEffect Stereotype: While the NextState stereotype is used to transition from one state to another, AfterEffect Stereotype is used to execute the after effect of a successful transition. The

after effects of a transition can be to call a particular activity diagram (e.g. timer) or to specify any other specific condition or action. Therefore, the *CreateActivity* attribute can be used to call a desired activity. Furthermore, *OtherAction* attribute can be used to specify a particular condition / action through OpaqueBehavior in the AfterEffect stereotype.

ActionOnReset Stereotype: It is used to model the concept of a reset condition which is frequently used in the domain of embedded systems. The reset port can be configured to a desired reset condition through the *FireOnReset* attribute. Furthermore, the required reset condition can be configured, through the *ResetOn/Off* attribute, that executes on the basis of reset status i.e. On or Off.

Acknowledgement Stereotype: It provides the acknowledgement status of a particular action i.e. successful or unsuccessful. The port that acknowledges the respective action can be defined through the *AcknowledgementPort* attribute. The condition that evaluates the acknowledgement as successful (true) or unsuccessful (false) can be defined through the *AcknowledgementCondition* attribute.

The aforementioned stereotypes simplify the modeling of transition level operations. However, transitions are only used to move from one state to another state and do not deal with the state operations. Therefore, it is essential to develop stereotypes for the management of state level operations in order to complete the modeling of behavioral requirements. In this regard, we extend *State* meta-class to develop different state level stereotypes in UMLSV, as shown in **Figure 4**.

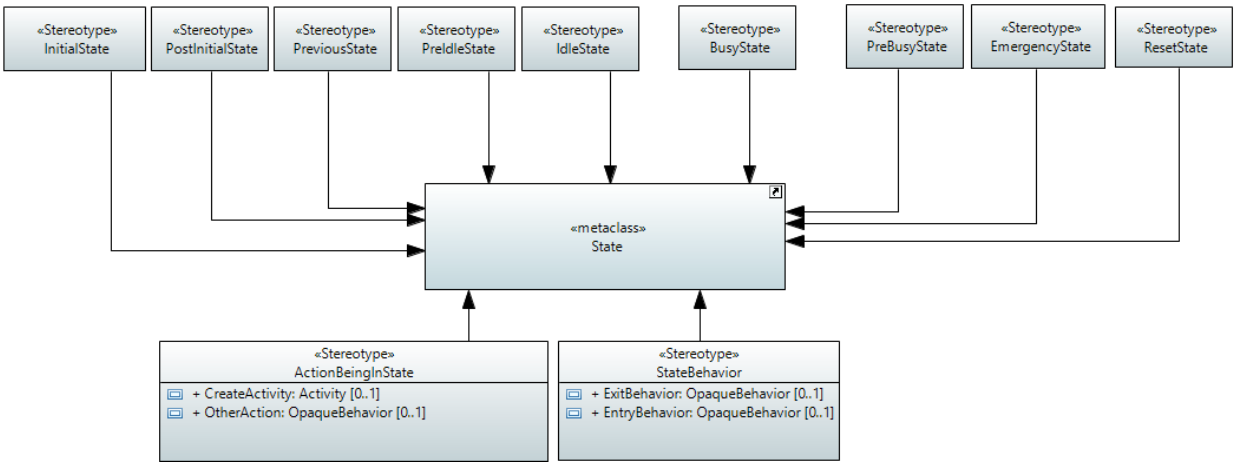

Figure 4: State Level Stereotypes

Stereotypes to Represent the Conceptual States of a System: The design of embedded systems is usually represented through a number of conceptual states like busy, idle etc. Although *State* meta-class provides notations to represent initial and final states of the system, it doesn't offer the features to represent other conceptual states of embedded systems. Therefore, we propose nine stereotypes, extended through the *State* meta-class, to represent the different states of a system, as

shown in **Figure 4**. Initial state of the system can be represented through InitialState stereotype. Similarly, PostInitialState, PreviousState, PreIdleState, IdleState, BusyState, PreBusyState, EmergencyState and ResetState stereotypes can be utilized to represent the conceptual states of the system as per requirements. The inclusion of conceptual state stereotypes in UMLSV not only simplifies the behavioral modeling but also assists in model transformation.

ActionBeingInState Stereotype: It is usually required to perform certain actions in different states of the system. The typical examples are applying a particular activity and execution of a specific condition. To manage such requirements, *ActionBeingInState* stereotype is developed. It comprises two attributes i.e. *CreateActivity* and *OtherAction.* A desired activity diagram can be invoked through the *CreateActivity* attribute. On the other hand, any specific condition or action can be defined and executed through the *OtherAction* attribute by means of OpaqueBehavior.

StateBehavior Stereotype: It is used to represent the behavior of a particular state while the system is entering or exiting that state. Consequently, the entering and exiting behaviors of a state can be specified through the *EntryBehavior* and *ExitBehavior* attributes respectively. It is important to note that a particular state may not have entering and existing behaviors simultaneously. Therefore, the multiplicity of *EntryBehavior* and *ExitBehavior* attributes is 0...1.

It can be argued that the objective of proposed stereotypes can be achieved through the existing SMD concepts. For example, there is no need to develop NextState stereotype because such functionality can be obtainable through UML *Transition* meta-class. Similarly, the entering and existing behaviors can be specified through UML *State* meta-class, therefore, there is no needed to develop StateBehavior Stereotype. Although the aforementioned stereotypes are based on the UML *State* and *Transition* meta-classes concepts, they significantly simplify the modeling of behavioral requirements. Particularly, we only consider the relevant concepts of *State* and *Transition* meta-classes and discard all irrelevant notations while developing the UMLSV stereotypes. For example, to move from current state to next state, it is usually required to only specify the constraint condition and target ports. Therefore, we only consider the two most relevant attributes of *Transition* meta-class in NextState stereotype. Similar is the case with other UMLSV stereotypes. Consequently, the proposed stereotypes provide two key benefits over the existing *State* and *Transition* meta-classes features as follows:

1) UMLSV stereotypes simplify the modeling of system behavior because each stereotype corresponds to a commonly recognized embedded systems operation like Reset, Acknowledgement etc. Furthermore, the proposed stereotypes only contain relevant details about a particular concept.

2) UMLSV stereotypes provide strong foundations to perform accurate model transformation for converting source UMLSV models into target low level SystemVerilog implementation. For example, it is straightforward to design appropriate transformation rules and navigate the source models because each stereotype represents a particular embedded system concept like Reset etc. This significantly simplifies the model transformation process (Section **3**).

To this point, we have presented the details of UMLSV stereotypes to model the structural and behavioral requirements of embedded systems. The application of aforementioned stereotypes is demonstrated through the Traffic Light Controller (Section **4.1**), Unmanned Aerial Vehicle (Section **4.2**), Elevator (Section **4.3**) and Car Collision Avoidance System (Section **4.4**) case studies.

2.3 Verification Requirements in UMLSV

To represent the embedded systems verification requirements in UMLSV, we use the temporal extension of OCL named as SVOCL. The technical details regarding the development of SVOCL can be found at [21]. In this article, we integrate SVOCL in UMLSV to represent verification requirements by means of SVA's. We strongly believe that the summary of SVOCL functions should be included in this article for completeness / readability. Therefore, in this section, we provide the summary of major SVOCL functions with examples. It is important to note that the focus of this section is to explain the practical usage of SVOCL functions in UMLSV as the technical aspects of SVOCL are already available in [21].

2.3.1 SVOCL Essential Functions for Temporal Properties / Constraints

We have used eight essential functions in SVOCL to express the temporal properties / constraints in UMLSV. The summary of functions is given in **Table 1**. The description of each function is as follows:

Table 1: Summary of SVOCL essential functions to represent the temporal properties in UMLSV

Serial No.	Functional Description to Express Temporal Properties / Constraints in UMLSV	Corresponding SVOCL Essential Function
1.	Chronological delay between two expressions	SVSeq (s, d, p)
2.	Successive recurrences of an expression	SVRep (p,r)
3.	To check the change in the value of an expression	SVChanged (expr)
4.	To check the stability in the value of an expression	SVStable (expr)
5.	To check that the value of an expression is changed from True to False	SVFell (expr)
6.	To check that the value of an expression is changed from False to True	SVRose (expr)
7.	Customized conditional statements	Disif expression
8.	Implications of behaviors/sequences/expressions	SVImplication (ant,con,t)

Chronological Delay: It is commonly utilized to express the temporal properties / constraints of embedded systems. In other words, it is used to verify the sequential/chronological delay between

the two terms/expressions in terms of clock cycles. Consequently, we have used "*SVSeq (s, d, p)*" function in SVOCL to express the chronological delay in UMLSV. The parameters "s" and "p" denote the first and second terms respectively whereas the parameter "d" denotes the amount of chronological delay. It is important to note that the parameter "d" can be given as a single value or range of values. For single value, any number in 1 to 99 can be used and 0 is not allowed in the beginning. On the other hand, range of values can be given through four digits where first two digits represent lower range value and last two digits represent the upper range value. This description of parameter "d" is also valid for other SVOCL functions. Finally, the return type of "*SVSeq (s, d, p)*" function is Boolean i.e. True in case the sequence is matched otherwise False.

For further clarification, consider a sample function "SVSeq (a, 5, b)" where parameter "d" (5) is given as a single value. It represents that expression "a" must hold 5 clock ticks later after expression "b". Consider another example "SVSeq ((x=y), 0204, (p=r))" where parameter "d" (0204) is given as a range of values. It denotes that "(x=y)" can hold from any 2 to 4 clocks after "(p=r)".

Successive Recurrences: It implies to check that a specific expression holds a value successively for a certain number of clock cycles. Therefore, we use "*SVRep (p, r)*" function. The parameter "*p*" denotes the expression and "*r*" denotes the number of recurrences. However, semantics of "SVRep" function are very simple that cannot be expressed alone. Consequently, "SVRep" function is used along with the "SVSeq" function to rationally express more composite and comprehensive recurrences. For example, "SVSeq (SVRep (x, 4), 3, b)" denotes that the expression "x" successively holds four times after three clock delays of expression "b". Likewise, "SVSeq (SVRep (w, 6), 2, (p=q)" denotes that if expression "p" is equal to expression "q" then after two clock cycles, expression "w" successively holds six times.

Implications of Behaviors/Sequences/Expressions: It refers to a condition in which in order for a behavior to happen, a prior sequence must have occurred. This prior sequence is recognized as antecedent. The following behavior is known as consequent. In other words, an antecedent sequence implies a consequent property expression. The outcome of implication can be True or False. The implications can be categorized into two types: In overlapping implication, the consequent is evaluated on the same clock tick of a successful antecedent. In non-overlapping implication, consequent is evaluated on the next clock tick after a successful antecedent. To express both (overlapping and non-overlapping) types of implications in UMLSV, we use a single function "SVImplication (ant,con,t)" in SVOCL.

The parameter "ant" denotes the antecedent expression and "con" denotes the consequent expression. The type of implication is denoted through "t" i.e. 1 for overlapping and 0 for non-overlapping. The return type of "SVImplication" function is Boolean i.e. True in case of successful implication otherwise False. For example, "SVImplication ((p=r), (y=z), 1)" denotes that if the expressions "p" and "r" are equivalent, then in the same cycle, expressions "y" and "z" are also equivalent. Likewise, "SVImplication ((p=r), (y=z), 0)" denotes that if the expressions "p" and

“r” are equivalent, then in the next cycle, expressions “y” and “z” will be equivalent. To manage complex implications, there is a provision to use other SVOCL functions (e.g. SVSeq etc.) in SVImplication function. For example, consider a comparatively complex example “SVImplication (SVSeq (a,3,b), c, 1)”. Here, SVSeq (a,3,b) function is used as an antecedent expression which shows that if the expression “a” holds after three clock cycles of expression “b” then expression “c” (consequent expression) also holds in the same cycle (as t=1).

Detection of Alterations in the Expression Values: It is frequently needed to detect the alterations/changes in the expression value. In other words, certain sample valued functions are needed to assess the sampled value of an expression with respect to the current and past time. Typical examples of these sample value functions are given in the following:

- We use “SVChanged(expr)” function to check whether the value of an expression is changed between the last time instance and the current time instance. Furthermore, we use “SVStable(expr)” function to detect whether the value of an expression is stable between the last time instance and the current time instance.
- In addition to SVStable and SVChanged functions, it is also essential to assess whether the value of an expression has changed from True in the last occurrence to False. Similarly, whether the value of an expression is changed to True which was False in the last time occurrence. To manage such requirements, we use “SVFell(expr)” and “SVRose(expr)” functions.

Conditional Statements: It is commonly needed to evaluate a specified conditional statement before executing a particular sequence (property expression). The property expression should be disabled in case the conditional statement is active at that time. In other words, the property expression is valid only if the conditional statement is lifted/removed. Furthermore, if the conditional statement becomes True while assessing the property expression, then outcomes of property expression should be disabled. A typical example of this phenomenon is to evaluate the “reset” condition while executing a particular expression. To manage such conditional statements in UMLSV, we use “Disifexp” expression in SVOCL. For example, the following chronological delay “SVSeq ((x=y), 2, (a=b))” will be disabled upon the activation of “reset”. This situation can be expressed through DisifExp as:

Disif *(reset)*
[
SVSeq ((x=y), 2, (a=b))
]
endDisif

In this section, we provide the summary of SVOCL essential functions to represent temporal properties / constraints in UMLSV. Particularly, the emphasis is to describe the practical usage of

SVOCL functions through examples rather to explain technical details. However, interested readers can found complete technical details about SVOCL in [21].

2.3.2 Integration of SVOCL in UMLSV

In order to integrate the SVOCL in UMLSV, we propose the "SVOCL Constraint" stereotype in the UMLSV to express the verification properties through essential functions (**Table 1**), as shown in **Figure 5**. We have extended the "SVOCL Constraint" stereotype through *Class* and *StateMachine* meta-classes, therefore, it can be applied to SYSML block and / or state machine diagrams as per requirements. The multiplicity of *Constraint* attribute is 1...*, therefore, multiple constraints can be defined in "SVOCL Constraint" stereotype. Consequently, the desired number of design verification requirements can be expressed through "SVOCL Constraint" stereotype by utilizing essential functions accordingly. The applicability of SVOCL functions for the representation of temporal constraints in UMLSV is further demonstrated in Section **4** (Validation) through four case studies.

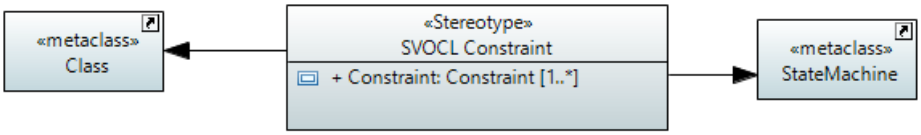

Figure 5: SVOCL Constraint Stereotype to Express the Design Verification Requirements

3. Implementation / Tool Support

The UMLSV profile can be imported in various modeling editors (e.g. MagicDraw [23], Papyrus [12] etc.) to model the design of a particular embedded system. Therefore, it is not required to develop any special modeling tool. In this article, we utilize the papyrus modeling editor to model the four case studies (as detailed in Section **4**). To complete the model-based development cycle, we implement the UMLSV transformation engine. Consequently, the SystemVerilog RTL code along with the SVAs code is generated for assertion based verification. The details are given in subsequent sections.

3.1 Architecture of UMLSV Transformation Engine

The major business logic of the UMLSV transformation engine is implemented in Java. However, to simplify the extraction of UMLSV elements and code generation, we utilize the Acceleo tool [25]. It provides sophisticated features to perform Model-to-Text (M2T) transformation technique. Furthermore, it also provides adequate support for Java language through Java services. The architecture of UMLSV transformation engine is shown in **Figure 6**.

There are four main components of the transformation engine i.e. Application Launcher, Java Services, UMLSV-to-SystemVerilog Parser and Code Generator. The description of each component is given below:

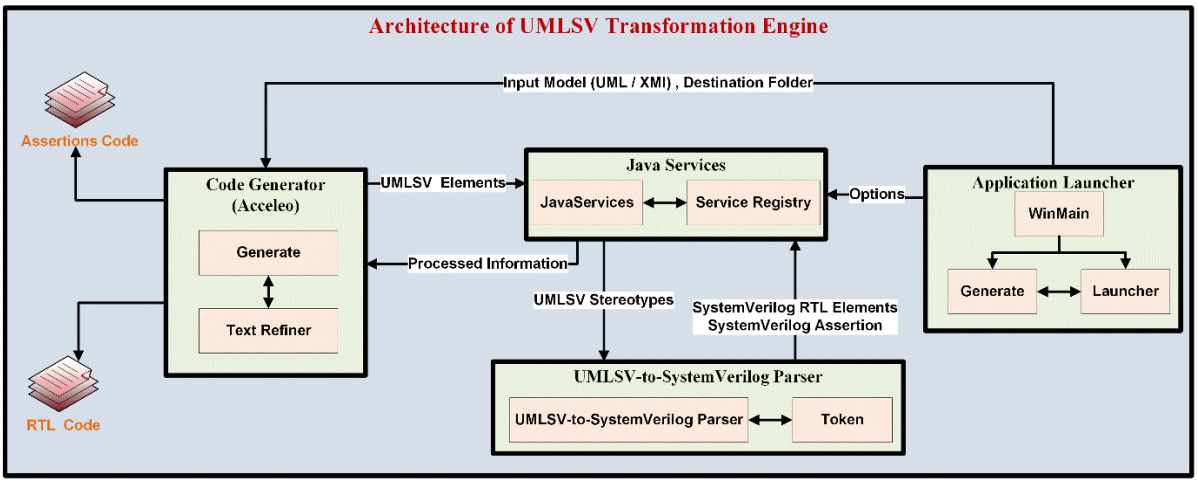

Figure 6: Architecture of UMLSV Transformation Engine

1. **Application Launcher:** This component is responsible to launch the user interface of the transformation engine. It has three sub-components i.e. WinMain (WinMain.java), Generate (Generate.java) and Launcher (Launcher.java). WinMain sub-component implements the user interface. Launcher and Generate sub-components are responsible to pass the Input Model and the Destination Folder attributes, provided by the WinMain, to the *Code Generator* component. Furthermore, various options (settings) are also passed to the *Java Services* component.
2. **Java Services:** This component acts as a main hub for all the other components. It has two sub-components i.e. JavaServices (Javaservices.java) and Service Registry (JavaServices.mtl). Various Java functions are implemented in the JavaServices sub-component to perform the required tasks. For example, it takes the UMLSV elements / stereotypes from the *Code Generator* component, invokes corresponding java functions to perform certain operations, and passes the stereotypes (in particular arrangement) to the *UMLSV-to-SystemVerilog Parser* component which parses the incoming stereotypes to the corresponding SystemVerilog elements / assertions and returns them back to the *Java Services* component. Consequently, this component performs further processing on the data provided by the *UMLSV-to-SystemVerilog Parser* component by considering the options which are passed by the *Application Launcher* component, as shown in **Figure 6**. The processed information is returned back to the *Code Generator* component. Service Registry sub-component is implemented to register java services in Acceleo.
3. **UMLSV-to-SystemVerilog Parser:** This component is responsible to parse the UMLSV stereotypes to equivalent SystemVerilog RTL constructs and assertions. It has two sub-components i.e. UMLSV-to-SystemVerilog Parser (UMLSVtoSVpasrer.java) and Token (token.java). UMLSV-to-SystemVerilog Parser sub-component takes the stereotypes from

the *Java Services* component, parses it to equivalent SystemVerilog assertions / RTL constructs by utilizing the Token sub-component and returns them back to the *Java Services* component. Basically, the constraints expressed through the SVOCL (Section **2.3**) are parsed to equivalent SVA's and other UMLSV stereotypes are parsed to the equivalent RTL constructs as per defined transformation rules (Section **3.2**).

4. **Code Generator:** Finally, SystemVerilog RTL and assertions code has been generated through this component. It has two sub-components i.e. Generate (Generate.mtl) and Text Refiner (TextRefiner.java). Three templates are implemented in the Generate sub-component to generate the SystemVerilog RTL and assertions files. Basically, Generate sub-component extracts the UMLSV elements from the model which are then passed to the *Java Services* component. Subsequently, Generate sub-component receives the processed information in the form of SystemVerilog RTL and assertions code from the *Java Services* component. Finally, it generates the required SV files in the given destination folder. Generate sub-component utilizes the Text Refiner sub-component to manage the formatting issues of the generated SV files.

It can be seen from **Figure 6** that the architecture of the transformation engine is based on a modular approach and there are four major components i.e. Application Launcher, Java Services, UMLSV-to-SystemVerilog Parser and Code Generator. Moreover, the components are loosely coupled with each other and able to work independently. Furthermore, the implementation is carried out in JAVA which is a well-known language to perform complex transformation and frequently used in several MBSE approaches (e.g. [43][45]). Consequently, UMLSV transformation engine is highly scalable and maintainable. For example, it is quite simple to enhance the user interface of the transformation engine by updating the Application Launcher component alone. Similarly, it is fairly possible to implement the Code Generator component through other model-to-text transformation tools (e.g. JET [16]) without altering other components. Finally, it is also straightforward to add or update the existing transformation rules by updating the Java Services and / or UMLSV-to-SystemVerilog Parser components.

User interface of the transformation engine provides various configuration options, as shown in **Figure 7**. The address of input model, developed through the UMLSV profile, can be specified through the respective *Browse* button. The user interface provides three options (i.e. Assertions, RTL and Both) for code generation, as given in the drop down list. It can be seen from **Figure 7** that *Behavior File Name* is currently disabled because only *Assertions* code is selected from the drop down list. The address of *Destination Folder* can be specified through the corresponding *Browse* button. *Reset* button clears the fields of user interface for new configurations. Once all the fields are properly filled, the *Transform* button is activated and the code can be generated by clicking on this button. The status of the transformation (i.e. successful or successful with errors) can be viewed in the *Transformation Status* area. Here, we provide the summary of UMLSV transformation engine. However, further details like user manual, installation guide, sample case studies, source code and an executable jar file can be downloaded from [26].

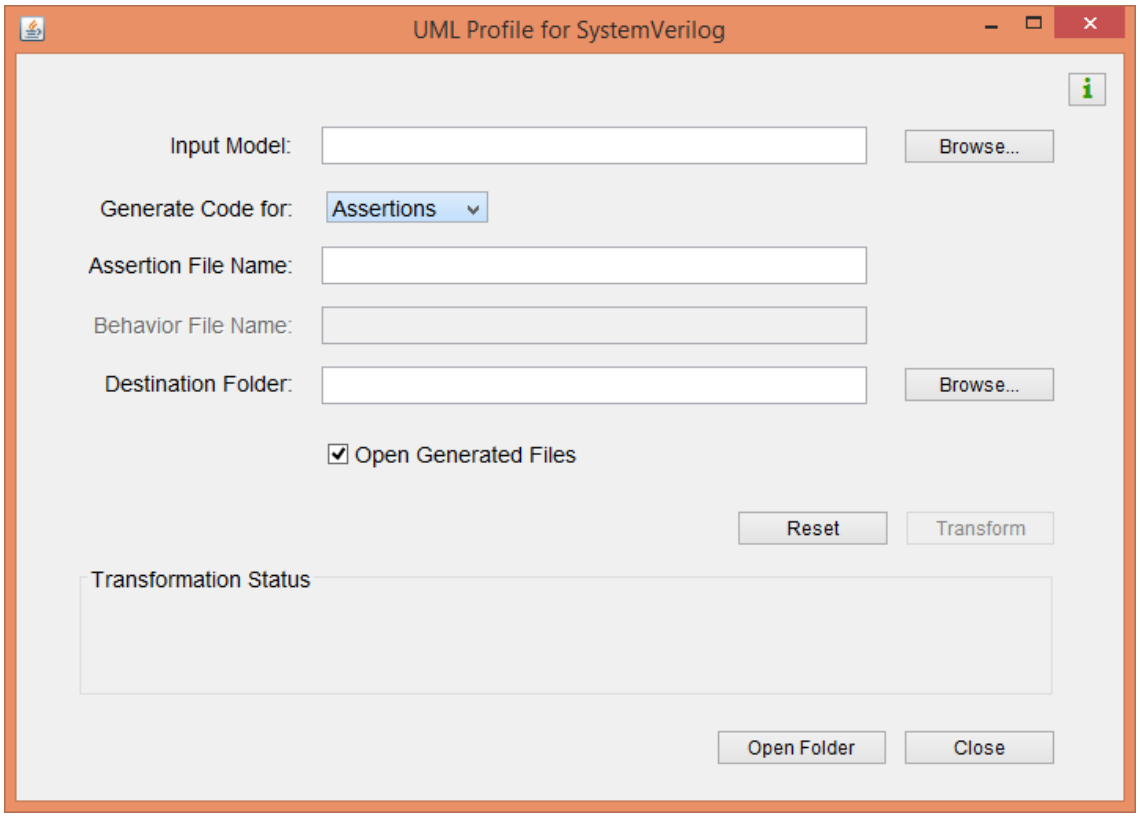

Figure 7: User interface of UMLSV Transformation Engine

3.2 Transformation Rules

It is essential to define some appropriate transformation rules in order to accurately generate the structure, behavior and assertions code from the UMLSV model. Therefore, we develop the transformation rules and implement them in the Java Services and UMLSV-to-SystemVerilog Parser components (Section **3.1**) of the transformation engine. In this section, we summarize the details of the transformation rules.

The structural code is mandatory for both the RTL and assertions code. Therefore, we perform logical mapping of the high level UMLSV structure modeling concepts (Section **2.1**) with the low level SystemVerilog constructs, as given in **Table 2**.

It has been analyzed that the flow ports are conceptually transformed to SystemVerilog registers where the flow port with Inout direction is transformed to SystemVerilog wire. Similarly, it has also been analyzed that different primitive data types are conceptually transformed to the equivalent SystemVerilog data types. In addition, signal type is utilized to represent the SystemVerilog clock / timer signal. On the basis of conceptual mapping that is shown in **Table 2**, we develop transformation rules to accurately generate the SystemVerilog structural code which is further incorporated in the behavioral and assertions code.

Table 2: Logical mapping between UMLSV structural modeling and SystemVerilog

Serial No.	UMLSV Structural Modeling	Comparable SystemVerilog Constructs
1.	Flow Ports	Registers
	Name	Register Name
	In Direction	Input logic / register
	Inout Direction	Wire
	Out Direction	Output logic / register
2.	Data Types	Data types
	Boolean	Boolean
	Real	Real
	Integer	Integer
	String	String
	Enumeration	Enumeration
3.	Signal	Clock / Timer

The logical mapping of high level UMLSV behavioral modeling concepts (Section **2.2**) to low level SystemVerilog constructs is given in **Table 3**.

Table 3: Logical mapping between UMLSV behavior modeling and SystemVerilog

Serial No.	UMLSV Behavioral Modeling Concepts	Comparable SystemVerilog Concepts
1.	State machine name	SystemVerilog module name
2.	FSM stereotype	Representation of a single/multiple FSM(s) within the main module
3.	State level stereotypes	State level code concepts
	State name	Name of the state in the context of code
	Initial state	Initial state in the code (Similar is the case with other such stereotypes)
	ActionBeingInState	Code logic for certain actions within a particular state
	StateBehavior	Code logic when the system enters / leaves a particular state
4.	Transition level stereotypes	Transition level code concepts
	NextState	Conditional code to transition between states

	AfterEffect	Code specification after the execution of a transition
	ActionOnReset	Code specification on reset
	Acknowledgement	Conditional code specification for acknowledgement status
5.	Other state machine elements	Equivalent SystemVerilog Concepts
	Fork	Parallel execution for blocks of code
	Join	Conditional statement by joining the incoming expressions
	Choice	Conditional branching e.g. If / else, Case

The conceptual mapping between the SVOCL functions and the SystemVerilog constructs is straightforward, as given in **Table 4**. However, it is quite possible that one essential function may call another functions recursively in order to express complex SVA's. For example, essential functions like SVSeq, SVRep etc. can be recursively called in SVImplication. This significantly increases the transformation complexity. To manage such recursive nature of the SVOCL in the transformation engine, we develop various transformation rules to handle all the possible recursions. These rules lead to accurately generate simple as well as complex SVA's. It is important to note that the transformation rules to generate SVA's code from SVOCL are already implemented in the previously published work [21]. Therefore, in UMLSV transformation engine, the implementation of SVOCL transformation rules is integrated in the UMLSV-to-SystemVerilog Parser component for the generation of SVA's code.

Table 4: Logical mapping between essential functions of SVOCL and SystemVerilog constructs

Serial No.	Essential Functions of OCL Extension	Equivalent SystemVerilog Constructs
1.	SVRose (expr)	$rose
2.	SVFell (expr)	$fell
3.	SVChanged (expr)	$changed
4.	SVPast (expr, ct)	$past
5.	SVStable (expr)	$stable
6.	SVRep (p,r)	* operator (Consecutive repetition)
7.	SVSeq (s, d, p)	## operator (Sequential delay)
8.	Disif expression	disable iff
9.	SVImplication (ant,con,t)	overlapping |-> and non-overlapping |=> implication

4. Validation

In this section, the applicability of UMLSV is demonstrated through four case studies i.e. Traffic Lights Controller (Section **4.1**), Unmanned Aerial Vehicle (Section **4.2**), Elevator (Section **4.3**) and Car Collision Avoidance System (Section **4.4**). For all the four case studies, papyrus modeling editor [12] is used to model the system design and its verification requirements. The details are summarized in subsequent sections.

4.1 Case Study of Traffic Lights Controller

The objective of Traffic Lights Controller is the traffic management at East-West (EW) and North-South (NS) roads intersection, as depicted in **Figure 8**. North-South is the main road, therefore, the green light is assigned to it most of the time to handle the traffic load. A sensor is installed at the East-West road (termed as EW sensor) to detect the presence of a vehicle on the EW road. On the activation of EW sensor, the green light should be assigned to the EW road provided that the green light on the NS road is ON for enough time. There is also an emergency sensor installed at the EW/NS junction for the efficient passage of emergency vehicles. Whenever the emergency sensor gets activated, the traffic lights of both the roads (EW & NS) will switch to the red for a minimum period of 3 clock cycles in order to pass the emergency vehicle without any interruption.

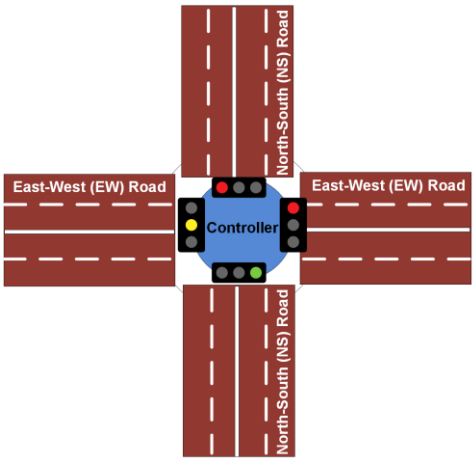

Figure 8: Traffic Lights Controller

4.1.1 Requirements Specifications

The structure of Traffic Lights Controller is modeled through the UMLSV guidelines (Section **2.1**). It can be observed from **Figure 9** that the flow ports are used to model the system registers / variables while the enumeration is used to model the set of traffic lights. Similarly, the Signal is used for typing the clock and timer flow ports in order to receive the signal on these ports at the required time instant. Moreover, the timer and clock are modeled in two separate UML activity diagrams which are provided as a built-in part of the UMLSV profile. Therefore, it is only required to import the timer and clock activities in the model. Relevant details can be found at [27].

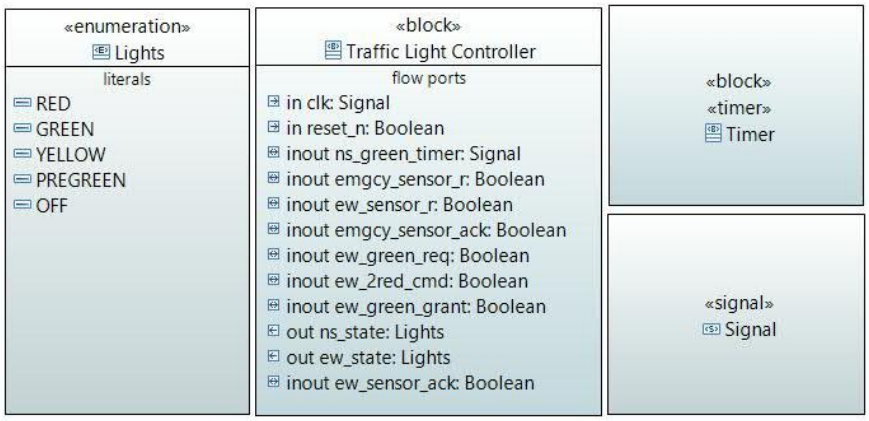

Figure 9: Structure of the Traffic Lights Controller in UMLSV

The behavior of Traffic Lights Controller is modeled through UMLSV stereotypes (Section **2.2**) with the help of SMD, as shown in **Figure 10**. The following behavioral requirements are considered:

- The NS light should stay green until one of the EW or emergency sensor is activated.
- If the NS green timer is three and the NS light is red, then the NS light should turn to green.
- The NS light should turn to yellow from green upon the activation of emergency sensor even if the value of NS timer is three.
- The NS green timer should increment on every clock for a maximum count of three. However, the NS green timer should reset to zero on the controller reset, or upon the NS yellow light. Similar is the case with EW green timer.
- The EW light should turn to yellow on the activation of emergency sensor or the EW timer has reached three.

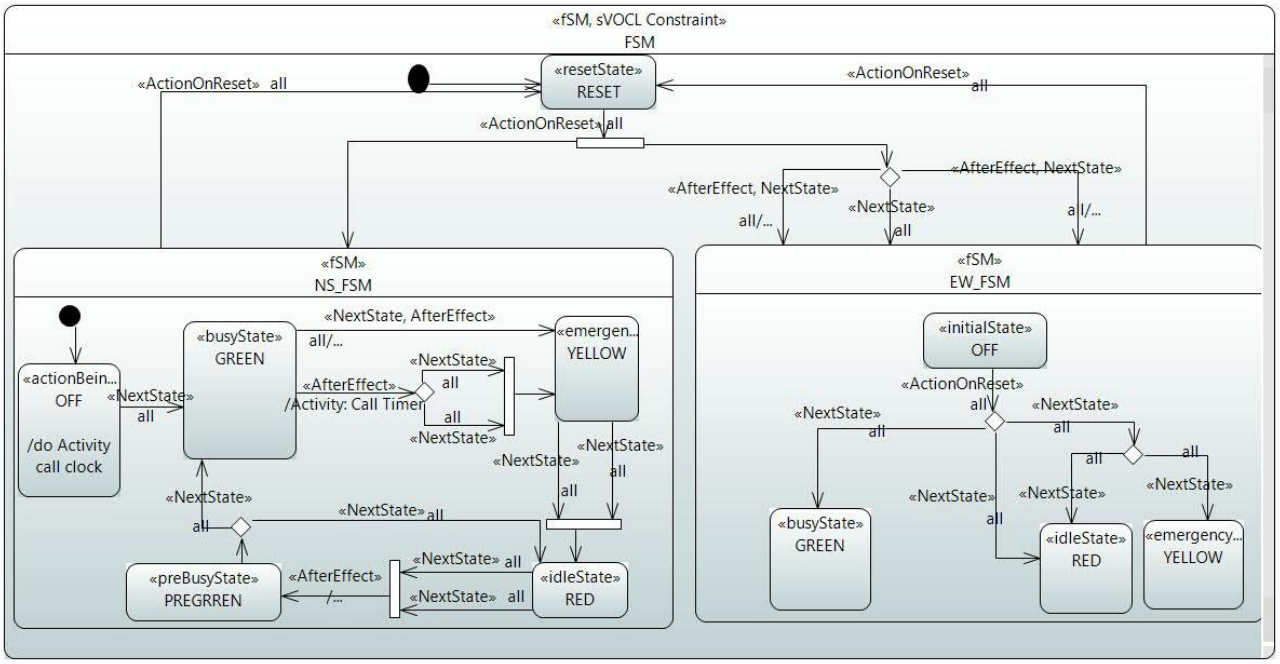

Figure 10: Behavior of Traffic Lights Controller in UMLSV

Two concurrent state machines named as EW_FSM and NS_FSM are modeled for the representation of EW and NS lights behavior respectively. State level stereotypes (Section **2.2**) are applied to represent different states of the Traffic Lights Controller. For example, *ResetState* stereotype is applied to represent the reset state of the system. Similarly, *InitialState*, *IdleState*, *BusyState*, *EmergencyState* and *PreBusyState* stereotypes of UMLSV are applied to represent the different states of the Traffic Lights Controller as per behavioral requirements. Furthermore, another state level stereotype *ActionBeingInState* is applied to call the timer activity.

In addition to the state level stereotypes, we also apply various transition level stereotypes (Section **2.2**) to model behavioral requirements. For example, *ActionOnReset* stereotype is applied to move from other states to the reset state on the activation of reset condition. Moreover, *NextState* stereotype is also applied to move from one state to another state on the success of a given constraint condition which is specified through ConditionForSuccessfulFiring attribute. Furthermore, *AfterEffect* stereotype is applied to execute the after effect of a successful transition. To manage complex behavioral requirements, both *NextState* and *AfterEffect* stereotypes are also applied on a single transition, as shown in the **Figure 10**. To summarize, we utilized both state and transition level stereotypes of UMLSV to model the design of traffic light controller, as shown in **Figure 9** & **Figure 10**. Now, the following design verification requirements of Traffic Lights Controller are expressed in the model through SVOCL (Section **2.3**):

1. **Constraint 1**: It assures that both the EW and the NS lights should not be green at the same time.
2. **Constraint 2**: It guarantees that whenever the reset gets activated then the NS light should switch to OFF.

3. **Constraint 3**: It assures that whenever the reset gets activated then the EW light should switch to OFF.
4. **Constraint 4**: This is a safety constraint which assures that there shouldn't be a direct switching of the NS light from green to red.
5. **Constraint 5**: This is a safety constraint which assures that there shouldn't be direct switching of the EW light from green to red.
6. **Constraint 6**: This constraint assures that if the emergency or the EW sensor are not activated and the NS light is green, then the next cycle is also green.
7. **Constraint 7**: This constraint assures that in case of emergency, the NS lights should switch from green to yellow to red.
8. **Constraint 8**: This constraint assures that in case of emergency, the EW lights should switch from green to yellow to red.
9. **Constraint 9**: This constraint assures that the NS and EW lights should not be green and yellow at the same time.
10. **Constraint 10**: This constraint ensures that the NS light should keep on green for ns_green_timer == 3 before it can change.
11. **Constraint 11**: This constraint confirms the correct sequence of NS and EW lights upon the activation of EW sensor.

We applied the *SVOCL Constraint* stereotype of UMLSV to express all the eleven verification requirements through essential functions of the SVOCL (Section **2.3.1**). We customized the Papyrus modeling editor [12] to support the essential functions of the SVOCL, as shown in **Figure 11**. Here, we include the screenshot of one complex verification requirement (Constraint 10) which is expressed in UMLSV. The interested readers can view the screenshots of all the design verification requirements at [27].

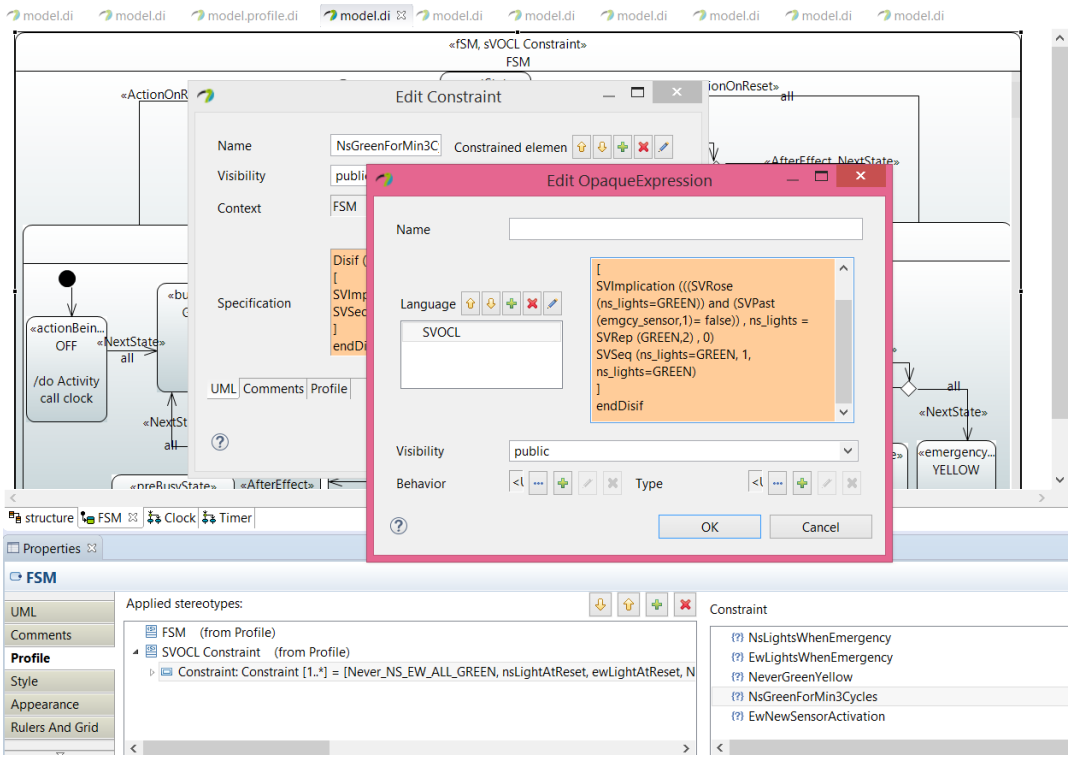

Figure 11: Expressing Design Verification Requirements of Traffic lights Controller in UMLSV

To this point, we have successfully modeled the structure (**Figure 9**), behavior (**Figure 10**) and design verification requirements (**Figure 11**) of the Traffic Lights Controller. Now, we utilize the transformation engine to generate the SystemVerilog RTL and assertions code from the Traffic Lights Controller model, as shown in **Figure 12**. The transformation engine generates two files, *model_RTL.sv* and *model_assertions.sv* for SystemVerilog RTL and assertions code respectively. The structural code is also generated and included in both the files accordingly.

4.1.2 Design Verification

In this section, we perform the design verification of Traffic Lights Controller by utilizing the generated code. Although the generated code can be utilized to perform the design verification in any UVM-compliant simulator of choice, we utilize Mentor Graphics QuestaSIM for design verification of Traffic Lights Controller due to its sophisticated ABV capabilities. The design verification (simulation) results are shown in **Figure 13**.

It can be seen in **Figure 13** that the constraint 11 has been violated during the simulation indicating the design errors. We inspect the source of this failure and take the corrective measures in the design accordingly. Finally, after intensive verification (simulation), the correctness of Traffic Lights Controller design has been confirmed. Here, we provide the summarized details of Traffic Lights Controller design verification and further information can be found at [28].

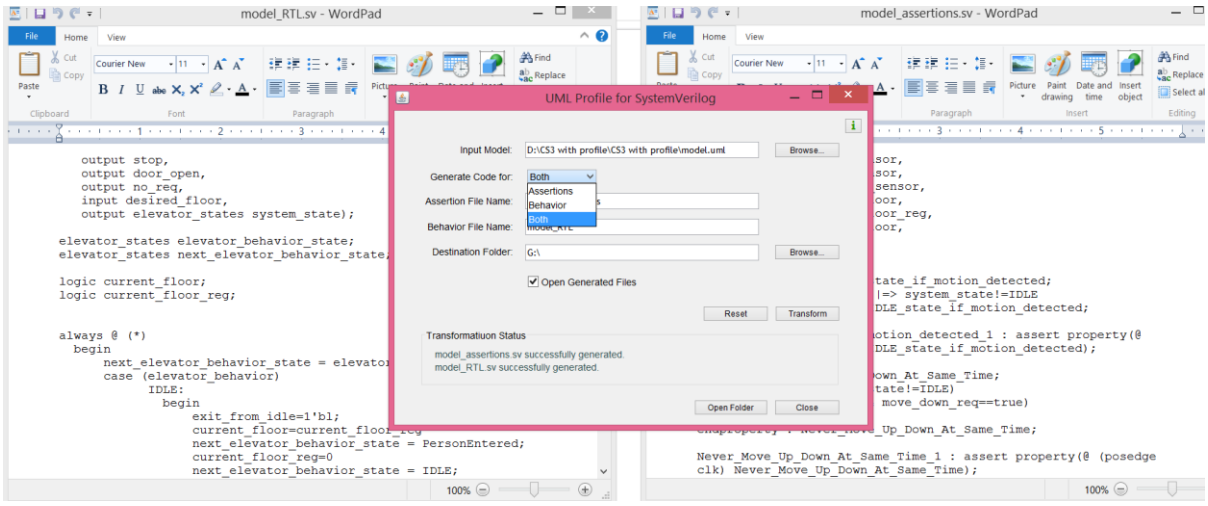

Figure 12: Generating RTL and Assertions Code through UMLSV Transformation Engine

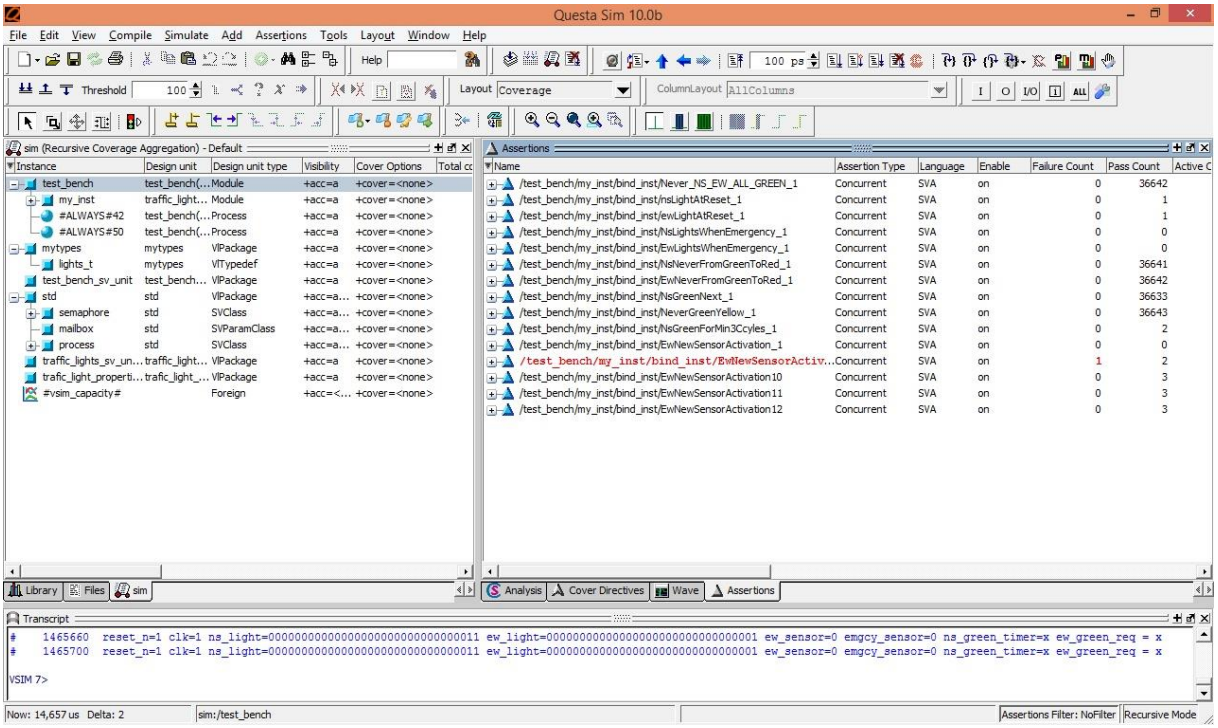

Figure 13: Design Verification of Traffic Lights Controller in QuestaSIM

4.1.3 Code Synthesis

To confirm our claim that the UMLSV transformation engine is capable of generating the fully synthesizable RTL code, we perform the code synthesis of Traffic Lights Controller design in Xilinx Vivado [29]. The results are given in

Table *5*.

Table 5: Code synthesis of traffic lights controller on Xilinx Zynq-xc7z020 device

FPGA Resource	% Used	Available	Utilized
Slice LUTs	0.03	53200	18
Slice Registers (as Flip-Flop)	< 0.01%	16400	13
BRAM	0.0 %	140	0

It can be seen from

Table *5* that for logic implementation, 18 LUTs and 13 slice registers (inferred from the registered output signals and binary state encoding) have been used in the Traffic Lights Controller design. The results (

Table *5*) assures that the UMLSV transformation engine is capable of generating fully synthesizable RTL code.

4.2 Unmanned Aerial Vehicle (UAV) System

The Unmanned Aerial Vehicle (UAV) is typically an aircraft without a pilot on board. It can be controlled remotely or fly autonomously on the basis of pre-programmed flight plans. UAVs have quite complex dynamic automation systems starting right from their flight to a safe landing.

4.2.1 Requirements Specifications

The structural and behavioral requirements of UAV case study are represented through UMLSV, as shown in **Figure 14** and **Figure 15** respectively. The following behavioral requirements are modeled:

- Initially, the system is in Flying state and continuously monitoring engine failure and GPS failure states. The system moves to either EngineFailure state or GPSFailure state depending on the values of engine_failure_sensor and gps_failure_sensor respectively. Similarly, system is also monitoring TerminationCommandRecieved, 5.8GHzLinkFailure, SoftGeoFenceBreach and DataLinkFailure states through termination_command_recieved, 5.8GHz_link_failue_sensor, geo_fencing_sensor and datalink_failure_sensor respectively.
- Once the system is in EngineFailure state, it has to be landed on emergency basis within 3 clock cycles. In TerminationCommandRecieved state, the system should move to FlightTerminationInitiated state after one clock cycle and subsequently move to ManuallyLandAircraft state within three clock cycles.
- Once the system is in SoftGeoFenceBreach state, it should move to RestoringFromNoFlyZone state on the next clock cycle and perform certain checks to move into respective state. Similarly, In DataLinkFailure state, system evaluate certain conditions (e.g. 900MHz_link_failure_sensor, 5.8GHz_link_failure_sensor etc.) to move into particular state accordingly.

- In case of GPS failure, the system continues normal flight on successful GPS auto restore. Otherwise, the system should move to FlightBackToStation state and subsequently move to either ReachedBackToStation state or AircraftLost state depending on the underlying conditions.

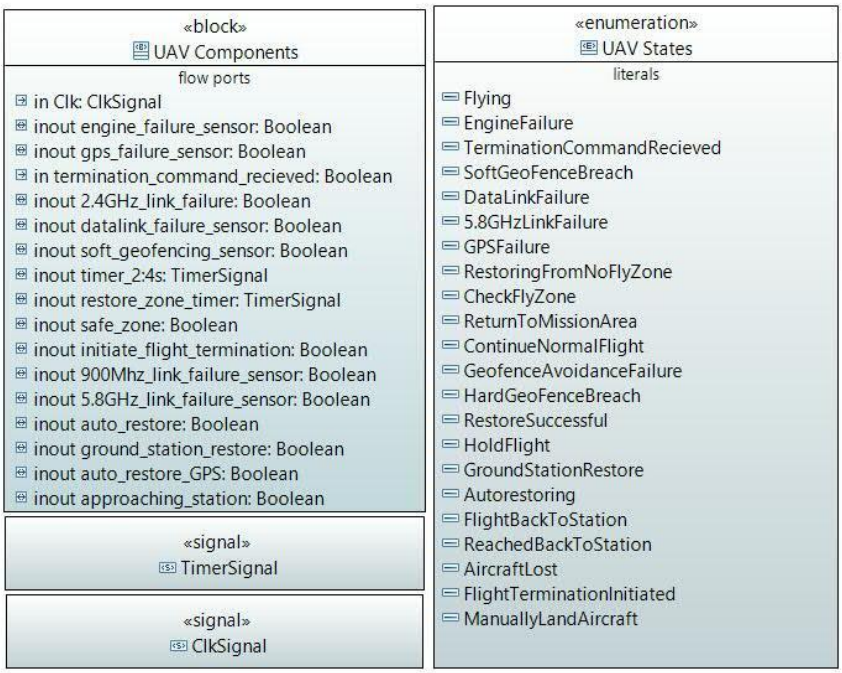

Figure 14: Structure of Unmanned Aerial Vehicle in UMLSV

The structure of UAV system is represented through SYSML BDD where flow ports are used to represent registers/sensors/variables of the system. Moreover, enumerations are used to represent system states. Furthermore, the Signal is used for typing the clock and timer flow ports as shown in **Figure 14**. On the other hand, the behavior of UAV system is modeled through UMLSV stereotypes, as shown in **Figure 15**. Particularly, the system is in Flying state initially. The *nextState* stereotype is applied on different transitions to move the system from Flying state to any or all other six states (i.e. EngineFailure, TerminationCommandRecieved, SoftGeoFenceBreach, DataLinkFailure, 5.8GHzLinkFailure and GPSFailure) depending on the success of a respective constraint condition. Similarly, other behavioral requirements of UAV system is modeled through UMLSV stereotypes, as shown in **Figure 15**. Once the design of UAV system is successfully modeled, the following safety constraints are identified and expressed in UMLSV:

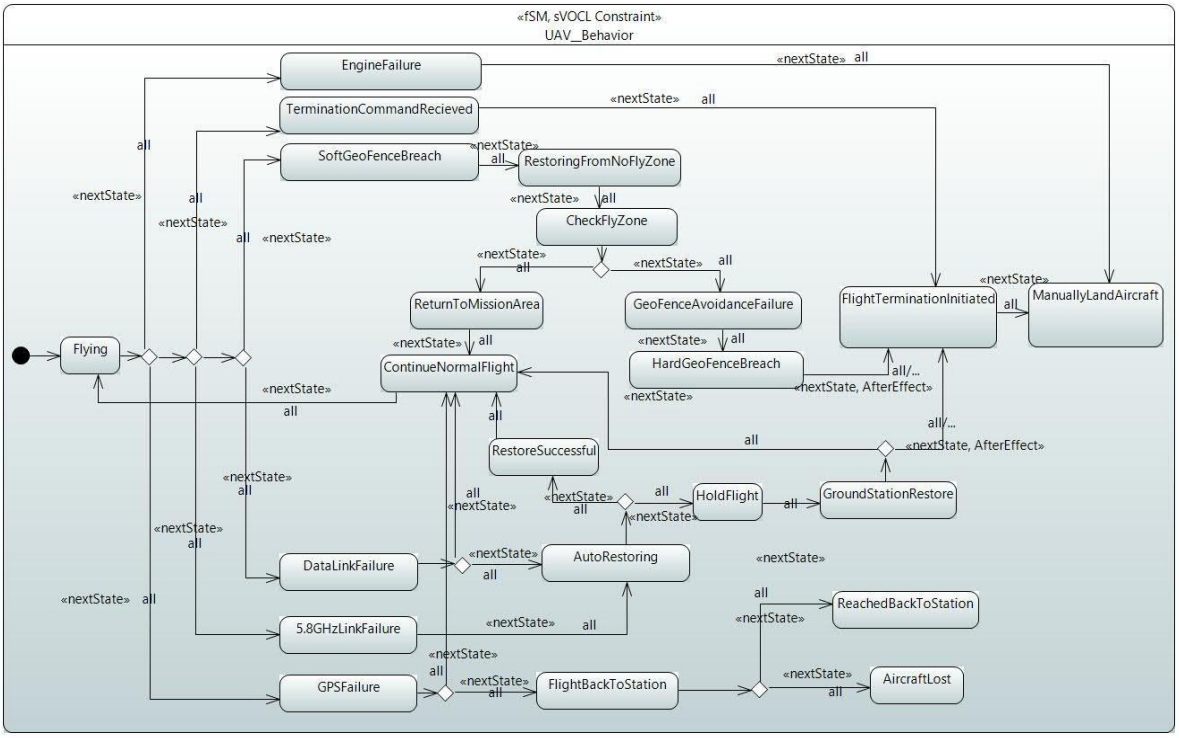

Figure 15: Behavior of Unmanned Aerial Vehicle in UMLSV

1. **Property 1:** In case of termination command during flight, the system should move to FlightTerminationInitiated within two clock cycles and subsequently enters into ManuallyLandAirCraft state within two to four clock cycles.
2. **Property 2:** From AutoRestoring state, the system should move to Flying state in 4 clock cycles on the activation of auto_restore and ground_station_restore.
3. **Property 3:** From RestoringFromNoFlyZone, the system should move to CheckFlyZone within 1 to 3 clock cycles. However, on the deactivation of safe_zone, the system should move to FlightTerminationInitiated state within 3 clock cycles and subsequently to ManuallyLandAirCraft state within 2 to 4 clock cycles.
4. **Property 4:** The system should move from RestoringFromNoFlyZone to CheckFlyZone within 1 to 3 clock cycles and upon the activation of safe_zone, system should move to Flying state after 3 clock cycles.
5. **Property 5:** In case of GPS failure with unsuccessful automatic restore, the system should go back to station within 4 clock cycles.

Here, we include the Property 1 in the UAV system model through *SVOCL Constraint* stereotype as shown in the **Figure 16**. The interested readers can view the screenshots of all the design verification requirements at [27].

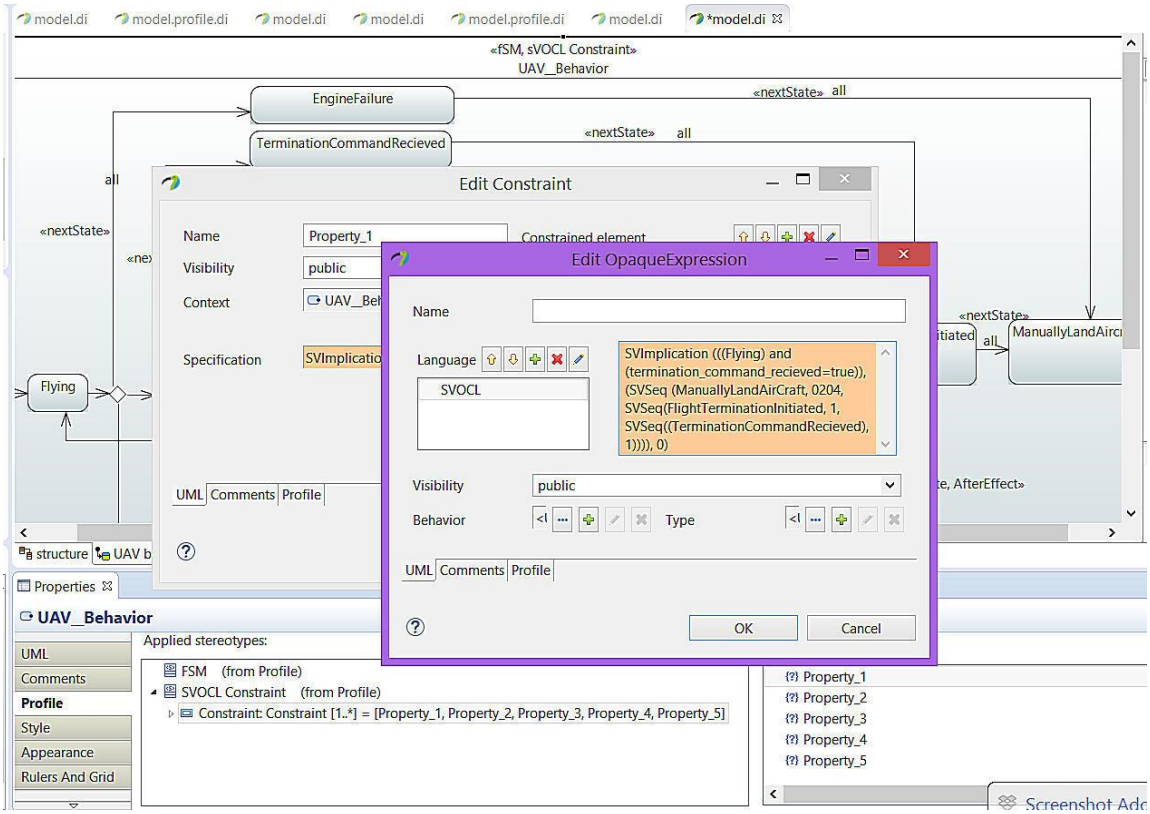

Figure 16: Verification Constraint of Unmanned Aerial Vehicle System in UMLSV

4.2.2 Design Verification and Code Synthesis

Once the UAV design and the verification requirements have been successfully modeled (Section **4.2.1**), we generate the corresponding SystemVerilog RTL and SVA's code through UMLSV transformation engine. However, we are not including the code generation information here as such details are already discussed in Section **4.1.1** for the Traffic Lights Controller. Similarly, we are not including the details of design verification and code synthesis for UAV case study because such details are already discussed in Section **4.1.2** and Section **4.1.3** respectively for the Traffic Lights Controller. The transformation engine, profile and the UAV model are available at [26] for further evaluation.

4.3 Elevator Case Study

This case study exemplifies the design of an elevator that moves the persons and luggage between different floors of a building. There are two sensors, attached in the elevator, to compute the number of peoples and the total weight respectively. The maximum permissible weight is 800 kg. The digital panel is installed in the elevator to select the desired floors in a sequence as per requirements. The emergency sensor is also connected to instantly stop and open the exit passage of moving elevator in case of emergency situations.

4.3.1 Requirements specifications

The structure and behavior of elevator is modeled through the UMLSV, as shown in **Figure 17** & **Figure 18** respectively.

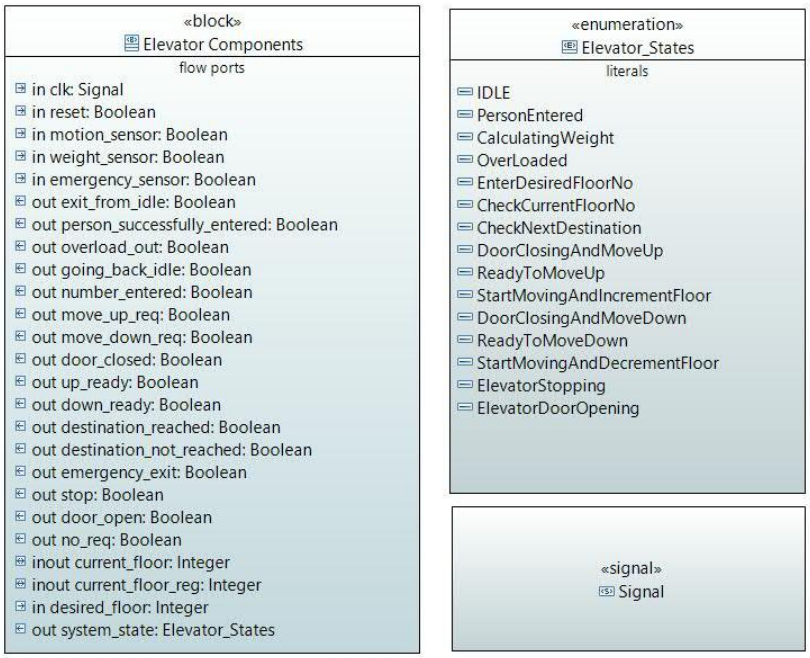

Figure 17: Modeling Structure of Elevator in UMLSV

It can be seen from **Figure 18** that the elevator is in idle state initially, therefore, UMLSV *IdleState* stereotype is applied. In addition, it is also required to wait for a particular time in Idle state before moving to the next destination. Therefore, *ActionBeingInState* stereotype is applied in Idle state to call clock activity. Furthermore, *ActionOnReset* and *AfterEffects* stereotypes are also applied on Idle state in order to manage reset and after effect conditions respectively. Once the condition given in the *NextState* stereotype is successful, the elevator should move from Idle state to PersonEntered state which is an initial state. Therefore, UMLSV *InitialState* stereotype is applied. The elevator should move from PersonEntered state to CalculatingWeight state in order to check the allowed weight limit. Logically, it is a post initial state, therefore, UMLSV *PostInitialState* stereotype is applied. From CalcualtingWeight state, the elevator can move to Overloaded state in case the weight is greater than allowed limits i.e. 800 kg. The weight limit condition is applied through *NextState* stereotype. From the Overloaded state, the elevator should move to Idle state for further processing. In this case, after effects and desired condition are applied by using *AfterEffects* and *NextState* stereotypes as shown in **Figure 18**. The elevator can also move to EnterDesiredFloorNo state from CalculatingWeight state if the weight is within allowed limits i.e. less than 800kg. Logically, EnterDesiredFloorNo is a pre busy state, therefore, UMLSV PreBusyState stereotype is applied. The elevator should move to CheckCurrentFloorNo state from the EnterDesiredFloorNo state for further processing.

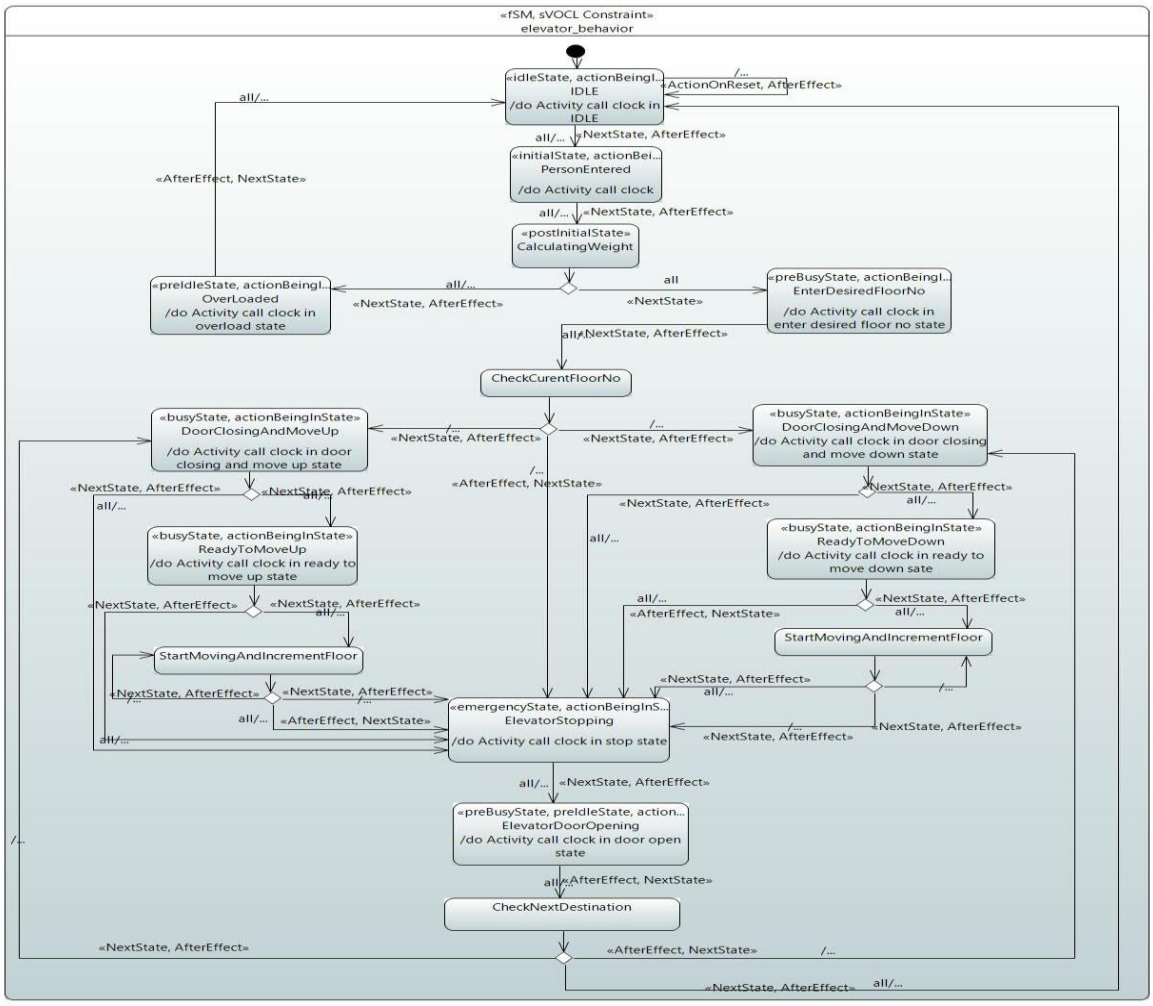

Figure 18: Modeling Behavior of Elevator in UMLSV

The elevator can either move to DoorClosingAndMoveUp or DoorClosingAndMoveDown state from the CheckCurrentFloorNo state depending on the current position of elevator and entered floor number. In DoorClosingAndMoveUp and DoorClosingAndMoveDown states, it is required to wait for 3 clock cycles to allow the proper closing of door. This requirement is managed by invoking clock activity through *ActionBeingInState* stereotype. From the DoorClosingAndMoveUp state, the elevator can either move to ReadyToMoveUp state or ElevatorStopping state (in case of emergency). The elevator can move from ReadyToMoveUp state to StartMovingAndIncrementFloor state and remained in this state until the given floor number is reached. In StartMovingAndIncrementFloor state, the increment in the floor number is performed through *AfterEffects* stereotype and the condition for the desired floor number is applied through *NextState* stereotype. The elevator should move to ElevatorStopping state from StartMovingAndIncrementFloor state once the desired floor number is reached. Similarly, the behavior of elevator from DoorClosingAndMoveDown state to ElevatorStopping state is modeled through UMLSV stereotypes as shown in **Figure 18**. Finally, the following five design verification requirements of the elevator are identified and modeled in UMLSV:

1. **Never_in_IDLE_State_If_Motion_Detected:** To confirm that whenever an elevator is moving, it should not be in the IDLE state.
2. **Never_Move_Up_Down_At_Same_Time:** To confirm the correct operating mechanism of elevator i.e. It shouldn't move in the upward and downward direction at the same time.
3. **Never_OpenDoor_When_Moving:** To confirm that when the elevator is moving either downward or upward, the door of elevator should not be opened.
4. **Must_Indicate_When_Weight_Limit_Exceed:** To confirm that that the overall weight of the elevator should be within the given limits (less than 800 KG).
5. **Status_During_Emergency:** To confirm that whenever the emergency sensor gets activated, the emergency exit should also be activated at the same time.

Here, we include the screenshot of one verification requirement (**Never_OpenDoor_When_Moving)** as shown in the **Figure 19**. The screenshots of other verification requirements of elevator can be found at [27].

After the successful modeling of elevator design and verification requirements, the UMLSV transformation engine is executed to generate the SystemVerilog RTL and SVA's code. Subsequently, design verification is performed in QuestaSIM through generated SystemVerilog code. However, we are not including design verification and code synthesis information here as such details are already discussed in Section **4.1.2** and Section **4.1.3** respectively for the Traffic Lights Controller. The transformation engine, profile and the elevator model are available at [26] for further evaluation.

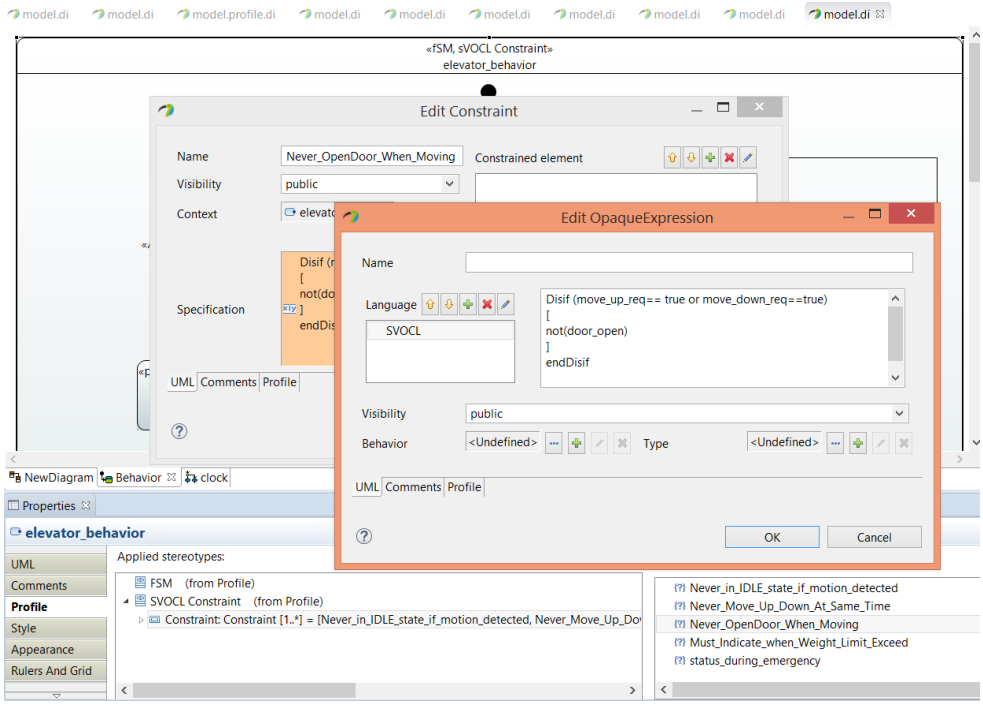

Figure 19: Expressing the Design Verification Requirements for Elevator Case Study

4.4 Car Collision Avoidance System

The objective of Car Collision Avoidance System is to increase the safety of vehicles by decreasing the number of road accidents. The Car Collision Avoidance System provides two main functionalities: radar detection and image tracking. The user can switch to any of the technique depending upon some particular requirements and weather conditions. During radar detection, waves are continuously emitted and the data is sent back to the main controller. If any obstacle is found in the vehicle's path, the distance of obstacle is calculated and some necessary actions are performed to avoid collision. On the other hand, the image tracking functionality employs a camera that takes the pictures of obstacles and passes it to the main controller. The main controller then interprets the images to decide about the presence of an obstacle. Once the obstacle is detected, necessary actions are performed to avoid collision.

4.4.1 Requirements specifications

The structural and behavioral aspects of Car Collision Avoidance System are shown in **Figure 20** and **Figure 21** respectively. It can be observed from **Figure 20** that the main car block consists of many parts like exhaust system, cooling system, wheels, ignition system, car collision avoidance module etc. However, the main structural requirements are captured through the car collision avoidance module. Finally, the states of CCAS are defined through enumeration.

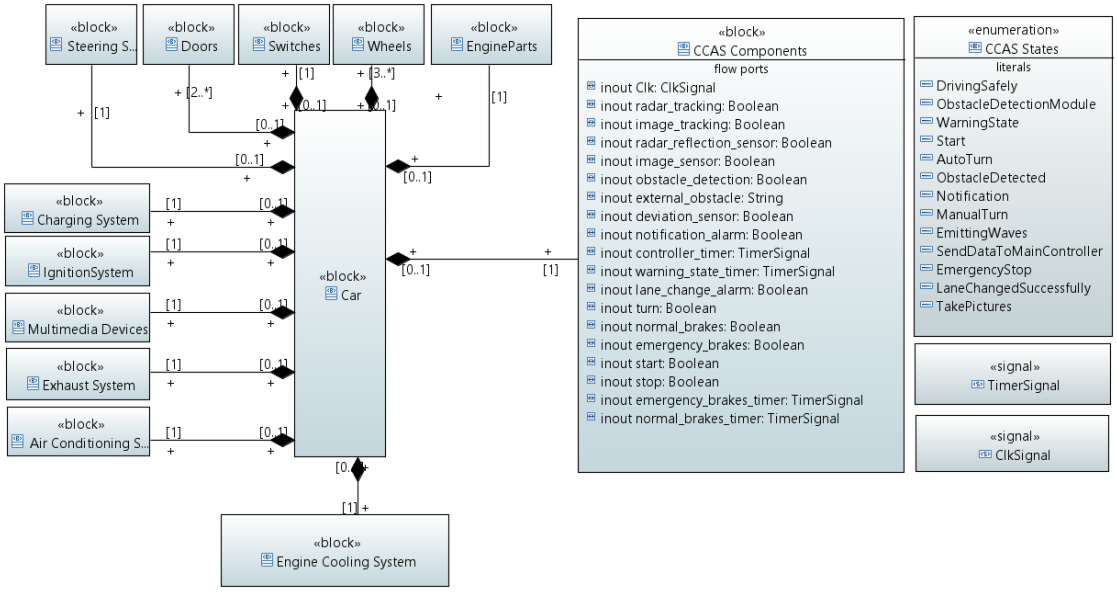

Figure 20: Structural Model of Car Collision Avoidance System

The summary of CCAS behavioral requirements (**Figure 21**) is as follows:

- The system begins with the *Start* state and moves to the *DrivingSafely* state on the positive edge of a clock. In the next clock cycle, it evaluates the activation of available obstacle detection modules (radar or image tracking).

- If the radar tracking is activated, the system moves to *EmittingWaves* state. The system moves to *TakePictures* state upon the activation of image tracking. In either of the case, the data is sent to the main controller after every three clock ticks through *SendDataToMainController* state. Finally, the system moves to *ObstacleDetected* state in case of an obstacle detection.
- The system moves from *ObstacleDetected* state to *ObstacleDetectionModule* state to calculate the distance of an obstacle. If the distance is less than 5 meters, the system moves to Imminent Collision Strategy composite state. If the distance is between 5-10 meters, the system moves to Near Collision Avoidance Strategy composite state. The system moves to Changing Lane Strategy composite state whenever the distance is between 10-20 meters.
    - In the Changing Lane Strategy, the two available options are auto turn and manual turn. The system moves to *AutoTurn* state on the activation of deviation_sensor and asserts lane_change_alarm to notify the driver. On the other hand, *ManualTurn* state allows the driver to change the lane manually. In either of the case (*AutoTurn* state or *ManualTurn* state), the system subsequently moves to *LaneChangedSuccessfully* state.
    - In the Near Collision Avoidance Strategy, the system first moves into *Warning* state on the same clock cycle. In the next clock cycle, the system enters into *Notification* state and asserts normal_brakes signal to reduce the speed of the car.
    - Finally, in the Imminent Collision Strategy, the system moves to *Notification* state (on the same clock tick) and generate alarm through notification_alarm signal. Subsequently, the system moves to *Warning* state on next clock tick and apply the emergency breaks through *EmergencyStop* state if the distance is still less than 5 meters.

It can be seen in the **Figure 21** that the UMLSV stereotypes are applied to model the behavior of Car Collision Avoidance System. We are not including the information about the applied UMLSV stereotypes here because such details are already discussed for Traffic Lights Controller, Unmanned Aerial Vehicle and Elevator case studies in Section **4.1.1**, Section **4.2.1** and Section **4.3.1** respectively. We identify five design verification requirements for Car Collision Avoidance System as follows:

1. **Property_1:** This property ensures that either the radar or the image tracking approach must be activated within the two clock cycles after the start of a car. Subsequently, the data should be sent to the main controller after three clock cycles.
2. **Property_2:** This property ensures that at least one of the three actions (i.e. imminent collision, near collision and changing lane) must be performed right after the three clock cycles once the obstacle is detected.

3. **Property_3:** This property ensures that the system is in Imminent Collision Strategy once the obstacle is less than five meters away. Furthermore, driver should be notified on the same clock cycle and the system must enter into warning state on the next clock cycle.
4. **Property_4:** This property makes sure that the emergency breaks will be applied on the same clock cycle provided that the distance from obstacle is less than five meters and the warning state timer is up to maximum limit. Subsequently, the system must be in the start state within four to six clock cycles.
5. **Property_5:** This property guarantees the correct execution of changed line functionality i.e. Lane must be changed within two to four clock cycles. Similarly, the system must be in DrivingSafely state within one to three clock cycles.

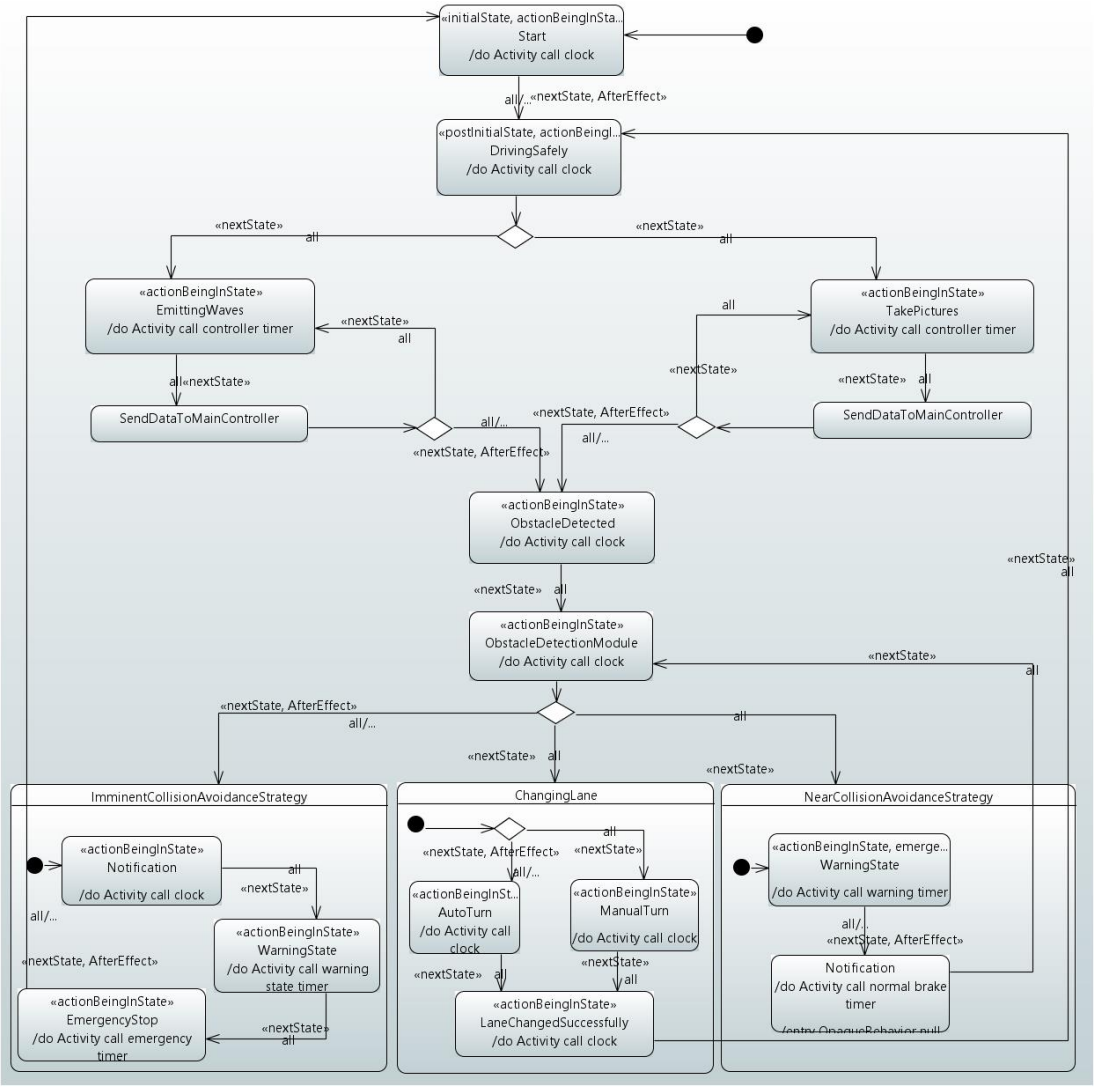

Figure 21: Behavioral Model of Car Collision Avoidance System

We express the aforementioned design verification requirements in UMLSV. Here, we are demonstrating the inclusion of first property in the model as shown in **Figure 22**.

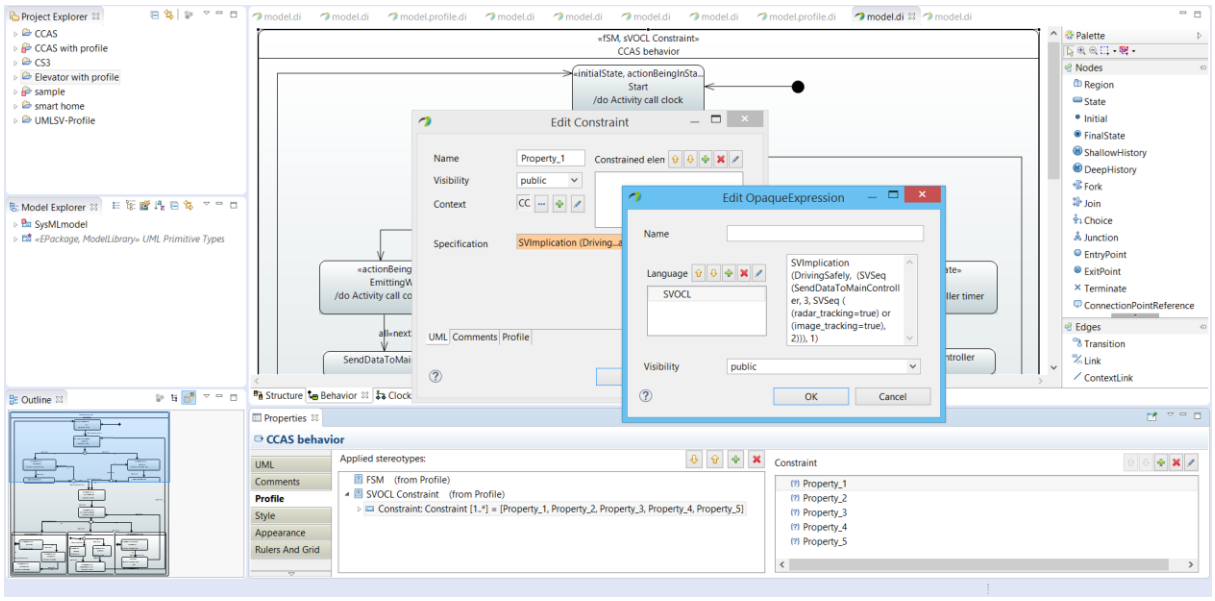

Figure 22: Expressing the Design Verification Requirements for Car Collision Avoidance System

We are not including the model transformation, design verification and code synthesis details of Car Collision Avoidance System because such details are already given for Traffic lights controller (Section **4.1**) case study. However, the complete model (including UMLSV profile) of Car Collision Avoidance System is available at [26] for further investigation.

So far, we have demonstrated the applicability of proposed framework through four case studies (i.e. Traffic lights controller - Section **4.1**, Unmanned Aerial Vehicle – Section **4.2**, Elevator - Section **4.3** and Car Collision Avoidance System - Section **4.4**). It is important to note that we have applied the proposed framework on other case studies as well. However, it is not possible to include all the case studies in the article. Furthermore, it is also difficult to fully describe the behavioral requirements of the given four case studies, particularly in the context of UMLSV. Therefore, we have publicized the UMLSV profile, transformation engine and models of all case studies, so that, the interested readers can further explore it. The details are available at [26].

4.5 Quality Measures

To this point, we have successfully demonstrated the application of UMLSV through four different case studies. However, we are not still able to confidently establish the benefits and practical usage of UMLSV without answering the following questions:

Q1) How efficient the UMLSV is, in terms of design productivity (total time required for design as well as its temporal verification), with respect to low level SystemVerilog implementation?

Q2) What is the quality of RTL and assertions code, automatically generated through the UMLSV transformation engine?

Q3) How good the proposed methodology is, in correcting the design errors, as compared to the conventional lower level implementation in SystemVerilog?

Q4) How easy to learn the proposed framework as compared to low level implementation in SystemVerilog?

To answer aforementioned questions, a comprehensive quantitative analysis of UMLSV has been performed. In the first phase, two industry experts were selected i.e. Expert A and Expert B. Expert A has more than five years of experience in embedded systems development through Verilog/SystemVerilog. On the other hand, Expert B has more than five years of experience in Model Driven Architecture (MDA) with specialization in UML based design and development. Design and verification requirements of two case studies (i.e. Traffic Lights and Elevator), as a plain text document, were given to both experts. Expert A was required to develop the RTL code and assertions for given case studies through SystemVerilog while Expert B was supposed to model the given case studies through UMLSV profile and subsequently transform the models into SystemVerilog RTL and assertions code through UMLSV transformation engine. The results are summarized in **Table 6**.

Table 6: Quantitative analysis of UMLSV for efficiency w.r.t Low Level SystemVerilog

Case Study	**RTL and Assertions Code through SystemVerilog**			**RTL and Assertions Code through UMLSV**			**Efficiency of UMLSV w.r.t Low Level SystemVerilog Implementation**
	Working hours for RTL implementation	Working hours for assertions implementation	Total Working Hours	Working hours for Design Modeling & Transformation	Working hours for assertions Modeling & Transformation	Total Working Hours	
Traffic Lights	21	9	30	15	8	23	(30-23)/(23) x 100 =**30%**
Elevator	19	5	24	8	3	11	(24-11)/(11) x 100 =**118%**

It can be seen from column 2 and column 3 of **Table 6** that the low level SystemVerilog implementation has taken 21 and 9 working hours to write the RTL and assertions code respectively for the Traffic Lights Controller. On the other hand, the RTL and assertions code for Traffic Lights Controller were completed in 15 and 8 working hours respectively through UMLSV profile (see column 5 and column 6 of **Table 6**). Therefore, it can be concluded that UMLSV is 30% efficient as compared to SystemVerilog low level implementation for Traffic Lights case study (as depicted in the last column of **Table 6**). For the Elevator case study, the efficiency of UMLSV is further improved up to 118% as compared to low level implementation. The primary reason for further improvement of UMLSV efficiency in the second case study is that Expert B gets familiar with the UMLSV profile after the modeling and transformation of the first case study. Therefore, it is easier for Expert B to model the Elevator design and assertions through UMLSV. Consequently, it can be concluded that the efficiency of UMLSV is significantly improved after getting the basic working mechanism of UMLSV profile.

It can be noted from **Table 6** that the efficiency of UMLSV is marginally better than the low level SystemVerilog implementation while dealing with the modeling and transformation of assertions through SVOCL (see column 3 and column 6 of **Table 6** for the comparison). Particularly, UMLSV only saves three working hours in both case studies for the modeling and transformation of assertions through SVOCL. In this regard, Expert B has also reported that the representation of assertions in SVOCL is a bit complex. Basically, SVOCL is developed to represent the major SVA's concepts in high level models. Consequently, it is required to include particular low level SVA's concepts like disable iff expression in SVOCL. As a result, there is some complexity involved in the representation of assertions through SVOCL. However, SVOCL is still marginally efficient as compared to pure low level SVA's. Moreover, it is able to successfully include the assertions in high level models, thus, significantly reducing the design verification gap as assertions code is also available along with the RTL code. Furthermore, the complexity of SVOCL is comparable with other renowned state-of-the-art property specification methodologies as referred in [21].

From the aforementioned quantitative analysis, it can be deduced that the proposed methodology significantly improves the design and verification time as compared to low level SystemVerilog implementation approach. However, the efficiency of UMLSV cannot be confirmed without analyzing the quality of RTL and assertions code, generated automatically through the UMLSV transformation engine. Therefore, the codes from both Expert A and B were given to test engineering to actually perform ABV in QuestaSIM. As a result, the corresponding test engineer confirmed that the SystemVerilog code for the system structure and assertions is generated by the UMLSV transformation engine with 100% accuracy for both case studies. However, there is an over specification of two control statements in the RTL codes. Furthermore, few minor syntax errors like missing semicolon, improper end statement etc. are also found. Importantly, test engineer confirmed that the required behavioral logic for both case studies is well incorporated in RTL codes that are generated through the UMLSV transformation engine. However, test engineer has spent two additional working hour to fix RTL codes, generated through UMLSV transformation engine, in order to perform simulation in QuestaSIM. The time utilized to correct RTL codes is very much acceptable because it is hard to generate RTL code with 100% accuracy through high level models by utilizing the concept of model transformation especially in case of a wide-ranging solution. Finally, the design verification of both case studies is successfully performed in QuestaSIM. Furthermore, it is also confirmed that the generated RTL codes are synthesizable and can be directly deployed on the target devices after successful verification.

In the second phase, the efficiently of UMLSV profile was further analyzed in Academia through two Master students i.e. Student A and Student B. Particularly, Student A was selected from Software engineering department and has the basic knowledge of UML. On the other hand, Student B was selected from Electrical engineering department with the basic knowledge of Verilog. The textual requirements of two case studies (i.e. Memory Model and Unmanned Ariel Vehicle) were given to both students where student A was required to generate the RTL and assertions code through UMLSV profile and student B was supposed to implement RTL and

assertions code directly through SystemVerilog. As student A and student B have very basic knowledge of UML and SystemVerilog respectively, there was a training time to learn the required concepts of UMLSV and SystemVerilog in order to perform actual implementation. Particularly, Student A was first required to learn the basic application of UMLSV stereotypes e.g. basic technical knowledge regarding the development of UMLSV stereotypes and the practical usage of stereotypes for the modeling of given requirements. On the other hand, Student B was required to learn basic SystemVerilog concepts particularly required to write the RTL and assertions code. In addition to this, Student B had to learn: 1) the basic syntax for control statements, variable declarations (e.g. ports, registers etc.) and clocking concepts 2) SystemVerilog assertions concepts like immediate / concurrent assertions with syntax and binding of assertions with the RTL. The results are summarized in **Table 7**.

Table 7*:* Quantitative analysis of UMLSV efficiency in academia

Case Study	**RTL and Assertions Code through SystemVerilog**				**RTL and Assertions Code through UMLSV**				**Efficiency of UMLSV w.r.t Low Level SystemVerilog**
	Working hours for Training	Working hours for RTL	Working hours for assertions	Total hours	Working hours for Training	Working hours for Design Modeling & Transformation	Working hours for assertions Modeling & Transformation	Total hours	
Memory	38	19	1	20	18	12	1	13	**53%**
UAV		41	19	60		18	13	31	**94%**

It can be seen from column 2 of **Table 7** that it requires 38 working hours for training the essential SystemVerilog concepts and syntax for the implementation of RTL and assertions, provided that the trainee has basic knowledge of Verilog. It is important to note that the advanced SystemVerilog features like object oriented programming concepts [5] (inheritance, polymorphism etc.), Direct Programming Interface (DPI) [8] etc. are not included in the training phase in order to perform competitive comparative analysis with the UMLSV. On the other hand, it requires 18 working hours for training UMLSV profile and transformation engine provided that the trainee has the basic knowledge of standard UML profile (see column 6 of Table 6). Consequently, *the learning time for low level SystemVerilog concepts is almost "double" as compared to UMLSV concepts, provided that the trainees have the basic knowledge of UML and Verilog respectively*.

Once the training was completed, both students implemented the given case studies in respective development techniques. It can be seen from column 5 of **Table 7** that 20 working hours were consumed to implement the Memory model in SystemVerilog. Particularly, 19 and 1 working hours were spent to implement RTL and assertions codes respectively. On the other hand, 13 working hours were consumed to model and transform the Memory model case study through UMLSV (see column 9 of **Table 7**). Particularly, 12 and 1 working hours were spent to model and transform RTL as well as assertions respectively. Therefore, UMLSV is 53% efficient as compared to low level SystemVerilog implementation. Similarly, for Unmanned Aerial Vehicle (UAV) case

study, UMLSV has reduced the total development time to almost half of the originally required time with low level implementation (shown as 94% improvement in the last column of **Table 7**).

In order to further confirm the quality of automatically generated RTL and assertions codes, the developed case studies from both students were given to a test engineer with more than 5 years of professional experience. Consequently, the test engineer confirmed that the structural as well as assertions code for both case studies, generated through the UMLSV profile, is 100% accurate. Furthermore, the major behavioral logic of the system design is also obtained by transforming UMLSV models. However, there are few syntax errors and over specification of control statements in RTL codes. The corresponding test engineer was able to fix the errors in the RTL codes of both case studies in just one working hour. Subsequently, he performed the design verification in QuestaSIM. During the verification process, an error is found in the UAV design of both students as one assertion failed during simulation. The reported error was given back to both students for corrections. Student A corrected the design in 4 working hours through the UMLSV profile. On the other hand, Student B spent 7 working hours to correct the design through SystemVerilog. Finally, the corrected RTL code of UAV from both students was given to test engineer. Again, he took 30 minutes to fix the errors in the corresponding RTL code. Therefore, it practically takes approximately 5 working hours (4 hours + 30 minutes) to fix the reported error through UMLSV. Consequently, it can be concluded that the corrections of errors in UMLSV is relatively simple as compared to the low level SystemVerilog implementation.

From the aforementioned quantitative analysis, it can be concluded that the UMLSV is a significantly efficient solution as compared to the low level SystemVerilog implementation. Furthermore, it is also analyzed that UMLSV allows the specification of verification assertions concurrently along with the system design (structure and behavior). On the other hand, the low level SystemVerilog implementation requires the completion of design (RTL code) before specifying the required assertions (for verification). This is what we have claimed in the introductory part of this article that the proposed method reduces the gap between design and its verification which leads to achieve important business objectives like time-to-market and productivity.

5. Comparison of Proposed Framework with State-of-the-Art

Model Based System Engineering (MBSE) is an attractive field for embedded systems and there exist several studies [4] dealing with the design and verification solutions. In this context, the selection of relevant studies for comparative analysis is an important aspect. The proposed framework in this article mainly deals with the UMLSV profile which is based on the concepts of UML and SYSML profiles. Therefore, we primarily select renowned UML-based state-of-the-art profiles for embedded systems to perform a balanced comparative analysis. In this regard, we define six evaluation parameters to investigate the pros and cons of the proposed framework with respect to state-of-the-art:

Exploited UML profiles: UMLSV is developed by utilizing the concepts of UML and SYSML profiles. Therefore, it is essential to investigate other UML-based profiles reported in the field of embedded systems. The *Exploited UML profiles* parameter evaluates the UML and its SYSML / MARTE profiles, utilized to develop a given state-of-the-art profile.

VM (Verification Method): In the context of MBSE for embedded systems, the design verification can be categorize into two types [4] i.e. Formal Verification (FV) and Dynamic Verification (DV). The proposed framework provides a complete transformation engine to automatically generate the target SystemVerilog code from the source UMLSV models to support DV. In this regard, it is important to evaluate the supported verification types in other state-of-the-art studies. Therefore, *VM (Verification Method)* parameter evaluates the method utilized in state-of-the-art profiles to perform design verification. This parameter is evaluated through three possible values i.e. FV, DV and Both (FV + DV).

Target Platform: In MBSE, automatically generated target models are utilized to perform design verification. For example, in the case of formal verification, Z notation and timed automata can be the possible target models. Alternatively, in the case of dynamic verification (simulation), the code of hardware design languages like VHDL, SystemC etc. can be the possible target models. The proposed framework fully supports the SystemVerilog target model to perform design verification. In this regard, it is essential to investigate the number of target models that are supported in the state-of-the-art profiles. Therefore, the *Target Platform* parameter evaluates the possible target models that are generated from the source high level models.

ABV (Assertion Based Verification): Assertion Based Verification (ABV) is a verification approach where the design is verified against some particular assertions. It provides several benefits over traditional verification methods like increased productivity through reduced verification time etc. ABV can be performed in different hardware languages like Verilog, C etc. However, SystemVerilog provides a complete and advanced platform for ABV through SVA's. The proposed framework fully supports ABV as SVA's are also generated through transformation engine along with the RTL code. In this context, it is essential to evaluate the support for ABV in other state-of-the-art profiles.

Property Specification Technique: It is always challenging to represent the complex temporal verification properties of the embedded systems at higher abstraction level. Therefore, researchers frequently work in this direction to introduce novel approaches. In the proposed framework, SVOCL is utilized to represent both simple as well as complex temporal properties of embedded systems at higher abstraction level. In this context, it is important to explore the property specification approaches in the relevant state-of-the-art profiles. Therefore, *Property Specification Technique* parameter evaluates the property specification approach of a given profile to specify the verification properties / constraints in the models.

Tool Support: The proposed framework provides a complete open source transformation engine to automatically transform the source high level models into the low level SystemVerilog RTL and assertions code. In this context, *Tool Support* parameter evaluates whether the profile under

consideration provides automated tool support to perform certain tasks like model transformation, design verification etc.

We perform a comparative analysis through the aforementioned evaluation parameters and the results are summarized in **Table 8**. The results of *Exploited UML profiles* parameter reveal that the combination of UML and its MARTE profiles is frequently utilized to develop profiles for embedded systems. The rationale for such a combination is the built-in features of MARTE profile to manage the temporal aspects of embedded systems. However, there exist few research studies where standard UML profile is extended individually in the context of formal semantics. For example, Luciano Baresi et al [50] propose Coretto UML (C-UML) where standard UML diagrams are extended through TRIO temporal logic semantics. Furthermore, Corretto Property Language (CPL) is developed for the representation of properties / constraints in the C-UML models.

Table 8: Comparative analysis of UMLSV with state-of-the-art profiles for embedded systems

Sr. #	**State-of-the-Art Profile**	**Exploited UML Profiles**	**VM**	**Target Platform's**	**ABV**	**Property Specification Approach**	**Tool Support**
1	Z-MARTE [41]	UML and MARTE	FV	Z notation	No	Z-MARTE	Yes
2	e profile [42]	UML and MARTE	DV	e HVL [30]	No	e profile	Yes
3	MADES Profile[43]	SYSML and MARTE	Both	Modelica TLF [31]	No	MADES Profile	Yes
4	Guglielmo et al [44]	UML	DV	C	Yes	PSL	Yes
5	GASPARD [45]	UML and MARTE	Both	VHDL SystemC SIGNAL[32]	No	MARTE Stereotypes	Yes
6	ModelicaML [46]	UML and SYSML	DV	Modelica	No	ModelicaML	Yes
7	DAM-Rail [47]	UML and MARTE	FV	RFT [33] BN [35] GSPN [34]	No	DAM-Rail	Yes
8	MOPCOM [48]	UML and MARTE	DV	VHDL SystemC C	No	MARTE Stereotypes	Yes
9	RecoMARTE [49]	UML and MARTE	FV	BZR Language [36]	No	RecoMARTE	Yes
10	C-UML [50]	UML	FV	TRIO	No	CPL	Yes
11	**UMLSV**	**UML and SYSML**	**DV**	**SystemVerilog**	**Yes**	**OCL Extension**	**Yes**

It can be analyzed from **Table 8** that the proposed profiles certainly provide higher level of abstraction, however, it is essential to consider the semantics of target models during the profile development for the accurate model transformation. For example, the semantics of Z notations are considered in Z-MARTE [41] profile. Similarly, the semantics of Modelica language are considered in ModelicaML profile. Before UMLSV, SystemVerilog was usually utilized at lower implementation level and there was a strong need to develop a higher abstraction layer for SystemVerilog in order to reduce the gap between design and its verification in the context of MBSE. This gap is now filled with the development of UMLSV as it provides the higher level of abstraction along with the generation of SystemVerilog RTL and assertions code.

It can be argued that the comparative analysis of UMLSV only considers UML based Domain Specific Modeling Languages (DSML's) for embedded systems, as given in **Table 8**, while other model based approaches are completely ignored. Actually, there are several studies (e.g. [19-20] etc.) available that deal with the model based development of embedded systems without employing UML. However, we believe that UML based DSML's are more appropriate for main comparative analysis (**Table 8**) because UMLSV is also developed through UML extension. Furthermore, it is hard to perform a balanced comparative analysis of UMLSV with other existing model based approaches due to the diversity of such approaches. Consequently, we only consider the renowned UML based DSML's for embedded systems, published in well-known journals, for the comparative analysis.

In fact, there exist a variety of model based approaches other than UML. For example, Bersani and Garcia-Valls [17] present a formal approach, based on temporal logic, for the online verification of dynamic behavior in cyber physical systems. Particularly, **C**onstraint **LTL** (Linear Temporal Logic) **o**ver **c**locks – CLTLoc is utilized to model the behavioral requirements along with temporal constraints of the system. The verification is performed through ae2zot tool[2].This approach is highly suitable for the verification of systems where behavioral requirements are altered dynamically and intensively based on temporal aspects. However, in contrast to UMLSV, this approach deals with the formal verification without any support for ABV. Furthermore, the development of models in CTCLoc is relatively complex as compared to UMLSV. In another research study, Fathabadi et al [18] proposes a model based approach and utilizes a mathematical language Event-B to model system requirements. Subsequently, formal verification is performed through ProB tool[3]. Finally, c code is generated for actual deployment after successful formal verification. In comparison with UMLSV, this approach is based on Event-B language that provides considerably low abstraction level as compared to UMLSV. Furthermore, this approach only deals with the formal verification and c code is finally generated for deployment only. Therefore, the correctness of the generated c code is still questionable. On the other hand, the UMLSV transformation engine generates SystemVerilog RTL and assertions code before performing design verification. Therefore, the final deployable RTL code is more reliable as it is

[2] https://github.com/fm-polimi/zot
[3] http://www3.hhu.de/stups/prob/index.php/Main_Page

thoroughly verified. However, in this context, more time is required for design verification in UMLSV as compared to Fathabadi et al [18] approach.

In addition to state-of-the-art approaches, it is important to mention the open source PolarSys[4] platform where several model-driven embedded solutions are proposed. For example, in CHESS project, a CHESS-ML profile is developed by utilizing the notations of UML, SYSML and MARTE profiles. Furthermore, a transformation tool is implemented to generate MAST[5] and DEEM[6] target models from the source CHESS-ML models to perform schedulability and dependability analysis respectively. In Papyrus-RT project, a complete modeling environment is provided for UML-RT [40] profile which is based on the concepts of capsules, ports, protocol and connectors to represent system structure and behavior. Furthermore, a transformation tool is implemented to generate C++ code from source models. In contrast to UMLSV, CHESS-ML and UML-RT do not provide higher abstraction layer for SystemVerilog. Consequently, both profiles are unable to transform high level source models into SystemVerilog RTL and assertions code. In fact, there doesn't exist any PolarSys project that provides higher abstraction layer for SystemVerilog and subsequently generate RTL and / or assertions code.

It can be argued that the fUML (Foundational UML) [22] can be applied in the proposed framework to avoid the utilization of UVM compliant third party simulation tool. This can be achieved by representing UMLSV semantics through fUML where the models can be directly executed to perform design verification. However, practically, it is almost impossible to utilize fUML in the proposed framework to provide a realistic simulation solution. Primarily, current fUML execution engine only provides simple verification features through simulation. On the other hand, UVM is a complete design verification methodology for integrated circuits with several advanced features. Consequently, to use fUML in the proposed framework, it is required to extend the current fUML / ALF execution engine in order to include all the UVM features which is practically impossible. In this context, Federico Ciccozzi et al [57] has carried out a systematic review to investigate the execution of UML models and observed that fUML is rarely used to provide a direct simulation solution.

It is analyzed from the investigation of *Verification Method (VM)* parameter that MADES and Gaspard profiles provide the support for both formal and dynamic verification. Consequently, both types of target platforms can be generated. For example, MADES supports Modelica and Temporal Logic Formulae (TLF) [31] to perform dynamic and formal verification respectively. It is also analyzed that the few studies deal with the formal design verification only. For example, DAM-Rail [47] profile supports advance formal verification support through Repairable Fault Tree (RFT) [33], Bayesian Network (BN) [35] and Generalized Stochastic Petri Net (GSPN) [34]. Currently, our proposed framework only supports dynamic verification through SVA's. We believe that it is highly suitable to provide both type of verification approaches (Formal and

[4] https://www.polarsys.org/list-of-projects
[5] https://mast.unican.es/
[6] http://rcl.dsi.unifi.it/projects/tools

Dynamic) for the design automation of complex and large embedded systems. Therefore, in future, we intend to include formal verification facility in the proposed framework.

Despite the frequent utilization of ABV at lower abstraction level, it has been observed that state-of-the-art UML-based solutions rarely offer the ABV support [37-39], [44]. Furthermore, most of these solutions [37-39] are partially completed as very few details are provided regarding the modeling methodology and relatively simple case studies are used to demonstrate the applicability without any model transformation support. The only complete model driven framework with the support of ABV is presented in [44], where the verification properties are expressed through PSL (Property Specification Language) and the target code is generated in C language. In contrast to our proposed framework, the work in [44] utilizes PSL to specify the verification properties which is considered in the middle of higher and lower abstraction level as properties /constraints are specified neither in the UML based design (higher level) nor in the target languages (lower level). Particularly, the verification properties are expressed in PSL through a separate editor after the completion of design and transformation process. This leads to a gap between design and its verification. Moreover, ABV is performed in C language that doesn't offer advanced ABV features as supported in SystemVerilog. Furthermore, a nonstandard radCHECK tool is developed in [44] for simulation with limited features, therefore, reliability of the tool is questionable. On the other hand, our proposed framework includes verification properties at higher abstraction level through SVOCL that allows the simultaneous modeling of assertions along with the system design and supports ABV through SystemVerilog. Furthermore, the automatically generated target code is fully compliant with UVM, and therefore, it is not required to develop any customized tool. In other words, ABV can be performed through existing standard and reliable simulators like QuestaSIM [10]. Based on aforementioned facts, it can be concluded that the proposed framework is the first model-driven solution that introduces the features of ABV by means of SVA's.

Another challenging aspect in the MBSE for embedded systems is the inclusion of system verification properties / constraints in the models due to the complexity of temporal constraints. Therefore, researchers frequently propose different approaches [15] to incorporate temporal constraints in the models. In this context, it can be analyzed from Table 8 that each state-of-the-art profile develops customized approach to specify the system properties / constraints. For example, in MADES profile, the Time diagram is proposed to specify the timing constraints of embedded systems. However, in Table 8, we have written MADES profile in Property Specification Approaches column to avoid the complex representation. In the proposed framework, we use temporal OCL extension (SVOCL) to express properties / constraints in UMLSV. SVOCL is particularly designed for SVA's and capable of expressing complex temporal assertions in UMLSV.

In the given research context, it is worth mentioning the Portable Test and Stimulus Standard (PSS) [54] which is a new verification language developed by Accellera. The initial specifications [54] were released in 2018, based on graph notations, with two major class elements

i.e. Actions and Objects. The main rationale behind PSS is to provide a simple specification format, currently through Domain Specific Language (DSL) and C++, to allow the test portability among verification elements and reuse it at different levels of abstractions. Accellera doesn't provide any reference implementation for PSS and, therefore, tool vendors acquire the required flexibility by incorporating PSS in their corresponding tools as per requirements. For example, Cadence provides PSS library in "Perspec System Verifier" tool [55]. Similarly, Questa provides PSS support through "inFact" tool [56]. In this context, the proposed framework is highly supportive for the integration of PSS concepts at higher abstraction level. Particularly, PSS allows the specification of verification aspects through two textual language i.e. DSL and C++. In this regard, the major PSS concepts like Actions, Objects etc., as given in the standard specification document [54], can be represented through UMLSV stereotypes in the proposed framework. Subsequently, transformation engine can be upgraded to generate PSS implementations (e.g. Cadence PSS library etc.) along with the RTL and assertions code from high level models. This leads to perform a comprehensive design verification through PSS compliant simulator / tool.

To summarize, we introduce the semantics of SystemVerilog in UMLSV to provide higher abstraction layer for SystemVerilog which is the first attempt in this direction to the best of our knowledge. Therefore, the two significant contributions of the proposed framework as compared to state-of-the-art are:

- It provides higher abstraction layer for SystemVerilog through UMLSV. Particularly, system design can be modeled through UMLSV stereotypes (Section **2**). At the same time, temporal properties / constraints can be included in the UMLSV model through SVOCL.
- It provides a transformation engine to automatically generate SystemVerilog RTL (synthesizable) and assertions code from the source UMLSV models.

6. Conclusion and Future work

This article presents a novel model-driven framework to specify both design and its verification requirements at higher abstraction level. Particularly, UMLSV (**UML** profile for **S**ystem**V**erilog) profile is proposed to bring the semantics of SystemVerilog from the lower abstraction level (RTL) to the higher abstraction level (MBSE). The UMLSV is developed by exploiting the concepts of UML and its SYSML profile. Particularly, various UMLSV stereotypes are proposed to model the design (structure and behavior) of embedded systems. Furthermore, a temporal extension of OCL is used to express the timing properties / constraints in the UMLSV by means of SystemVerilog Assertions (SVA's). This leads to represent the system design and its verification requirements collectively at higher abstraction level, providing the foundation to perform Assertion Based Verification (ABV) in early development phases.

As a part of solution, a UMLSV transformation engine is developed to generate the Synthesizable SystemVerilog RTL code, along with the SVA's code. The application of UMLSV is demonstrated through four industrial case studies. The results prove that the UMLSV profile is capable of modeling both the design and its verification requirements collectively for a variety of

embedded systems. Furthermore, it is also verified that the UMLSV transformation engine is capable of generating synthesizable SystemVerilog RTL code, along with the SVA's code, such that the design verification can be performed instantly.

This research provides the application of SystemVerilog in MBSE and can be further extended in multiple directions. One probable direction is to include the simulation requirements in UMLSV, so that, the UVM-compliant test benches can also be generated in SystemVerilog along with RTL and assertions code. Another possibility is the investigation of similarities between the semantics of a particular formal verification approach (e.g. timed automata) and SystemVerilog. Such investigations will provide the basis for the extension of UMLSV to support both formal as well as dynamic verification approaches.

Acknowledgement

This project is funded by NSTIP (National Science Technology, Innovative Plan), Saudi Arabia under the Technology Area "Information Technology Strategic Priorities" and Track "Software Engineering and Innovated Systems". We acknowledge the support of KACST (King Abdul-Aziz City for Science and Technology) and STU (Science and Technology Unit) Makkah.

References

[1] Cormac Driver, Sean Reilly, Eamonn Linehan, Vinny Cahill,And Siobhan Clarke: Managing Embedded Systems Complexity with Aspect-Oriented Model-Driven Engineering, ACM Transactions on Embedded Computing Systems (TECS) 2010, Volume 10 Issue 2, Article No. 21 DOI: 10.1145/1880050.1880057

[2] Sebastien Guillet, Florent De Lamotte, Nicolas Le Griguer, Eric Rutten, Guy Gogniat And Jean-Philippe Diguet: Extending UML/MARTE to Support Discrete Controller Synthesis,Application to Reconfigurable Systems-on-Chip Modeling, ACM Transactions on Reconfigurable Technology and Systems 2014, Volume 7 Issue 3, Article No. 27 DOI: 10.1145/2629628

[3] Stéphane Lecomte, Samuel Guillouard, Christophe Moy, Pierre Leray and Philippe Soulard: A co-design methodology based on model driven architecture for real time embedded systems, Mathematical and Computer Modelling 2011, Volume 53, Issues 3–4, Pages 471–484 DOI: 10.1016/j.mcm.2010.03.035

[4] Muhammad Rashid, Muhammad Waseem Anwar and Muhammad Amir, "Towards the Tools Selection in Model Based System Engineering for Embedded Systems - A Systematic Literature Review", Journal of Systems and Software (JSS) 2015, Volume 106, Pages 150-163.

[5] Tom Fitzpatrick: SystemVerilog for VHDL users, Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE), 2004.

[6] Bengtsson, J.E., Yi, W.: Timed Automata: Semantics, Algorithms and Tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004)

[7] J. M. Spivey, The Z Notation: A Reference Manual. Englewood Cliffs, NJ, USA: Prentice-Hall, 1992.

[8] Hassan Sohofi and Zainalabedin Navabi: System-level assertions: approach for electronic system-level verification, IET Computers & Digital Techniques 2015, Volume 9, Issue 3, Pages 142-152.

[9] IEEE SystemVerilog Standard 1800-2009. .http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=5354133

[10] Mentor Graphics Questa SIM, Last Accessed March 2018 http://www.mentor.com/products/fv/questa/

[11] Kyuchull Kim: Functional Verification of A Safety Class Controller For Npps Using A Uvm Register Model, Journal of Nuclear Engineering and Technology 2014, Volume 46, Issue 3, Pages 381-386

[12] Papyrus Modeling Editor, Last Accessed December 2018. Available from: http://www.eclipse.org/modeling/mdt/papyrus/

[13] Object Management Group 2012. OMG System Modeling Language Specification. Retrieved March 6, 2012 from

http://www.omg.org/spec/SysML/1.3/

[14] Object Management Group, Unified Modeling Language standard, Last Access February 2018. https://www.omg.org/spec/UML/2.5/About-UML/

[15] Muhammad Rashid, Muhammad Waseem Anwar, Farooque Azam and Muhammad Kashif: Model-Based Requirements and Properties Specifications Trends for Early Design Verification of Embedded Systems, IEEE 11th System of Systems Engineering Conference (SoSE) 2016.

[16] Eclipse Model-to-Text Project – JET, Last Accessed April 2018 https://www.eclipse.org/modeling/m2t/

[17] Marcello M. Bersani and Marisol García-Valls : Online verification in cyber-physical systems: Practical bounds for meaningful temporal costs, Journal of Software: Evolution and Process, Wiley 2018, Volume 30, Issue 3, pp 1-25.

[18] Asieh Salehi Fathabadi, Michael J.Butler, ShengYang, Luis Alfonso, Maeda-Nunez, James Bantock, Bashir M.Al-Hashimi and Geoff V.Merrett : A model-based framework for software portability and verification in embedded power management systems, Journal of Systems Architecture 2018, Volume 82, Pages 12-23.

[19] Huafeng Zhang, Yu Jiang, Han Liu, Hehua Zhang, Ming Gu and Jiaguang Sun : Model driven design of heterogeneous synchronous embedded systems, 31st IEEE/ACM International Conference on Automated Software Engineering (ASE) 2016.

[20] Haitao Zhang, Guoqiang Li, Daniel Sun, Yonggang Lu and Ching-Hsien Hsu : Verifying cooperative software: A SMT-based bounded model checking approach for deterministic scheduler, Journal of System Architecture 2017, Vol 81, pp 7-16.

[21] Muhammad Waseem Anwar, Muhammad Rashid, Fraooque Azam and Muhammad Kashif : Model-based design verification for embedded systems through SVOCL: an OCL extension for SystemVerilog, An International Journal of Design Automation for embedded systems 2017, Vol 21, Issue 1, Pages 1-36.

[22] OMG fUML – Foundational UML standard, Last Accessed October 2018. https://www.omg.org/spec/FUML/

[23] Magic Draw, Last Accessed, August 2018. http://www.nomagic.com/products/magicdraw.html

[24] Accellera Universal Verification Methodology Standard, Last Accessed Nov. 2018. http://www.accellera.org/downloads/standards/uvm

[25] Eclipse Acceleo (M2T), Last Accessed December 2018. https://eclipse.org/acceleo/

[26] UMLSV Transformation Engine with sample case studies, Last Accessed Nov. 2017. http://modeves.com/umlsvte.html

[27] UMLSV Guidelines. Last Accessed Dec. 2018. http://modeves.com/umlsvman.html

[28] UMLSV Profile, Design verification details, Last Accessed October 2018. http://www.modeves.com/dvquesta.html

[29] Xilinx Vivado Design Suite, Last Accessed February 2017, http://www.xilinx.com/products/design-tools/vivado.html

[30] IEEE Computer Society, IEEE std 1647-2008, IEEE Standard for the Functional Verification Language e., Standard IEEE Std 1647-2008, IEEE, NY, USA, August 2008.

[31] Formal Dynamic Semantics of the Modelling Notation," Tech. Rep., 2010, http://www.mades-project.org/

[32] L. Besnard, T. Gautier, P. Le Guernic, and J.-P. Talpin. Compilation of Polychronous Data Flow Equations. In S. Shukla and J.-P. Talpin, editors, Correct-by-Construction Embedded Software Synthesis: Formal Frameworks, Methodologies, and Tools. Springer, 2010.

[33] Codetta Raiteri D,Iacono M, Franceschinis G,Vittorini V, "Repairable fault tree for the automatic evaluation of repair policies. In: Proceedings of the2004 international conference on dependable systems and networks; 2004. Pp.659–68.

[34] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis, Modelling with Generalized Stochastic Petri Nets, ser. Wiley Series in Parallel Computing. John Wiley and Sons, 1995.

[35] Charniak E. Bayesian networks without tears: making Bayesian networks more accessible to the probabilistically unsophisticated.AIMagazine1991; 12 (4):50–63.

[36] G. Delaval, H. Marchand, and E. Rutten. 2010. Contracts for modular discrete controller synthesis. In Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'10). ACM Press, New York, 57–66.

[37] Emad Ebeid, Davide Quaglia and Franco Fummi: Generation of SystemC/TLM code from UML/MARTE sequence diagrams for verification, IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS) 2012, Pages 187-190 DOI: 10.1109/DDECS.2012.6219051

[38] A. Banerjeey S. Ray, P. Dasgupta P and P. Chakrabarti: A Dynamic Assertion-based Verification Platform for Validation of UML designs, ACM SIGSOFT Software Engineering Notes 2012, Volume 37 Issue 1,Pages 1-14 DOI: 10.1145/2088883.2088891

[39] Doron Drusinsky, James Bret Michael, Thomas W. Otani and Man-Tak Shing: Validating UML Statechart-Based Assertions Libraries for Improved Reliability and Assurance, Second International Conference on Secure System Integration and Reliability Improvement 2008. Pages 47-51 DOI: 10.1109/SSIRI.2008.54

[40] Bran Selic: Using UML for modeling complex real-time systems, In Languages, Compliers and Tools for Embedded Systems (LCTES) 1998, pages 250-260

[41] Siru Ni, Yi Zhuang, Zining Cao, and Xiangying Kong: Modeling Dependability Features for Real-Time Embedded Systems, IEEE Transactions On Dependable And Secure Computing 2015, Volume 12, Issue 2, Pages 190 – 203.

[42] Eamonn Linehan and Siobhan Clarke: An aspect-oriented, model-driven approach to functional hardware verification, Journal of Systems Architecture, Elsevier 2012, Volume 58, Issue 5, Pages 195-208, DOI: 10.1016/j.sysarc.2011.02.001

[43] Imran R. Quadri, Etienne Brosse, Ian Gray, Nicholas Matragkas, Leandro Soares Indrusiak, Matteo Rossi, Alessandra Bagnato and Andrey Sadovykh: MADES FP7 EU Project: Effective High Level SysML/MARTE Methodology for Real-Time and Embedded Avionics Systems, 7th International Workshop Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC) 2012, Pages 1-8, DOI: 10.1109/ReCoSoC.2012.6322882

[44] Giuseppe Di Guglielmo, Luigi Di Guglielmo, Andreas Foltinek,, Masahiro Fujita, Franco Fummi, Cristina Marconcini and Graziano Pravadelli: On the integration of model-driven design and dynamic assertion-based verification for embedded software, Journal of Systems and Software, Volume 86, Issue 8, August 2013, Pages 2013–2033. DOI: 10.1016/j.jss.2012.08.061

[45] Abdoulaye Gamatié, Sébastien Le Beux, Éric Piel, Rabie Ben Atitallah, Anne Etien, Philippe Marquet, and Jean-Luc Dekeyser: A Model-Driven Design Framework for Massively Parallel Embedded Systems, ACM Transactions on Embedded Computing Systems (TECS) 2011, Volume 10 Issue 4, Article No. 39. DOI: 10.1145/2043662.2043663

[46] Schamai W.. Modelica Modeling Language (ModelicaML) A UML Profile for Modelica, technical report 2009:5, EADS IW, Germany, Linkoping University, Sweden, 2009

[47] S. Bernardi, F. Flammini, S. Marrone , N. Mazzocca, J. Merseguer, R. Nardone and V. Vittorini: Enabling the usage of UML in the verification of railway systems: The DAM-rail approach, Reliability Engineering & System Safety 2013, Volume 120, Pages 112–126 DOI: 10.1016/j.ress.2013.06.032

[48] Stéphane Lecomte, Samuel Guillouard, Christophe Moy, Pierre Leray and Philippe Soulard: A co-design methodology based on model driven architecture for real time embedded systems, Mathematical and Computer Modelling 2011, Volume 53, Issues 3–4, Pages 471–484 DOI: 10.1016/j.mcm.2010.03.035

[49] Sebastien Guillet, Florent De Lamotte, Nicolas Le Griguer, Eric Rutten, Guy Gogniat And Jean-Philippe Diguet: Extending UML/MARTE to Support Discrete Controller Synthesis,Application to Reconfigurable Systems-on-Chip Modeling, ACM Transactions on Reconfigurable Technology and Systems 2014, Volume 7 Issue 3, Article No. 27 DOI: 10.1145/2629628

[50] Luciano Baresi, Angelo Morzenti, Alfredo Motta, Mohammad Mehdi Pourhashem K and Matteo Rossi : A Logic-Based Approach for the Verification of UML Timed Models, ACM Transactions on Software Engineering and Methodology (TOSEM) 2017, Vol 26, Issue 2, Article No. 7.

[51] Martin Grant and Muller Wolfgang : UML for SOC Design, Springer book 2005.

[52] Grotker T., Liao S., Martin G. and Swan S. : System Design with SystemC, Springer book 2002.

[53] Lavagno Luciano, Martin Grant and Selic Bran : UML for Real: Design of Embedded Real-Time Systems, springer book 2003.

[54] Accellera Portable Test and Stimulus Standard (PSS), Last Accessed Sept. 2019. https://www.accellera.org/downloads/standards/portable-stimulus

[55] Cadence Perspec System Verifier, Last Accessed Sept. 2019 , https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/software-driven-verification/perspec-system-verifier.html

[56] Mentor Graphics Questa inFact Tool, Last Accessed Sept. 2019, https://www.mentor.com/products/fv/infact/

[57] Federico Ciccozzi, Ivano Malavolta and Bran Selic : Execution of UML models: a systematic review of research and practice, Journal of Software and System Modeling 2018, pp 1-48.