A TASM-based Requirements Validation Approach for Safety-critical Embedded Systems

Jiale Zhou, Yue Lu, and Kristina Lundqvist

School of Innovation, Design and Engineering
Mälardalen University, Västerås, Sweden
{zhou.jiale, yue.lu, kristina.lundqvist}@mdh.se

Abstract. Requirements validation is an essential activity to carry out in the system development life cycle, and it confirms the completeness and consistency of requirements through various levels. Model-based *formal methods* can provide a cost-effective solution to requirements validation in a wide range of domains such as safety-critical applications. In this paper, we extend a formal language Timed Abstract State Machine (TASM) with two newly defined constructs *Event* and *Observer*, and propose a novel requirements validation approach based on the extended TASM. Specifically, our approach can: 1) model both functional and non-functional (e.g. timing and resource consumption) requirements of the system at different levels and, 2) perform requirements validation by utilizing our developed toolset and a model checker. Finally, we demonstrate the applicability of our approach in real world usage through an industrial case study of a Brake-by-Wire system.

1 Introduction

With the growing complexity of safety-critical systems, requirements are no longer merely specified at the outset of the systems development life cycle (SDLC). On the contrary, there is a continuum of requirements levels as more and more details are added throughout the SDLC, which can roughly be divided into two categories in terms of high-level and low-level requirements [2]. High-level requirements describe what features the proposed system has (i.e. features hereafter) and low-level requirements state how to develop such a system (i.e. requirements hereafter). Studies have revealed that most of the anomalies discovered in late development phases can be traced back to hidden flaws in the requirements [9] [11], such as contradictory or missing requirements, or requirements that are discovered to be impossible to satisfy features at the late phase of development. For this reason, requirements validation is playing a more and more significant role in the development process, which confirms the correctness of requirements, in the sense of consistency and completeness [20]. In details, consistency refers to situations where a specification contains no internal contradictions in the requirements, while completeness refers to situations where the requirements must possess two fundamental characteristics, in terms of neither

objects nor entities are left undefined and the requirements can address all of the features.

In order to increase the confidence in the correctness of the requirements, model-based *formal methods* techniques have been to a large extend investigated into the field of requirements validation [7] [10]. In these techniques, the system design derived from requirements is often specified in terms of analyzable models at a certain level of abstraction. Further, features are formalized into verifiable queries or formulas and then fed into the models to perform model checking and/or theorem proving. In this way, the requirements are reasoned about to resolve contradictions, and it is also verified that they are neither so strict to forbid desired behaviors, nor so weak to allow undesired behaviors. However, such *formal methods* techniques also suffer from some limitations, such as how to ease the demand of heavy mathematics background knowledge to perform theorem proving, and how to model the target without having the state explosion problem of model checking occurred.

To tackle with the aforementioned limitations, we propose an approach to requirements validation using an extended version of the formal language Timed Abstract State Machine (TASM), which contains new constructs *TASM Event* and *TASM Observer*. Additionally, TASM has shown its success in the area of systems verification in [18] [19], with some distinctive features: 1) TASM supports the formal specification of both functional behaviors and non-functional properties of safety-critical systems w.r.t. timing and resource consumption and, 2) It is a literate language being understandable and usable without requiring extensive mathematical training, which avoids obscure mathematics formulae and, 3) TASM provides a toolset [16] to execute the pertaining TASM models for the purposes of analysis. The *Observer* technique [4] has an origin in the model-based testing domain where it has been used to specify and observe coverage criteria as well as verify such observable properties, but without changing the system's behaviors. The applications and advantages of using the *Observer* technique inspire us to exploit it to perform requirements validation, which makes a detour on the state explosion issue of model checking by not adding new states in the analysis. To be specific, our approach consists of three main steps:

- ***Requirements modeling*** models requirements by using various constructs in TASM.
- ***Features modeling*** translates features into our newly defined TASM observers that are used for the later analysis.
- ***Requirements validation*** contains four kinds of validation checking on focus, i.e. *Logical Consistency Checking*, *Auxiliary Machine Checking*, *Coverage Checking*, and *Model Checking*, as in the consistency and completeness checking of requirements.

The main contributions of this work are three-fold: 1) We extend the TASM language with two newly defined constructs in terms of *Event* and *Observer* and, 2) We propose a novel approach to requirements validation by using the extended TASM language and, 3) We demonstrate the applicability of our approach through a case study. The remainder of this paper is organized as follows:

An introduction to the TASM language and its extension is presented in Section 2. Section 3 introduces the Brake-by-Wire (BbW) system and its requirements. Our approach to requirements validation is described and demonstrated by using the BbW system in Section 4. Section 5 discusses the related work, and finally concluding remarks and future work are drawn in Section 6.

2 TASM Language and Its Extension

Figure 1 shows the meta-model of the extended TASM language in UML class diagram. The constructs included in the dashed rectangle are the new TASM constructs defined in this work. Section 2.1 gives an overview of the TASM language and Section 2.2 presents the extension of TASM.

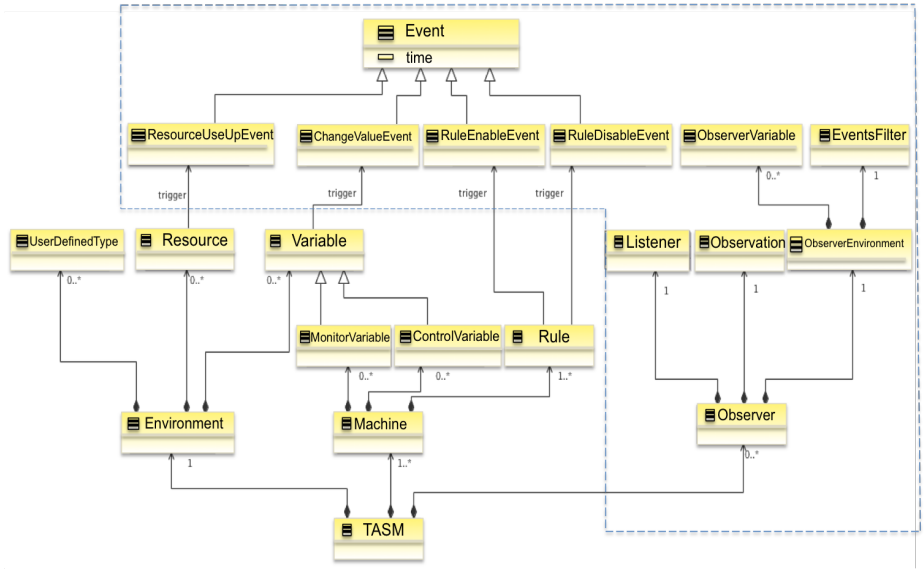

Fig. 1. The Meta-model of the extended TASM language.

2.1 Overview of TASM

TASM [16] is a formal language for the specification of safety-critical systems, which extends the Abstract State Machine (ASM) [5] with the capability of modeling timing properties and resource consumption of applications in the target system. TASM inherits the easy-to-use feature from ASM, which is a literate specification language understandable and usable without extensive mathematical training [8]. A TASM model consists of two parts – an environment and a set of main machines. The environment defines the set and the type of variables, and the set of named resources which machines can consume. The main machine is made up of a set of monitored variables which can affect the machine execution, a set of controlled variables which can be modified by machines, and a set of

machine rules. The set of rules specify the machine execution logic in the form of "if *condition* then *action*", where *condition* is an expression depending on the monitored variables, and *action* is a set of updates of the controlled variables. We can also use the rule "else then *action*" which is enabled merely when no other rules are enabled. A rule can specify the annotation of the time duration and resource consumption of its execution. The duration of a rule execution can be the keyword *next* that essentially states the fact that time should elapse until one of the other rules is enabled.

TASM describes the basic execution semantics as the computing steps with time and resource annotations: In one step, it reads the monitored variables, selects a rule of which *condition* is satisfied, consumes the specified resources, and after waiting for the duration of the execution, it applies the update set instantaneously. If more than one rules are enabled at the same time, it non-deterministically selects one to execute. As a specification language, TASM supports the concepts of parallelism which stipulates TASM machines are executed in parallel, and hierarchical composition which is achieved by means of auxiliary machines which can be used in other machines. There are two kinds of auxiliary machines - *function* machines which can take environment variables as parameters and return execution result, and *sub* machines which can encapsulate machine rules for reuse purpose [16]. Communication between machines, including main machines and auxiliary machines, can be achieved by defining corresponding environment variables.

2.2 The Extension to TASM

Our extension to TASM consists of two main parts, i.e. *TASM Event* and *TASM Observer* (Event and Observer hereafter, respectively) as shown in Figure 1.

Definition 1 *TASM Event (EV). TASM Event E defines the possible types of an event instance, including ResourceUsedUpEvent, ChangeValueEvent, RuleEnableEvent, and RuleDisableEvent. An event instance e is triggered by the corresponding TASM construct, which is a tuple* $< E, t >$*, where E is the type of the event instance, and t is the time instant when the instance occurs.*

The events of $ChangeValueEvent$ type is triggered by a specific TASM environment variable whenever its value is updated,, which can be referenced in the form of *VariableName->EventType*. The $ResourceUsedUpEvent$ is triggered by the case whenever the resource of the application is consumed totally, which can be referenced in the form of *ResourceName->EventType*. The $RuleEnableEvent$ and $RuleDisableEvent$ are triggered whenever a specific TASM rule is enabled or disabled, respectively, which can be referenced in the form of *MachineName->RuleName->EventType*.

Definition 2 *TASM Observer. An observer is a tuple* $< ObserverEnvironment, Listener, Observation >$*, where:*

- $ObserverEnvironment$ *is a tuple* $< ObserverVariable, EventsFilter >$*, where* $ObserverVariable$ *is a set of variables that can be used by both* $Listener$ *and* $Observation$*, and* $EventsFilter$ *can be configured to filter out events irrelevant to the observer.*

- *Listener specifies the observer execution logic in the form of "**listening** condition **then** action", where the condition is an expression describing the sequence of the occurrence of events and the action is a set of actions updating the value of observer variables when the condition evaluates to be true.*
- *Observation is a predicate of the TASM model, which can evaluate to be either true or false, depending on the value of corresponding observer variables.*

In this work, we only introduce the informal execution semantics of *Observer*, as depicted in Figure 2, and the formal semantics is considered as part of our future work. Basically, in the runtime, the TASM model often produces massive events according to the modeled application. After the $EventsFilter$ removes the irrelevant events, the remaining events will be logged in the local database, namely $EventsLog$. Next, the *Listener* defined in *Observer* will evaluate its *condition* based off of the sequence of logged events. Since *regular expression* is usually used as a sequential search pattern, the specification of the event sequence follows the syntax and semantics of regular expression. If the *condition* is satisfied, then the *action* will start to update the observer variables. Once all of the updates are executed, the *Observation* will be concluded based on the updated observer variables. A running TASM model (representing the target system) can be observed by several observers at the same time.

For a better understanding, we give an example of *Observer* as shown in Figure 3, where *eventA* and *eventB* are $RuleEnableEvent$ type, and *eventC* and *eventD* are $RuleDisableEvent$ type. The observer variables include a Boolean variable *ov* (initiated as false) and a Time variable *time* (initiated as zero). $ChangeValueEvent$ and $ResourceUsedUpEvent$ are regarded as irrelevant events and removed by the $EventsFilter$, the $RuleEnableEvent$ and $RuleDisableEvent$ events are logged in the *Eventslog* database. As shown in line 9 in Figure 3, the expression of the *Listener* condition in regular expression, represents the event sequence that begins with *eventA*, followed by arbitrary events (represented by ".*") in the middle, and ends with two events in terms of either *eventB and eventD*, or *eventC and eventD*. If the condition evaluates to be satisfied, the observer variable *ov* will be assigned as true, and *time* as the interval between *eventA* and *eventD*. In this example, if the events sequence in the condition is detected and the interval *time* is larger than 100, the *Observation* will be concluded as a true predicate.

3 Case Study

Our case study is a Brake-by-Wire (BbW) system which is a demonstrator at a major automotive company [13]. The BbW system aims to replace the mechanical linkage between the brake pedal and the brake actuators. Further, the BbW system consists of micro-controller units, sensors, actuators and communication bus, which interprets driver's operation and operating conditions, through sensors, to decide on the desired brake torque of the brake actuators for appropriate brake force on each wheel.

The features that the BbW system should possess are described as follows:

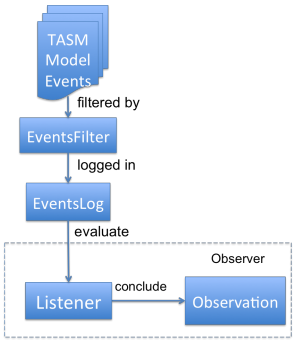

Fig. 2. The workflow of the Observer execution.

```
ObserverVariables:{
  Boolean ov := false;
  Time time := 0;
}
EventsFilter:{
  filter out: ChangeValueEvent,
       ↪ResourceUsedUpEvent;
}
Listener:{
  listening eventA.*(eventB | eventC)
       ↪eventD then
    ov := true;
    time := eventD.t - eventA.t;
}
Observation:{
  ov == true and time > 100;
}
```

Fig. 3. An example of the TASM Observer.

- **Req H1**: The system shall provide a base brake functionality where the driver indicates that she/he wants to reduce speed so that the braking system starts decelerating the vehicle.
- **Req H2**: When the brake pedal is not pressed, the brake shall not be active.
- **Req H3**: The time from the driver's brake request till the actual start of the deceleration should be no more than 300 ms.

The list of requirements for the BbW system in our work is as follows:

- **Req L1**: The brake torque calculator shall compute the driver requested torque and send the value to the vehicle brake controller, when a brake pedal displacement is detected.
- **Req L2**: The vehicle brake controller shall decide the required torque on each wheel and each of the required wheel torque values is sent together with the sensed vehicle velocity to the Anti-lock Braking System (ABS) function on respective wheel.
- **Req L3**: The ABS function shall decide appropriate braking force on each wheel, based on the received torque request, current vehicle velocity and wheel angular velocity.

4 The TASM-based Approach to Requirements Validation

In this section, we will introduce our approach that addresses the issue of formalizing and validating requirements specifications written in natural language. Further, our approach is based on the use of the extended TASM language to formalize both requirements and features. We will go into details about each step by introducing the adhering sub-steps and show an illustration by using the BbW system. Specifically, Section 4.1 and Section 4.2 discuss modeling of the requirements and features respectively, and Section 4.3 presents the analysis and results of requirements validation of the BbW system.

4.1 Requirements Modeling

The first step of our approach is to analyze the low-level requirements (i.e. , requirements) in natural language and formalize them by using the corresponding TASM models. This step contains five sub-steps, as shown in Figure 4:

Fig. 4. The sub-steps of requirements modeling.

- **Step 1: Requirements Preprocessing** distinguishes functional requirements from non-functional requirements.
- **Step 2: Components Identification** extracts the possible software components of the system referred in the functional requirements and maps them onto TASM main machines.
- **Step 3: Connections Identification** identifies the connections between different software components, according to a certain type of interactions.
- **Step 4: Behavior Specification** specifies the behaviors of components, which implement different system functionalities.
- **Step 5: Property Annotation** adds timing and resource consumption annotations to the relevant TASM model.

Requirements Preprocessing. At this step, we need to distinguish functional requirements from non-functional requirements. The functional requirements will be formalized into executable TASM models, and non-functional requirement in terms of timing and resource consumption requirements can provide useful information for property annotation. In the BbW system, all the requirements, i.e. *ReqL1*, *ReqL2* and *ReqL3*, are functional requirements.

Components Identification. The identification of the system components and the mapping of each component onto a TASM main machine is of importance in the process. In order to do so, we recommend the following two tasks:

- *Identification of the external (or environmental in other words) components that interact with the system.*
- *Identification of the internal components that compose of the system.*

At this step, a list of main machines will be defined for the BbW system, as shown in Table 1.

Connections Identification. In the TASM model, asynchronous communication between different main machines can be implemented by using a set of variables, which ignores the transmission delay between machines. On the contrary, the common form of inter-process communication (IPC) is message-passing, which considers the transmission delay and bandwidth consumption as

Main Machine	Quantity	Category	Description
DRIVER	1	External Entity	model the driver's behavior
VEHICLE	1	External Entity	model the behavior of the vehicle
TORQUE_CALC	1	Micro-controller	calculate the driver's requested torque
BRAKE_CTRL	1	Micro-controller	calculate the requested torque per wheel
ABS_CTRL	4	Micro-controller	calculate the brake force on each wheel
BRAKE_ACTU	4	Actuator	perform the brake force on each wheel
WHLSPD_SENSOR	4	Sensor	sense the rotating speed of each wheel
VCLSPD_SENSOR	1	Sensor	sense the moving speed of the vehicle
PEDAL_SENSOR	1	Sensor	sense the position of the brake pedal
COMMU_BUS	1	Bus	the communication bus

Table 1. The TASM main machines model the entire Brake-by-Wired system.

unavoidable. To this end, we define a main machine with the annotation of time and bandwidth as a means of modeling the communication bus. In our case study, the sensors in the BbW system communicate with the corresponding controllers through ports using signals, where transmission delay can be ignored. Further, a specific TASM main machine i.e. COMMU_BUS (in Table 1) models the communication bus, which is responsible for the communication between the brake controller and the ABS controllers.

Behavior Specification. There is no silver-bullet to model the behaviors of various components in TASM. Based on our experiences, we recommend the following steps:

- *Identification of possible states of the target system*: A user-defined type is used to represent the possible states, and a state variable is defined to denote the current state of the system.
- *Identification of the transition conditions of states*: The conditions of a certain machine rule are given, according to the corresponding value of the state variable and the transition conditions.
- *Identification of the actions when the system enters a specific state*: The actions of machine rules are specified, based on the behaviors of a component and the next possible state.

In the BbW system, all of the identified components (i.e. TASM main machines) are divided into five categories according to different functionalities: external entity, micro-controller, actuator, sensor, and bus. For reasons of space, we do not list all the rules used by the identified TASM main machines. Instead, we list the rules of four typical templates in our case study, i.e. micro-controller, actuator, sensor, and bus. In order to have a better understanding on the proposed sub-steps, we discuss the specification of a micro-controller component in detail.

A micro-controller component is activated by an event, and it reads a set of variables and performs a sequence of computation after being activated. When it finishes execution, the result will be used by other components. Therefore, the micro-controller component typically has three possible states – WAIT (initial state), COMPUTE, and SEND: The WAIT sate denotes that the micro-controller is waiting for activation and, the COMPUTE state represents that the micro-controller is performing computation. The SEND state introduces that the micro-controller is sending the results to other components. Figure 5 shows

the rules of the TASM main machine, which models the micro-controller. PERFORM_COMPUTATION() and SEND_RESULT() are sub machines.

Figure 6 shows the machine rules that model an actuator, and PERFORM_ACTUATION() is a sub machine. Figure 7 shows the rules of the TASM main machine, which models a sensor. Measure_Quantity() is a function machine. Figure 8 shows the machine rules, which models the communication bus. Get_Message() is a function machine and TRANSMITTING_MESSAGE() is a sub machine.

```
R1:Activation{
  if ctrl_state=wait and new_event=
      ↪True then
    ctrl_state := compute;
    new_event  := False;
}
R2:Computation{
  t:=computation_time;
  if ctrl_state = compute then
    PERFORM_COMPUTATION();
    ctrl_state := send;
}
R3:Send{
  if ctrl_state = send then
    SEND_RESULT();
    ctrl_state := wait;
}
R4:Idle{
  t := next;
  else  then
    skip;
}
```

Fig. 5. The TASM main machine models the micro-controller component.

```
R1:Trigger{
 if actu_state=wait and new_event=
     ↪True then
   new_event  := False;
   actu_state := actuate;
}
R2:Actuation{
 t:=actuation_time;
 if actu_state=actuate then
   PERFORM_ACTUATION();
   actu_state := wait;
}
R3:Idle{
 t:= next;
 else then
   skip;
}
```

Fig. 6. The TASM main machine models the actuator component.

```
R1:Sample{
  if sensor_state = sample then
    sensor_value :=
        ↪Measure_Quantity();
    sensor_state := send;
}
R2:Send{
  if sensor_state = send and
      ↪sensor_value >= threshold
      ↪then
    observer_value  := sensor_value
        ↪;
    new_sample_value:= True;
    sensor_state    := wait;
}
R3:Wait{
  t := period;
  if sensor_state = wait then
    sensor_state := sample;
}
```

Fig. 7. The TASM main machine models the sensor component.

```
R1:Transmit{
  if bus_state=idle and new_message
      ↪=True then
    bus_message := Get_Message();
    bus_state := engaged;
}
R2:Send{
  t:=bus_delay;
  band:= bandwidth;
  if bus_state = engaged then
    TRANSMITTING_MESSAGE();
    bus_state := idle;
}
R3:Wait{
  t := next;
  else then
    skip;
}
```

Fig. 8. The TASM main machine models the communication bus component.

Non-functional Property Annotation. The accurate estimation of the pertaining non-functional properties of the target system is playing a paramount role in performing non-functional requirements validation. The Property Annotation step can be carried out in the following ways:

- The estimates can be determined based upon the non-functional requirements specified in the low-level requirements.
- The estimates can be obtained by using existing well-known analysis methods, e.g. Worst-Case Execution Time (WCET) Analysis [12] for time duration of rules.
- The estimates can be determined based upon the information in the related hardware specifications, e.g. the time duration and power consumption of a communication bus transferring one message.
- However, in some cases, the estimates can also be given by the experiences of domain experts, if the accurate estimation is not possible.

We annotate the aforementioned TASM models with time duration and resource consumption, and the annotation terms *computation_ time*, *actuation_ time*, *period*, *bus_ delay* and *bandwidth* are either a specific value or a range of values, which are given by our domain knowledge for simplicity.

4.2 Features Modeling

Our approach proceeds with the formalization of high-level requirements, i.e. , features. At this step, each feature will be translated into corresponding TASM observer(s). The formalization consists of the following sub-steps:

- **Step 1: Listener Specification** specifies the possible events sequence which represents the observable functional behaviors or non-functional properties required by the feature, and the corresponding actions taken on observer variables when the sequence is caught by the Listener.
- **Step 2: Observation Specification** formalizes a predicate depending on the observer variables. If the predicate of the Observation holds, i.e. evaluates to be true, it implies that the satisfaction of the feature can be observed in the system.
- **Step 3: Events Filtering** identifies the interesting events and filters out the irrelevant events by specifying $EventsFilter$.
- **Step 4: Traceability Creation** links the specified Observer to the textual requirements. The link is used for requirements traceability from the formalization to natural language requirements in order to perform coverage checking.

In the BbW system, there are three features i.e. *ReqH1*, *ReqH2* and *ReqH3*. The specification of *Observer* is illustrated by applying the proposed steps to *ReqH1*. To be specific, *ReqH1* states "The system shall provide a base brake functionality where the driver indicates that she/he wants to reduce speed so that the braking system starts decelerating the vehicle", and the interesting events sequence consists of three parts. The first part "PEDAL_SENSOR->Send->RuleEnableEvent" denotes the event that is triggered when the Send

rule of the PEDAL_SENSOR main machine is enabled, which models the behavior that the brake pedal is pressed by the driver. The second part ".*" has the same semantic with the counterpart defined in *regular expression*, which means an arbitrary number of events regardless of their type. The last part "BRAKE_ACTU->Actuation->RuleEnableEvent" represents the event that is triggered after the Actuation rule of the BRAKE_ACTU main machine is executed, i.e. disabled, which models the behavior that the brake actuator acts on the wheels i.e. decreases the speed of the vehicle. When the events sequence is detected, the Observation "ov == true" evaluates to be true, which indicates that the satisfaction of *ReqH1* can be observed in the TASM model.

```
ObserverVariables:{
  Boolean ov := false;
}
EventsFilter:{
  filter out: ChangeValueEvent, ResourceUsedUpEvent, RuleDisableEvent;
}
Listener:{
  listening PEDAL_SENSOR->Send->RuleEnableEvent .* BRAKE_ACTU->Actuation->
      ↪RuleEnableEvent then
    ov := true;
}
Observation:{
  ov == true;
}
```

Fig. 9. The Observer of Req H1.

4.3 Requirements Validation

Validation of the formalized requirements aims at increasing the confidence in the validity of requirements. In this work, we assume that there is a semantic equivalence relation between the requirements and TASM models, and between features and observers. This is built upon the fact that the TASM models and observers are derived from the documented requirements and features, by following the proposed modeling steps based on our thorough understanding of the BbW system. The validation goal is achieved by following several analysis steps, based on the use of the derived TASM models and observers which may help to pinpoint flaws that are not trivial to detect. Such validation steps in our approach are:

- **Logical Consistency Checking**. The term of logical consistency can be intuitively explained as "free of contradictions in the specifications". In our work, the logical consistency checking can be performed on the executable TASM models, i.e. requirements, by our developed tool TASM Toolset. Two kinds of inconsistency flaws can be discovered. One kind of flaw is that two machine rules are enabled at the same time, which is usually caused by the fact that there exist unpredictable behaviors in the requirements. The other is that different values are assigned to the same variable at the same time, which is usually caused by the fact that there exist hidden undesired behaviors in the requirements.

- **Auxiliary Machine Checking**. Auxiliary machines include function machine and sub machine. When the TASM TOOLSET starts to execute the TASM model, if there exists any undefined auxiliary machine, the tool will detect this situation, stop proceeding, and generate an error message. The existence of undefined auxiliary TASM machines, in terms of functions and sub machines, violates the completeness of TASM model specifying requirements. The undefined auxiliary TASM machines are usually caused by the lack of detailed descriptions of the proposed system's behaviors.
- **Coverage Checking**. Coverage checking corresponds to checking whether the desired behaviors specified in features can be observed in the TASM model, which is an important activity of requirements completeness checking. To perform the coverage checking, all of the features are translated into observers which observe the execution of TASM models at runtime. If the *Observation* holds, the corresponding feature can be regarded as covered by the requirements.
- **Model Checking**. The TASM machines can be easily translated into timed automata through the transformation rules defined in [16]. The transformation enables the use of the UPPAAL model checker to verify the various properties of the TASM model. This check aims at verifying whether the TASM model is free of deadlock and whether an expected property specified in a feature is satisfied by the TASM model. It is necessary to stress that the essential difference between *Model Checking* and *Coverage Checking* is whether a property is exhaustively checked against a model or not. Although a sound property checking is desired, in some cases *Model Checking* will encounter state explosion problem, which limits its usefulness in practice.

We follow the validation steps to check the validity of the requirements of the BbW system. First, we use the TASM TOOLSET to perform *Logical Consistency Checking* on the formalized TASM model. As in the fact that there are no inconsistency warnings reported by the tool, we therefore proceed the validation steps with *Auxiliary Machine Checking*. As shown in Figure 5, 6, 7 and 8, there exist some undefined auxiliary machines in the TASM models of those typical components, which also have been detected by our TASM TOOLSET. For instance, in the *ABS_CTRL* main machine (a micro-controller component), the *PERFORM_COMPUTATION* sub machine is not defined, which implies that the requirements need to specify in more details about how *"The ABS function shall decide appropriate braking force on each wheel"*. Next for *Coverage Checking*, since the observations are determined to be held according to the results of the TASM observers in the runtime, the satisfaction of requirements towards features is therefore reached. On the note about *Model Checking*, we first translate the TASM model into timed automata, and then check the deadlock property as well as the *ReqH3* requirement via the UPPAAL model checker. The corresponding results are: 1) *Deadlock free* is *satisfied* and, 2) the *ReqH3* is *satisfied*. Although the case study is a demonstrator, it is an illustrative example to show how to follow our proposed approach to perform requirements validation at various levels.

5 Related Work

In addition to the aforementioned related work, there are some other interesting pieces of work deserved to be mentioned as follows. Event-B [1] is a formal state-based modeling language that represents a system as a combination of states and state transitions. Iliasov [10] showed how to use Event-B for systems development, where the system constraints are formalized as a set of visualized proof obligations which can be synthesized as use cases. Such proof obligations are then reasoned about their satisfaction in the corresponding Event-B model. Mashkoor et al. [14] proposed a set of transformation heuristics to validate the Event-B specification by using animation.

Cardei et al. [6] presented a methodology that first converts SysML requirements models into a requirements model in OWL, and then performs the rule-based reasoning to detect omissions and inconsistency. Becker et al. [3] provided a formalization for self-adaptive systems and the corresponding requirements, which enables a semi-automatic analysis of performance requirements for self-adaptive systems. Cimatti et al. [7] introduced a series of techniques that have been developed for the formalization and validation of requirements for safety-critical systems. Specifically, the methodology consists of three main steps in terms of informal analysis, formalization, and formal validation. Scandurra et al. [17] proposed a framework to automatically transform use cases into ASM models, which are used to validate the requirements through scenario-based simulation. MARTE [15] is a UML profile for modeling and analysis of RTES, covering both functional and non-functional properties of the system. Nevertheless, to our best knowledge, there has not been any work about using MARTE for the purposes of requirements validation.

6 Conclusions and Future Work

In this paper, we have proposed a novel TASM-based approach to requirements validation. The approach 1) uses the extended TASM language to model the documented requirements and, 2) performs the requirements validation by using two tools in terms of the TASM Toolset and the model checker Uppaal. Our case study using a Brake-by-Wire (BbW) system developed by a major automotive company, has shown that our approach can achieve the goal of requirements validation via *Logical Consistency Checking*, *Auxiliary Machine Checking*, *Coverage Checking*, and *Model Checking*. Even if limited in complexity, the BbW system consists of a number of parts presenting the real world safety-critical systems, such as micro-controllers, sensors, actuators, and communication buses.

In this work, the validity of our TASM model towards requirements and features is built upon our thorough understanding of the BbW system, and hence TASM models are semantic preserving. Moreover, we have observed model validation issue as a common problem with model-based approaches. This is getting more complicated when the system's non-functional properties are considered. To address the situation, as future work, we will combine our proposed modeling approach with a set of assistant techniques, such as rule/pattern-based algorithm

to semi- or fully-automatically transform natural languages into TASM models. The future work also includes a wider industrial validation of our approach, and the improvement of our current TASM Toolset. Such improvement will not only facilitate our evaluation but also power up our analysis with statistical methods [12] and probabilistic modeling patterns.

References

1. J.-R. Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.
2. A. T. Bahill and S. J. Henderson. Requirements development, verification and validation exhibited in famous failures. *Syst. Eng*, 2005.
3. M. Becker, M. Luckey, and S. Becker. Performance analysis of self-adaptive systems for requirements validation at design-time. In *Proceedings of QoSA'13*, pages 43–52, New York, USA, 2013. ACM.
4. J. Blom, A. Hessel, B. Jonsson, and P. Pettersson. Specifying and generating test cases using observer automata. In *Proceedings of FATES'04*, pages 125–139. Springer-Verlag, 2005.
5. E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.
6. I. Cardei, M. Fonoage, and R. Shankar. Model based requirements specification and validation for component architectures. In *Systems Conference, 2008 2nd Annual IEEE*, pages 1–8, 2008.
7. A. Cimatti, M. Roveri, A. Susi, and S. Tonetta. From informal requirements to property-driven formal validation. In *Proceedings of FMICS'09*, pages 166–181. Springer-Verlag, Berlin, Heidelberg, 2009.
8. E. M. Clarke and J. M. Wing. Formal methods: state of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, Dec. 1996.
9. A. Ellis. Achieving safety in complex control systems. In *Proceedings of SCSC'95*, pages 1–14. Springer London, 1995.
10. A. Iliasov. Augmenting formal development with use case reasoning. In *Proceedings of Ada-Europe'12*, pages 133–146. Springer Berlin Heidelberg, 2012.
11. N. G. Leveson. *Safeware: System Safety and Computers*. ACM, NY, USA, 1995.
12. Y. Lu. *Pragmatic Approaches for Timing Analysis of Real-Time Embedded Systems*. PhD thesis, Mälardalen University, 2012.
13. MAENAD. `http://www.maenad.eu`, 2013.
14. A. Mashkoor, J.-P. Jacquot, and J. Souquières. Transformation Heuristics for Formal Requirements Validation by Animation. In *Proceedings of SafeCert'09*, York, United Kingdom, 2009.
15. OMG. `http://www.omgmarte.org/`, 2013.
16. M. Ouimet. *A formal framework for specification-based embedded real-time system engineering*. PhD thesis, Department of Aeronautics and Astronautics, MIT, 2008.
17. P. Scandurra, A. Arnoldi, T. Yue, and M. Dolci. Functional requirements validation by transforming use case models into abstract state machines. In *Proceedings of SAC'12*, pages 1063–1068, NY, USA, 2012. ACM.
18. Z. Yang, K. Hu, D. Ma, and L. Pi. Towards a formal semantics for the AADL behavior annex. In *Proceedings of DATE'09*, pages 1166–1171, 2009.
19. J. Zhou, A. Johnsen, and K. Lundqvist. Formal execution semantics for asynchronous constructs of aadl. In *Proceedings of ACES-MB'12*, pages 43–48, 2012.
20. D. Zowghi and V. Gervasi. The three cs of requirements: Consistency, completeness, and correctness. In *Proceedings of REFSQ'02*, 2002.