Modelling an Automotive Software System with TASTD*

Diego de Azevedo Oliveira[1] and Marc Frappier[1]

Université de Sherbrooke, Sherbrooke QC J1K 2R1, Canada

Abstract. At the ABZ2020 Conference, the case study track proposed to model an Adaptive Exterior Light System and a Speed Control System: the former controls different exterior lights of a vehicle while the latter controls the speed of a vehicle. This paper introduces a model for these two case studies using Timed Algebraic State-Transition Diagrams (TASTD). TASTD is an extension of Algebraic State-Transition Diagrams (ASTD) providing timing operators to express timing constraints. The specification makes extensive use of the TASTD modularity capabilities, thanks to its algebraic approach, to model the behaviour of different sensors and actuators separately. We validate our specification using the cASTD compiler, which translates the TASTD specification into a C++ program. This generated program can be executed in simulation mode to manually update the system clock to check timing constraints. The model is executed on the test sequences provided with the case study. The paper provides a comparison between the TASTD model and other solutions presented at the ABZ2020 Conference. The advantages of having modularisation, orthogonality, abstraction, hierarchy, real-time, and graphical representation in one notation are highlighted with the proposed model.

Keywords: ASTD · real-time model · ABZ2020 case study · TASTD · formal method

1 Introduction

The ABZ2020 Conference case study [9] describes an adaptive exterior light system (ELS) and a speed control system (SCS). The ELS controls several lights, which are parts of various subsystems, like controlling side lights and comfort functions. The SCS is a function that tries to maintain or adjust the vehicle's speed according to various external influences. Both systems are examples of software systems present in modern vehicles.

In this article, we present our specification of the ABZ2020 case study to demonstrate the usefulness of TASTD as a modelling language. We use ASTD

* Supported by organization Supported by Public Safety Canada's Cyber Security Cooperation Program (CSCP) and NSERC (Natural Sciences and Engineering Research Council of Canada).

tools to generate executable code in C++, which could be deployed in an embedded system. First, we specify our model with the ASTD editor, eASTD. Second, we produce executable code with the ASTD compiler, cASTD.

To facilitate comparison with existing work, the structure of this paper follows the one proposed in the *call for paper* of the ABZ2020 case study track. The subsequent subsections briefly present TASTD, its supporting tools, and the distinctive features of our approach.

This paper is structured as follows. Section 2 describes our modelling strategy. In Section 3, we take into account all requirements for both systems (ELS and SCS), except a minor one which deals with the graphical user interface. Section 4 presents the validation process of the case study model, and the discussion around the verification of the model. Previous solutions identified flaws in the case study documentation. We confirm such issues in Section 5. Section 6 compares our TASTD model with those presented at the ABZ2020 Conference. Lastly, Section 7 concludes the paper.

1.1 TASTD

Timed Algebraic State-Transition Diagrams (TASTD) [4] is a time extension for ASTD [18]. ASTD allows the composition of automata using CSP-like process algebra operators: sequence, choice, Kleene closure, guard, parameterized synchronization, flow (the AND states of Statecharts), and quantified versions of parameterized synchronization and choice. Each ASTD operator defines an ASTD type that can be applied to sub-ASTDs. Elementary ASTDs are defined using automata. Automaton states can either be elementary or composite; a composite state can be of any ASTD type. Within an ASTD, a user can declare attributes (i.e., state variables). Actions written in C++ can be declared on automata transitions, states, and at the ASTD level; they are executed when a transition is triggered. These actions can modify ASTD attributes and execute arbitrary C++ code. Attributes can be of any C++ type (predefined or user defined).

TASTD introduces time-triggered transitions, i.e., transitions triggered when conditions referring to a global clock are satisfied. In ordinary ASTDs, only the reception of an event from the environment can trigger a transition. The special event Step labels the timed-triggered transitions. Automata transitions can be labelled with the special event Step. Step is treated as an event; its only particularity is that it is evaluated on a periodical basis. The specifier determines the value of the period according to the desired time granularity required to match system timing constraints. TASTD also introduces new ASTD timing operators that can perform Step transitions: delay, persistent delay, timeout, persistent timeout, and timed interrupt. TASTDs rely on the availability of a global clock called cst, which stands for *current system time*. If the guard of a Step transition is satisfied, the transition can be fired. TASTD is fully algebraic, TASTD operators can be freely mixed with ordinary ASTD operator.

1.2 TASTD support tools

TASTD specifications can be edited with a graphical tool called eASTD and translated into executable C++ programs using cASTD. The generated C++ programs can be used as an actual implementation of the TASTD specification. cASTD can generate code for simulation, where a manual clock, which the specifier controls, replaces the system clock. The specifier can decide to advance the clock to a specific time; the simulator will generate the Step events necessary to reach the specified time. Environment events can be submitted at these specified times. We use a simulation to validate the provided scenarios discussed in Section 4.

A new tool called pASTD is under development; it will permit to specify TASTD attributes and actions using the Event-B language and generate proof obligations for invariants declared on automata states and TASTDs. These proof obligations are represented as theorems of a synthetic Event-B context that can be proved using the Rodin platform. Such an Event-B-annotated ASTD specification could then be refined into an implementation by transforming actions into B0 actions, proving their refinement, and translating them into C using the Atelier B tools.

1.3 Distinctive features of our modelling approach

Modularisation, Hierarchy And Orthogonality ASTD is an algebraic language, in the sense that an ASTD is either elementary, given by an automaton, or compound, given by a process algebra operator applied to its components. This algebraic approach streamlines modularity. A model can be decomposed into several parts which are combined with the process algebra operators. As it will be described in Section 2, the case study is decomposed into several parts which are specified separately and then connected with ASTD types synchronisation or flow. Each ASTD contains a name, parameters, variables, transitions, actions, and states (an initial state is required). An ASTD state may be of any ASTD type, called sub-ASTD, and share its variables, transitions, and states with its parents. With this modular and hierarchical structure, isolating an ASTD and modifying its behaviour does not produce side effects in other ASTD. Modularity also makes the specification easier to understand, because each component can be analysed separately.

Time In TASTD, time is integrated into its syntax and its semantics. As portrayed by the case study requirements, time management is implemented with clock variables or using TASTD operators. That allows us to produce executable code satisfying the time constraints.

Graphical representation With ASTD graphical representation, to understand the behaviour of an ASTD is to reason about its transitions and states. ASTD visualisation is an advantage over other formal methods that only use textual representation, which makes their specification harder to understand.

2 Modelling Strategy

This section describes our modelling strategy and how the model is structured and provides insights into how we approached the formalization of the requirements. The complete model is found in [3].

Model structure Our specification mainly uses two ASTD operators to structure the model. These are flow, denoted by Ψ, and parameterized synchronisation, denoted by $|[\Delta]|$. The flow operator is inspired from AND states of Statecharts, which execute an event on each sub-ASTD whenever possible. The parameterized synchronization operator executes two sub-ASTDs in parallel, and they must synchronize on a set of events Δ. If Δ is empty, then the parameterized synchronisation is an interleave, denoted by $|||$. We can draw an analogy between these three operators and Boolean operators. Operator $||[]||$ acts like a conjunction: $E_1||[\{e\}]||E_2$ can execute an event e iff both E_1 and E_2 can execute it. It expresses a conjunction of ordering constraints on e given by E_1 and E_2. It is a *hard* synchronisation. Operator Ψ acts like an inclusive OR: $E_1 \Psi E_2$ can execute an event e iff either E_1, or E_2, or both E_1 and E_2 can execute it. It is a *soft* synchronisation. Operator $|||$ looks like an exclusive OR: $E_1 ||| E_2$ will execute e on either E_1 or E_2, but on only one of them; if both E_1 and E_2 can execute e, then one of them is chosen nondeterministically.

ELS and SCS systems are loosely coupled [1], which means that each component can handle some requirements independently. At start, we divide our model into the elements that the user or the environment can manipulate, such as buttons, switches, and sensors, and the response on the actuators after manipulating those elements. We call the former group the *sensors* and the latter group the *actuators*. Figure 1 shows the ASTD Control, composed of sensors and actuators. Each green box is a call to the ASTD of that name. ASTD Sensors combines the various sensor ASTDs using an interleave operator; no synchronisation is needed between the sensors, because each sensor has its own distinct set of events. Operator $|||$ being commutative and associative, ASTD Sensors is shown here as an n-ary ASTD. The ASTD model of a sensor describes the physical ordering constraints on the events of that sensor. For instance, the ignition key cannot do event putIgnitionOn without doing first insertKey. We shall illustrate such an ASTD in Section 3.1.

The actuators are partitioned into two parts: speed actuators and light actuators. The actuators are composed using a flow operator, because a sensor event may influence several actuators, and a sensor event might influence an actuator, depending on state. Thus, actuators are not synchronized, but composed with a flow.

ASTD Control composes sensors and actuators also with a flow. ASTD CAR in Figure 2 is the root (main) ASTD. It synchronises ASTDs Control and Sensors. This means that ASTD Sensors is called twice: once within Control in a flow, and once again at the root level in ASTD CAR in a synchronisation. This particular pattern is used to enforce a priority on ordering constraints between sensors and

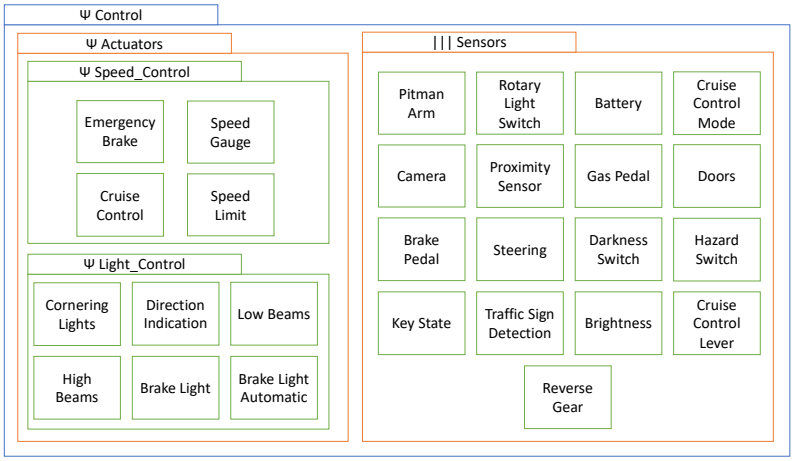

Fig. 1. ASTD Control composing sensors and actuators

actuators. In order to accept a sensor event, it must first satisfy the physical ordering constraints of that sensor. An actuator may refuse a sensor event that is accepted by the corresponding sensor ASTD, because in its current state, the actuator ASTD is not influenced by the sensor event and can ignore it. On the other hand, a sensor event should not be accepted by ASTD CAR if the actuator ASTD can execute it, but the sensor ASTD cannot execute it; that would violate the physical ordering constraints of the sensor. Thus, using simply a flow between sensors and actuators is insufficient, because it would allow the system to accept a sensor event through the actuators, even if it is refused by the sensor ASTD. This is why ASTD Control alone is insufficient and cannot be the main ASTD. ASTD CAR synchronises Control with Sensors on sensor events, so that CAR accepts a sensor event when both Control and Sensors can execute it. ASTD Control always accepts sensor events that Sensors can accept, because it combines Actuators and Sensors with a flow, which is not blocking.

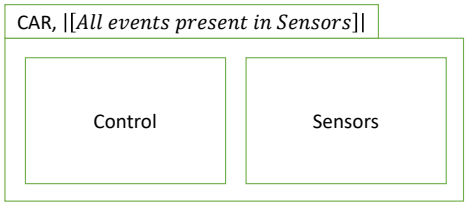

Fig. 2. Main ASTD of ELS and SCS

Communication with shared variables ASTD allows the use of shared variables, which are called *attributes* in the ASTD notation. An attribute declared in an ASTD may be used in guards and actions of its sub-ASTDs. Attributes are used to communicate the state of a sensor to the actuator ASTDs; this allows for the reduction of the number of states in automata. Sensor ASTDs update attributes describing the state of a sensor. Actuator ASTDs read these attributes to determine the acceptance of an event and to compute the actuator response. For flow and synchronisation ASTDs, shared attributes must be used with care, because their sub-ASTDs are executing in sequence. The semantics of the ASTD requires commutativity on the execution of the actions in a flow $E_1 \Psi E_2$, such that it terminates on the same values of the attributes whether either E_1 or E_2 is executed first. Commutativity is easily ensured in our specification, because only the sensor ASTDs update the sensor attributes, and sensor events in actuator ASTDs do not read the value of sensor attribute in their guards or actions.

Table 1 presents the attributes declared in each ASTD. Attributes declared in the root ASTD CAR indicate the current state of the sensors. For example, attribute *keyState* indicates the state of the ignition key (*KeyInserted, NoKeyInserted, KeyInIgnitionOnPosition*). ASTD Speed_Control shares attribute *speedLimit*, a Boolean to indicate if the speed limit is on, and *emergencyBrake*, to indicate if an emergency brake is necessary.

Component	Variables
CAR (root)	pitmanArmUD, pitmanArmFB, lightSwitch, keyState, hazardSwitch, armoredVehicle, darknessMode, reverseGear, voltageBattery, cameraState, steeringAngle, highBeamOn, currentSpeed, engineOn, SCSLever, cruiseControlMode, rangeRadarSensor, gasPedal, brakePedal, sCSLever, safetyDistance, rangeRadarState, speedMode, trafficSignDetectionOn, allDoorsClosed, oncommingTraffic, brightnessSensor, cruiseControlOn
Actuators	setVehicleSpeed
Light_Control	brakeLight, blinkLeft, blinkRight, tailLampLeft, tailLampRight, lowBeamLeft, lowBeamRight, corneringLightLeft, corneringLightRight
Speed_Control	emergencyBrake, speedLimit

Table 1. Shared variable by components

The complete model is composed of 66 automata, 1 closure, 26 synchronisation, 14 flow, 33 call, 1 persistent guard, 7 persistent delay, and 2 delays, for a total of 150 ASTDs. These numbers are artificially high, because n-ary ASTDs are currently not supported by the editor eASTD. Thus, an n-ary ASTD is represented by $2n - 1$ ASTDs, instead of simply $n + 1$ ASTDs. For instance, an interleave $E_1 \,|||\, E_2 \,|||\, E_3$ is represented by 5 ASTDS ($E_{123}, E_{12}, E_1, E_2, E_3$),

because E_{12} represents the interleave ASTD composing E_1 and E_2, and E_{123} composing E_{12} with E_3.

Formalization of the requirements Tables 2 and 3 relate ASTDs and requirements listed in [9]. Some requirements are present in several ASTDs as they are related to different actuators. Time requirements, such as ELS-1 and SCS-8, are covered with the use of event Step. SCS-30 is the only requirement not covered since it concerns user interface representation.

ASTD	Requirements
directionIndication	ELS-1, ELS-2, ELS-3, ELS4, ELS-5, ELS-6, ELS-7, ELS-8, ELS-9, ELS-10, ELS-11, ELS-12, ELS-13, ELS-23, ELS-29, ELS-47
lowBeams	ELS-14, ELS-15, ELS-16, ELS-17, ELS-18, ELS-19, ELS-21, ELS-22, ELS-28, ELS-29, ELS-47
corneringLights	ELS-24, ELS-25, ELS-26, ELS-27, ELS-29, ELS-45, ELS-47
highBeams	ELS-30, ELS-31, ELS-32, ELS-33, ELS-34, ELS-35, ELS-36, ELS-37, ELS-38, ELS-42, ELS-43, ELS-44, ELS-46, ELS-47, ELS-48, ELS-49
brakeLightAut	ELS-29, ELS-39, ELS-40, ELS-47
reverseLightAut	ELS-29, ELS-41, ELS-47

Table 2. Cross-reference between ASTDs and requirements for adaptive exterior light system of [9]

3 Model Details

This section shortly describes the main modelling elements of our specification following the structure explained in the previous section.

3.1 Sensors ASTDs

Sensor ASTDs describe the physical ordering constraints and the valid states that the sensors can attain. For example, Figure 3 shows the ASTD key. This ASTD is an automaton, and its states are NoKeyInserted, KeyInserted, KeyInIgnitionOnPosition, with initial state NoKeyInserted. The transitions represent valid movements of the key. On each transition, the attribute *keyState* is updated. Event putIgnitionOn turns the engine on, and attribute *engineOn* becomes true. Event putIgnitionOff sets attribute *engineOn* to false as the engine turns off.

ASTD	Requirements
cruiseControl	SCS-1, SCS-2, SCS-3, SCS-4, SCS-5, SCS-6, SCS-7, SCS-8, SCS-9, SCS-10, SCS-11, SCS-12, SCS-13, SCS-14, SCS-15, SCS-16, SCS-17, SCS-18, SCS-19
automatedControlVehicleAhead	SCS-20, SCS-21, SCS-22, SCS-23, SCS-24, SCS-25, SCS-26
emergencyBreakSignals	SCS-27, SCS28
speedLimitControl	SCS-29, SCS-31, SCS-32, SCS-33, SCS-34, SCS-35
trafficSignDetection	SCS-36, SCS-37, SCS-38, SCS-39
cameraAndProximity	SCS-40, SCS-41
brakePedal	SCS-42
brakeLightAutomatic	SCS-43
Not implemented	SCS-30

Table 3. Cross-reference between ASTDs and requirements for speed control system of [9]

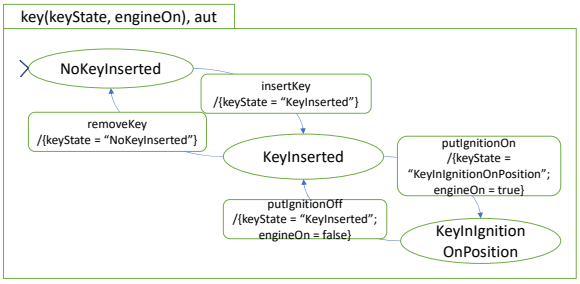

Fig. 3. Automaton ASTD key

3.2 Actuators ASTDs

Actuators depend on the sensors to act. Attributes describing the sensors' state affect how the actuators can be executed.

Consider ASTD DirectionIndication of Figure 1, which is a flow between ASTD BlinkControl that indicates if the blink is tip blinking, hazard switch blinking or non-tip blinking, and ASTD BlinkBulb, that indicates if the light bulb is on or off. For the sake of simplicity, we show an excerpt of the transitions between states off and tip from sub-ASTD BlinkControl in Figure 4. State off indicates that blinking shall stop after completing the previous signal, whereas tip indicates that tip blinking shall be executed. Those two states have five transitions that depend on the pitman arm, hazard switch, key state, and time. The transition from off to tip through event movePitmanArmUD is guarded on the position in which the pitman arm is moving and the key state. The guard is to conform to

requirements ELS-1, ELS-5, and the statement that direction blinking is only available when the ignition is on. Executing the transition changes the value of attributes *pitmanArmUDP*, *tip_timer*, and *NbrCycles*. *pitmanArmUDP*, stores the value of the last pitman arm position and is later used to define which side shall blink, which is related to ELS-3. Attribute *tip_timer* acknowledges how long the user holds the pitman arm, which is related to ELS-4. *NbrCycles* is a counter to determine how many blinking cycles are necessary to stop, related to ELS-7 and ELS-3.

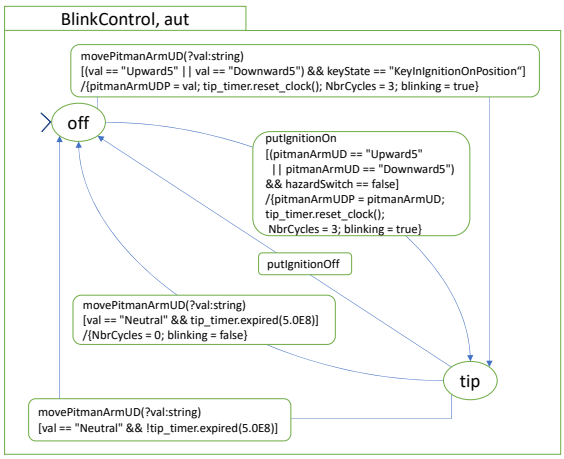

Fig. 4. ASTD BlinkControl, extract with states tip and off

Similarly, Figure 5 is an excerpt from sub-ASTD BlinkBulb. State off indicates that the light is off due to a dark cycle or no blinking. State on means that the light is on. These two states alone have six transitions between them. Transition from off to on through event movePitmanArmUD, is related to ELS-1, ELS-10, ELS-11, ELS-13. It has a guard on the position to which the pitman arm is moving, the key state, the hazard switch, and the cycle timer. Attribute *cycle_timer* acknowledges for how long the bulb is bright or dark and is used to accomplish ELS-1, ELS-10. *hazardSwitch* indicates if the hazard switch is on. Executing this transition resets *cycle_timer* and performs function *blinkLightsOn*, that transition, satisfying other requirements, turns on the blinking lights.

Moving the pitman arm from Neutral to Upward5, in a state where only the engine is on, will execute transitions present in ASTDs PitmanArm, BlinkBulb, and BlinkControl. This results in the activation of the right direction blinking light.

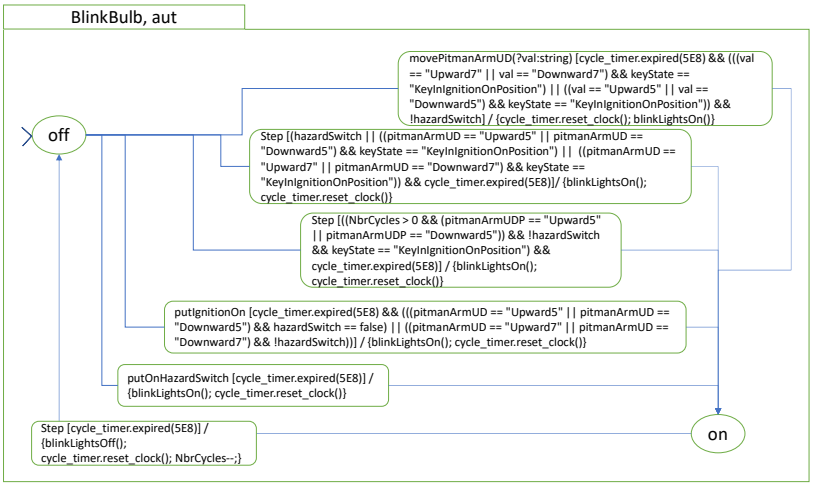

Fig. 5. ASTD BlinkBulb, extract with states on and off

3.3 Modelling time requirements

Some requirements (e.g., ELS-1 and SCS-7) determine specific behaviour for distinct components during a specific time interval. In TASTD, with each event Step, the system acknowledges the passage of time. So, choosing a Step interval value that allows the specification to successfully achieves all requirements is mandatory.

For this case study, we choose the value of Step as 0.05 seconds. With that Step value, we accomplish every requirement, even ELS-40, and ELS-8, which have a different flashing rate than ELS-1. ELS-40 asks for a pulse ratio of 360 ± 60 flashes per minute. In other words, 1/12 of a second bright and 1/12 of a second dark. For ELS-40, the chosen step value accomplishes the requirement because we can have 60 flashes less per minute. In the worst-case scenario, with a step of 1/20 second, there are 300 flashes per minute. ELS-8 demands a fixed pulse ratio of 1:2, which means 1/3 of a second bright and 2/3 of a second dark, without a safe range. With our chosen Step, we complete each cycle at 1.05 seconds, which has approximately 57 cycles and slightly misses the requirement. Additionally, the ratio of 1/3 means a pulse of $0.\widehat{3333}$ seconds, and we would miss the requirement at any chosen step value. We would accomplish the requirement if ELS-8 had a tolerance as in ELS-1 or ELS-40.

At each occurrence of a Step event, the flow ASTD Actuators executes every transition labelled with Step whose guard holds, in each automaton under its scope. In Figure 5, we have a transition Step from state on to state off. This transition is responsible for finishing a bright cycle and turning off the direction blinking light. In our simulations, every 0.05 seconds, the system receives a Step event, to react to the passage of time. The guard cycle_timer.expired(5E8) of the

Step transition from state on to off ensures that it is executed only after 0.5 seconds (i.e., every ten steps) in the state on. In our specifications, nanoseconds (ns) are used as time units, so 5E8 denotes $5x10^8$ ns = 0.5 s.

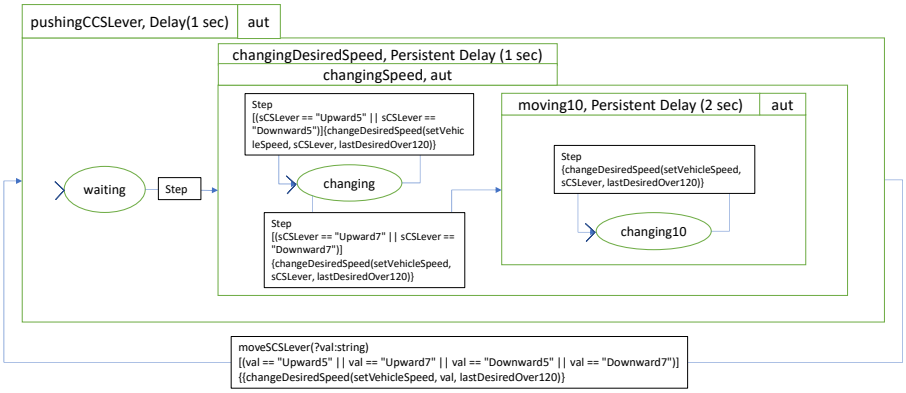

Fig. 6. TASTD pushingCCSLever

Figure 6 shows TASTD pushingCCSLever, an excerpt of the cruise control ASTD. TASTD pushingCCSLever is related to requirements SCS-7 to SCS-10. In summary, those requirements mean: if the driver pushes the cruise control lever to an upward or downward position within the first or second resistance level and holds it there for two seconds, the desired speed of the cruise control is adjusted every second (every two seconds for positions at 7°, beyond the pressure point) following the lever position.

TASTD pushingCCSLever is a delay. It allows for idling at least d time units before the first event. Once the first event occurs, the TASTD may continue its execution without delay. It has transition moveSCSLever, related to SCS-1 to SCS-12. Function changeDesiredSpeed changes the desired speed to match the input from the cruise control lever, and attribute *lastDesiredOver120* is related to SCS-39. The initial state of pushingCCSLever is state waiting, and it does nothing but waits one second.

The second state of pushingCCSLever is a Persistent Delay. It allows idling for at least d time units before executing each event of its sub-ASTD. The persistent delay of one second means that each Step inside changingDesiredSpeed waits at least one second to be accepted. If the lever position is at a 7°, then the step moves the active state to the sub-TASTD moving10. moving10 is a persistent delay of two seconds, related to SCS-8 and SCS-10. If the lever position is at a 5°, then with each Step, the active state remains at state changing, and the delay continues as one second, which is related to SCS-7 and SCS-9. It is worth noting that the first constraint of two seconds, from SCS-7 to SCS-10, is satisfied with

the delay that waits one second and the first persistent delay on the first event that waits for another second.

4 Validation and Verification

To validate our model, we use interactive animation of the specification with the executable code generated by the cASTD compiler for simulation. The compilation is automatic, and no human modification is necessary after production. We execute the compiled code and compare the results with the provided scenarios [8]. Our model satisfies the scenarios provided in the case study, with minor differences in current speed due to insufficient information in the case study on calculating it when accelerating or decelerating. To overcome this difference, we added to our model an event that changes the current speed to a chosen value. We use this function when execution arrives at row "target speed reached" for each scenario, mainly to continue the simulation with the same speed as provided in the trace. Figure 7 presents the TASTD responsible for calculating the speed at each Step. At each Step, currentSpeed is calculated, and it can be adjusted with updateSpeed if deceleration or acceleration is insufficient.

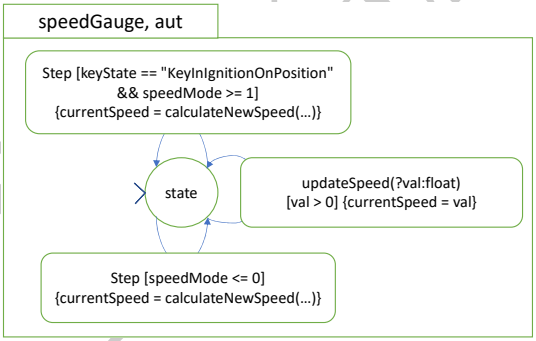

Fig. 7. TASTD speedGauge

Changes to the model due validation Validation is a method to help ensure that a specification's behavior reflects its requirements. During the interactive animation, we found a divergence between our model and the requirements for the low beams and direction indication. The provided traces and their interpretation were determinant to solving ambiguities and providing the correct behaviour.

5 Specification Ambiguities and Flaws

In ABZ2020, several authors [1,6,11,13,15,16] found different ambiguities and flaws in the case study document [9]. We confirm such ambiguities. Although updated versions were provided after their feedback, the document still has ambiguities concerning their statements.

For example, ELS-42, which [11] mentioned, has yet to be changed. There is no indication of what happens with the high beams in the case of sub-voltage, only that it is not available. What we modelled is in case of sub-voltage, the high beams are still on, because of ELS-43, which states that even in the case of sub-voltage, if the light rotary switch is in position Auto and the pitman arm is pulled (which it is with ELS-42), the high beam is activated.

6 Comparison

In this Section, we compare our solution with the approaches and techniques previously used to model the ABZ2020 case study.

In [16], ELS is specified with Event-B. Their model is verified by proving the generated proof obligations for invariants using Rodin and validated with animation in ProB [12]. During their formalisation, authors identified and reported several ambiguities in the requirements, which were addressed in the newer versions of the document. Event-B does not provide visualisation or modularisation mechanisms. A model is developed by successive refinements. Each refinement can add new behavior, but in a somewhat restrictive manner, because new events cannot modify existing variables. Thus, when a variable is introduced, all events that must update it must be introduced at the same time. Independent system components are typically added one at a time, and separately proved. Thus, an interleave ASTD is typically modeled as two (or more) successive refinements. A flow or synchronisation ASTD must be modeled in the same refinement in Event-B, because the state changes must be modeled at the same time, or either abstracted and later refined. Invariants are global; to attach them to a specific state, one must use invariants of the form $ifInStateX \Rightarrow propertyOfStateX$. If an event appears in more than one component, its specification becomes increasingly complex, because its guard is enriched with new ordering constraints, in a monolithic manner. It makes it difficult to analyse those components independently. It also makes it hard to modify a specification, because the refinements are closely coupled, and moving one aspect from one refinement to another is a complex reengineering task, which involves reproving several proof obligations. Event-B events have many guards with many variables for a sizeable system like the ELS and SCS case studies, which makes it hard to understand the behaviour of an event. For example, in the final refinement, the event to turn the key to the ignition on position has 16 guards and 17 elementary actions. In TASTD, each component is specified separately. Synchronization and flow ensure that constraints imposed on an event in several components are defined separately and can thus be analysed and (hopefully) proved separately. In our specification,

the event that ignites the engine is decomposed over nine transitions in ASTDs low beams, direction indication, and key. This modularisation streamlines the understanding of the behaviour of that event. On the other hand, one has to go over all these transitions to get a complete picture of the behavior of the event.

The work in [15] presents an Event-B specification of the SCS. As in [16], the authors found ambiguities in the SCS requirements. Again, the lack of visualisation makes their specification harder to understand, and the modularisation of TASTD over event-B shows a significant advantage of our approach.

In [11], authors present a verified low-level implementation using MISRA C. MISRA C is a language derived from the automotive industry, which is close to C. To verify their specification, the authors implement ELS and SCS in C and perform unit tests. Afterwards, they perform formal verification with the CBMC model checker [5]. However, as stated in [16], even if this approach has the advantage of directly producing the executable code, its correctness cannot be guaranteed since model checking on a limited scope does not ensure the absence of bugs. The authors also provide a list of ambiguities that they found in [9].

In [6], ELS is specified with Electrum [14]. Electrum extends Alloy [10] with mutable relations and temporal logic. The authors do not address requirements needing arithmetic operations, because of the limitations of Alloy for model checking integer values. Their model uses signatures to model the structural aspect and predicates to capture the system's behavior. Verification and validation of their specification use animation through **run** instructions exercising simple behaviours of the system and a validator for complex requirements. With the Alloy Analyzer, the authors can provide a visual animation of the states during the execution of the system. Alloy being a model-based notation like Event-B, it suffers from the same weaknesses in terms of modularization., whereas ASTD modularisation, thanks to its algebraic nature, allows a specifier to isolate a component.

In [1], authors use Abstract State Machine (ASM) to model both ELS and SCS. They use the ASMETA framework to edit, simulate and animate their machines. Similar to event-B, their approach is refinement based, where they start with a simple machine and add details through refinements. ASM also allows for modularity. Their validation is with interactive animation. Requirements verification is performed through model checking using AsmetaSMV [2], which supports CTL and LTL. A downside of their specification is that they cannot deal with continuous time. Thus they do not address requirements that demand time management. Additionally, they mention ambiguities in the case study document but do not state them.

The work in [13] presents a specification with a subset of the case study in classical B and Event-B, then compare the two. With classical B, they found advantages with its modularisation capabilities. With Event-B, the advantage is in the proving environment, which generates more straightforward proof obligations than classical B. The authors divided their modelling strategy into three phases: 1) an exploratory phase with editing and animation, in which they used classical B for its rich substitution language. 2) a synthesis phase with a refinement-

based approach with classical B, in which components were integrated, and the authors added safety invariants verified using model checking. 3) a verification phase, where they manually translated the classical B specification to Event-B and then proved and model checked.

7 Conclusions

To summarize, we have presented a TASTD model for the adaptive exterior light and speed control system case study of ABZ2020. Our model considers all the requirements, excluding SCS-30. We validate our model through interactive animation and comparison with the validation scenarios proposed in the case study.

The main advantages of modelling with TASTD in comparison with other methods presented in ABZ2020 are the following.

- The algebraic approach allows for the decomposition of a specification into very small components which are easier to analyse and understand. In particular, the behavior of an event that affects several components can be separately specified in each component. The synchronisation and flow operators can be used to indicate how these components interact over these events (i.e., hard or soft synchronisation).
- Communication by shared attributes permits to simplify automata of a specification and reduce the number of automaton states.
- The graphical nature of TASTD allows for an easier understanding of a specification. Automata and process algebra operators makes it easier to understand the ordering relationship between events.
- TASTD provides a simple, modular approach to deal with timing requirements.
- TASTD, with its compiler cASTD, can generate C++ code that can be deployed into an embedded system. It is also capable of generating code for simulation, in order to check scenarios.

TASTD currently lacks supports for verification. As future work, we intend to extend TASTD with invariants that can be attached to automaton states and ASTD themselves, thus allowing to decompose the verification of properties into small parts. Attributes could be written using the mathematical language of classical B or Event-B and actions could be written using the rich generalized substitution language of classical B. Proof obligations will be generated as theorems of Event-B contexts and proved using Rodin, which provides a nice proving environment. A translation from ASTD to B has been proposed in [7,17], but it produces monolithic, complex POs. We hope that this new approach will help to simplify proof obligations.

This case study is, until now, the most extensive specification defined with TASTD in the number of attributes and ASTDs, with 150 ASTDs, 50 attributes, and generated executable code of 11MB. It demonstrated that the editor was not ready for a specification with many ASTDs, and the compiler was unprepared

for a specification with many attributes. Thanks to this model, both the editor and the compiler were improved to deal with large specifications.

The editor must still be extended to deal with large specifications and n-ary operators. For instance, Figure 1 was manually prepared for this paper to remove superfluous intermediate binary ASTDs that make the specification harder to read.

Additionally, during model development, we considered creating a new ASTD type to avoid the double call to ASTD Sensors in our solution. With this type, we want to describe the idea of a control ASTD A_1 (e.g., ASTD Sensors of our case study) and a controlled ASTD A_2 (e.g., ASTD Actuators of our case study), which, we believe, is a recurring pattern in control specifications. ASTD A_1 and A_2 would be "partly" synchronised through a set Δ of events, in the following sense. An event e of Δ would be executed iff A_1 can execute it. Thus, A_2 is executed iff A_1 can execute e and if A_2 can execute e. if A_1 can execute e, then it does, independently of the capacity of A_2 to execute e.

Another modification to ASTD that we plan to introduce is to allow for the definition an order of execution of the operands of binary operators synchronization, flow and choice. For interleaving and choice, it would allow the specifier to remove nondeterminism and choose which operand will be tested first for execution. For synchronisation and flow, it would allow to determine the order in which the operands will be executed; thus, the second ASTD to execute could reliably use the values of the attributes updated by the first ASTD executed.

References

1. Arcaini, P., Bonfanti, S., Gargantini, A., Riccobene, E., Scandurra, P.: Modelling an automotive software-intensive system with adaptive features using asmeta. In: International Conference on Rigorous State-Based Methods. pp. 302–317. Springer (2020)
2. Arcaini, P., Gargantini, A., Riccobene, E.: Asmetasmv: a way to link high-level asm models to low-level nusmv specifications. In: International Conference on Abstract State Machines, Alloy, B and Z. pp. 61–74. Springer (2010)
3. de Azevedo Oliveira, D., Frappier, M.: Case Study ABZ 2020 TASTD Model. https://github.com/DiegoOliveiraUDES/casestudyABZ2020-tastdmodel (2023), [Online; accessed 06-January-2023]
4. de Azevedo Oliveira, D., Frappier, M.: Technical report 27 - extending astd with real-time. https://github.com/DiegoOliveiraUDES/astd-tech-report-27 (2023), [Online; accessed 28-January-2023]
5. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ansi-c programs. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 168–176. Springer (2004)
6. Cunha, A., Macedo, N., Liu, C.: Validating multiple variants of an automotive light system with electrum. In: International Conference on Rigorous State-Based Methods. pp. 318–334. Springer (2020)
7. Fayolle, T.: Combinaison de méthodes formelles pour la spécification de systèmes industriels. Theses, Université Paris-Est ; Université de Sherbrooke (Québec, Canada) (Jun 2017), https://theses.hal.science/tel-01743832

8. Houdek, F., Raschke, A.: Validation sequences for abz case study "adaptive exterior light and speed control system" v1.8 (2019)
9. Houdek, F., Raschke, A.: Adaptive exterior light and speed control system. In: International Conference on Rigorous State-Based Methods. pp. 281–301. Springer (2020)
10. Jackson, D.: Software Abstractions: logic, language, and analysis. MIT press (2012)
11. Krings, S., Körner, P., Dunkelau, J., Rutenkolk, C.: A verified low-level implementation of the adaptive exterior light and speed control system. In: International Conference on Rigorous State-Based Methods. pp. 382–397. Springer (2020)
12. Leuschel, M., Butler, M.: Prob: an automated analysis toolset for the b method. International Journal on Software Tools for Technology Transfer **10**(2), 185–203 (2008)
13. Leuschel, M., Mutz, M., Werth, M.: Modelling and validating an automotive system in classical b and event-b. In: International Conference on Rigorous State-Based Methods. pp. 335–350. Springer (2020)
14. Macedo, N., Brunel, J., Chemouil, D., Cunha, A., Kuperberg, D.: Lightweight specification and analysis of dynamic systems with rich configurations. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 373–383 (2016)
15. Mammar, A., Frappier, M.: Modeling of a speed control system using event-b. In: International Conference on Rigorous State-Based Methods. pp. 367–381. Springer (2020)
16. Mammar, A., Frappier, M., Laleau, R.: An event-b model of an automotive adaptive exterior light system. In: International Conference on Rigorous State-Based Methods. pp. 351–366. Springer (2020)
17. Milhau, J., Frappier, M., Gervais, F., Laleau, R.: Systematic translation rules from astd to event-b. In: International Conference on Integrated Formal Methods. pp. 245–259. Springer (2010)
18. Nganyewou Tidjon, L., Frappier, M., Leuschel, M., Mammar, A.: Extended algebraic state-transition diagrams. In: 2018 23rd International Conference on Engineering of Complex Computer Systems (ICECCS). pp. 146–155. Melbourne, Australia (2018)