Formal Verification of Event-driven Health Applications

Mohammad-Reza Gholami¹,∗ and Hanifa Boucheneb²

¹,² Department of Computer and Software Engineering, Ecole Polytechnique of Montreal, Canada

Abstract—Software does an important task in almost every part of our everyday life, particularly in systems designed for the healthcare, aircraft navigation and automotive. One of the important objectives of software engineering is to support developers for building systems that function reliably even they are complex. In this paper, we show how Model-Based Design is used to model a safety-related application. By applying formal verification techniques, we also define specific properties to ensure that a software system satisfies its correctness criteria. We use the formal approach to study and verify the properties of a medical device known as Endotracheal intubation. The system is modeled in a concurrent manner and synchronization between components is done through events. We present how the system is modeled in the Simulink and Stateflow and present formalization of some safety and temporal requirements based on the events issued from the controller. In order to formally prove the defined properties, we employ Simulink Design Verifier toolset.

Keywords—Formal Methods; Formal Verification; Design Verification; Model-Based Design; Linear Temporal Logic; Safety-Critical.

I. INTRODUCTION

The increasing complexity of embedded systems (e.g. avionics, health and automotive systems) conveys the producers of safety-critical applications to use more systematic processes for development. Traditional design processes are not fast enough in discovering the errors in requirements; hence, the whole process would be longer and more expensive.

A key goal of software engineering is to construct systems that operate reliably regardless of their complexity [1]. A promising approach to achieve this goal is to use formal methods, which are mathematically based languages, tools and techniques for specifying and verifying such systems. Formal methods can significantly increase our understanding of a system by disclosing inconsistencies, ambiguities, and incompleteness in the early design phase so that they can be eliminated in order to ensure the appropriate behavior of the system.

Software vendors typically spend large amounts of time of their total development time and resources on software testing [2]. The severity of the defects also increases the detection time and the cost of total development budget for a software product. So there is a need for a sophisticated discipline of software engineering to ensure that all the expected requirements were satisfied by means of the specified software. In other words, software verification refers to the process that can determine whether the artifacts of one software-development phase fulfill the specified requirements produced during the previous phase.

In a safety critical system, even a single error in the source code and an associated malfunction of the system can cause high costs and could even endanger the health of people. In critical real-time embedded systems, verification and validation activities are becoming huge and quite costly when the complexity and size of the systems grows.

Formal verification can be performed by employing Model-Based Design [3] in order to specify formal model of the system. Model-Based design facilitates the addressing of difficulties and complexities existing in control system design by providing an executable specification which implies that the model exists as more than just a document, but as a functional part of the design process [4]. It also provides a single design environment that enables developers to use a single model of their entire system for data analysis, model

∗ Corresponding author can be contacted via the journal website.
visualization, testing and validation, and ultimately product deployment, with or without automatic code generation [5]. Furthermore, model-based design creates a structure for software reuse that permits established designs to be effectively and reliably upgraded in a more simplistic and cost effective manner.

In this paper, we study how to verify some safety and liveness requirements of an event-based system using Simulink Design Verifier. We use the formal approach to verify properties of an endotracheal intubation, which is a specific type of tracheal tube that is inserted through the patient’s mouth to maintain an open airway [6]. This medical device is illustrated in Figure 1. Using Simulink/Stateflow, we come up with a model with parallel components, where event passing and synchronization is efficiently provided.

![Figure 1. Endotracheal intubation](image)

The rest of the paper is organized as follows: Section 2 presents some background and previous work related to this research topic. Employed tools are described in Section 3. We present our case study in Section 4 which consists of definition, properties, and implementation of the model. The outcomes of our implementation are analyzed in Section 5. Finally, Section 6 concludes our paper.

II. BACKGROUND AND RELATED WORK

Formal methods enhance verification process by using formal notations and concepts in writing requirements and specifications. In formal methods, mathematical and logical techniques are used to express, investigate, and analyse the specification, design, documentation, and behavior of both hardware and software. The word formal in formal methods derives from formal logic and means “to do with form” [8]. In formal logic, dependence on human intuition and judgement is avoided in evaluating the arguments. In order to constitute an acceptable statement or a valid proof, formal logic employs a restricted language with very precise rules for writing assumptions, theorems, and proofs. In formal methods for computer science, languages are enriched with some of the ideas from programming languages and are called specification languages, but their underlying interpretation is usually based on a standard logic.

Formal verification in the field of software means the automated proof of specified properties on the code without executing the program. Also, it ensures that a design conforms to some precisely expressed notion of functional correctness [9]. The main benefits of formal verification in comparison to testing (dynamic verification), are its soundness and exhaustiveness. Specifications in formal methods are well-formed mathematical statements which are used to specify a property that needs to be verified in the system [10].

Many projects currently use MathWorks Simulink and Simulink Coder [11] which formerly known as Real-Time Workshop for at least some of their modeling and code development [12]. This kind of Design focuses on using executable system models as the foundation for the specification, design, implementation, test, and verification [13]. The executable specification in Model-based design, replaces parts or all of the paper format of the system specification and requirements as the main deliverable between design stages. It consists of an executable model of the application algorithm that can be simulated. The next step of Model-Based Design is known as model elaboration which consists of transforming the executable specification into a more design based form [14]. In Section 3, we briefly explain how to build an executable model by using Simulink and Stateflow.

Recently, some efforts have been made in order to employ formal methods in designing critical systems. In particular, Jiang et al. [15] developed a real-time Virtual Heart Model (VHM) for modeling the electro-physiological operation of proper functioning and malfunctioning. They introduced a timed-automaton model to define the timing properties of the heart and used Simulink Design Verifier as the main tool for designing their model.

Simulink/Stateflow has also been used in [16] to model a train tracking function for an automatic train protection system. The model was implemented based on the requirements specification document in which safety and functional properties were originally written in natural language. The authors of [16] used Simulink Design Verifier for verification and validation. They also had a positive experience when they used this tool for the safety-critical function in the railway transportation domain.

Another case example for a medical device has been presented in [17] where an iterative approach is applied for system verification based on software architectural model. They employed Simulink/Stateflow for describing the component level behavior of the model and used Simulink Design Verifier for proving the system level properties to establish component-level properties.

Above mentioned works used Simulink Design Verifier for proving safety properties which is supported by this tool in nature. In this paper, authors...
employ Simulink/Stateflow to model a system having different components. In addition, components of the system are designed to act in parallel and synchronization between them is accomplished by events. Furthermore, some liveness properties are also formalized in the model to describe our method for proving this kind of properties. For the formal analysis, we use Simulink Design Verifier, which intensively employs the BMC and K-Induction features of the PROVER[18] engine to establish the satisfiability of the proof objectives. We also use this tool to verify design issues like Integer overflow, Division by zero, Assertions and Violations of design properties to eliminate runtime issues of the model.

III. SIMULINK AND STATEFLOW

The executable model can be built by Simulink which is an environment for multi-domain simulation and Model-based Design for dynamic and embedded systems. Mode logic in Simulink models is described in terms of hierarchical state machines specified in a variant of Statecharts called Stateflow [11].

Stateflow is a widespread model-based development environment is Matlab/Simulink toolset, which is used in several industries, such as aerospace, medical, and automotive. It uses a variant of the finite state machine notation established by Harel [19] and provides the language elements required to describe complex logic in a natural, readable, and understandable form. Since it is tightly integrated with MATLAB and Simulink, it can provide an efficient environment for designing embedded systems that contain control and supervisory. In particular, Stateflow diagram enables the graphical representation of hierarchical and parallel states as well as transitions between them and inherits all simulation and code generation capabilities from Matlab toolset.

A state is called as superstate when it contains other states and a state is called substate when it is contained by a supersate. When a state consists of one or more substates, it has decomposition that can be either parallel (AND) or exclusive (OR) decomposition. All substates at a particular level within the same state must have the same decomposition.

In parallel (AND) decomposition, states can be active at the same time and the activity of each parallel state is essentially independent of other states.

We can use our defined Events to trigger actions in parallel states of a Stateflow chart. Broadcasting of an event can trigger a transition and/or an action. The actions can be executed either as a part of a transition from one state to another or based on the activity status of a state which can be entry, during, exit, and on event actions. For instance, while the state Fill is active, c = GetElapsed() is executed every time unit.

The general form of a transition in Stateflow is presented in Figure 2. It shows the behavior of a simple event, condition and transition action specified on a transition from one exclusive (OR) state to another. Initially, state Start is active and entry action is executed, which sets the variable tempo to 15. When the event eFill is received, the chart root detects that there is a valid transition to state Fill as a result of the event eFill, so it validates the condition and if the result is true, the Condition Action immediately gets executed and completed. Conversely, the state Start remains active and no Condition Action executes if the condition is false. The state Start is marked as inactive and the Transition Action is executed and completed when the transition destination Fill has been determined to be valid. States can have different actions such as: entry, during, exit, and on event-name which are being executed based on the current status of the active state.

A. Simulink Library

To build models in Simulink, blocks are the main elements that are used, and they are hosted in the library. A Simulink block has sets of input and output ports. A block with N input and M output ports defines a function which describes each of the signals at the output ports as a (possibly time-dependent) expression of the signals at the input ports. Formally, a block is a tuple $(P_i, P_o, f)$, where $P_i$ is the set of input ports, $P_o$ is the set of output ports and $f : \mathbb{R}^M \rightarrow \mathbb{R}^N$ is a function which defines the behavior of the block [20]. In the following, some blocks from the standard Simulink library is briefly described:

1) Embedded Matlab Function block

The Embedded MATLAB Function Block facilitates writing the MATLAB m-code which can be incorporated into a Simulink model. This block is placed in the User Defined Functions Library and can be inserted into a model in the same way as any other Simulink blocks. We use this block whenever there is a need to implement part of the logic of the property by program code.
2) **Subsystem block**

A subsystem is a set of blocks that we replace with a single block called a Subsystem block. As our model increases in size and complexity, we can simplify it by grouping blocks into subsystems.

3) **Function-Call Subsystem**

This block represents a subsystem that can be invoked as a function by another block. In other words, a function-call subsystem is a subsystem that another block can invoke it directly during a simulation. It is similar to a function in a procedural programming language. Invoking a function-call subsystem is equivalent to invoking the output methods of the blocks that the subsystem contains in sorted order. The block that invokes a function-call subsystem is called the function-call initiator. Moreover, the Stateflow, Function-Call Generator, and S-function blocks can all serve as function-call initiators.

4) **Temporal Logic Operators**

These operators are used in the Stateflow and control the execution of a chart in terms of time. In state actions and transitions, you can use two types of temporal logics: event-based and absolute-time. Event-based temporal logic keeps track of recurring events, and absolute-time temporal logic defines time periods based on the simulation time of your chart.

For event-based temporal logic, the following operators \textit{after}, \textit{before}, \textit{every} and \textit{temporalCount} can be used in the Stateflow. For instance, in the provided model for our case study, the operator \textit{after} is used in the \textit{Timers} and \textit{Cylinders} states.

**B. Simulink Design Verifier**

Simulink Design Verifier [21] is a tool set of Matlab which uses formal methods to identify hard to find design errors in the models without requiring extensive tests or simulation runs. Moreover, it enables us to perform model analysis within the Simulink environment, in order to verify the designs and validate the requirements early, without having to generate code.

We can use Simulink, MATLAB functions, and Stateflow to express formal requirements. It also provides a set of building blocks and functions that can be used to define and organize verification objectives. The block library provided, includes blocks and functions for test objectives, proof objectives, assertions, constraints. In addition, a dedicated set of temporal operators like Detector, Extender and Within Imply blocks are also provided in order to model the verification objectives with temporal aspects. Following to this, Within Imply block that is used in our implementation is briefly described:

1) **Within Imply block**

The Within Imply block captures the within implication by observing whether the \textit{Obs} input is true for at least one step within each true duration of the first input \textit{In}. Whenever \textit{Obs} is not detected within a particular input true duration, the output becomes false for one time step in the step that follows the input true duration. This block captures the behaviour: \textit{(Within} \textit{In}) \Rightarrow \textit{Obs}.

In the example illustrated in Figure 3, model sample time is considered as 1 second.

- In Figure 3a, although \textit{Obs} is observed within the first true duration of \textit{In} (steps 1...4), but it is not observed within the second true duration of \textit{In} (steps 5...10), so \textit{Out} becomes false for one time step after the \textit{In} signal becomes false (step 11).
- In Figure 3b, \textit{Obs} is not observed within the first true duration of \textit{In}, so \textit{Out} becomes false for one time step after the \textit{In} signal becomes false.

![Figure 3. Within Imply Simulink Block](image-url)
• In Figure 3c, Obs is observed within the true duration of In (at time step 4), so Out remains true until the end of the simulation. The input is true if Obs becomes valid for at least one time every true duration of the input.

IV. CASE STUDY

This case study aims to show and familiarize how a system works with a framework where time aspects are combined with multi task programming. In this case study, we are modeling and verifying the properties of a Filling system of balloons of an intubation probe. An intubation probe is placed to ensure continuous passage of air to the lungs and introduce oxygen sensors, aspiration probes to the lung for patient treatment.

As illustrated in Figure 4, this system consists of two balloons, two access valves for manual inflation, two pressure sensors, a power distributor, a pump and an air tank. The pump is actuated by a gear motor and a transmission by a cylinder rack. The pumped air is propelled through the power distributor (B and D) to one of the balloons or outside. The probe has several buttons (Start, Stop, Duration, Pressure, StopAlarm) and LEDs (L1, L2, Alarm). The Alarm LED reports the anomalies. L1 and L2 are witnesses indicating the inflated balloons, and the button StopAlarm allows the user to stop the alarm. The system controlled by a Programmable Controller who is responsible for controlling the commands and messages which are sent to or received by other components. We are using Simulink and Stateflow as an integrated tool environment for modeling, and Simulink Design Verifier for verification of some properties.

A. The model

A model is known as abstract representation of a system. Software model is actually the ways of expressing a software design, and in order to express the software design some kind of abstract language or pictures are usually used. Software modeling needs to deal with the entire software design, including interfaces, interactions with other software, and all the software methods [22]. Engineers can model the system using a modeling language which it can be graphical or textual [23].

According to the description of the case study, the first step when modeling the Intubation, is to pinpoint the superstates in the system and their interactions. One of the most important parts of the design is to find out which superstates should be parallel (AND) and which ones should be exclusive (OR).

In the Intubation Stateflow, there are ten distinguished blocks corresponding to each component, which are illustrated in Figure 5. All of these blocks are working in a parallel execution order. These blocks are represented with ten different states in the model. In every moment of running the model, at least one substate has to be active in each state. These states are: Controller, Distributor, Cylinder, Balloons, Alarm, Lights and Timers. These states are designed to be parallel(AND) because a change in these states is allowed at every time step.
The states corresponding to each component, interact together through sending direct broadcast event and one simplified function (Initialize) to make the model smaller, initialize variables and the status. In other words, using function helps to group the different actions that are associated to each transition. Direct event broadcasting is used to prevent receiving an error pertaining to recursion in the Stateflow chart. The temporal logic operator after is also used whenever there was a need to control the execution of states in terms of time. In figures, some defined functions in the Stateflow are removed for the sake of simplicity. To explain and describe the model, this section is divided into three different parts as follows:

- **Inputs & Local variables, ranges and values.**
- **Events.**
- **Components (Parallel (AND) States).**

In the following, we give a short description of components, including some screenshots taken from the Simulink to complement the description of the Stateflow.

**Variables:**

We maintain the current state and values of the components using local variables. For simplification, we used integer values to represent the corresponding physical step of the components. The input type variable corresponds to the variable whose value is coming from the Simulink model. Conversely, the output type variable is modified and used in the Stateflow, and it is accessible through the Simulink model. Both input and output type variables, their range, and default values are defined and set in the declaration of the stateflow, for example, for the cylinder component:

$$\text{int nCylPos} = -1$$

For this specific variable, the value can be set to either -1, 0 or 1 which respectively correspond to the position of the cylinder as: rear, center and front. Similarly, the different values for nBalloonState correspond to different status of the balloon, such as: Empty, NotFull and Full. The target pressure of the balloon in which the balloon is considered as full inflated, is stored in nPressure. The variable nDuration, sets the amount of the time that an inflated balloon should remain full before controller sends a deflation command. Some input and output variables for the Stateflow diagram listed in Table 1.

**Table 1. Variables in Intubation Stateflow**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>nCylPos</td>
<td>Output</td>
<td>-1, 0, 1, 2</td>
</tr>
<tr>
<td>nBalloonState</td>
<td>Output</td>
<td>-1, 0, 1</td>
</tr>
<tr>
<td>bLightState</td>
<td>Output</td>
<td>true, false</td>
</tr>
<tr>
<td>bAlarm</td>
<td>Output</td>
<td>true, false</td>
</tr>
<tr>
<td>bStopAlarm</td>
<td>Input</td>
<td>true, false</td>
</tr>
<tr>
<td>bStart</td>
<td>Input</td>
<td>true, false</td>
</tr>
<tr>
<td>nPressure</td>
<td>Input</td>
<td>10, 20, 30 mins</td>
</tr>
<tr>
<td>nDuration</td>
<td>Input</td>
<td>12, 18, 24</td>
</tr>
</tbody>
</table>

**Events:**

Different events were defined to model the communication between different components of the system. According to their usage, they are defined in the Stateflow as Directed Event Broadcast. The relationship between events and the corresponding
component, is listed in the Table 2. The status column illustrates if the event is sent to components of the chart or is sent out from the Stateflow to the Simulink model of the system. As such, the event \textit{exFillB1} means that the event will be sent out from \textit{Distributor} component whenever a Fill attempt is required to inflate balloon 1. Similarly, the external event \textit{exEmptyB1} is sent out from \textit{Distributor}, for every deflating requests. In addition, the event \textit{exFillB1} is also sent from \textit{Balloon1} component out from the Stateflow whenever balloon 1 is completely inflated.

### Table 2. Events

<table>
<thead>
<tr>
<th>Component</th>
<th>Events</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alarm</td>
<td>cStartAlarm, cStopAlarm</td>
<td>Internal</td>
</tr>
<tr>
<td>Light1</td>
<td>eLight1On, eLight1Off</td>
<td>Internal</td>
</tr>
<tr>
<td>Light2</td>
<td>eLight2On, eLight2Off</td>
<td>Internal</td>
</tr>
<tr>
<td>Distributor</td>
<td>eFill, eEmpty, cxFillB1, cxFillB2</td>
<td>Internal, External</td>
</tr>
<tr>
<td>Cylinder</td>
<td>vPlus, vMinus</td>
<td>Internal</td>
</tr>
<tr>
<td>Balloons</td>
<td>exB1Full, exB2Full</td>
<td>External</td>
</tr>
</tbody>
</table>

**Stateflow:**

This section describes each component that we have modeled as different states. We explain the nature and the interaction of each one of them.

1) **Controller**

The model of the controller was realized from the specification based on the provided Grafcet [24]. In order to model the controller some local variables representing the steps, transitions and actions and some local variables representing the channels and reflecting the value of local actions in the system were also defined. The local variables \textit{B} and \textit{D} are defined as \textit{boolean}, and being used to set the channels and activate them (\textit{B} and \textit{D} are shown in Figure 4).

The entire controller is included in a single superstate. In order to do a specific action, the controller set the values for the channels and send the event to the \textit{Distributor} state. Table 3 presents events and channels used by the controller for specific actions.

Instead of modeling the user’s interaction with the system, we use local variables to simulate the choice of target pressure and the targeted cycle time (\textit{nPressure} and \textit{nDuration}). Those values are set at the beginning and are left untouched during the simulation. Our model assumes that at the beginning the cylinder is in the position \textit{rear} and both balloons are considered \textit{Empty}.

One of the major building blocks of our controller which is responsible to fill and empty the balloons is illustrated in Figure 6. In this figure, in order to inflate the first balloon, on the entry action of the state \textit{S2} corresponded values for \textit{B} and \textit{D} are set then the event \textit{eFill} is sent to the \textit{Distributor}.

2) **Cylinder**

The cylinder has three substates: \textit{rear}, \textit{center} and \textit{front}. This state is constantly waiting to receive the events \textit{vPlus} or \textit{vMinus} from the controller to change its position forward or backward. To model the 2 seconds delay for each position transition, we included an in-between location, and used an \textit{after} temporal operator between each position. Once the delay is exhausted, the cylinder position changes. We use a local variable \textit{nCylPos (rear = -1, center = 0, front = 1}}
and In-between = 2) to store the current position of the cylinder. Initially, the state rear is active. We set this variable to 2 when the cylinder is in transition between two positions. The controller has guards using the transitional value to ensure the cylinder state has completed its movement.

3) Pressure Distributor

This state receives the specified event from the controller in order to launch the selected action for inflating or deflating the desired balloon. The selected action is relevant to the current value of the local variables B and D which are set to true or false by the controller before sending the event eFill or eEmpty. In addition, to complete the entire function, it also sends specific events to states Cylinder and Balloon for their relevant actions. Table 3 shows the events and variables used for specific functions.

<table>
<thead>
<tr>
<th>Table 3. Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Distributor Channel</strong></td>
</tr>
<tr>
<td>B=0, D=1</td>
</tr>
<tr>
<td>B=1, D=0</td>
</tr>
<tr>
<td>D=0 or B=0, D=0</td>
</tr>
<tr>
<td>B=0, D=1</td>
</tr>
<tr>
<td>B=1, D=1</td>
</tr>
<tr>
<td>B=0, D=0 or B=0, D=0</td>
</tr>
</tbody>
</table>

Initially, the state Init is active and waits to receive a specific event from Controller to complete the selected function. The state FillB1, on the rightmost side of Figure 7 has to run iteratively until the destined balloon is inflated. Similarly, the emptiness of the balloon is ensured by running the state EmptyB1 on the leftmost side of this figure.

The state of this component is formally defined by sets of states, actions and transitions. First thing to remember is that the composition of states can be either a Parallel AND or an Exclusive OR composition.

Outgoing transitions for Distributor state as illustrated in Figure 8, are defined as:

\[ TR = \{ tr_1, tr_2, tr_3, tr_4, tr_5, tr_6, tr_7, tr_8, tr_9, tr_{10}, tr_{11}, tr_{12} \} \]

A state definition sd, is a triplet composed of actions \( A = \{ entry, during, exit \} \), executed respectively upon entering, during, and exiting the state, an internal composition (Parallel AND, Exclusive OR), and a list of outgoing transitions [25].

The state definition list SD [25], which associates state definitions sd with corresponding substates in the Distributor, is denoted as:

\[ SD = \{ Dist : sd_0; Init : sd_1; EmptyB1 : sd_2; FillB1 : sd_3; EmptyB2 : sd_4; FillB2 : sd_5 \} \]

The definition of the states for the Distributor depicted in Figure 7, can be listed as:

- \( sd_0 = (AND) \)
- \( sd_1 = (OR, \{ tr_1, tr_3, tr_4, tr_5 \}) \)
- \( sd_2 = ((A,e), OR, \{ tr_2, tr_3 \}) \)
- \( sd_3 = ((A,e), OR, \{ tr_2, tr_5 \}) \)
- \( sd_4 = ((A,e), OR, \{ tr_6, tr_7 \}) \)
- \( sd_5 = ((A,e), OR, \{ tr_8, tr_9 \}) \)

The sample trace information represented in Table 4, illustrates the status of events and evolution of states over the time, when the controller tries to inflate the first balloon which is empty. Values of B and D channels specifies the current requested function to the pressure distributor component. During the process some events are handled or triggered by Distributor component and is denoted as follows: Event(Recv) is an internal event that is issued from the controller to distributor component to tell which of these functions (Inflate, Deflate, Move Cylinder) is requested. Event(Send) is an internal event that will be sent from Distributor state to Cylinder state to perform the movement. Event(Exit) is sent from Distributor state out from the Stateflow chart. This event later will be handled by other subsystem block in the Simulink model.
4) Balloons

The state balloon has three substates: Empty, NotFull and Full. In this model, we consider two different superstates corresponding to each balloon. Initially, the state Empty is active in both balloons. The transition between substates has a guard; so, the movement is done when one of the events eFill or eEmpty is received from the state Distributor. We use a local variable nBalloonState (Empty = -1, NotFull = 0, Full = 1) to store the current status of the balloon.

Outgoing transitions for Balloon1 state as illustrated in Figure 8, are defined as:

\[ TR = \{ tr_1, tr_2, tr_3, tr_4, tr_5, tr_6, tr_7 \} \]

The state definition list SD, which associates state definitions sd with corresponding substates in the Balloon1, can be denoted as:

\[ SD = \{ Balloon1 : sd_0, Empty : sd_1, NotFull : sd_2, Full : sd_3 \} \]

The definition of the states for the Stateflow depicted in Figure 8, can be listed as:

- \( sd_0 = (AND) \)
- \( sd_1 = ((Ae), OR, \{ tr_1, tr_2 \}) \)
- \( sd_2 = ((Ae), OR, \{ tr_3, tr_4, tr_5, tr_6 \}) \)
- \( sd_3 = ((Ae), OR, \{ tr_7, tr_8 \}) \)

5) Timers

In our model, timers are designed as two different components: The InflateTimer which is responsible for the inflation time of a balloon, and the HoldTimer which is the time that a balloon should maintain the status Full.

The state InflateTimer contains three substates: State Wait which is initially active, the state Start which is activated by the controller while requesting for a Inflate procedure, and Stop that gets activated whenever the maximum time is reached. The temporal logic operator after is used as a transition condition from state Start to state Stop. The InflateTimer is illustrated in Figure 9:

\[ TR = \{ tr_1, tr_2, tr_3, tr_4 \} \]

Outgoing transitions for the inflateTimer state as illustrated in Figure 9, are defined as:

\[ TR = \{ tr_1, tr_2, tr_3, tr_4 \} \]

The state definition list SD, which associates state definitions sd with corresponding substates in the inflateTimer can be denoted as:

\[ SD = \{ inflateTimer : sd_0, Wait : sd_1, Start : sd_2, Stop : sd_3 \} \]

The definition of the states for inflateTimer that is depicted in Figure 9, can be listed as:

- \( sd_0 = ((Ae), AND) \)
- \( sd_1 = (OR, \{ tr_1, tr_2, tr_3 \}) \)
- \( sd_2 = (OR, \{ tr_4, tr_5 \}) \)
• \( sd_3 = (OR, \{tr_3, tr_4\}) \)

6) Alarm

The alert state contains two substates: Off and On. Initially, the state Off is active. When the timer exceeds from the specified threshold or if any anomaly happens, the controller stops the system operation and sends the event eAlarm. Once the event received, the state On will be activated and remains in this state until the user stops the alarm.

V. METHODS AND ANALYSIS

This section describes our method and results of formal verification using Design Verifier with Simulink and Stateflow. Before verification, we run the simulation for our provided model using predefined input parameters in order to ensure that the model can be executed properly.

A. Properties

The term property refers to a logical expression of signal values in a model. For example, we can specify that a signal in a model should attain a particular value during execution of the system. The Simulink Design Verifier software can then prove the validity of such safety properties. This is done by performing a formal analysis of the model to prove or disprove the specified properties. If the software disproves a property, it provides a counterexample that demonstrates a property violation.

The developer can specify properties by using two blocks provided in the Simulink Design Verifier library. The Proof Objective block is used to define the values of a signal that the Simulink Design Verifier software will prove. The Proof Assumption block is used to constrain the values of a signal during a proof [23].

The definition of properties comes with the execution order of contained blocks (e.g. a, b, c, ...). The update functions of each block in the property gets executed respectively after other blocks in the main model. The modeled system consists of the following properties:

1. A balloon should be inflated in less than or equal three fill attempts.
2. The fill command remains active until corresponding balloon is inflated.
3. There must be no anomaly alarm (False alarms).
4. The pressure in each balloon never exceeds a predetermined value.

The term \( Pre(x,t) \) is used for Predecessor in provided equations when defining specified properties (It corresponds to Unit Delay block in Simulink library). As such, The predecessor of signal \( x \) at time \( t \) is denoted by \( Pre(x,t) \) that corresponds to the value of that signal at time step \( t-1 \). The term \( Pre'(x,t) \) is also used as the transitive closure of \( Pre(x,t) \). The following section details each property and the results obtained by Simulink Design Verifier:

1) Property 1

The goal of this property is to verify the number of fill attempt events that are sent by the controller, in an inflate procedure, meets the number specified in the requirement specifications. Consider the scenario that the controller needs to inflate balloon 1 at the beginning of the system run. At start, we assume that the balloon is empty (has 6 cm of water) and the cylinder is at rear position, and each Fill attempt increases 6 cm of water to the balloon's pressure. If the target pressure is set to 18 cm of water, the controller needs to issue 2 fill attempts in the inflation procedure.

Figure 10, illustrates the implementation of this property in Simulink. The Distributer component sends out an external event from the Stateflow whenever a fill attempt is issued.

![Figure 10. Formalization of property 1](image)

The Function-Call Subsystem is a triggered subsystem and increases the count of the attempts whenever an event is received.

In Figure 10, the Logical Operator \( LO1 \) validates \( isFill \) and \( isRun \) input signals, and ensures that 'b' becomes false as soon as the inflating procedure is terminated. The Relation Operator \( RO1 \) validates the count input against the desired value. Afterwards the Within Implies block checks if the input 'c' was true for at least one time step during the time steps that 'b' is true.

In Figure 11, as per definition of Within Implies block shown in Eq. (4), it captures the within
implication by observing whether the input ‘c’ is true for at least one time step within each true duration of the first input ‘b’.

Figure 11. Internal functions in property I

Formalization: Let M be a Simulink model, \( \pi = s_0, s_1, s_2, \ldots \) be a sequence of states, and property I denoted by \( p \):

\[ M \models p \iff G \left( b(t) \Rightarrow c(t) \right) \]

Proof: The block function of this property can be written as:

\[ f_d(b, c, t) = \]

\[
\begin{cases} 
1 & \text{if } \text{count} = \text{desired} \\
0 & \text{if } \text{count} \neq \text{desired} 
\end{cases}
\]

(3)

Finally, \( d \) as the output of this property, must hold true during the entire execution of the system and corresponds to the result of \( b(t) \Rightarrow c(t) \).

Simulation and trace of the property \( p \) is done by specifying the time offset \( T_o = 0 \) and sample time period \( T_s = 1 \text{sec} \). In the following, the active state in Cylinder component as well as output variables of the Stateflow are shown for these situations:

- If cylinder is at rear position.
- If cylinder is at center position.

Table 5, denotes the evolution of variables over the time when the cylinder is at rear position. We assume that the inflate procedure for balloon 1 is started at time step \( i \) by the controller. Initial values are also shown in the table at time 0.

As per design of the controller, when a fill command is issued but if the cylinder position is in front position, the controller holds the fill command, sends backward command to the cylinder and re-issues the fill command and sends the external event. Table 6, displays the evolution of the variables over the time when the cylinder is at center position.

Figure 12. Using Standard Simulink Block
In terms of block execution sequence, the execution order of Implies block is after ‘Intubation’ chart, and Proof Objective block is after the Implies block. Thus, in each time step of the execution, inputs of the Stateflow are evaluated first and then its outputs are updated and fed to the specified property including the Implies block.

The implementation of the function $f_a$ in Figure 11 is illustrated in Figure 12.

To explain, the purpose of using the Unit Delay block is to prevent causing the Algebraic-loop. As depicted in Figure 12 the evaluation of the Sum block is done in each sample time, so direct feedback the output of the Sum block to one of its inputs causes an algebraic loop and reports an error in compile time, so there is a need to use Unit Delay block which has an internal state and stores the previous input. The initial state of this block is set to zero.

We also proposed another implementation for the function $f_a$ to investigate the verification time of this property when two different implementations are used. To address this, we employed an Embedded Matlab function which has the same functionality and increments the output count as soon as receiving the external event from the controller. Figure 13a represents the subsystem's internal blocks along with its corresponding code.

| Table 5. Evolution of variables over the time – Cylinder at rear |
|---------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| Cylinder | rear | rear | rear | front | center | front | center | center | center |
| isRun  | 0    | 0    | 0    | 1     | 1     | 1     | 1     | 1     | 1     |
| isFill | 0    | 0    | 0    | 1     | 1     | 1     | 1     | 0     | 0     |
| event  | -    | -    | -    | 1     | -     | 1     | -     | 0     | 0     |
| count  | 0    | 0    | 0    | 0     | 1     | 1     | 2     | 2     | 2     |

| Table 6. Evolution of variables over the time – Cylinder at center |
|---------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| Cylinder | center | center | center | front | center | front | center | center | center |
| isRun  | 0    | 1    | 1    | 1     | 1     | 1     | 1     | 1     | 1     |
| isFill | 0    | 0    | 1    | 1     | 1     | 1     | 1     | 0     | 0     |
| event  | -    | -    | 1    | -     | 1     | -     | 1     | -     | -     |
| count  | 0    | 0    | 0    | 0     | 1     | 1     | 2     | 2     | 2     |

![Figure 13. Function-Call Subsystem Block](image)
2) **Property 2**

The goal of this linear temporal property is to make sure when a controller initiates an inflation procedure, the fill signal remains enable until the corresponding balloon is inflated completely. As per design of the controller, when the state corresponding to the inflation of first balloon is activated, it activates the Filling signal and send the appropriate events to the Distributor and Cylinder respectively in order to complete the inflating process. Since in each movement of the Cylinder component adds a specific pressure (6 cm of water) to the balloon, this sequence should be executed more than one time, based on the target pressure that is defined for the balloons in the configuration (18 cm of water for an inflated balloon).

![Figure 14. Property 2](image)

Figure 14, illustrates the implementation of this property. The input Running is used to bound the time steps of the verification to the time that the system is running.

As illustrated in Figure 15, this function can be divided into four different sub-functions that are denoted as:

- \( f_x(Filling,t) \)
- \( f_x(balloon\_stat,t) \)
- \( f_x(Running,t) \)
- \( f_x(a,b,c,t) \)

![Figure 15. Internal functions in property 2](image)

The function \( f_x(a,b,c,t) \) is constructed by using an Embedded Matlab function and its output is updated at each time step \( t \) after the preceding functions a, b, and c. The output of this function over the time is represented in Eq. (5). Moreover, whenever the Running signal is true and when the Filling signal is true, it must be remain true until the BalloonStat1 signal becomes true (Full = 1). The code corresponding to MATLAB Function 2 is provided in Appendix 1.

**Formalization:** Let \( M \) be a Simulink model, and property 2 denoted by \( p \):

\[
M \models p \iff G (f_x(a,b,c,t) = true)
\]

**Proof:** The block function of this property can be written as: \( f_x(a,b,c,t) \), where the output should hold true during the execution at any time step \( t \). This property has three inputs and one output which is connected to a Proof Objective block (P-block).

\[
f_x(a,b,c,t) = \begin{cases} 
0 & \text{if } (¬c_{(t)} \land ¬Pre(b,t) \land ¬b_{(t)}) \\
1 & \text{otherwise} 
\end{cases}
\]

3) **Property 3**

The goal of this property is to assure the model does not generate any false alarms. To address this property we have designed a statement that validates the opposite condition and applied a 'not' to it.

This property ensures that the controller never ends up in a state where we have \('Filling == true'\) (which means the process is undergoing) and we have the alarm light on without having detected any anomaly. The anomaly detection always sets the variable 'bError' to true. Hence, if 'bError' is set to false, means there is no anomaly and the state 'Alarm.Off' in the controller is active.

Figure 16, illustrates the implementation of this property. The input Filling is used to bound the verification to the time that the system is in the balloon inflation process.
Formalization: Let M be our Simulink model, and property 3 is denoted by p:

\[ M \vDash p \iff G \text{true} \]

**Proof**: The block function of this property can be written as:

\[ f_d(a,b,c,t), \text{ where its output should hold true during the execution of the program at each } t \text{ time steps. This property has three inputs and one output which is connected to a Proof Objective block.} \]

As shown in Figure 16, the output of Embedded Matlab function at each time step, is also the result of the property and should be hold true at any time step t. The value of \( d \) over time in the Figure 16, corresponds to the output of the function \( f_d(a,b,c,t) \), which is already declared in Eq. (5). The function \( f_d \) is constructed by using an Embedded Matlab function and its output is updated at each time step \( t \) after the preceding blocks (AND, RO1 and RO2).

4) **Property 4**

The goal of this property is to validate that the model do not permit the pressure within the two balloons to exceed the maximum value configured by the user. This property can be modeled in simulink as represented in Figure 17.

**Figure 16. Property 3**

**Figure 17. Property 4**

We have modeled this property by performing a simple comparison between the balloon pressure and the target pressure when the status of balloon is reported as full by the controller.

\[ f_t(a,b,t) = -a_i(t) \lor b_i(t) \]  

\[ (6) \]

The Relational Operator block RO1 validates if the balloon status is full by comparing the input value BalloonStat1 to the constant FULL \((-1 = \text{empty}, 0 = \text{notfull}, 1 = \text{full})\). In addition, RO2 validates if the pressure reported by the controller at the time step \( t \) is equal to the configured target pressure. The output value of the Imply block over the time should be always true to prove that the property is satisfied. The function corresponding the this block can be represented as Eq. (6). This property also has two inputs, and its output is updated at each time step \( t \) after the preceding blocks (RO1 and RO2).

**B. Verification**

Formal verification of specified properties are done on a computer having an Intel Core i7 CPU with 6GB of RAM. We employed Matlab version 2013b and Simulink Design Verifier toolbox is also used as the verification engine.

Table 7 depicts the result of the property proving for different properties with their parameters. Since we had two different implementation for the Function-call Subsystem block in property I, they are denoted as 1 and 2 in the result table. (Implementations are shown in Figures 12,13a).

As can be seen in results illustrated in Table 7, the verification time varies in different properties. Some has shorter and some has longer analysis time. The column Elapsed in results table, specifies the time that verification engine spends to find appropriate test cases to prove or disprove the corresponding property. The column Params & Conditions in the table denotes parameter values and conditions that are set to verify a particular property.

For instance, based on the design of the system, a cylinder has a movement lag which is defined as 2
seconds by default. In addition, each cylinder movement increases the pressure equal to 6 cm of water, and if the target pressure for balloon is defined as 18 cm (Empty balloon has 6 cm), the cylinder needs to move two times. Every request for inflate a balloon should be done within 15 seconds which is defined in InflateTimer. As such, if the cylinder lag set to 7 seconds, the total inflation time including time spent for other states exceeds 15 seconds, as a result the property becomes falsified.

In fact, complex properties spend more analysis time than others. On the other hand, using the Proof Assumption block and defining the appropriate parameter value, is one of the ways that can reduce the verification time.

### Table 7. Verification Results

<table>
<thead>
<tr>
<th>Property</th>
<th>Params &amp; Conditions</th>
<th>Elapsed</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1_1</td>
<td>Fill attempts &gt; 2</td>
<td>17, m, 42</td>
<td>Failed</td>
</tr>
<tr>
<td>1_2</td>
<td>Fill attempts = 2</td>
<td>17, m, 43</td>
<td>Satisfied</td>
</tr>
<tr>
<td>1_3</td>
<td>Fill attempts &gt; 2</td>
<td>15, n, 14</td>
<td>Failed</td>
</tr>
<tr>
<td>2</td>
<td>Cylinder lag=7 sccs</td>
<td>1, n, 3, 24</td>
<td>Satisfied</td>
</tr>
<tr>
<td>2_2</td>
<td>Cylinder lag=4 sccs</td>
<td>1, n, 3, 24</td>
<td>Satisfied</td>
</tr>
<tr>
<td>3</td>
<td>Cylinder lag=2 sccs</td>
<td>3</td>
<td>Failed</td>
</tr>
<tr>
<td>4</td>
<td>Cylinder lag=6 sccs</td>
<td>2, n, 4, 25</td>
<td>Satisfied</td>
</tr>
<tr>
<td>5</td>
<td>Set point=18, Target &lt;=18</td>
<td>1, n, 10</td>
<td>Satisfied</td>
</tr>
<tr>
<td>6</td>
<td>Set point=18, Target &lt;16</td>
<td>30, m, 56</td>
<td>Failed</td>
</tr>
</tbody>
</table>

### VI. CONCLUSION

In this paper, we used the formal approach and model-based design in order to specify and formally verify the functionalities of a medical device known as Endotracheal intubation. We proposed a formal model of the system as well as specification blocks of its linear properties, and formally verified the specified properties using Simulink Design Verifier. The system is modeled with parallel components in Simulink and Stateflow, where the event passing/handling and synchronization is efficiently provided. The chart is also optimised to avoid having possible issues such as State Inconsistency, Conflicting transitions, Data Range Violations and Cyclic Behavior in the Stateflow. We also employed Simulink Design Verifier toolset to prove correctness of the model as well as the safety and some temporal properties. In this effort, property proving as well as simulation traces for different components are done based on the discrete timing concept. In addition to the previous work, the subsystem advantage is used for encapsulation the specified properties. Moreover, the capability of triggered subsystems to handle the event from another components in the model is also used. The authors plan to extend this work for more linear temporal properties, in order to overcome the limitations of the Simulink for specifying such properties. These properties will be available through Simulink library as different customizable blocks.

### REFERENCES


[17] Murugesan, Anitha and Whalen, Michael W and Rayadurgam, Sanjai and Heimdahl, Mats PE. Compositional verification of a medical device system. Proceedings of the 2013 ACM SIGAda annual conference on High integrity language technology,


APPENDIX 1

Function: Embedded MATLAB Function 2

```
1: function out = within(p, q, act)
2:     |
3:     persistent pre, done, res;
4:     if isempty(pre) then
5:         pre = false;
6:     endif
7:     |
8:     if isempty(done) then
9:         done = false;
10:    endif
11:    |
12:    if isempty(res) then
13:        res = false;
14:    endif
15:    |
16:    if act && not(done)
17:        if not(pre) then
18:            if q then
19:                done = true;
20:            elseif p && not(q) then
21:                pre = true;
22:            endif
23:        endif
24:    endif
25:    |
26:    if p && pre then
27:        res = true;
28:    endif
29:    |
30:    if not(p) && pre then
31:        if not(q) then
32:            res = false;
33:        else
34:            res = true;
35:            done = true;
36:        endif
37:    endif
38:    |
39:    elseif act && done
40:        res = true;
41:    elseif not(act) && & pre then
42:        if done then
43:            res = true;
44:        elseif q then
45:            res = true;
46:        else
47:            res = false;
48:        endif
49:    elseif not(act) && not(pre then
50:        res = false;
51:    endif
52:    out = res;
53: }
```