


Modelling a Mechanical Lung Ventilation System using TASTD^{*}

Alex Rodrigue Ndouna¹  and Marc Frappier¹ 

Université de Sherbrooke, Sherbrooke QC J1K 2R1, Canada
{Alex.Rodrigue.Ndouna,Marc.Frappier}@USherbrooke.ca

Abstract. For the ABZ2024 conference, the proposed case study consists of modelling an adaptive outdoor mechanical lung ventilation system. The mechanical lung ventilator is intended to provide ventilation support for patients that are in intensive therapy and that require mechanical ventilation. The system under study is made up of two main software components: the graphical user interface (GUI) and the controller, this paper introduces a model for the controller part of the software system 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 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 · ABZ2024 case study · TASTD · formal method

1 Introduction

The ABZ2024 Conference case study [5] describes an mechanical lung ventilator system. This system is made up of two main software elements: the graphical user interface (GUI) and the controller. The GUI is a touchscreen panel that displays the information needed to check the respiratory condition, allows parameter settings, and displays ventilation parameters and alarm settings. When the controller receives operator input from the GUI, it communicates with the valve controllers, serial interfaces, and other subcomponents and sends them

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

commands. A major critical aspect of the system is its temporal behavior. Many properties and constraints have explicit temporal requirements (like after 7 seconds ..)

In this article we present the specification of the controller component for the ABZ2024 case study with TASTD [3] to demonstrate the usefulness of TASTD as a modelling language. We use ASTD tools [6] to generate executable code in C++, which could be deployed in an embedded system. First, we specify our model with the ASTD editor, **eASTD** [6]. Second, we produce executable code with the ASTD compiler, **cASTD** [6].

This paper is structured as follows. Section 2 provides a brief introduction to TASTD, its support tools, and highlights the distinctive features of our approach. Section 3 describes our modelling strategy. In Section 4, we take into account all the requirements of the controller component, except a minor one which deals with the graphical user interface. Section 5 presents the validation process of the case study model, and the discussion around the verification of the model. We discuss issues and flaws identified in the case study documentation [5] in Section 6. Lastly, Section 7 concludes the paper.

2 Distinctive Features of our Approach

2.1 An Overview of TASTD

Timed Algebraic State-Transition Diagrams (TASTD) [3,4] is a time extension for ASTD [9]. 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. **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.

2.2 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 5.

2.3 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 3, 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 nested ASTDs. 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 the 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.

3 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 [8].

Model structure Our specification mainly uses seven ASTD operators to structure the model. These are the flow operator, denoted by Ψ , the interleaving operator, denoted by \parallel , the sequence operator, denoted by \rightarrow , the kleene closure operator, denoted by \star , the timeout operator, denoted by *Timeout*, the choice operator, denoted by $|$, and the guard operator, denoted by \Rightarrow .

The flow operator is inspired from AND states of Statecharts, which executes an event on each sub-ASTD whenever possible. The interleaving executes two sub-ASTDs in parallel, this operator looks like an exclusive *OR*: $E_1 \parallel E_2$ 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. The sequence operator allows for the sequential composition of two ASTDs. When the first ASTD reaches a final state, the second one can start its execution. This enables decomposition of problems into a set of tasks that have to be executed in sequence. The Kleene closure operator comes from regular expressions; it allows for iteration on an ASTD an arbitrary number of times (including zero). When the sub-ASTD is in final state, it enables to start a new iteration. The Timeout operator allows for verifying if a TASTD receives an event in a period of time, if not, then another TASTD has the right to execute. In the Timeout operator, the first transition of the first TASTD has to occur before d time units, otherwise the timeout is executed. The choice operator allows a choice between two sub-ASTDs; once a sub-ASTD has been chosen, the other sub-ASTD is ignored. It is essentially the same as a choice operator in a process algebra. The guard operator guards the execution of its sub-ASTD using a predicate or a function declaration of its parent ASTD. The first event received must satisfy the guard predicate; once the guard has been satisfied by the first event, the sub-ASTD execute the subsequent events without further constraints.

At the start, we divide our model into the elements that the user or the environment can manipulate, such as buttons, 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 Controller, composed of sensors and actuators. In our case study solution, the notion of sensors covers both the sensor coming from the hardware part of the system and all the system configuration parameters coming from the GUI. The notion of actuators includes both the list of parameters that the controller transmits to the GUI when the request is made by the user, the parameters that the controller sets at the hardware level and the list of parameters resulting from the controller capturing indicators for its own operation.

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.

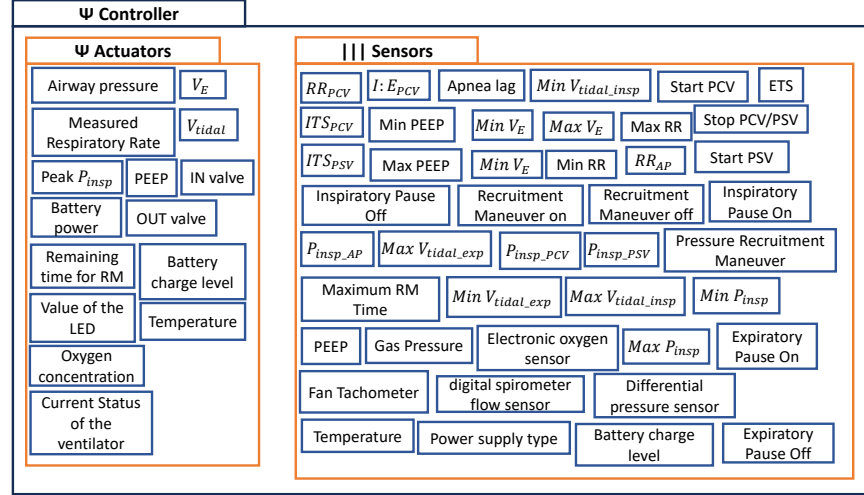


Fig. 1. ASTD Controller composing sensors and actuators

ASTD Main in Fig. 2 is the root (main) ASTD. This ASTD just allows us to start or stop the ventilator. It is mainly made up of the ASTD Controller and two events; the *power_on* event, which is triggered when the user presses the start button, and the *power_off* event, which is triggered when the user presses the stop button.

The controller's machine state diagram of Fig. 4.1 in [5] is made up of 6 main states: *Start-up*, *FailSafe*, *SelfTest*, *VentilationOff*, *PCV* and *PSV*. To take advantage of the modularisation, hierarchy, originality and above all reusability options offered by ASTDs, we're going to decompose our controller into modules corresponding to each state of the state diagram and, using TASTD operators, link these modules together. The same principle of decomposition has been applied to each of the modules individually. The TASTD of the controller is shown in Fig. 3, where the details of ASTD Sensors and Actuators are hidden, since they are shown in Fig. 1.

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 other ASTDs; this allows for the reduction of the number of states in automata. Sensor ASTDs update

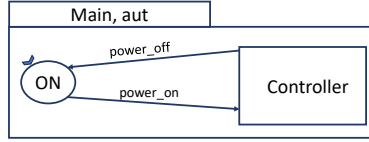


Fig. 2. Main ASTD of Mechanical Lung ventilator controller

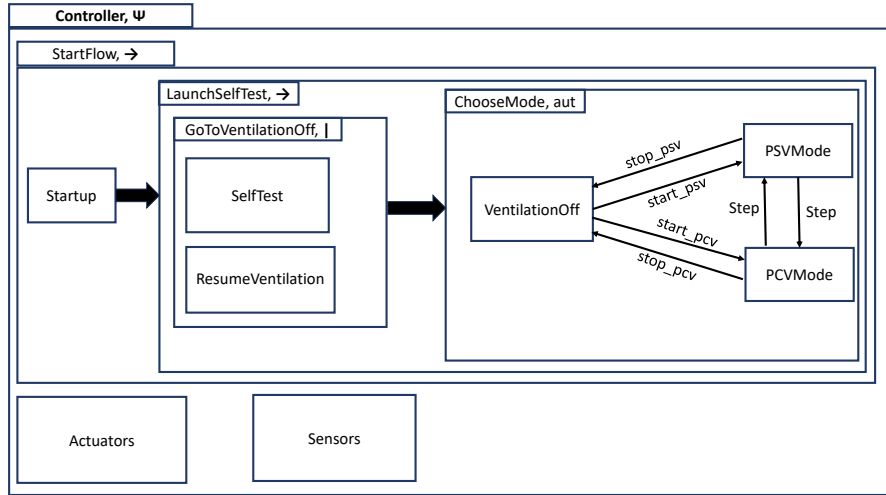


Fig. 3. Controller ASTD of Mechanical Lung ventilator

attributes describing the state of a sensor. Other 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.

In ASTDs, attributes can have primary types, but using the eASTD tool it is also possible to define more complex types in C++ through the notion of class or struct, an option that is very useful depending on the problem to be solved. So our model integrates a new data type called *Parameters* which contains both the sensors coming from the GUI and those coming from the hardware initialised to their default values in accordance with the specification of our case study. The notion of a complex type is even more effective and interesting in this case, as it allows us to associate utility functions with our complex type, which can then

be used to update it. In order to comply with the FUN5.1 specification, we have defined a *read_config* utility function associated with our type, which allows us to initialise a *Parameters* object from the values of the attributes saved in the config file. The *readConfig.h* [8] header file in our model contains the definition of our *Parameters* type.

Table 1 presents the attributes declared in each ASTD. Attributes declared in the root ASTD *Controller* indicate the current state of the sensors. For example, attribute *parameters.start_PCV* indicates whether the user has triggered PCV mode. ASTD *parameters.battery_charge_level* indicates the battery charge level.

Component	Variables
Main (root)	<i>parameters, power_status</i>
Controller	<i>error_value, paw_drop_in, last_inspiration_time, move_to_pcv, mode, peak_pinsp, ve, rr, peep, fio2, vtidal, updated_parameters</i>
StartUp	<i>adc_timeout, pressure_sensor_timeout, error, processes_count</i>
InspirationPhaseEnd	<i>inspiration_phase_timer</i>
Initialization	<i>attempt, status</i>
CheckSensors	<i>pressure_sensor_valid_response</i>
PCVModeExpiration	<i>its_trigger_window</i>
PSVModeExpiration	<i>its_trigger_window, inspiration_time</i>
LaunchLoop	<i>enable_loop</i>

Table 1. Shared variable by components

We want to come back to two important variables in our model, *parameters* and *updated_parameters*. These 2 variables are of the *Parameters* type presented above but with the only difference that *parameters* will contain the values of the attributes at the start of the breathing cycle, values which will be used throughout the current breathing cycle, whereas *updated_parameters* will contain the same values as *parameters* but including the updates made to the attributes during the current cycle. This is because changes made to attributes during the current cycle can only be used by the controller during the next cycle. Attribute *updated_parameters* therefore contains updates made to attributes during the current cycle and will be used to initialise *parameters* at the start of the next cycle so that this cycle can take into account changes made to attributes during the previous cycle. Before the start of the first cycle, *updated_parameters* and *parameters* have the same attribute values, values taken from the configuration file.

The complete model is composed of 73 automata, 5 sequence, 5 synchronisation or interleaving, 4 flow, 28 call, 13 guard, 8 timeout, 15 closure, 7 choice, and 2 delays, for a total of 160 ASTDs.

Formalization of the requirements Tables 2 relates ASTDs and requirements listed in [5]. Some requirements are present in several ASTDs because

they have to be verified by several components. This is the case, for example, for CONT.42 and CONT.43 functionalities, which must be verified by both the PCVMode and PSVMode ASTDs. However, as ASTDs allow for modular modelling, we are going to implement these functionalities in 2 different ASTDs and they will just be called in the PCVMode and PSVMode ASTDs, thus avoiding duplication of code. Time requirements, such as CONT.37, CONT.36.2 and CONT.25, are covered with the use of event **Step**, the use of Timeout ASTD or the use of Delay ASTD type.

ASTD	Requirements
Controller	CONT.1, CONT.2, CONT.3, CONT.4, CONT.5, CONT.6, CONT.7, CONT.8, CONT.9, CONT.10
StartUp	CONT.12, CONT.13, CONT.14, CONT.15, CONT.16
SelfTest	CONT.17, CONT.18, CONT.19
VentilationOff	CONT.38
PCVMode	CONT.20, CONT.21, CONT.22, CONT.23, CONT.24, CONT.25, CONT.26, CONT.27, CONT.28, CONT.39, CONT.40, CONT.41, CONT.42, CONT.43, CONT.44, CONT.45
PSVMode	CONT.29, CONT.30, CONT.31, CONT.32, CONT.33, CONT.34, CONT.35, CONT.36, CONT.37, CONT.39, CONT.40, CONT.41, CONT.42, CONT.43, CONT.44, CONT.45
FailSafe	CONT.38
Main (root)	CONT.11, CONT.38

Table 2. Cross-reference between ASTDs and requirements for mechanical lung ventilator of [5]

4 Model Details

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

4.1 Sensors ASTD

The ASTD **Sensors**, is a component that allows all parameter changes to be listened to in the form of events from the GUI or hardware, with the aim of modifying the controller’s behaviour by updating certain attributes of variables shared in the specification. **Sensors** is an interleave ASTD made up of several sub-ASTDs such as: **HWSensors** which is an ASTD for listening to changes coming from the hardware, **PCVModeSensors**, **PSVModeSensors** and **AlarmThresholdsSensors** which are ASTDs for listening to changes coming from the GUI concerning the configuration of PCV mode, PSV mode and alarm management respectively. Fig. 4 shows the **PCVModeSensors** ASTD.

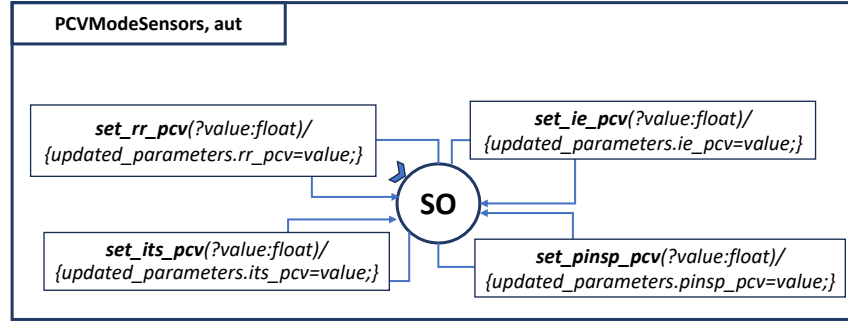


Fig. 4. Automaton ASTD PCVModeSensors

ASTD PCVModeSensors is an automaton with only one state, S0, which is the initial state. This automaton does not have a final state because here we want to listen to updates continuously, so there is no need for an accepting state which would stop listening to these updates. The transitions here represent the update actions coming from the GUI. As transitions we have: *set_rr_pcv* which updates the *rr_pcv* attribute with a float value passed as a parameter, the update is performed using the instruction *updated_parameters.rr_pcv=value*. In the same logic *set_ie_pcv* updates the *ie_pcv* attribute, *set_its_pcv* updates the *its_pcv* attribute and *set_pinsp_pcv* updates the *pinsp_pcv* attribute. Having a single state makes all these transitions reflexive, which means that changes to each attribute can be listened to infinitely. The other sub-ASTDs of the Sensors ASTD follow the same implementation and operating logic as the PCVModeSensors.

4.2 Actuators ASTD

As mentioned in Section 3 presenting the structure of our model, we have defined the actuators in our model as elements that enable our model to provide information to external elements. For our model we have detected fourteen main actuators. These actuators are divided into two main groups: actuators enabling communication with the GUI (*PAW*, *Peak Pinsp*, *RR*, *Temperature*, *V_E*, *V_{tidal}*, *FiO₂*, *PEEP*, *Battery power*, *Battery charge level*, *Remaining time for RM*, *current status of the ventilator*), and those enabling communication with the hardware which are *IN Valve* and *OUT Valve*. Our ASTD Actuators will only focus on the actuators communicating with the GUI, as the actions or events enabling the actuators to be modified in the direction of the hardware are all actions internal to the fan start-up management process or the breathing cycle management process; thus the updating of the latter will be carried out by the *StartFlow* ASTD in Fig. 3.

The Actuators ASTD is of type flow and is made up of twelve sub-ASTDs of the closure type, each corresponding to the management of one of the twelve

GUI actuators. the sub-ASTDs are of the closure type in order to be able to express the notion of continuously listening to requests for information from the GUI for each of the actuators. Each of the closure ASTDs is made up of an automaton ASTD with 3 states and 2 transitions; a transition representing the request from the GUI and an automatic **Step** transition that we use to simulate the controller's response to the request made by the GUI. The Fig. 5 shows the **LoopGetPaw** ASTD used to process any request for information on the value of *PAW*.

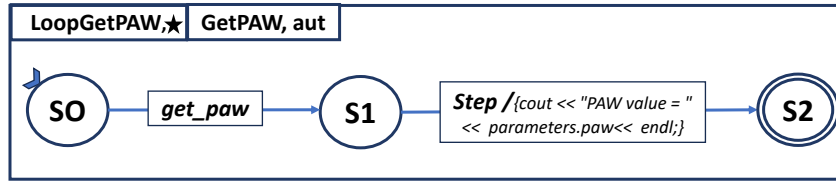


Fig. 5. Automaton ASTD **LoopGetPAW**

The structure of the **LoopGetPAW** sub-ASTD follows the general structure described above, with a closure-type **LoopGetPAW** ASTD for infinite processing of requests, and an automaton-type **GetPAW** sub-ASTD for processing a request for a *PAW* value. The **GetPAW** ASTD has three states **S0**, **S1** and **S2**, with **S0** as the initial state and **S2** as the final state. **S0** and **S1** are linked by a *get_paw* transition which corresponds to a request for a *PAW* value from the GUI, **S1** and **S2** are linked by an automatic **Step** transition which is a simulation of the response from the controller to the *get_paw* request. Here the response simulation only displays the value of the attribute to the console.

4.3 Main(root) ASTD

The general structure of the main ASTD has already been presented in Section 3 but here we return to each of the elements making up this ASTD in greater detail. The main ASTD presented in Fig. 6 is made up of two states: the **ON** state, which is the start state of our model, and the **StartUpCall** state, which is a Call type ASTD enabling a call to be made to another ASTD contained in another diagram, which enables us to take advantage of the modularity offered by ASTDs. The **StartUpCall** ASTD is used to call the ASTD **Controller**, which is responsible for managing the ventilator start-up process and managing the different breathing modes. The two states are linked by two transitions: the *power_on* transition from the **ON** state to the **StartUpCall** state, this transition refers to the action of starting the ventilator by pressing the **ON** button. The second transition *power_off* between the **StartUpCall** state and the **ON** state corresponds to the stopping of the ventilator by the user, as shown in Fig. 6. It allows for an

important action to be carried out, that of saving the main parameters in the configuration file before stopping the ventilator. The parameter values saved in this way will be used as parameter initialisation values the next time the program is started. This save is made by calling the *save_config* function, which is a utility function of our *Parameters* type that saves certain attribute values in the configuration file.

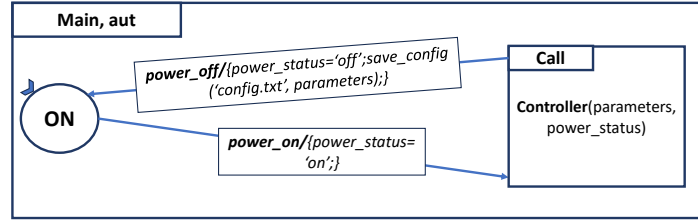


Fig. 6. Automaton ASTD Main

4.4 Controller ASTD

The controller ASTD is the element which contains the implementation of the internal operation of our controller, its overall structure was presented in Section 3. It is made up of a *StartFlow* ASTDs and two ASTDs of type call to *Actuators* and *Sensors*. *Actuators* and *Sensors* have been described above. Here we will focus on the *StartFlow* ASTD.

According to Fig. 4.1 of the specification document [5], the controller's state diagram comprises 5 main states: *StartUP*, *FailSafe*, *SelfTest*, *VentilationOff*, *PCVMode*, *PSVMode*. The transition from one state to another takes place in a precise order depending on the triggering of certain events or when the processing carried out on the previous state has been completed successfully. Fig. 7 shows our *StartFlow* ASTD, which is of the sequence type, i.e., an ASTD that allows for moving from one ASTD to another when the first ASTD has reached a final state.

The first ASTD type is a call to the *Startup* ASTD, which implements the ventilator startup process. Once the startup process has been successfully completed, the *StartFlow* ASTD allows it to switch to another sequence ASTD called *LaunchSelfTest*, which in turn allows it to combine in sequence a choice ASTD called *GoToVentilationOff* and an automaton ASTD called *ChooseMode*, which triggers the *VentilationOff* ASTD. This ASTD either executes ASTD *SelfTest*, represented by an ASTD call, before switching to the *VentilationOff* ASTD, or switches directly to the *VentilationOff* ASTD by executing the *resume_ventilation* transition in accordance with the specification. The ASTD *ChooseMode* permits,

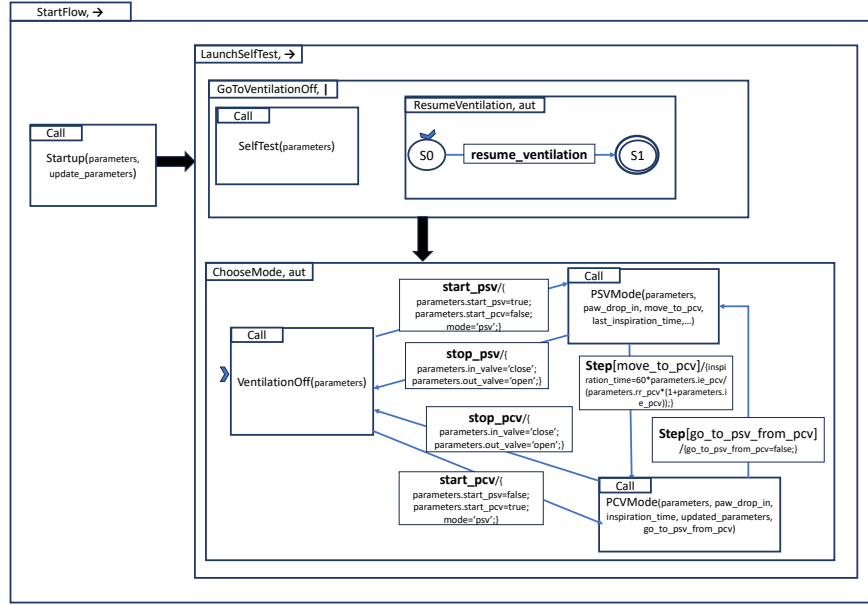
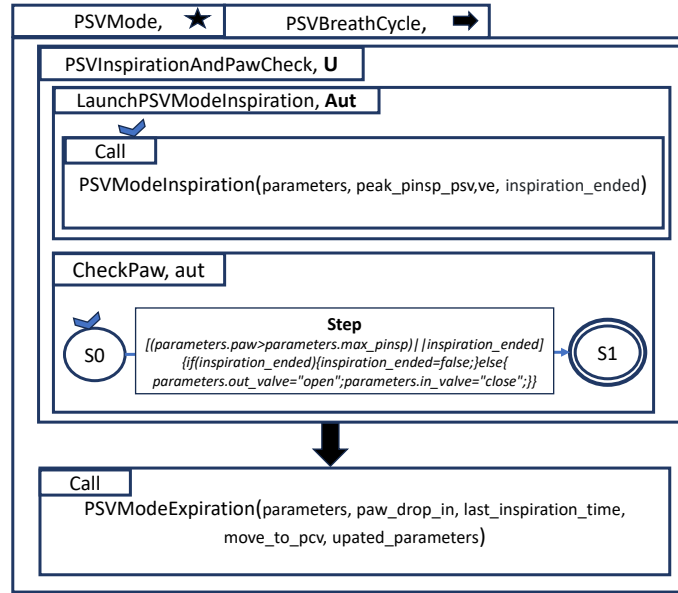


Fig. 7. StartFlow ASTD

after executing the **VentilationOff** ASTD as its initial state, to switch to the PCV mode, represented here by the call to ASTD **PCVMode**, by executing the *start_pcv* transition. It also permits to switch to the PSV mode, represented by the call to ASTD **PSVMode**, by executing the *start_psv* transition. The **Choose-Mode** ASTD can also be used to toggle back and forth between PSV and PCV modes via time-triggered **Step** transitions. For example, to switch from PCV-Mode to PSVMode, the user has to trigger the *start_psv* operation while in PCV mode, so that at the end of the inspiration time in PCV mode, we switch to PSV mode. This is shown in our model in Fig. 7 by the fact that the variable *go_to_psv_from_pcv* is equal to true.

One of the most important aspects of the case study, if not the most important, is the management of breathing cycles. Breathing is done in PCV mode or in PSV mode. Both modes have the same general operating scheme, which consists of a loop of breathing cycles. A breathing cycle generally consists of two stages: an inhalation stage and an exhalation stage.

Fig. 8 shows the proposed model for managing breathing in PSV mode. It is made up of six hierachically nested ASTDs. The first **PSVMode** closure ASTD allows us to model the infinite repetition of the breathing cycle. The second sequence type ASTD **PSVBreathCycle** enables us to model the notion of a breathing cycle by putting two ASTDS into sequence: the ASTD **PSVInspirationAndPawCheck** which implements the inspiration logic and a call to ASTD

Fig. 8. *PSVMode* ASTD

PSVModeExpiration which implements the exhalation management logic. The **PSVBreathCycle** ASTD can therefore be used to pass from inspiration represented by the **PSVInspirationAndPawCheck** ASTD to expiration represented by the call to the **PSVModeExpiration** ASTD, with the **PSVMode** ASTD simply allowing the cycle to be repeated. It should also be noted that the **PSVInspirationAndPawCheck** ASTD is a flow, which in certain cases allows it to skip the inspiration phase and trigger the expiration phase when $PAW > MAX\ PINSP$, a condition which is verified here by the second sub-ASTD of **PSVInspirationAndPawCheck** called **CheckPaw**.

4.5 Modelling Time Requirements

A critical aspect of the system is its temporal behavior. Many properties and constraints have explicit temporal requirements (like after 10 seconds ...) [5]. In TASTD, a clock tick is represented by the triggering of a **Step** event. To manage time constraints, it is therefore important to choose the time difference or time interval between two clock ticks, i.e. the time difference between two consecutive **Step** events. For this case study, we chose an interval value of 0.1 seconds. With this step value, we meet all the temporal constraints of our case study. The main temporal constraints are found in the process of managing respiratory cycles (inspiration and expiration): CONT.22, CONT.25, CONT.36.2, CONT.37, CONT.41.2, CONT.42.2, CONT.43.1, CONT.45.

After analysis, we can see that all these temporal constraints are either of ASTD type Timeout or Delay. A Timeout is a temporal constraints where we wait for an event to occur during a period of d units of time, and if the event does not occur, the ASTD stops waiting and goes on to execute another ASTD. A Delay ASTD waits for at least d units of time before accepting the first event. We can also express these constraints using automatic Step transitions guarded with conditions on timers. But for our model we have opted for Delay and Timeout for a more explicit representation of the timing constraints.

Fig. 9 shows our solution to the CONT.22 time constraint, which stipulates that in PCV mode, after a given inspiration time, which we will call *inspiration_time*, the controller checks whether the user has manually triggered the inspiration pause or whether he has triggered the recruitment maneuver. Here we'll use a Delay that waits for an *inspiration_time* delay before calling the *InspirationPhaseEnd* ASTD, which is supposed to check whether the user has triggered the inspiration pause or the recruitment maneuver.

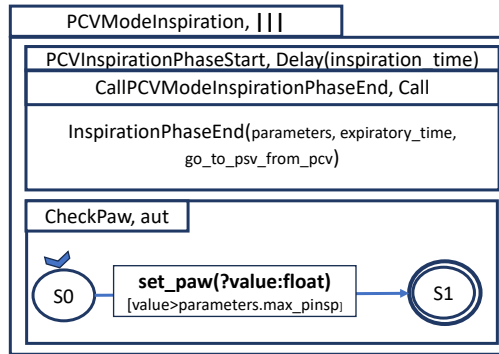


Fig. 9. PCVModelInspiration ASTD

Fig. 10 shows our solution for modelling the CONT.42.2 constraint. We have opted to use the Timeout operator via the *DetectExpiratoryPauseOff* ASTD of type Timeout, which allows the user to wait for *parameters.max_exp_pause* seconds before triggering an *expiratory_pause_off* transition contained in the *CheckExpiratoryPauseOff* ASTD. If the transition is not triggered, the switch to the *StopExpiratoryPause* ASTD is triggered to end the inspiration pause and, as *StopExpiratoryPause* has only one state, which is final, this final state status will allow the user to exit the *DetectExpiratoryPauseOff* ASTD and start the next inspiration.

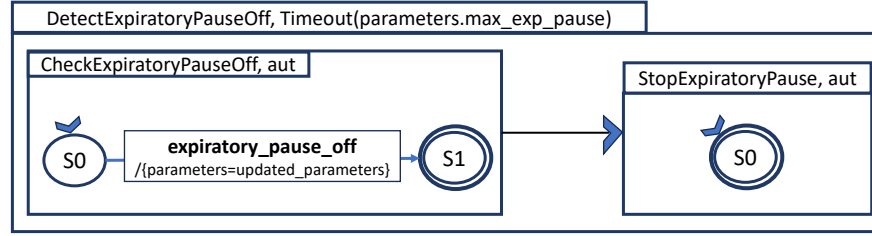


Fig. 10. DetectExpiratoryPauseOff ASTD

5 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 [7]. Our model satisfies the scenario given for the case study, although the proposed scenario only covers part of the controller's operating workflow. For the rest of the workflow we have generated test cases in compliance with the specification to validate the PCV mode and PSV mode steps which were not taken into account by the scenario proposed with the case study. We have implemented 12 test cases to test our specification: 5 test cases to simulate different scenarios in PSV mode, 6 test cases to simulate different scenarios in PCV mode and one test case to simulate an execution scenario from ventilator start-up to execution of the **VentilationOff** step in the ventilator operation workflow.

6 Specification Ambiguities and Flaws

In the ABZ2024 case study, specifically in the modelling of the controller component, we detected several cases of ambiguity or lack of information, particularly in the modelling of the **SelfTest** step (FUN.6, CONT.17, CONT.18, CONT.19) where there is no indication of how the controller is supposed to receive the results of the self-tests carried out by the controller and the GUI. There is also a lack of information, particularly at the level of the communication verification stage with the sensors, where CONT.15 states that a maximum of 5 connection attempts must be made with the pressure sensor and after these 5 attempts it is deduced that there is an error and the controller switches to **FailSafe** mode, but there is no mention of how long to wait before being able to attempt a new connection. We have the same situation for CONT.16 with the attempt to initialise the external ADC. To solve the problem we have assumed that this time for each case will be taken as a parameter of our controller.

7 Conclusions

To summary, we have presented a TASTD model for the Controller component of the ABZ2024 case study software system. Our model considers all the requirements. 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 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.

The development of models for the controller component, as well as their validation and documentation, required approximately two months. The modelling and analysis phase lasted almost 80 hours. The validation process took 40 hours, during which the model was updated to ensure that all the start-up stages of the system's ventilation were respected, that the system shut down safely for every error that occurred, to ensure that the switchover between PCV and PSV mode was carried out correctly and also that all the required time constraints were respected, as this was defined as one of the main critical aspects.

Currently, TASTD has a number of shortcomings, one of which is the size of the generated C++ code. Each ASTD is represented by a C++ structure, and nested ASTDs generate nested structures. Our specification is large and contains many deeply nested ASTDs, which induces very long prefixes to access variable x in the most nested ASTDs. During the code generation process, inefficient representation of these prefixes generates huge expressions on which several substitutions must be recursively applied. The problem could be solved by generating more modular C++ code. As a result, `cASTD` is currently unable to compile the main ASTD representing the entire controller system. We thus had to compile its sub-ASTDs independently and test them individually. In future work, we will be working on the generation of more compact C++ code by `cASTD`.

TASTD currently also lacks supports for verification. As future work, we intend to extend TASTD with Event-B[1] and CCSL[2]. These extensions will enable us to have in a single specification certain modules modeled in Event-B and others in CCSL or TASTD. These modules will enable us to translate TASTD specifications into Event-B or CCSL and thus take advantage of the verification tools offered by these two languages to be able to verify our TASTD specifications.

References

1. Abrial, J.: Modeling in Event-B. Cambridge University Press (2010)
2. André, C.: Syntax and Semantics of the Clock Constraint Specification Language (CCSL). Research Report RR-6925, INRIA (2009), <https://inria.hal.science/inria-00384077>
3. de Azevedo Oliveira, D., Frappier, M.: TASTD: A real-time extension for ASTD. In: Glässer, U., Campos, J.C., Méry, D., Palanque, P.A. (eds.) Rigorous State-Based Methods - 9th International Conference, ABZ 2023, Nancy, France, May 30 - June 2, 2023, Proceedings. Lecture Notes in Computer Science, vol. 14010, pp. 142–159. Springer (2023). https://doi.org/10.1007/978-3-031-33163-3_11, https://doi.org/10.1007/978-3-031-33163-3_11
4. de Azevedo Oliveira, D., Frappier, M.: Technical Report 27 - Extending ASTD with real-time. <https://github.com/DiegoOliveiraUDES/astd-tech-report-27> (2023)
5. Bonfanti, S., Gargantini, A.: The Mechanical Lung Ventilator Case Study. In: Rigorous State-Based Methods 10th International Conference, ABZ 2024, Bergamo, Italy, June 25–28, 2024, Proceedings, Lecture Notes in Computer Science, vol. 14759. Springer (2024)
6. Frappier, M.: ASTD support tools repo. <https://github.com/DiegoOliveiraUDES/ASTD-tools> (2023), [Online; accessed 26-January-2024]
7. Gargantini, A.: SCENARIO VERS 1.5. https://github.com/foselab/abz2024_casestudy_MLV/blob/main/scenarios_v_1_5.pdf (2023)
8. Ndouna, A.R., Frappier, M.: Case Study ABZ 2024 TASTD Model. <https://github.com/ndounalex/casestudyABZ2024-tastdmodel> (2024)
9. 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)