Fraunhofer
IESE

Assuring Functional Safety in Automotive Software Through Pattern-based Requirements Development

Authors:
Ganesh J. Pai
Andreas Roeser

IESE-Report No. 013.10/E
Version 1.0
April 2010

A publication by Fraunhofer IESE

Fraunhofer IESE is an institute of the Fraunhofer Gesellschaft.
The institute transfers innovative software development techniques, methods and tools into industrial practice, assists companies in building software competencies customized to their needs, and helps them to establish a competitive market position.

Fraunhofer IESE is directed by
Prof. Dr. Dieter Rombach (Executive Director)
Prof. Dr. Peter Liggesmeyer (Director)
Fraunhofer-Platz 1
67663 Kaiserslautern

Abstract

An emerging standard for functional safety in road vehicles, the ISO 26262, is expected to impose greater stringency on the practice of automotive systems and software engineering. In particular, a need exists for increased rigor during requirements development not only to be compliant with some key parts of the standard, but also to obtain early assurance of functional safety. In this paper, we present pattern-based requirements development, using patterns of timed automata and property specifications, as a feasible, rigorous and model-based method to address this need. As preliminary validation, we apply it to verify functional safety requirements for a simple but non-trivial real example of an embedded car-window controller.

Table of Contents

1 Introduction

The modern road vehicle contains an appreciably large number of embedded software-based systems: a premium car, for instance, contains around 100 electronic control units with approximately 1GB of software [1]. An increasing amount of the functionality controlled by this software is safety-related or safety-critical e.g., adaptive cruise control or electronic stability control. It is vital to assure, early during development, that such software will function as required; early identification of errors provides the greatest opportunity for risk mitigation [2].
The ISO 26262 [3] is an emerging standard for functional safety in road vehicles, complying with which is intended to provide the required assurance at the systems level. When published, it is expected to be applied to all new road-vehicle development; it is also expected to impose more stringent constraints on the practice of automotive systems and software engineering. The draft international standard suggests, for instance:

- 26262-4.6: Demonstration via analysis that *technical safety requirements* are consistent with *functional safety requirements*.[1]

- 26262-8.9: Usage of formal verification (or an appropriate combination of formal and informal verification) for software functionality that is assigned a certain automotive safety integrity level (ASIL)[2] to show that the goal is actually achieved.

To demonstrate compliance to such constraints, some modifications are foreseen to how requirements development (especially specification and associated quality assurance) will be instantiated in practice. Indeed, the standard itself implies a need for greater rigor during requirements development, especially for automotive systems software.
In practice, requirements for automotive systems are largely first specified in natural language; then within the paradigm of model-based development (MBD), they are re-specified using 'semi-formal' (graphical) models e.g., SysML. The rationale is that such models can be not only more easily validated with customers, but also informally verified e.g., via inspection, and subsequently used for automatic code generation.

[1] Summarily, technical safety requirements refer to how a service is provided; functional safety requirements relate to what services a system architecture provides.

[2] An ASIL is assigned based on a hazard analysis; it represents a classification of the technical risk reduction measures used to achieve acceptable residual risk.

In this paper, we present pattern-based requirements development using timed automata (TA) patterns and property specification patterns as a feasible and rigorous alternative that is both compatible with MBD and compliant with the guidelines from the ISO 26262 standard.
Although the usage of patterns in software engineering is not new, to the best of our knowledge, our work is the first to apply TA patterns and property specification patterns together to provide early assurance of (1) functional safety for automotive software and (2) compliance with key requirements from the corresponding standard.
The basic approach for pattern-based requirements development is given in section II. Then, in section III, we provide an overview of the formalisms and patterns used. As a preliminary validation of our approach, in section IV, we apply it to specify and analyze functional safety for a simple but real example system: an embedded controller regulating the movement of a car window. Section V discusses the individual specifications and findings resulting from applying pattern-based requirements development to the evaluation example. In section VI, we discuss our contribution with respect to compatibility with industrial practice and related work in the literature. Section VII concludes the paper.

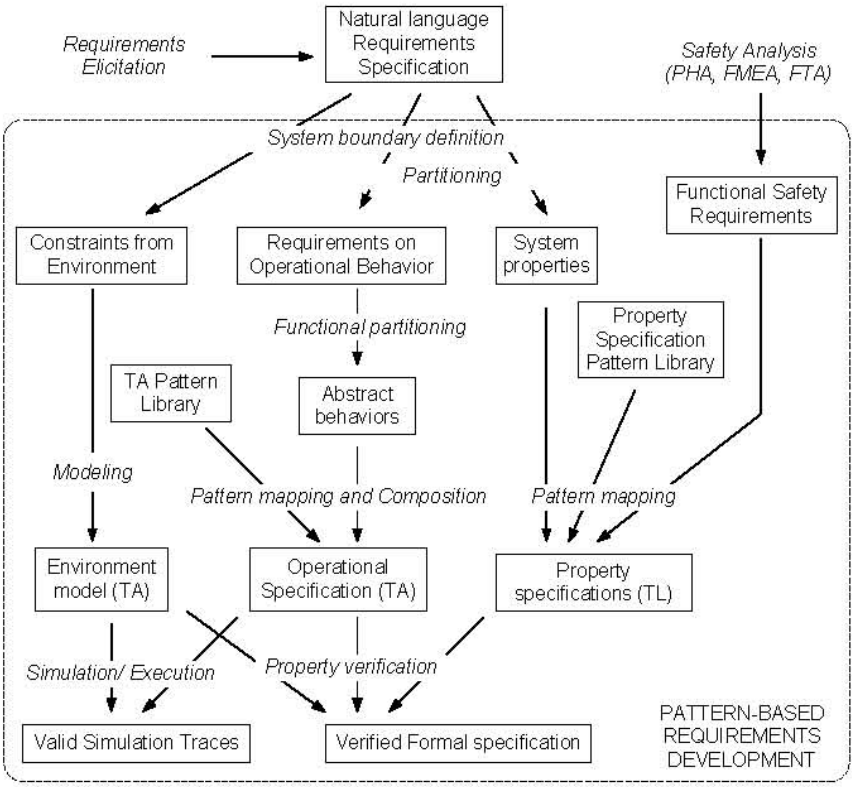

Figure 1: Pattern-based requirements development

2 Pattern-based Requirements Development

We iteratively apply pattern-based requirements development (Fig. 1) on natural language (NL) requirements specifications; starting with system boundary definition, we identify monitored and controlled variables at the machine-environment boundary.
In practice, the NL specifications (assumed to be available from traditional elicitation techniques) have been observed to contain a mixture of constraints from the environment, descriptions of, and/or requirements on operational behavior, and properties of the system under development. Based on domain knowledge, the specifications undergo partitioning along these dimension.
The safety engineering process recommended by the ISO 26262 occurs in parallel with the systems engineering process: the outcome of hazard analysis and risk assessment (using techniques such as preliminary hazard analysis (PHA), failure modes and effects analysis (FMEA), fault tree analysis (FTA), etc.) yield functional safety requirements which are also considered as part of the requirements pool.
Subsequently, the requirements on operational behavior undergo functional partitioning, where abstract behaviors are identified or defined. In principle, this is a lightweight and restricted application of functional analysis [4], where the defined abstract behaviors are the functions that will achieve the operational requirements. At this stage, rather than undertaking a complete functional analysis and allocation e.g., as in [5], it suffices to create a functional breakdown structure (FBS) and allocate, say, preliminary timing requirements to the identified functions.
Thereafter, we formally specify each abstract behavior by picking an appropriate TA pattern or a composition of TA patterns from a pattern library. Presently, we pick patterns based on the correspondence between the semantics of a pattern and the behavior to be specified. We envision a semiautomatic approach to guide the choice of the patterns using domain models[3]. The formal specification of the overall operational behavior is created by composing the TA patterns specifying the constituent abstract behaviors.
In parallel, we use the library of property specification patterns to formally specify both the identified system properties and the functional safety requirements, using an appropriate temporal logic (TL). In our case, we used a subset of timed computational tree logic (TCTL) [6]. The constraints imposed by the environment on the system are also specified using TA.
The result is a formal specification of (1) the required system behavior and the environment modeled as a network of communicating TA, and (2) the system properties to be fulfilled as TL statements. We check the validity of the opera-

[3] An avenue of future work, very briefly discussed in section VI.

tional specification and the model of the environment, in part, by simulation or execution of the TA model.
We achieve a clean partition between technical safety requirements and functional safety requirements by noting that during the early phases, a technical safety requirement i.e., how a service is provided, is contained in part by the requirements on expected operational behavior; whereas a functional safety requirement i.e., what service is to be provided, is a property to be exhibited by the system.
Thus, providing assurance that the technical safety requirements are consistent with the functional safety requirements amounts to demonstrating that the operational specification satisfies the appropriate system properties e.g., using a formal verification technique such as model-checking. As a consequence, both the constraints imposed by the standard i.e., on the product and the process, are met.

3 Specification Mechanisms

3.1 Networks of Timed Automata

In this section, we provide a brief overview of networks of communicating TA. We chose TA for formal specification, primarily since their semantics and extensions permit specifying real-time behavior; additionally, it affords a common formalism to model the operational behavior of the system, and its environment.

TA are finite state machines (FSM) equipped with real-valued clock variables [7]. Graphically, they are represented as directed graphs with edges and locations. The clock variables *C* permit the specification and measurement of elapsed time between events. Timing constraints such as propagation delays, execution and response times are specified as predicates on the values of *C*. Formally, we have [8]:

Definition 3.1: A timed automaton *T* is a tuple (L, l_0, C, A, E, I), where L is a finite set of locations and $l_0 \in L$ is the initial location; C is a finite set of clocks; A is an alphabet of actions; $I : L \to B(C)$ assigns invariants to locations; and $E \subseteq LxB(C)xAx2^C xL$ is the finite set of edges: $e = \langle l, g, a, r, l' \rangle \in E$. Another notation for e is $l \xrightarrow{g,a,r} l'$.

Here, l and l' are locations, g is the set of clock constraints guarding e, a is the action of e, r is the set of clocks that is reset by e; *B*(C) is the set of clock constraints over *C*, i.e., the set of Boolean combinations of atomic constraints of the form c ⊗ *x*, where c ∈ C $x \in$ x $\mathbb{N}$ and ⊗ ∈ {<, ≤, =, ≥,>}.

A labeled transition system (LTS) defines the semantics of TA, where a state comprises a current location and a clock valuation, while a transition between states is either a delay or an action. Formally, we have [8]:

Definition 3.2: A clock valuation function u : $C \to \mathbb{R}_{\geq 0}$ is a function from the set of clocks C to non-negative real numbers and $u_0 = 0, \forall x \in C$.

Definition 3.3: For a TA T , a state (or configuration) of T is a pair $(l, u) \in Lx\ \mathbb{R}^C$ where u is the clock valuation function and l is the current location. The semantics of T is defined as the LTS $(S, s_0, \to)$, where $S \subseteq Lx\ \mathbb{R}^C$ is the set of states and $\mathbb{R}^C$ is the set of all clock valuations; $s_o = (l_0, u_0)$ is the initial state; the transition relation is $\to \subseteq Sx\{(R)_{\geq 0} \cup A\}xS$.

The transition relation → is composed of (1) Action tran sitions:

$(l, u) \xrightarrow{a} (l', u')$ iff $\exists e = \langle l, g, a, r, l' \rangle \in E$ such that

$u \in g, u' = [r \to 0]u \wedge u' \in I(l')$; and (2) Delay transitions: $(l,u) \xrightarrow{d} (l,u+d)$ iff $\forall d' : 0 \le d' \le d \Rightarrow u + d' \in I(l)$. For $d \in (R)_{\ge 0}, u + d$ maps each clock $x \in C$ to the value $u(c) + d$. The clock valuation that maps each clock in r to 0 and agrees with u over $C \setminus r$ is $[r \to 0]u$.

The above definitions can be used to define guards and invariants as sets of clock valuations with $u \in I(l)$. When a delay transition is taken, all clocks are increased by the same delay value, the system is delayed in the current location, and location invariants during a delay must not be violated. Alternatively, an action transition follows an outgoing enabled edge only if the current clock valuation satisfies the edge guard.

Parallel composition of n TA, creates a network of communicating TA $\mathbf{T} = \{T_i\} = \{(L_i, l_i^o, C, A, E_i; I_i)\}$ where $(1 \le i \le n)$, having a common set of actions and clocks. Such networks, augmented with communication channels and global variables permit the modeling of concurrent and/ or cooperative behaviors.

The TA can transmit on channel $a(a!)$, or receive on channel $a(a?)$. If A is the set of channel names, the set of actions $A = \{a? \mid \in A\} \cup \{a! \mid a \in A\} \cup \{\tau\}$ represents synchronization actions performed via channels for inter-process communication, and internal synchronization independent actions (τ) respectively. In addition, a set $U \in A$ represents urgent channels, which model synchronization without delay. Edges containing a synchronization action on an urgent channel therefore do not contain clock guards.

3.2 Timed Automata Patterns

Recurring types of operational behavior e.g., behaviors triggered by external events, timeouts, etc., are applicable to real-time embedded systems. We use TA patterns [9] to abstract and specify such generalized and recurring high-level behavior in operational requirements specifications.

Graphically, a TA pattern is shown as a triangle with a circle attached at one of the vertices (see Fig. 3). The triangle is an abstract representation of the automaton while the circle represents its initial states. The edge of the triangle opposite the vertex with the attached circle, represents the set of final states. Depending on the pattern to be used, one or more locations (shown as circles), corresponding transitions to these locations (shown as directed arcs) also appear in the graphical representation, and they may be annotated with the relevant transition guards. In [9], a set of 13 TA patterns are formally defined. Here, we briefly summarize those patterns which we applied to the evaluation example:

1) *Event prefixing*: Specifies the precedence of an event before an action in the TA.

2) *External choice*: Specifies the resolution of a choice by the first action of the available alternative automata.

3) *Deadline*: Specifies the behavior that the automaton execution completes at, or before, a specified time.

4) *Delay*: Specifies that the execution of the automaton is delayed by exactly t time units.

5) *Recursion*: Recursive invocation of the automaton in one of its states, used to specify non-terminating reactive systems.

6) *Timed interrupt*: Preemptive interruption of behavior in an automaton by any other automaton after a time lapse.

7) *Event interrupt*: Similar to the Timed interrupt pattern, where automaton execution is interrupted by an event.

8) *Wait-until*: Parallel composition of a given automaton and the Delay pattern, constraining the automaton behavior by forcing its execution to finish no earlier than a defined time.

9) *Time out*: Composition of the External choice and Delay patterns, modeling the behavior of a time lapse and an alternative execution.

10) *Periodic repeat*: Composition of the Deadline, Wait-until and Recursion patterns, modeling the repetitive behavior of an automaton which must terminate before time t.

3.3 Property Specification Patterns

Property specification patterns are a generalized description of the permissible state or event sequences in a system modeled as an FSM [10]. They provide a systematic approach to classify, structure and formalize requirements describing system properties.
In use, they are effectively *templates* with which recurring requirements types can be formally stated in a declarative form using an appropriate TL. Property specification patterns are useful not only because of their formal foundations, but also because of the structured grammar that accompanies them; consequently, they can be applied for precisely (re)stating natural language requirements.
We found the patterns developed in [10] and [11] to be the most applicable for

the evaluation example, especially in formalizing those system properties containing timing constraints. As an example, the globally scoped real-time bounded invariance specification pattern is given in TCTL as $\mathbf{A}\Box(P \Rightarrow \mathbf{A}\Box_{\leq T} Q)$. Here, if P and Q are propositions, and T is a timing deadline, then this pattern specifies that whenever P holds, Q also holds for at least T time units. A comprehensive list of property-specification patterns including patterns for real-time constraints and probabilistic constraints is available in [10], [11] and [12].

4 Evaluation Example and Approach

4.1 Car Window Movement Controller

Now, we present the example system used to evaluate our method: a simple, real, but non-trivial example of a car-window movement controller (Fig. 2). As shown in the figure, the window-movement controller represents software functionality embedded in a door-control device (TSG)[4]. The TSG also contains other embedded functionality such as lighting control, seat adjustment, etc. The TSG interfaces with the external environment via three connectors, the first of which connects it to all the sensors and actuators within the door i.e., (1) a motor that physically moves the window, and (2) a set of sensors that indicate window movement and whether the window is completely open (closed). All relevant sensors and actuators that are not located inside the door are attached to the TSG via the second connector. The third connector interfaces the TSG to the controller area network (CAN) bus and the power supply.
We mainly considered the controller that regulates the driver-side window movement (TSG_VL). Consequently, only a subset of the inputs and outputs from the original requirements document were necessary for specification, and the most relevant connector for our purposes was the first.

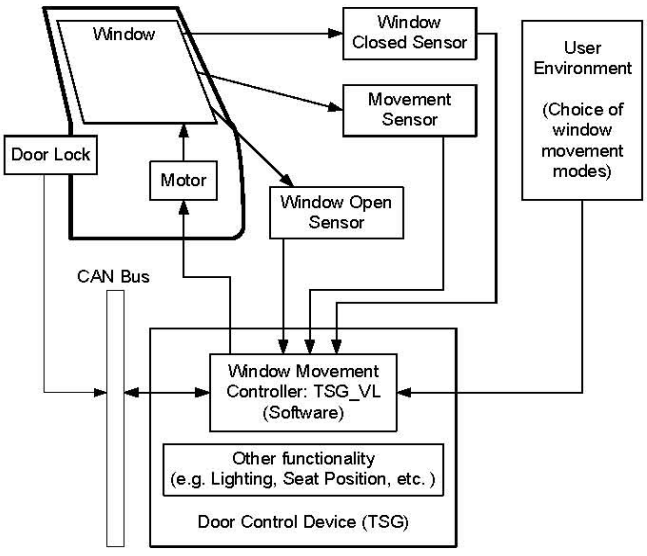

Figure 2: Car-window movement controller system

[4] TSG abbreviates Türsteuergerät: door-control device in German

Using the window movement buttons on the door, the driver controls window movement by selecting a movement direction (*Up, Down*) and a movement mode (*Manual, Automatic*). In the *Automatic* movement mode, the movement process ends only when the window is completely open (closed). Additionally, the window can be controlled by messages sent over the CAN bus e.g., from other electronic systems in the car such as an automatic door-locking mechanism.
Based on the sensor input and the chosen window movement mode, TSG_VL is required to react within a specified deadline by sending the appropriate control signal to the window motor, which then moves the window to the desired position. Window movement persists as long as (1) the appropriate stimulus exists, (2) the window is neither fully opened nor fully closed, and (3) an error has not occurred. When errors are detected, the controller initiates error-handler routines.
A special error-handling routine (which imposes a safety requirement on the system) is *jam protection* i.e., when an obstacle is detected during upward window movement, the movement is immediately stopped and the window is retracted automatically until it is completely open.

4.2 Approach

The original requirements [13] for this system were documented in natural language (NL) and contained a mixture of required operational behavior and system properties. They were constructed, in part, by an experienced systems engineer from a large auto manufacturer. Therefore, the writing style and complexity of the requirements was considered to be representative of requirements that systems engineers in the domain would address in practice.
Applying our method (as described in section II), we systematically constructed the formal requirements specification of the car-window movement controller, comprising:

1) The TA model specifying the requirements on the controller operational behavior (TSG_VL).

2) The environment model comprising 3 TA i.e. for the window sensors (win_sensor), the window motor (win_motor) and the user-environment (env_user). Since the only communication over the CAN bus relevant for the controller was a signal from the door locking mechanism we decided not to model the CAN bus in its entirety; rather we modeled only the signal of interest as part of the stimuli from the user environment.

3) The system properties formalized using property specification patterns and TCTL.

We used the UPPAAL tool [14] for specification since it offered an intuitive GUI for modeling, together with tightly integrated simulation and verification capabilities. The choice of the tool, however, necessitated the use of a *subset* of TCTL as the temporal logic for specification. Since it mainly allows checking reachability properties, temporal logic operators cannot be nested, resulting in a restriction on the variety of properties that can be specified (and checked). In general, the tool allows the use of (1) path formulae for universality (A) and existence (E), and (2) state formulae for global (□) and eventual (◊) states. In addition to these, the usual Boolean operators (*AND*, *OR*, *NOT*) and the special operators *leads to* (-->), and *imply* (⇒)) are available.
To validate that the correct operational behavior was specified, we used the simulator available in the tool. The simulator executes the specification either in interaction with the user, or by randomly picking execution paths; then it animates the execution by generating sequence charts which show the interactions among the communicating TA, and generates execution traces that can be inspected. Subsequently, we verified whether the operational specification satisfies the system properties by using the model-checker available in the tool.

5 Results and Findings

This section documents the outcome of applying patternbased requirements development to the evaluation example: we present the TA specification of the controller operational behavior in detail, and only briefly describe the environment model[5]. Then we present the specification of the functional safety requirements, followed by our findings.

5.1 Controller Specification

By functional partitioning of the controller operational behavior requirements, we first identified constituent abstract behaviors: *controller active and waiting for input*, *window movement* (upward/ downward), *mode choice*, *mode switching*, *sensor polling*, *error handling* and *controller reset*. Then, based on TA pattern semantics, the abstract behaviors were (manually) mapped to the following patterns: *Event prefixing*, *External choice*, *Deadline*, *Periodic repeat*, *Event interrupt*, and *Timed interrupt*.
To model the operational specification of TSG_VL as a TA pattern (Fig. 3), we compose the TA patterns of the individual abstract behaviors. The flat TA model encapsulated by the composed TA pattern model is shown in Fig. 4. In creating this flat model we also define the appropriate internal locations and transitions, channel names, state variables, global variables, clocks and guards.

As shown in Fig. 4, initially the controller is active and waiting for input. Recall that TSG VL is connected to the sensors, the window motor and the movement input via the first connector. Hence all channels or global variables of the form connector1SignalX model a corresponding port inthe physical connector over which the respective signal is sent.

[5] A more comprehensive description of the environment model is available in [15].

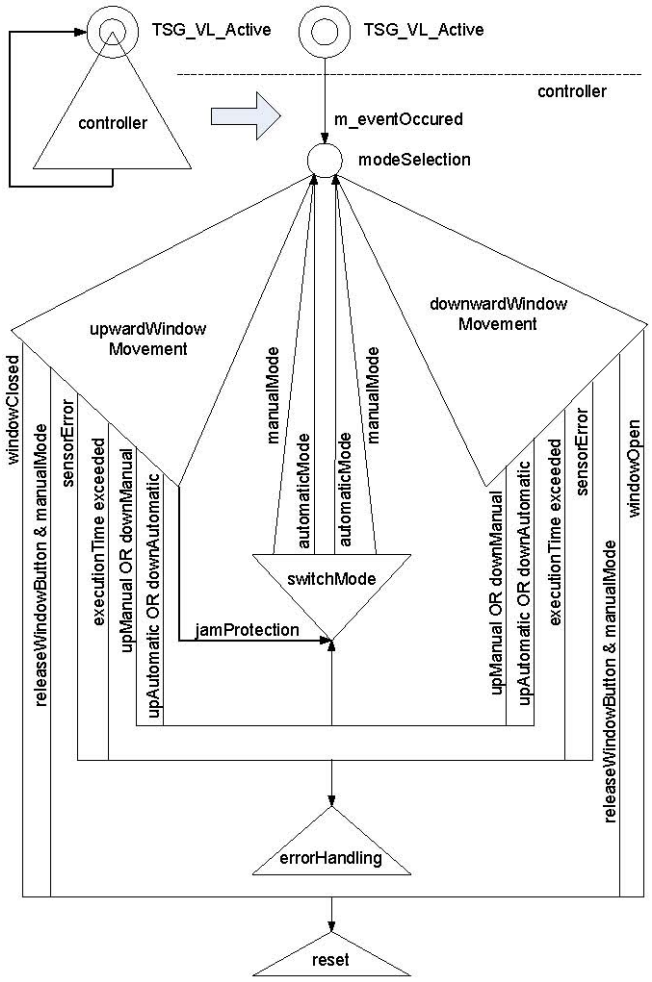

Figure 3: Timed automata pattern model of TSG_VL

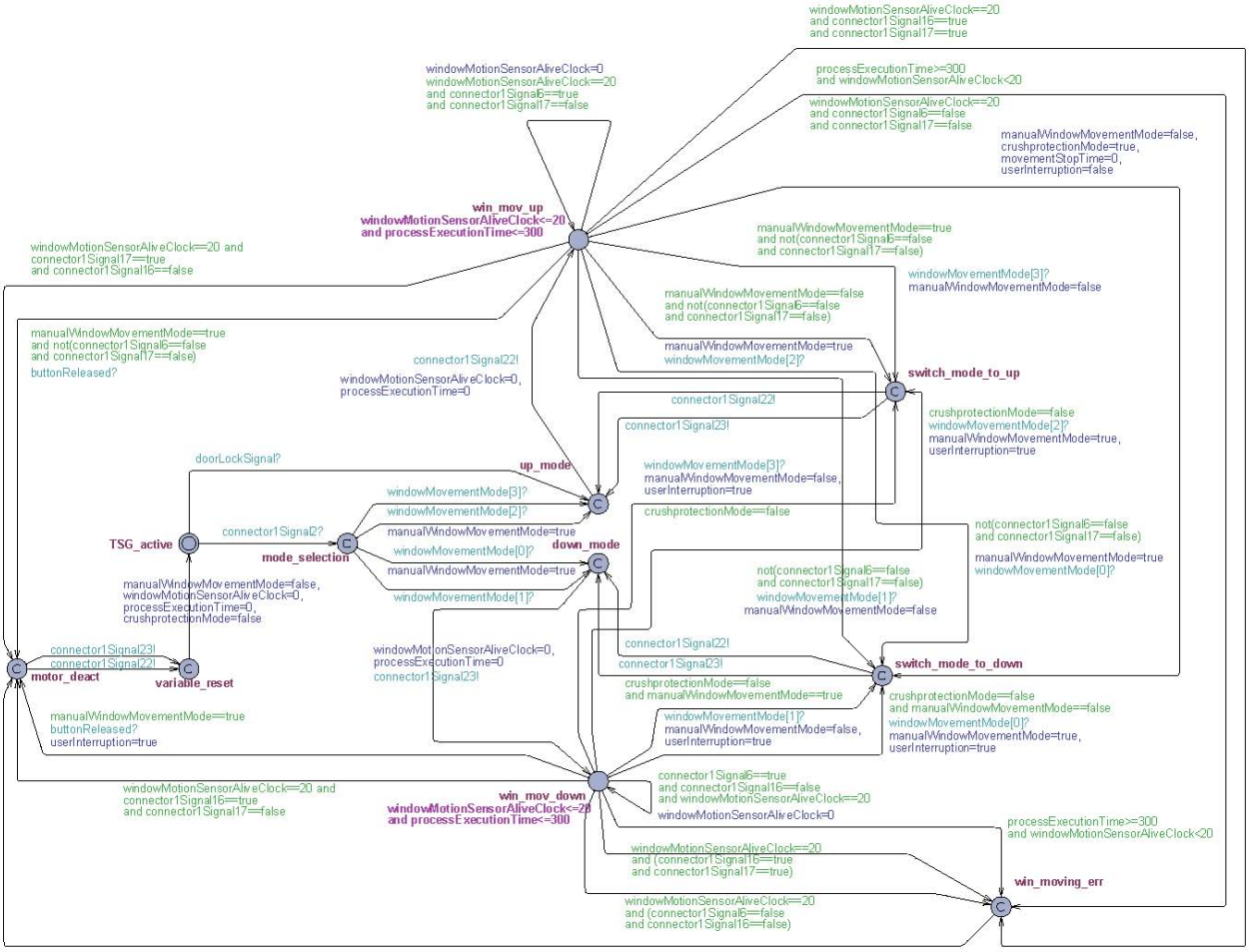

Figure 4: Timed automata model of TSG_VL

When an input mode is selected by the driver, the env_user TA) automaton synchronizes with TSG_VL via the channel connector1Signal2. This is modeled as an urgent channel i.e., no time elapses and the user input is communicated to the controller as and when it occurs. Based on the selected movement mode (Manual, Automatic) and direction (Up, Down) the TSG_VL automaton moves the window upwards (the location win_mov_up) or downwards(the location win_mov_down).

The clocks processExecutionTime and windowMotion- SensorAliveClock are used to specify timing deadlines on the movement process, and the periodic polling duration for the sensors respectively. As per the requirements, the former was 3s (300 time units), and the latter was 0.2s (20 time units).

Mode switching initiated from the user-environment is modeled by the locations switch_mode_to_up and switch_mode_to_down. When these locations are reached, the clocks for window-movement process execution and sensor polling are reset, the window-movement in the current direction is halted and

restarted in the chosen movement direction. The location motor_deact models motor deactivation, with which window movement is stopped. Violation of the timing requirements results in an error, causing the automaton to transition to the win mov error location. Additionally, this error state is reached for combinations of sensor values that indicate illegal window positions (clarified in section V-B1). Thus, at the level of the operational specification, the error state is an abstraction encapsulating different error types e.g., those resulting from timing constraint violations, faults in the sensors, and faults occurring during window movement.
A special safety requirement applicable during upward window movement is that of *jam protection*. When the TSG_VL detects from the sensors, after an upward window movement process has been initiated, that the window is neither moving nor completely closed, then a local variable crushProtectionMode is set to TRUE, and the window is forced to automatically open completely. This specifies, to an extent, the technical safety requirement for the window movement controller.

5.2 Environment Specification

1) *Window Sensor Model*: The win_sensor TA models the changes occurring in the window position, rather than the individual sensors and their state changes. A need to manage the complexity of the overall system model motivated this abstraction, and it allowed us to realize one sensor model rather than three separate ones.
To arrive at this model, we enumerated the different combinations of sensor states, and reduced it to legal or illegal values of a *window_position* state variable. For example, the combination of sensor values given by $\{$ *not_moving, opened, not_closed* $\}$ is equivalent to the window position $\{$ *opened* $\}$; whereas the combination $\{$ *moving, opened, closed* $\}$ is assigned to the window position $\{$ illegal $\}$. In fact, such illegal combinations of sensor values indicate faults either in the window, the motor or in the sensors, and it must trigger an appropriate error handling routine from TSG_VL.
During this process of defining the variable values for *window_position*, we uncovered missing requirements: e.g., the responses of the window controller was undefined for illegal combinations of sensor values.

2) *Window Motor Model*: The win_motor automaton models the motor states and its corresponding state changes. State changes occur after TSG_VL communicates the appropriate (starting or stopping) signal, over the communication channels (between TSG_VL and win_motor). We assumed no motor starting and stopping latency; this information was not available in the original requirements document either.

3) *User Environment Model*: The env_user TA models the choice of window movement modes and its execution is asynchronous to that of TSG_VL. A special *user input observer* automaton was also created to implement the *urgent edge* pattern [16]. This pattern allows such asynchronous input to be processed as quickly as possible, without blocking the env_user TA.
We introduced some additional timing constraints in the env_user model, rather than in TSG_VL model, to deal with the following scenarios:

i) *Physical constraints on the window-movement mode choice*: E.g., the requirements indicated that when the automatic window-movement mode is selected, the stimulus persists for a maximum of 0.5 seconds due to device physics. Thus, even if the user were to initiate a mode switch from the automatic mode to the manual mode instantaneously after selecting the automatic mode, the corresponding signal would not be recognized for up to 0.5 seconds.

ii) *The window-movement mode choices are abused*: The requirements for this case were ambiguous or missing from the NL requirements. Therefore, we introduced timing constraints that enforce a minimum duration between the choice of window-mode changes, preventing the driver from instantaneously and repetitively switching between different window-movement modes.

5.3 Specification of System Properties

Here, we mainly discuss the formalization of the *jam protection* safety requirement[6]. In the NL requirements document, this requirement was given as:

"Jam protection: If a jammed object is recognized, the current upward movement of the window is stopped immediately (within 20 ms), and the window is moved downward into the down position."

We formalized this statement using TCTL and real-time property specification patterns [11] as the following 4 properties:

1) **SR1: It is never the case that sensor polling time is greater than 20ms during window movement.** This is formalized using the safety pattern, with TCTL as:

$$A\Box\ \mathit{NOT}((\text{TSG\_VL.win\_mov\_down}\ \mathit{OR}\ \text{TSG\_VL.win\_mov\_up})\ \mathit{AND}\ \text{TSG\_VL.windowMotionSensorAliveClock} > 20)$$

[6] Ref. [15] provides a comprehensive set of system properties that werespecified.

2) **SR2: Whenever a jammed object is recognized the controller switches to the jam protection mode**. In specifying this property, we used the globally scoped bounded response real-time pattern. UPPAAL, however, does not permit nesting of the temporal logic operators, instead providing a special *leads to* (-->) operator whose semantics imply bounded response. For this particular requirement, the initialization of jam-protection had no deadline, resulting in an un-timed instantiation of the pattern. We specify this property in TCTL as shown:

((TSG_VL.windowMotionSensorAliveClock = 20) *AND*
(TSG_VL.win_mov_up) *AND*
(Win_Sensor.sensorclock < 19) *AND*
(connector1Signal6 = FALSE) *AND*
(connector1Signal17 = FALSE))
--> ((TSG_VL.switch_mode_to_down) *AND*
(TSG_VL.crushprotectionMode = TRUE))

3) **SR3: The jam protection mode leads to the upward window movement stopping within 20ms.**
In contrast to SR2, this property requires a response within a specified time bound, and also uses the real-time bounded response pattern. To specify this property, we introduced a local clock movementStopTime, which only measures the time until upward movement is stopped, without affecting the functional or timing behavior of the system. As mentioned earlier, we did not model the timing latency arising from the physics of the motor. The formal specification of the property is given as:

((TSG_VL.switch mode to down) *AND*
(TSG_VL.movementStopTime = 0) *AND*
(TSG_VL.crushprotectionMode = TRUE) *AND*
(Win_Motor.motor moving up direction)) -->
((Win Motor.motor is off) *AND*
(TSG_VL.movementStopTime < 20) *AND*
(TSG_VL.manualWindowMovementMode = FALSE))

4) **SR4: It is never the case that downward window movement is interrupted until the window is completely open.** This property is used to verify that the user cannot abort the downward movement process once started (during jam-protection). Once again, we introduce a special local variable userInterruption whose initial value is FALSE, and does not affect the functional or timing behavior. The variable is set to TRUE, if the user chooses (via the env_user automaton), a mode that would normally interrupt downward movement e.g., *Close Window Manually*. To state SR4, we

specify that it is never the case that this variable is TRUE during jam-protection. That is,

A□ *NOT*((TSG_VL.crushprotectionMode = TRUE)
AND (TSG_VL.userInterruption = TRUE))

Further, we state that when the jam-protection mode is enabled, then only the error state or the motor deactivated states are reachable. This is intended to prevent mode switches between different movement modes when the hazard occurs and jam-protection is initiated. Thus:

((TSG_VL.win mov down) *AND*
(TSG_VL.crushprotectionMode = TRUE))
--> ((TSG_VL.motor deact) *OR*
(TSG_VL.win_moving_err)

These TCTL properties comprise the functional safety requirement of the window-control system i.e., they specify, to an extent, what services the system should provide.

5.4 Findings

The original requirements document for the evaluation example contained 6 pages with approximately 60 distinct statements in NL. Of these, 13 were system properties including the functional safety requirement, all of which we formalized using property specification patterns. The remainder contained a mixture of operational behavior requirements and statements about the system boundary. Although the NL requirements were reasonably detailed and well documented, the formalization process helped to identify a total of 29 defects. These were classified as:

- *17 undocumented assumptions*: 10 of these assumptions were made through the course of formal modeling, while the remaining 7 were discovered during the formalization procedure, and needed explicit documentation.

- *8 missing requirements*: All 8 were discovered during the process of building the pattern-based TA model of the system; 2 of these concerned timing criteria; 1 dealt with the problem of determining process priorities. The remaining 5 were found during the definition of the system boundary, even before formalization of the system requirements had begun. These 5 defects pertained to unspecified behavior for illegal combinations of sensor values.

- *3 incorrect requirements*: All three were cases of non-determinism, which were detected through model-checking. Although we also simulated the system to validate its behavior, these cases reflected subtleties which may not have been found via inspection or simulation, unless the system were simulated enough times to exercise the defects.

- *1 ambiguous requirement*: This dealt with an unclear delineation of the system boundary.

Although we were unable to validate with the original developers of the requirements, the assumptions made to correct the defects, the UPPAAL model-checker successfully verified that the operational behavior specification satisfies the safety requirement.
Verification times for the properties derived from the toplevel requirements ranged from a few seconds to two hours, using an Intel dual-core 2GHz processor with 2GB of RAM. Table 1 shows the resources used in verifying the safety property of the window-movement controller system.

Table I
RESOURCES USED FOR SAFETY PROPERTY VERIFICATION

Property	Verification time (s)	Memory (KB)
SR1	1.172	130644
SR2	1155.39	164980
SR3	354.687	94904
SR4	7206.281	61420

6 Discussion

6.1 Suitability for Industrial Usage

Based on the findings presented in section V-D, we assert the preliminary validity of our approach: in particular, pattern-based requirements development provides a means for obtaining early assurance that technical safety requirements are consistent with functional safety requirements, thus complying with some key constraints imposed by the ISO 26262 standard.

Although we evaluated our approach on a real system from the domain, we claim with restriction the feasibility of our approach in industrial practice: we did not attempt a detailed quantitative measurement of costs and effort involved, nor did we perform a full-fledged assessment of feasibility for use in an actual industrial development process. Such an assessment would be necessary to gauge the scalability of applying our method e.g., in terms of state-space explosion, when developing larger, more complex automotive systems.

Nevertheless, our approach is compatible with existing practices[7] e.g., after functional analysis, requirements are modeled using MATLAB Simulink/ Stateflow models which are validated by simulation. Formality may be then introduced in this process by translating the models to formal specifications, and applying model checking to verify system properties [17].

In principle, our approach deviates from this practice by systematically creating a model-based formal specification earlier in the requirements development process i.e., immediately after functional analysis. Iterative formal specification at this stage assists in clarifying ambiguous requirements and identifying missing, derived and/ or incorrect requirements. Indeed, formal requirements modeling has been empirically demonstrated to be valuable not only for identifying design errors early [18] but also for improving the quality of natural language requirements themselves [19].

Using TA patterns to specify operational behavior not only assists in hierarchically structuring and composing requirements specifications, but also supports downstream engineering activities: (1) During architecture design, TA patterns are directly available as formal specifications of component behavior since they are mapped to abstract behaviors after functional partitioning; (2) Since there is early assurance (during requirements development) of consistency between the technical and functional safety requirements, to assure that an architecture and its components meet the functional safety requirements, it suffices to (1) verify that the components realizing an abstract behavior meet the TA pattern specification of that behavior and (2) verify that the composed architecture meets the

[7] Based on a requirements development process description provided by a large international automotive systems supplier.

corresponding composed TA specification of the overall operational behavior.
In this paper, we have not reported on how a requirements engineer may pick an appropriate TA and/or property specification pattern from the corresponding pattern library. This is a promising avenue of future work, where we anticipate connecting the pattern libraries to interconnected domain models e.g., ontologies of domain specific concepts, standards and systems engineering concepts. The idea is to guide the choice of the appropriate patterns by applying inference on such domain models [20] and by using structured natural language having underlying formal semantics.
We performed neither hazard analysis to identify additional safety requirements nor ASIL assignment: this process is out of the scope of the work presented here. In practice, a parallel safety engineering process involves these activities, the results of which, in conjunction with the ISO 26262 standard, will dictate the process measures to be employed for risk mitigation e.g., whether informal verification would suffice instead of formal verification.
Although the standard does not strictly mandate a formally-based approach, we advocate it: the idea is that when a strong safety claim requires justification e.g., Safe automobile behavior in the presence of uncovered failures, an approach such as ours guarantees that a sub-claim in the argument chain holds e.g., verified compliance to the functional safety requirement.
Nonetheless, some barriers exist to adopting formally techniques in practice; namely, (1) a lack of well-integrated tool chains that support the introduction of our approach into the critical path of existing industrial processes (2) limitations from the existing tools in terms of the flexibility of expressing and verifying properties (3) the relatively high learning curve in acquiring familiarity with TA and temporal logic, and consequently (4) the perception of reduced comprehensibility and accessibility of formal methods. Addressing these barriers are potential avenues for future work.

6.2 Related Work

Our usage of specification patterns and timed automata is, by itself, not new: requirements patterns [21], object analysis patterns [22], timed automata [23], timed automata patterns [9], design patterns [24], and variations on property specification patterns [10] such as those supporting realtime properties [11], probabilistic properties [12] or safety specific properties [25], have all been used individually to model a variety of real-time embedded systems, including automotive systems.
However, our usage of TA patterns together with property specification patterns in a systematic way to specify requirements, such that there is (1) early assurance of compliance with key constraints imposed by the automotive safety standard ISO 26262, and (2) validation of compatibility with,and feasibility for use during, requirements development inthe automotive systems domain is new.
The work in [22] is the closest counterpart to ours, where requirements are

formalized using object analysis patterns: here, after functional analysis, structural patterns (using UML class diagrams) specify the functional architecture, and behavioral patterns (using timing extended UML statecharts) specify the behavior that the functional architecture should exhibit. Then, the UML models are converted to a suitable formal representation, after which they undergo modelchecking against system properties of interest (which are themselves formally stated using real-time property specification patterns).
Our approach is similar to the work in [22], in adopting a pattern-based approach to requirements development, and identifying functional partitioning as the point at which patterns can be applied for specification. Furthermore, our approach is compatible with theirs since the formal model for behavioral specifications in both are effectively time augmented FSM.
The use of UML class diagrams to specify the functional architecture, and statechart diagrams for behavioral specification, as in [22], is analogous to industrial practice, where MATLAB Simulink and Stateflow models respectively, are used instead. Both are informal in their semantics, and both have to be converted to a formal representation for automatic verification. Our approach differs firstly in providing an earlier stage of formal modeling. Secondly, in [22], the environment is modeled as equivalence classes of the input space, whereas we explicitly model the environment as a TA. Although this is restrictive when modeling a continuous environment and also increases the state-space to be explored during verification, it provides an intuitive way to specify synchronous or cooperative behaviors, characteristic of real-time automotive embedded systems.

7 Conclusion

The main contribution of this paper is a rigorous approach, sing pattern-based requirements development, to provide arly assurance of (1) functional safety in automotive software nd (2) compliance to key constraints imposed the ISO 6262 standard.
To validate our approach, we applied it to a simple but non-trivial and real evaluation example: a car-window movement controller. In particular, we formally specified the requirements using timed automata patterns and property specification patterns; then using a combination of simulation, execution and formal verification (using modelchecking), we verified that the expected operational behavior of the evaluation example (containing the technical safety requirement) is consistent with its functional safety requirements. Through the process of constructing the specification and subsequently via verification, we uncovered an appreciable number of requirements defects, some of which were subtle and may not have been detectable otherwise.
We believe that the industrial acceptance of an approach such as ours is hinged mainly on how automated the implied process steps are, and less significantly on how comprehensible formally specified requirements are, in comparison with natural language requirements.
To the best of our knowledge, our approach is the first to show the feasibility of using timed automata patterns together with property specification patterns in the context of an upcoming safety standard (ISO 26262) that sets stronger constraints on the practice of automotive systems and software engineering.

References

[1] C. Ebert and C. Jones, "Embedded Software: Facts, Figures and Future," IEEE Computer, vol. 42, no. 4, pp. 42–52, April 2009.

[2] B. Boehm, Soft. Eng. Economics. Prentice Hall, 1981.

[3] ISO/DIS 26262 Road Vehicles - Functional Safety, International Organization for Standardization, 2009.

[4] I. C. on Systems Engineering (INCOSE), INCOSE Syst. Eng. Handbook v3.1, 2007.

[5] M. Eriksson, K. Borg, and J. Boerstler, "The FAR approach - functional analysis/ allocation and requirements flowdown using use case analysis," in Proc. of the 16th Intl. Symp. Of INCOSE, Jul. 2006.

[6] R. Alur, C. Couroubetis, and D. Dill., "Model-checking for real-time systems," in Proc. 7th IEEE Symp. Logic in Comp. Sci., 1990, pp. 414–425.

[7] R. Alur and D. Dill, "A theory of timed automata," Theor. Comp. Sci., vol. 126, no. 2, pp. 183–235, 1994.

[8] U. Sorensen and C. Thrane, "Slicing for UPPAAL," Master's thesis, Aalborg University, Dept. of Comp. Sci., June 2007.

[9] J. Dong, P. Hao, S. Qin, J. Sun, and W. Yi, "Timed automata patterns," IEEE Trans. Soft. Eng., vol. 34, no. 6, pp. 844–859, November/December 2008.

[10] M. Dwyer, G. Avrunin, and J. Corbett, "Property specification patterns for finite-state verification," in Proc. 2nd Workshop on Formal Methods in Soft. Practice, 1998, pp. 7–15.

[11] S. Konrad and B. Cheng, "Real-time specification patterns," in Proc. ICSE, 2005, pp. 372–381.

[12] L. Grunske, "Specification patterns for probabilistic quality properties," in Proc. ICSE, 2008, pp. 31–40.

[13] F. Houdek and B. Paech, "Das Türsteuergerät - eine Beispielspezifikation" Fraunhofer IESE, Technical Report, in German 2002.02/D, January 2002.

[14] K. Larsen, J. Bengtsson, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL - a tool suite for automatic verification of realtime systems," in Hybrid Systems III, LNCS 1066. Springer-Verlag, 1995, pp. 232–243.

[15] A. Roeser and G. Pai, "Rigorous model-driven requirements engineering: A case study with timed automata," Fraunhofer IESE, Technical Report, May 2009.

[16] G. Behrmann, A. David, and K. Larsen, "A tutorial on UPPAAL," LNCS, no. 3185, pp. 200–236, 2004.

[17] S. Miller, M. Whalen, and D. Cofer, "Software model checking takes off," Comm. of the ACM, vol. 53, no. 2, pp. 58–64, 2010.

[18] R. Lutz, "Analyzing software requirements errors in safetycritical embedded systems," in Proc. IEEE Intl. Symp. Requirements Engineering, Jan. 1993.

[19] S. Easterbrook et al., "Experiences using lightweight formal methods for requirements modeling," IEEE Trans. Software Eng., vol. 24, no. 1, pp. 4–14, Jan. 1998.

[20] T. Breaux, A. Ant´on, and J. Doyle, "Semantic parameterization: A process for modeling domain descriptions," ACM Trans. Softw. Eng. Methodol., vol. 18, no. 2, 2008.

[21] S. Konrad and B. Cheng, "Requirements patterns for embedded systems," in Proc. of the IEEE Intl. Req. Eng. Conf. (RE), 2002.

[22] S. Konrad, B. Cheng, and L. Campbell, "Object analysis atterns for embedded systems," IEEE Trans. Soft. Eng., vol. 30, no. 12, pp. 970–992, Dec. 2004.

[23] M. Lindahl, P. Pettersson, and W. Yi, "Formal design and nalysis of a gearbox controller," Springer Intl. Jnl. of Soft. ools for Tech. Transfer, vol. 3, no. 3, pp. 353–368, 2001. 24] B. Douglass, Real-time Design Patterns. Addison-Wesley, 003.

[25] F. Bitsch, "Safety-patterns - the key to formal specification of safety-requirements," in Proc. COMPSAC, LNCS 2187, 2001, p. 176–190.

.

Document Information

Title:	Assuring Functional Safety in Automotive Software Through Pattern-based Re-quire-ments Development
Date:	April 2010
Report:	IESE-013.10/E
Status:	Final
Distribution:	Public