VSMoN: Runtime Monitoring Based Data-driven Remote Vital Sign Monitoring System

Rahul Bharadwaj Pendyala[*], **Abhinandan panda**[*], **and Srinivas Pinisetty**[*]
[*]Indian Institute of Technology Bhubaneswar, India

ABSTRACT It is reported that many health diseases leading to critical conditions (and deaths) could have been avoided or prevented with regular health monitoring that can enable early detection. With the alarming global health threat, better medical assistance may be possible if vital signs are monitored on a regular basis. In this regard, the proposed informal methods lack formal soundness and guarantee when used in health monitoring systems. Moreover, the ability of domain/healthcare experts to express the healthcare policies to be monitored is one of the most important inputs in developing such a system. Thus, we propose a domain-specific language (DSL) that can be used by experts to describe healthcare policies. Since the application is related to healthcare, the correctness of the monitoring system built from the policies is of utmost importance. We thus rely on formal monitoring approaches to synthesize monitoring code from policies, which necessitates translating policies expressed in our DSL to the Timed Automata formalism. To illustrate the feasibility of the proposed approach, a prototype implementation of a client-server-based mobile application has been developed. We examine preliminary performance assessments using the prototype developed, finding that the response time for monitoring sufficiently large amounts of data is reasonable.

KEYWORDS Domain-Specific Language, Timed Automata, Client-Server Architecture, Formal Runtime Monitors, Remote Health Monitoring.

1. Introduction

According to a survey by the World Health Organization (WHO), about 1.7 million people under the age of 75 died in the European Union (EU) in 2013. Nearly 34% of those deaths could have been prevented if the patients had received better medical treatment (Burkeil n.d.). Heart attacks and strokes account for around half of all preventable fatalities. Diabetes and hypertension have grown increasingly frequent, owing to hectic lifestyles, poor dietary habits, etc (Agarwal & Lau 2010). As a result, there is a need to find a solution for such illnesses that may lead to costly medications for a long time by identifying the diseases at their onsets. Traditional periodic manual vital sign monitoring risks unknown patient health deteriorating due to the limited frequency of monitoring. In the case of ischemic stroke therapy, the affected person's neurological symptoms like speech problems and vital signs consisting of blood coagulation index ought to be monitored in real-time after the administration of recommended tissue plasminogen activator (rt-PA) (Lansberg et al. 2012). If any vital signs are out of range, the stroke crew will issue corresponding remedy orders. In case of any delay in making such time-bounded decisions may lead to unavoidable consequences. Therefore, in such scenarios, the conventional health center, and doctor-targeted health monitoring paradigm now seem ineffective in safely coping with the growing variety of patients and sicknesses (Kakria et al. 2015).

Fortunately, advances in technology have the potential to help provide a variety of solutions in response to the growing demand for medical assistance. Using wearable sensors to monitor real-time vital signs in a patient, such as an electrocardiogram (ECG), photoplethysmography (PPG), and blood pressure (BP), presents a new problem because it generates a huge volume of data that must be monitored. It becomes difficult for medical practitioners to examine and monitor such enormous amounts of data in order to diagnose complex illness patterns and give proper medical care. Hence, there is a need for automatic

JOT reference format:
Rahul Bharadwaj Pendyala, Abhinandan panda, and Srinivas Pinisetty. *VSMoN: Runtime Monitoring Based Data-driven Remote Vital Sign Monitoring System*. Journal of Object Technology. Vol. 23, No. 2, 2024.

http://dx.doi.org/10.5381/jot.2024.23.2.a4

analysis of large samples of vital signs-related data.

In this setting, the current decision support systems rely on staff observation-based simulations and semi-automated informal methods based on the Internet of Things (IoT) (Ghosh et al. 2016; Khoi et al. 2015), artificial intelligence (Gondalia et al. 2018; Hijazi et al. 2016) that can coordinate with health workers to enhance the affected person's healthcare. The primary concern in using such informal approaches based on artificial intelligence and machine learning lies in their credibility when operating in safety-critical medical care scenarios.

Runtime verification (RV) (Francalanza et al. 2017; Bartocci et al. 2018) is a dynamic approach of monitoring to ensure that a run of a system under inspection satisfies or violates a particular desired policy (φ) during execution. In the formal method-based RV approaches, such as (Bauer et al. 2011; Falcone et al. 2009), an RV monitor is synthesized automatically from a formal high-level specification of a set of policies that is correct by construction. The generated RV monitors are used to verify a stored execution of a system (offline verification) or the current live execution of a system (online) with respect to the desired correctness policy φ. Formal RV approaches have been proposed for the automated identification of vital signs from physiological signals such as ECG and PPG (panda et al. 2021b,a; Panda et al. 2022). However, there is no specific setup proposed for how these results can be utilized to build a vital sign monitoring system. Moreover, medical/domain experts may also want to specify policies for remote/continuous health monitoring. It may not be possible for them to describe the intended policies in formalisms such as Timed Automata (TA) used in these works. As a result, the challenge is to derive a monitoring code from the vital sign policies that are specified by domain experts that are correct by construction.

In this paper, we address a few of the above challenges by proposing a complete end-to-end setup for the vital sign remote health monitoring system using runtime monitoring approaches that guarantee formal soundness.

The key contributions of the paper are:

- *Design of DSL and translating into TA formalism:* We propose a DSL to specify time-intervened safety-critical policies for a medical care scenario and also an algorithm to translate a given timed policy expressed using DSL into its corresponding timed automaton from which monitoring code can be generated.
- *Setup/architecture of the system:* Because healthcare policies change over time, and updating them on all local systems can be inconvenient, we propose a complete end-to-end client-server architecture-based setup in which RV Monitors are generated on the server rather than having a standalone application that runs all the systems locally.
- *Prototype and performance evaluation:* We develop a prototype of the proposed setup, including all the modules, such as a DSL parser and TA generation from DSL. Also, a mobile application based on client-server architecture as a proof-of-concept that facilitates the above contributions and also examines preliminary performance assessments using the prototype developed, finding that the response time for monitoring sufficiently large amounts of data is acceptable. By sufficiently large amounts of data, we mean that we tested our prototype with 1GB traces, whereas a normal trace collected over a duration of one minute would be around 1KB.

This paper is organized as follows: Section 2 presents an outline of our proposed approach. Section 3 briefly presents some of the preliminaries related to timed automata and runtime verification via some examples. An overview of the software architecture of our monitoring system is discussed in Section 4. In Section 5, we present the DSL framework and discuss the translation of policies to TA. Finally, we discuss implementation details in Section 6.

2. Overview of the proposed approach

In this section, we present a high-level overview of the proposed approach that facilitates the real-time health tracking of a person's vital signs associated data, immediate feedback of health data, and timely intervention of medical assistance. The structural overview of our proposed monitoring system is presented in Fig. 1.

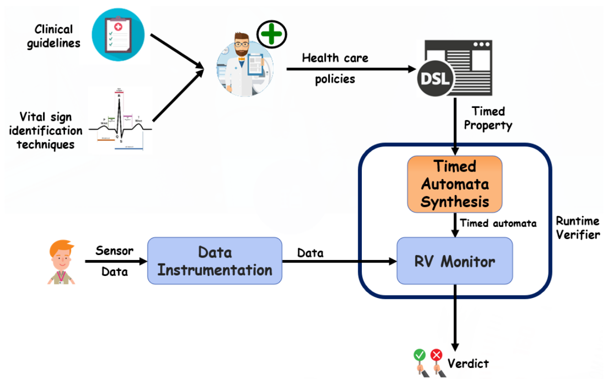


Figure 1 Structural outline of our model

In this context, a key component in developing such a system is knowing which healthcare policies to monitor. There are approaches for the identification of vital signs from physiological signals like ECG and PPG (panda et al. 2021b,a). Moreover, based on such inferences or their knowledge of clinical guidelines, domain/healthcare experts may want to describe the policies to be monitored. Hence, there is a need for a DSL that has all the relevant constructs so the domain expert can easily understand and describe those healthcare policies. As shown in Fig. 1 we propose a DSL that allows domain experts to describe healthcare policies. The data instrumentation component of the system collects all relevant vital sign data such as blood pressure, heart rate, temperature, and so on from the wearable sensors. This data is fed to the RV Monitor, which checks for violations of safety policies of interest.

We can manually implement the monitors once we have all of the healthcare policies that need to be monitored. However, the correctness of the monitoring code itself is uncertain. There are tools and frameworks to automatically generate monitoring code that is correct by construction using formal runtime monitoring approaches (Bauer & Falcone 2012; Barringer et al. 2004; Colombo et al. 2009; Chen & Roşu 2007). However, to utilize these frameworks, we need to translate these policies in DSL into formal models/specifications required to synthesize the monitoring code using formal monitoring frameworks. In this work, we leverage runtime monitor generation approaches proposed in (Pinisetty et al. 2017), which require the policies specified as timed automata to be given as input. Thus, we propose an algorithm for automatically translating the intuitive DSL policies into the TA formalism as shown in the timed automata synthesis module of Fig. 1.

The policies in the TA formalism can be used by the runtime monitor generation tool to synthesize monitors to check for policy violation, as shown in the RV Monitor module of Fig. 1.

To demonstrate our framework, we use the following illustrative example policies that can be considered for monitoring.

Policy $\varphi 1$***:*** Event HeartRate cannot be high for throughout 10-time units.[1]

Policy $\varphi 2$***:*** Event BloodPressure cannot be high within 20-time units after an Event low HeartRate.

Policy $\varphi 3$***:*** Event BloodPressure cannot be high within 10-time units after an Event high HeartRate.

In later sections, we elaborate on the different modules described in Fig. 1 and we will use these key policies as examples to demonstrate the approach.

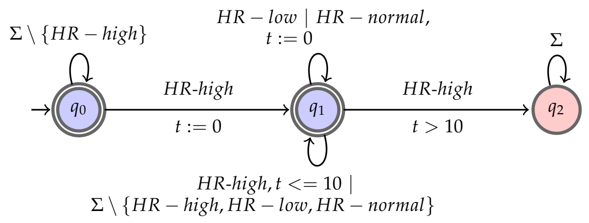
3. Preliminaries and Background

In this work, we utilize a formal runtime verification monitoring approach (Pinisetty et al. 2017) to generate monitors from high-level policies. The monitor is usually automatically extracted from the policy φ (which the monitor should verify). The RV approach (Pinisetty et al. 2017) requires the policy to be monitored to be formalized as TA. In this section, we will recall the TA formalism briefly via an example. We also briefly recall the approach that we apply to generate monitors from policies described as TA.

Timed Automata: We consider timed policies as languages that can be described by deterministic timed automata in this work. A timed automaton is a finite-state automaton (a graph with a finite number of nodes and a finite set of labeled edges) with timing constraints imposed by a finite number of real-valued clock variables. Here we omit the details of the formal syntax and semantics of TA (Alur & Dill 1994). We discuss the well-known TA model via an example.

Example 1 (Timed automaton) *The TA in Fig. 2 represents policy $\varphi 1$ (i.e, Event HeartRate cannot be high for throughout 10-time units). In the TA in Fig. 2, the set of locations is $L = \{q_0, q_1, q_2\}$, where q_0 is the initial location, and Σ ={HR-low, HR-high, HR-normal, BP-low, BP-high, BP-normal, Temp-low, Temp-high, Temp-normal } is the set of events. The set of real-valued clocks is $X = \{t\}$. Transitions occur between locations depending on events. On the transitions, there are guards with constraints on clock values such as $t \leq 10$, and resets of clocks. When the first event HR-high occurs, the TA moves to q_1 from q_0, and the clock t is reset to 0. If the event HR-high occurs while the TA is in location q_1 and $t \leq 10$, the TA remains in location q_1; otherwise, it moves to location q_2. The location q_2 (non-accepting) should never be reached for the policy $\varphi 1$ to be satisfied over the runs.*

[1] Time duration mentioned in the policy is only for illustration purposes.



$\Sigma \setminus \{HR-high\}$
$HR-low \mid HR-normal,$
$t := 0$
Σ
q_0
HR-$high$
$t := 0$
q_1
HR-$high$
$t > 10$
q_2
HR-$high, t <= 10 \mid$
$\Sigma \setminus \{HR-high, HR-low, HR-normal\}$

Figure 2 Timed automaton defining policy $\varphi 1$

Runtime verification monitors from policies defined as TA: There are approaches proposed for obtaining runtime monitors for timed policies, form the specification of the policies in the form of TA (Bauer et al. 2011; Pinisetty et al. 2017).

Given a timed policy denoted as φ defined as TA $\mathcal{A}_\varphi$, it is straightforward to generate an RV monitor (denoted as M_φ) for the timed policy φ using the approaches discussed in (Bauer et al. 2011; Pinisetty et al. 2017).

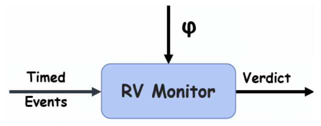


Figure 3 Runtime monitor

The monitoring algorithm proposed in (Pinisetty et al. 2017) requires the policy φ defined in the timed automaton format and relies on operations on timed automata to check whether the observed input stream satisfies the policy φ or not. In this work, we consider the system being monitored as a black-box as usual in RV approaches (Bauer et al. 2011; Falcone et al. 2009).

The system execution is not influenced or changed by a verification monitor. Given a finite timed word σ over the alphabet Σ (where a timed word is a stream of timed events, where each event contains information about the event and the timestamp) denoting the current observation, the RV monitor emits verdicts true (T), and false (F) (conclusive) and currently true (CT), and currently false (CF) (inconclusive verdicts).

The RV monitor for φ denoted as M_φ returns verdict T if for any continuation σ satisfies φ. It returns verdict F if any continuation of σ falsifies φ. Monitor M_φ returns inconclusive verdict CT if σ satisfies φ, and not all continuations of σ satisfy φ. It returns an inconclusive verdict CF if σ falsifies φ, and there is a continuation of σ that satisfies φ.

We use an example to provide some insight into the expected behaviour of RV monitors (see (Pinisetty et al. 2017; Bauer et al. 2011) for formal details).

Example 2 (Example illustrating behaviour of an *RV* monitor) *Consider policy $\varphi 1$, defined formally by the TA in Fig. 2.*

Table 1 Example behavior of RV monitor for policy $\varphi 1$ illustrated in Fig. 2

σ	$M_{\varphi}(\sigma)$
$(HR\text{-}high, 1)$	CT
$(HR\text{-}high, 1) \cdot (BP\text{-}low, 3)$	CT
$(HR\text{-}high, 1) \cdot (BP\text{-}low, 3) \cdot (BP\text{-}low, 5)$	CT
$(HR\text{-}high, 1) \cdot (BP\text{-}low, 3) \cdot (BP\text{-}low, 5) \cdot (HR\text{-}high, 14)$	F

Consider an input timed word $\sigma = (HR\text{-}high, 1) \cdot (BP\text{-}low, 3) \cdot (BP\text{-}low, 5) \cdot (HR\text{-}high, 14)$. The step-wise behavior of the monitor for policy $\varphi 1$ is shown in Table 1. At $t = 1$, when the current observed input is $\sigma = (HR\text{-}high, 1), \sigma$ satisfies the policy $\varphi 1$ (the automaton in Fig. 2 moves to location q_1), so the monitor provides verdict CT in the first step. Similarly, the verdict provided by the monitor is CT in the next two steps at $t = 3$ and at $t = 5$, since the event in these two steps is $BP\text{-}low$, and the automaton will remain in location q_1. At $t = 14$, after observing the event $(HR\text{-}high, 14)$ (i.e., when the current observed input is $\sigma = (HR\text{-}high, 1) \cdot (BP\text{-}low, 3) \cdot (BP\text{-}low, 5) \cdot (HR\text{-}high, 14)$, the policy $\varphi 1$ is falsified by σ and for any extension of σ falsifies the policy $\varphi 1$. In the automaton, the location reached will be q_2, from which there is no path to any accepting location. Thus, the monitor provides a conclusive verdict (false).

Remark 1 *To monitor multiple policies, one possibility is to combine all the TAs using the TA product construction (Alur & Dill 1994) and to synthesize one monitor for the resulting policy. Alternatively, we can also synthesize one monitor per policy and execute all the monitors simultaneously (by feeding input to all the monitors in parallel) and the individual verdicts of a policy can be reported.*

4. Proposed system: Architecture

In this section, we discuss the software architecture of the proposed monitoring model. The solution should be designed in such a way that it can be easily adapted to simple devices like smartphones or wearable devices. In general, standalone applications in which all modules are run locally on the device are not suitable for our model because healthcare policies can evolve and are frequently updated. As a result, whenever there is a new policy update, we must update the local machines, which is inconvenient. We thus consider a client-server architecture for the proposed system that can accommodate evolving/changing policies to be monitored on the server side. For such a software toolchain to be feasible, we must address the following modules.

- A front-end that allows administrators to express safety-critical policies using the proposed formal specification language.
- A front-end that provides functionalities that enable users to initiate remote online or offline runtime monitoring and receive results on the client side.

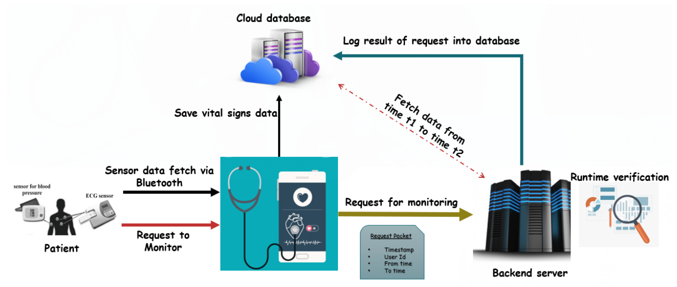


Figure 4 Architecture and setup

- A common backend server that processes requests, and generates runtime monitors for the policies specified by domain experts.

Architecture: The software architecture of our model is presented in Fig. 4. Wearable device sensors, such as a blood pressure sensor, an ECG sensor, and a temperature sensor, are worn by a patient. The data from those sensors is intercepted and sent via Bluetooth to the mobile software. This information is saved in a cloud database and will be available for future offline monitoring. In the proposed client-server architecture, only small or low computational tasks are implemented in client modules with limited hardware capacity. Users can access the client module, which allows them to start or stop data monitoring and view the results of previous monitoring requests. As per the setup, request validation and input would be handled from the client side. The server will be responsible for processing the client's request and returning the result. In the server module, the admin can input the safety-critical policies that are consistent with our DSL. All the heavy computations like data parsing and monitoring take place in the server module. After monitoring, the monitoring verdicts are logged into the database.

Remark 2 *We have not examined any load-balancing strategies as of yet, but it would be helpful to utilise a load-balancer that distributes the traffic load among the servers, which would increase the application's performance.*

Data Collection from device sensors: In recent times, wearable sensors have attracted great interest because of their ability to offer continuous, real-time physiological data. We employ accessibility technologies like Bluetooth to capture sensor data automatically. Sensor data is saved in JSON format, along with a reading and a timestamp. To reduce the load on the server, we read the sensor data every 10 seconds (this interval can be configurable) and push it to the server. NoSQL database is used to store this data for later analysis. Because wearable sensors cannot store data indefinitely, the mobile application saves the data collected from sensors locally first, then periodically uploads it to the central server, as shown in Fig. 5. If the mobile application is not connected to the server, it can still take new measurements, but it will only be synchronized with the server once connectivity is established. This method of updating data at regular intervals allows the client to keep the application running even if there is no Internet connection. We also consider that the wearable device is secure and is not programmable wirelessly, and thus an attacker cannot access it.

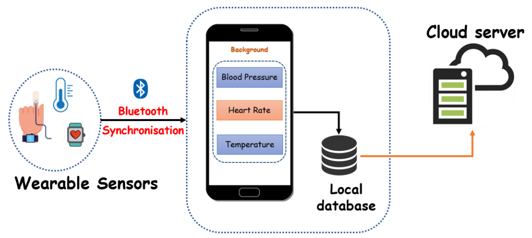


Figure 5 Data collection

5. Methodology

In this section, we present the DSL that can be used by domain experts for specifying vital sign policies to be monitored (in Section 5.1). Later in Section 5.2, we present an approach for the translation of policies expressed in DSL into the TA formalism. Once we have the policies to be monitored in TA format, the monitoring code can be synthesized using the formal RV approaches recalled in Section 3.

5.1. DSL for Specifying Policies

A DSL is a language that is designed to be used in the context of a given domain. Our scenario does not involve engineers specifying all the monitoring requirements; instead, healthcare practitioners or domain experts may be able to define the policies to be monitored. Medical practitioners may find it challenging to express healthcare policies in formalisms such as timed automata.

The domain expert should be able to dynamically remove, add new policies, or update existing policies to be monitored. As a result, it is not like traditional development, where healthcare practitioners informally specify policies, and developers work to develop monitors that reflect those policies. Thus, to make our approach more tangible, we propose a language with simple, intuitive constructs that domain experts can easily relate to and understand to describe healthcare policies to be monitored. In real life, there are times when the decisions we make, determine what should be done next. Similar situations arise in defining time-intervened healthcare policies. A healthcare policy should be written in such a way that it can inform us what action or set of actions we should take if event X occurs at a specific point in time. It's possible that, depending on these decisions, we'll want to pay attention to the next event, Y, to decide whether or not to take any action. We thus need language constructs that can be used to define which decisions should be made when a series of events occur. We intend to draw parallels with well-known programming languages such as C, where decision-making expressions such as if-else statements determine the direction of the program execution flow. Hence, we want to design a DSL that provides constructs that are comparable to decision statements in C but with a few customizations through which our timed policies can be specified.

Our DSL is specified in the form of context-free grammar. Every word labeled within < .. > is a non-terminal and a

Domain-Specific Language		
<start>	→	*begin := <dsl> end;*
<dsl>	→	*<matched − stmt>*
	|	*<unmatched − stmt>*
<matched − stmt>	→	*if <conditional-stmt>then <matched − stmt> else <matched − stmt> endif ;*
	|	*return <result>*
<unmatched−stmt>	→	*if <conditional-stmt> then <unmatched−prime>*
	|	*return <result>;*
<unmatched−prime>	→	*<dsl> endif;*
	|	*<matched−stmt> else <unmatched-stmt> endif;*
<conditional−stmt>	→	*(<event> = <condition>, <timestamp>)*
<timestamp>	→	*time <relop> intnum*
<event>	→	*HR*
	|	*BP*
	|	*TEMP*
<condition>	→	*high <condition-prime>*
	|	*low<condition-prime>*
	|	*normal<condition-prime>*
<condition-prime>	→	*/<condition>*
	|	*δ*
<relop>	→	<=
	|	>=
	|	==
	|	<
	|	>
<result>	→	*safe*
	|	*unsafe*

Table 2 DSL for specifying healthcare policies

terminal otherwise. The structure of DSL is presented in Table-2. Let us briefly look into the constructs provided by our DSL.

- A <*matched-stmt*> is either an if-then-else statement containing no unmatched statements or any statement which is not an if-then-else statement and not an if-then statement.
- A <*unmatched-stmt*> is an if-then statement (with no else-part) or an if-then-else statement where unmatched statements are allowed in the else-part (but not in the then-part) which is comparable to context-free grammar of if-else statements.[2]
- The construct <*conditional-stmt*> can be used to specify the occurrence of an event at a specific point in time.
- The construct <*event*> is used to specify on what event we are defining the condition like *heart rate (HR), blood pressure (BP) or temperature (TEMP).*
- The construct <*condition*> is used to define whether the event's condition is *high, low, or normal*.
- The <*condition-prime*> construct is used to indicate the condition of the event <*blood pressure*>, as it accepts values such as *high/low* or *low/high*. δ rule is similar to the epsilon rule in Context-Free grammar i.e. it is a rule defined with the empty string.
- The construct <*result*>, which takes the values *safe* or *unsafe*, is used to specify the final decision of the policy after observing a set of events.
- The construct <*relop*> provides relational operators that can be used to provide timing relations between a set of events.

Now, let us consider the policies mentioned in Section 2 and see how they can be expressed using the proposed DSL.

Example 3 (Policies specified using the DSL) *Let us consider Policy $\varphi 1$ from Section 2 which states that if the event heart rate is high at time t=0 and if even after time t=10, the condition of the event heart rate is high, then it must lead to an unsafe state (In medical terms this policy concerns with Tachycardia). In Fig. 6 we illustrate how it is specified using the proposed DSL.*

Policy $\varphi 2$ from Section 2 states that if the event heart rate is low at time t=0 and the condition of event blood pressure continues to be high for more than 20-time units, then it must lead to an unsafe state (In medical terms this policy concerns with Bradycardia). In Fig. 6 we illustrate how it is specified using the proposed DSL.

Policy φ1: *Warning for Tachycardia*

```
begin:=
    if ( HR = HIGH , TIME = 0 ) then
        if ( HR = HIGH , TIME > 10 ) then
            return unsafe;
        else
            return safe;
        ;
    else
        return safe;
    ;
end;
```

Policy φ2: *Warning for Bradycardia*

```
begin:=
    if ( HR = LOW , TIME = 0 ) then
        if ( BP = LOW , TIME >= 20 ) then
            return safe;
        else
            return unsafe;
        ;
    else
        return safe;
    ;
end;
```

Figure 6 Policy φ1 and Policy φ2 mentioned in Section 2 expressed using the DSL

5.2. DSL to Timed Automata Translation

The idea for translating a policy into its corresponding TA is based on the notion of constructing a control flow graph from a given code segment. We shall use a stack-based algorithm to construct a TA from a given timed policy specified using the proposed DSL. The proposed approach is illustrated in Algorithm 1. A healthcare policy described using the DSL proposed in subsection 5.1 that is syntactically valid is provided as input in a text document. The output of this algorithm is a graph-like structure that matches the timed automaton syntax. This output is further processed to make it suitable for our subsequent modules. The algorithm reads the input text file in a linear fashion. The typical if-else block starts with an if statement then moves on to the else part, and finally ends with the end-if keyword. To ensure a correct mapping with respective if-else blocks, we use these keywords to decide what should be pushed onto the stack and what should be popped. The basic idea is that whenever an if keyword is encountered, we push the corresponding conditional if statement into the stack, as shown in line 8. When we encounter an else statement, we only pop once if the top element is an *if* statement; otherwise, the *else* statement is pushed to the top of the stack as shown in lines[13-21]. Whenever an element is popped from the stack, we create a new state in our final automaton and make the current stack top element the parent of this newly created state. Furthermore, if *end-if* is encountered, we begin popping elements until a start state(q_0) is reached. As a result, for each pop operation, we create a new state and make its parent the current stack top element after this element is popped and also add a self-loop to the parent for events related to the event in the condition and reset the clock, as shown in lines[23-31]. When we encounter a *return* statement, we mark the current stack top as an accepting location if the value associated with the return keyword is *safe*, and the non-accepting location otherwise as shown in lines[32-39]. Finally, for every state in the TA add self-transitions for missing (don't care) events as shown in lines[46-48].

Let us now further discuss how the elements generated by the proposed algorithm are mapped to the syntax of timed automata. Each conditional statement (if/else) creates a new location in the automaton. So, each time an if or else statement is popped from the stack, a new Node is created (a location in the corre-

[2] The constructs matched-stmt and unmatched-stmt are used to eliminate ambiguity in the grammar.

Algorithm 1 TA generation from a policy in DSL

```
Input: A text file containing timed policy as specified in the
    policy $\varphi 1$ using the DSL proposed in 5.1
Initialise start Node $\leftarrow q_0$
AcceptingStates = [], NonAcceptingStates=[]
stateList=[$q_0$]
eventsMap $\leftarrow$ Map[EventName]ListOfRelatedEvents
Initialise Stack stack
stack.push($q_0$)
Read lines in text file using a buffered reader
while $(line \in lines)! = null$ do
    if line.startsWith(if) then
        stack.push("if condition") ;
        stateList.add( makeNode( stack.top() ) );
        stateList.add( makeNode( !stack.top() ) );
    else if line.startsWith(else) then
        if stack.top() == if then
            condition $\leftarrow$ stack.top().condition;
            Node $\leftarrow$ stack.pop();
            Node.parent $\leftarrow$ stack.top();
            transition $\leftarrow$ [Node.parent,Node, condition];
            Node.addTransition( transition );
            stack.push(else condition);
        else
            stack.push(else condition);
    else if line.startsWith(endif) then
        while stack.top() ! = $q_0$ do
            condition $\leftarrow$ stack.top().condition;
            Node $\leftarrow$ stack.pop();
            Node.parent $\leftarrow$ stack.top();
            transition $\leftarrow$ [Node.parent, Node, condition];
            Node.addTransition( transition );
            selfLoopEvents $\leftarrow$ eventsMap[condition.event]
            stack.top().addTransition([Node.parent,
    Node.parent, parentConditions, resetClock])
    else if line.startsWith(return) then
        if line.split[' '][1] == safe then
            stack.top().isAccepting $\leftarrow$ true;
            AcceptingStates.append( stack.top() ) ;
        else
            NonAcceptingStates.append( stack.top() );
            transition $\leftarrow$ [stack.top(), stack.top(), $\Sigma events$] ;
            stack.top().addTransition( transition );
    else
        continue;
    for Node s: AcceptingStates do
        s.addTransition([s, s.parent, $\Sigma$ events]);
        s.parent.isAccepting $\leftarrow$ true)
    for Node s: stateList do
        events $\leftarrow$ [$\Sigma$ events $\setminus$ {events in transitions of state}]
        s.addTransition([s, s, events]);
    return startNode $q_0$
```

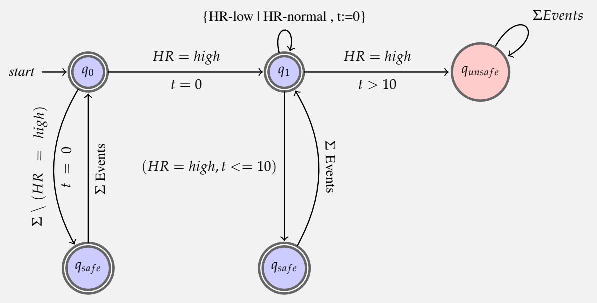


Figure 7 TA corresponding to Policy $\varphi 1$ mentioned in Fig.6 obtained from the proposed Algorithm.
For readability, we have omitted transitions for $\Sigma \setminus \{HR - high, HR - low, HR - normal\}$

sponding TA). Each State or Node has attributes like *locationId*, *isAccepting*, *ParentNode*, *List<Transition>*. Each Transition object consists of attributes like sourceNode, destinationNode, condition, and clockResetFlag that includes action and guard constraint similar to TA semantics. Each time a State N_i is created using the *makeNode()* function with a *locationId*, a transition T or edge is created between State N_i and StateN_p (parent State) associated with stack top. A transition from N_i to N_p consists of *action* and *guard*.

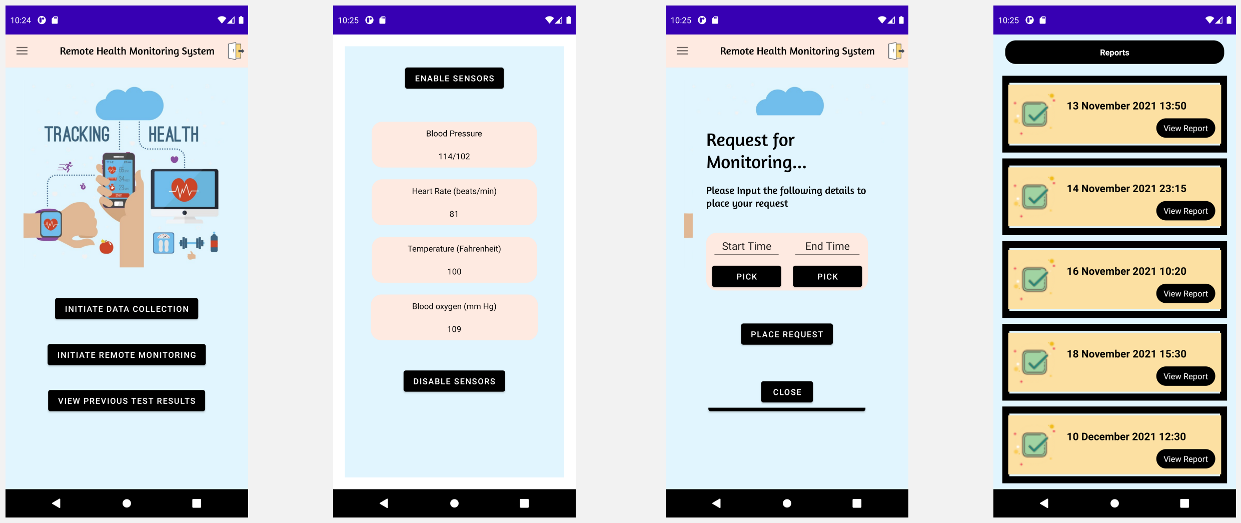
As per our DSL, every conditional statement is associated with the following elements *(<event> = <condition> ,<timestamp>)*. From this, *<event> = <condition>* construct corresponds to *action* associated with the transition T and *<timestamp>* which is essentially *time <relop> intnum*, corresponds to *guard* constraint associated with that Transition T and *intnum* is an integer describing clock value. The TA corresponding to policy $\varphi 1$ that is obtained by following this algorithm is presented in Fig. 7.

The source code along with a further detailed explanation of Algorithm 1 is available at: github.com/rahulpr22/taAlgo and also discussed in Appendix A. Once we have policies expressed as TA, we can use established RV monitoring algorithms (Pinisetty et al. 2017) that generate monitoring code that is correct by construction.

Remark 3 *Any policy that can be described using a timed automaton with a single clock can be expressed using our DSL. A user can specify multiple policies, each corresponding to a TA with a single clock. All policies can be monitored concurrently (by generating a monitor for each policy and executing them concurrently) or by automatically composing all of the TAs using TA product construction.(Alur & Dill 1994; Bengtsson & Yi 2003). Further optimization steps can be added to the TA generation procedure to reduce the number of states in the resulting automaton. For example, locations q_{safe} in Fig. 7 can be merged into a single state, and self-loop can be added to state q_1 eliminating the transitions to state q_{safe}.*

Since every line of the policy stated in the text document will be pushed and removed from the stack only once, the runtime complexity of this algorithm is O(n).

Remark 4 (RV monitor from the policy as timed automaton)



(a)Client Dashboard (b)Data Generation (c)Request monitoring (d)View Results

Figure 8 Client Interfaces

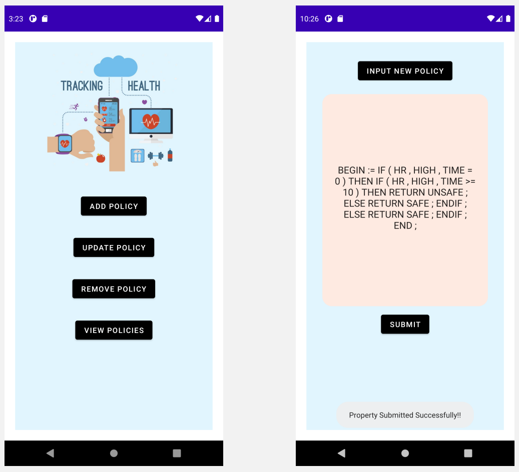
As discussed in Section 3, we use the RV monitoring approach/framework provided in (Pinisetty et al. 2017) to obtain the RV monitor from the policy in the TA format. How the monitor obtained using the approach in (Pinisetty et al. 2017) behaves is illustrated via an example (Example 2 in Section 3.)

Remark 5 (Discussion on the translation approach) *Let us consider a policy φ specified using the DSL in Table-2 that is fed into our TA generation algorithm.*
Input validation: If the input policy φ is not as per the DSL syntax, our parsing algorithm will raise an error and will exit.
Termination: If the input policy specified is valid and properly formatted, the algorithm reads each line in the policy and processes the conditions, and returns statements until all the lines of policy are evaluated, and finally terminates.
State and Transition Creation: The algorithm creates states and transitions based on the input policy. For each "if" statement encountered, it creates a new state in the Nodes map and a corresponding state object in the TA stack. It also creates transitions between states based on the conditions and stack operations.
Parent-Child Relationships: The algorithm establishes parent-child relationships between states as follows. When an "else" statement is encountered, it pops a state from TA and sets its parent as the previous state in the stack. This ensures the correct hierarchical structure of states.
Accepting and Non-Accepting States: The algorithm identifies and categorizes accepting and non-accepting states as follows. When a "return" statement is encountered, it checks the return value and sets the accepting status of the current state accordingly. Accepting states are added to the accepting list, and non-accepting states are added to the non-accepting list.
Transition Connections: The algorithm establishes transitions between states based on the conditions and stack operations. Transitions are created from the parent state to the current state with the condition obtained from the stack. Additionally, if a state is accepting, a transition from the state to its parent with the condition Σ events is created.

Based on the above analysis, the algorithm satisfies the properties of soundness. It processes the input policy, creates states and transitions, establishes parent-child relationships, identifies accepting and non-accepting states, connects transitions between states, and generates the expected output.

6. Implementation and Evaluation

In order to demonstrate the practicality of the proposed approach, we developed a prototype that shows how formal runtime monitoring techniques can be integrated with the design and development of a software application. The application consists of two modules, client and server, as described in Section 4.



```
BEGIN := IF ( HR , HIGH , TIME =
0 ) THEN IF ( HR , HIGH , TIME >=
10 ) THEN RETURN UNSAFE ;
ELSE RETURN SAFE ; ENDIF ;
ELSE RETURN SAFE ; ENDIF ;
END ;
```

Figure 9 Admin Interfaces

The *client module* is written in *JAVA* and focuses on the application's basic capabilities, such as user interfaces and their functionality. The application includes two types of users: normal users and admin users (also known as application managers). The online registration portal, user login page, and user dashboard are some of the client-side application's features. The

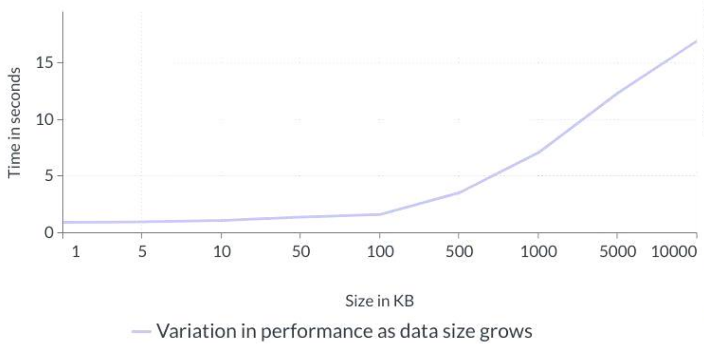


Fig. 10(a)

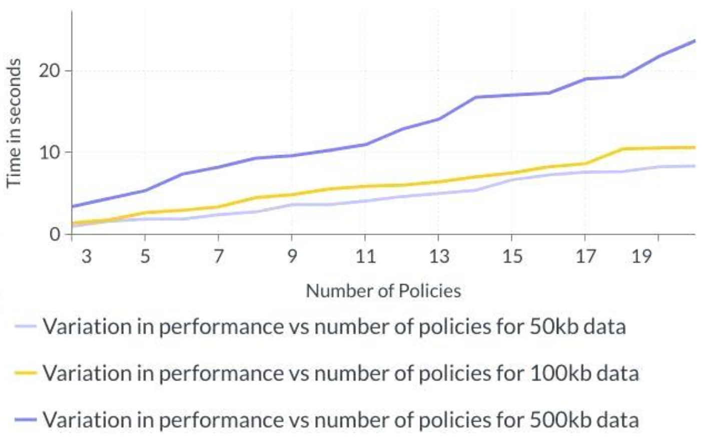


Fig. 10(b)

Figure 10 Fig.(a) depicts performance variation with increase in size of the data and Fig.(b) depicts performance variation with increase in number of policies.

normal user can utilize the user dashboard activity to start the data-gathering process, initiate remote health monitoring, and access or see their prior monitoring findings. The user dashboard page of the application is presented in Fig. 8(a).

The *data collection module* takes vital sign sensor data and parses it into a format that may be used to identify events. This data is then utilized to evaluate various Boolean formulas for generating associated events, which are then stored in a cloud-based NoSQL database along with their timestamps. To test the validity of our approach with real-time data, we used pseudorandom generators that mimicked sensor data. The implementation of the interface for data generation is presented in Fig. 8(b). The client can send a request to the backend server to initiate monitoring, and the user can specify from what time to what time the data should be monitored; the implementation of the interface to request for monitoring is shown in Fig. 8(c). The client can also access previously computed reports based on their date of creation, and the implementation of the interface for the same is present in Fig. 8(d).

The admin user can specify the healthcare policies that are syntactically correct with respect to the proposed DSL. The implementation of the same is presented in Fig. 9. The interface also includes a backend written in *JAVA* to assist in validating the syntax correctness of policies specified. For the DSL proposed in 5.1 we implement a predictive top-down LL(1) parser(Lasser et al. 2019) in order to check the syntactic correctness of policies specified using the DSL.

The backend server uses runtime monitoring libraries/implementation provided in (Pinisetty et al. 2017) that generates runtime monitors by taking policies specified as TA (stored in XML format) as input. The backend monitoring algorithms are written in Python with 550 lines of code. When it comes to the proposed client-server architecture, the client and server are both completely independent of one another. The client component will act as a bridge between the application and the backend server, defining how the two interact. The client and the service's backend interact or communicate using Rest APIs as the front-end component. The client will be in charge of invoking specific URL endpoints with provided parameters such as start and end times and returning the incoming JSON responses parsed as a JSON object to the application in order to update its UI Components. Fig. 1 depicts the overall structure of the implementation details.

This mobile application was tested on a Google Pixel 4 Android 11 phone with 8GB of RAM. The server end of REST is stateless, which means that the server doesn't have to store anything across requests, i.e., no data is stored on the server while request transfers are being processed, so the session is saved on the client's end. This means that there doesn't have to be much communication between servers, making it horizontally scalable. To assess the efficiency and scalability of our proposed vital sign monitoring system, using a set of example policies (policies $\varphi1$, $\varphi2$, $\varphi3$ discussed earlier), we did some preliminary performance assessment, as shown in Fig. 10.

We observed that for trace sizes under 100KB, performance is nearly identical, but after that, computational performance grows approximately linearly with data size, as shown in Fig. 10(a). However, it was observed that 1GB of trace data could be verified in less than a minute, which is regarded acceptable given that the size of a trace collected over a one-minute period is approximately 1KB. We have also observed that performance varies almost linearly with the number of policies to be verified and is consistent across different sizes of trace, such as 50KB, 100KB, and 500KB, as shown in Fig. 10(b). Because all of the heavy computations are done on the server side, our application performs consistently on almost all high-end and low-end devices. The source code for all the modules that are implemented can be found at github.com/rahulpr22.

7. Conclusion and Future work

We have designed and developed a setup for data-driven remote vital sign monitoring that makes use of formal runtime monitoring approaches. We proposed a domain-specific language that can be used to specify safety-critical policies for a medical care scenario. We proposed a technique to translate a given timed policy into its corresponding timed automaton from which monitoring code can be generated. We developed a mobile application as a proof-of-concept to test the feasibility of our approach. The preliminary performance evaluations using the prototype revealed that the response time for monitoring sufficiently large amounts of data is quite reasonable.

We are currently working on ways to complete our approach by focusing on privacy-preserving aspects of the proposed architecture, which includes securing healthcare data and controlling access to healthcare policies using attribute-based encryption techniques. We would also like to run a user study and see whether healthcare workers find the DSL usable and, if not, what aspects of the DSL they struggle with. We would also like to customize this application to the patient's condition so that only those healthcare policies relevant to a patient's medical condition are set to be monitored.

References

Agarwal, S., & Lau, C. T. (2010). Remote health monitoring using mobile phones and web services. *Telemedicine and e-Health*, *16*(5), 603–607.

Alur, R., & Dill, D. L. (1994). A theory of timed automata. *Theoretical computer science*, *126*(2), 183–235.

Barringer, H., Goldberg, A., Havelund, K., & Sen, K. (2004). Rule-based runtime verification. In *International workshop on verification, model checking, and abstract interpretation* (pp. 44–57).

Bartocci, E., Falcone, Y., Francalanza, A., & Reger, G. (2018). Introduction to runtime verification. In E. Bartocci & Y. Falcone (Eds.), *Lectures on runtime verification - introductory and advanced topics* (Vol. 10457, pp. 1–33). Springer. doi: 10.1007/978-3-319-75632-5_1

Bauer, A., & Falcone, Y. (2012). *Decentralised ltl monitoring. formal methods, lncs# 7436.* Springer.

Bauer, A., Leucker, M., & Schallhart, C. (2011). Runtime verification for ltl and tltl. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *20*(4), 1–64.

Bengtsson, J., & Yi, W. (2003). Timed automata: Semantics, algorithms and tools. In *Advanced course on petri nets* (pp. 87–124).

Burkeil. (n.d.). *Eurostat. eu report on amenable and preventable deaths statistics.* Retrieved from https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Amenable_and_preventable_deaths_statistics&direction=next&oldid=337528

Chen, F., & Roşu, G. (2007). Mop: an efficient and generic runtime verification framework. In *Proceedings of the 22nd annual acm sigplan conference on object-oriented programming systems, languages and applications* (pp. 569–588).

Colombo, C., Pace, G. J., & Schneider, G. (2009). Larva—safer monitoring of real-time java programs (tool paper). In *2009 seventh ieee international conference on software engineering and formal methods* (pp. 33–37).

Falcone, Y., Fernandez, J.-C., & Mounier, L. (2009). Runtime verification of safety-progress properties. In *International workshop on runtime verification* (pp. 40–59).

Francalanza, A., Aceto, L., Achilleos, A., Attard, D. P., Cassar, I., Monica, D. D., & Ingólfsdóttir, A. (2017). A foundation for runtime monitoring. In *Runtime verification - 17th international conference, RV 2017, seattle, wa, usa, september 13-16, 2017, proceedings* (Vol. 10548, pp. 8–29). Springer. doi: 10.1007/978-3-319-67531-2_2

Ghosh, A. M., Halder, D., & Hossain, S. A. (2016). Remote health monitoring system through iot. In *2016 5th international conference on informatics, electronics and vision (iciev)* (pp. 921–926).

Gondalia, A., Dixit, D., Parashar, S., Raghava, V., Sengupta, A., & Sarobin, V. R. (2018). Iot-based healthcare monitoring system for war soldiers using machine learning. *Procedia computer science*, *133*, 1005–1013.

Hijazi, S., Page, A., Kantarci, B., & Soyata, T. (2016). Machine learning in cardiac health monitoring and decision support. *Computer*, *49*(11), 38–48.

Kakria, P., Tripathi, N., & Kitipawang, P. (2015). A real-time health monitoring system for remote cardiac patients using smartphone and wearable sensors. *International journal of telemedicine and applications*, *2015*.

Khoi, N. M., Saguna, S., Mitra, K., & hlund, C. (2015). Irehmo: An efficient iot-based remote health monitoring system for smart regions. In *2015 17th international conference on e-health networking, application & services (healthcom)* (pp. 563–568).

Lansberg, M. G., O'Donnell, M. J., Khatri, P., Lang, E. S., Nguyen-Huynh, M. N., Schwartz, N. E., ... others (2012). Antithrombotic and thrombolytic therapy for ischemic stroke: antithrombotic therapy and prevention of thrombosis: American college of chest physicians evidence-based clinical practice guidelines. *Chest*, *141*(2), e601S–e636S.

Lasser, S., Casinghino, C., Fisher, K., & Roux, C. (2019). A verified ll(1) parser generator. In *Itp.*

panda, A., Pinisetty, S., & Roop, P. (2021a). Runtime verification of implantable medical devices using multiple physiological signals. In *Proceedings of the 36th annual acm symposium on applied computing* (pp. 1837–1840).

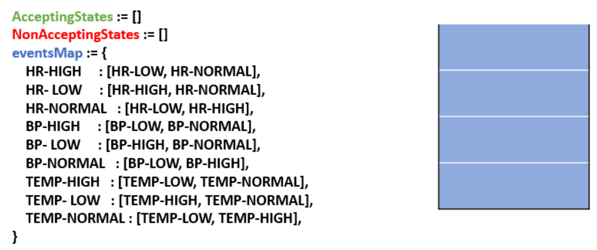
panda, A., Pinisetty, S., & Roop, P. (2021b). A secure insulin infusion system using verification monitors. In *Proceedings of the 19th acm-ieee international conference on formal methods and models for system design* (pp. 56–65).

Panda, A., Pinisetty, S., & Roop, P. (2022). Policy-based diabetes detection using formal runtime verification monitors. In *2022 ieee 35th international symposium on computer-based medical systems (cbms)* (pp. 333–338).

Pinisetty, S., Jéron, T., Tripakis, S., Falcone, Y., Marchand, H., & Preoteasa, V. (2017). Predictive runtime verification of timed properties. *Journal of Systems and Software*, *132*, 353–365.

A. Appendix

The Algorithm for constructing TA for a given timed policy expressed using DSL is presented in Section 5.2. The brief explanation of the algorithm is presented here via an example. Let us consider policy $\varphi 1$ presented in Section 5.1 The initial state of the algorithm is as follows:



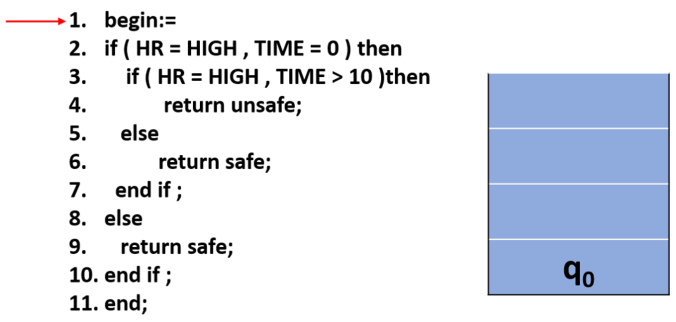
```
AcceptingStates := []
NonAcceptingStates := []
eventsMap := {
   HR-HIGH     : [HR-LOW, HR-NORMAL],
   HR- LOW     : [HR-HIGH, HR-NORMAL],
   HR-NORMAL   : [HR-LOW, HR-HIGH],
   BP-HIGH     : [BP-LOW, BP-NORMAL],
   BP- LOW     : [BP-HIGH, BP-NORMAL],
   BP-NORMAL   : [BP-LOW, BP-HIGH],
   TEMP-HIGH   : [TEMP-LOW, TEMP-NORMAL],
   TEMP- LOW   : [TEMP-HIGH, TEMP-NORMAL],
   TEMP-NORMAL : [TEMP-LOW, TEMP-HIGH],
}
```

Figure 11 Initial state

1. Initially a start state q0 is pushed int the stack following line-5 of the algorithm.

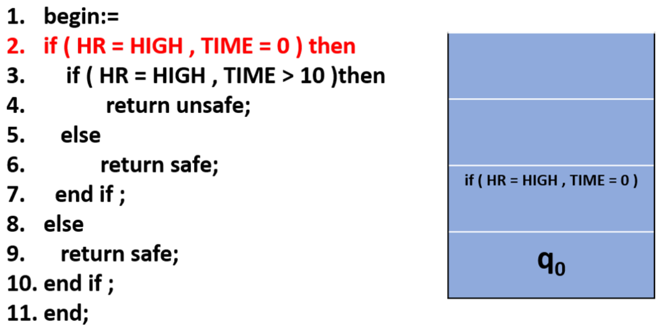
2. When a line starting with if statement is encountered push that line into stack following line-8 of the algorithm as shown in Fig. 13.

3. When a line starting with if statement is encountered push that line into stack following line-8 of the algorithm as shown in Fig. 14.



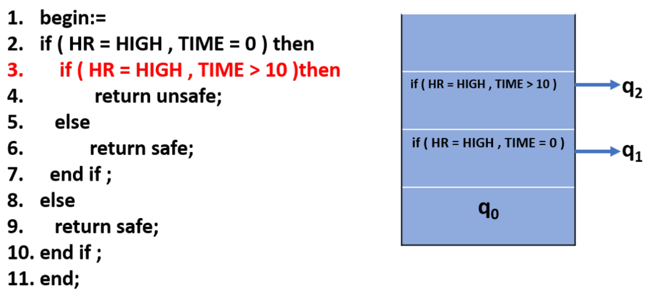
```
1.  begin:=
2.  if ( HR = HIGH , TIME = 0 ) then
3.      if ( HR = HIGH , TIME > 10 )then
4.          return unsafe;
5.      else
6.          return safe;
7.      end if ;
8.  else
9.      return safe;
10. end if ;
11. end;
```

Figure 12 Step-1



```
1.  begin:=
2.  if ( HR = HIGH , TIME = 0 ) then
3.      if ( HR = HIGH , TIME > 10 )then
4.          return unsafe;
5.      else
6.          return safe;
7.      end if ;
8.  else
9.      return safe;
10. end if ;
11. end;
```

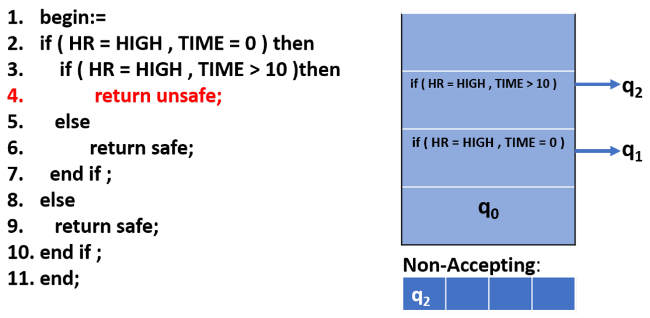
Figure 13 Step-2



```
1.  begin:=
2.  if ( HR = HIGH , TIME = 0 ) then
3.      if ( HR = HIGH , TIME > 10 )then
4.          return unsafe;
5.      else
6.          return safe;
7.      end if ;
8.  else
9.      return safe;
10. end if ;
11. end;
```

Figure 14 Step-3

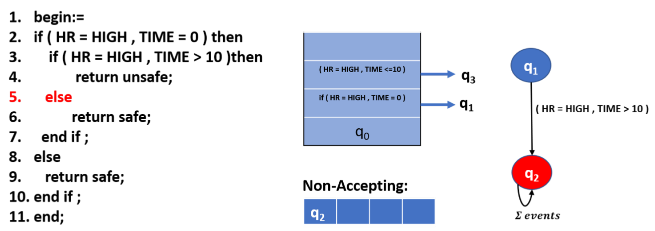
4. When a line starting with return unsafe statement is encountered mark the state corresponding to the stack top as non-accepting, following line-36 of the algorithm.



```
1.  begin:=
2.  if ( HR = HIGH , TIME = 0 ) then
3.      if ( HR = HIGH , TIME > 10 )then
4.          return unsafe;
5.      else
6.          return safe;
7.      end if ;
8.  else
9.      return safe;
10. end if ;
11. end;
```

Figure 15 Step-4

5. When a line starting with else statement is encountered, if the current stack top is a line starting with if statement we pop the current stack top and make the next stack top as is parent and then we push the else statement into the stack, following lines-[13-19] of the algorithm.

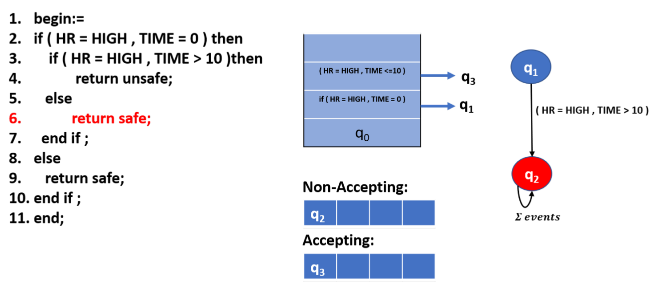


```
1.  begin:=
2.  if ( HR = HIGH , TIME = 0 ) then
3.      if ( HR = HIGH , TIME > 10 )then
4.          return unsafe;
5.      else
6.          return safe;
7.      end if ;
8.  else
9.      return safe;
10. end if ;
11. end;
```

(HR = HIGH , TIME <=10)
q_3
if (HR = HIGH , TIME = 0)
q_1
q_0
Non-Accepting:
q_2
q_1
(HR = HIGH , TIME > 10)
q_2
Σ events

Figure 16 Step-5

6. When a line starting with return safe statement is encountered mark the state corresponding to the stack top as non-accepting, following line-33 of the algorithm.

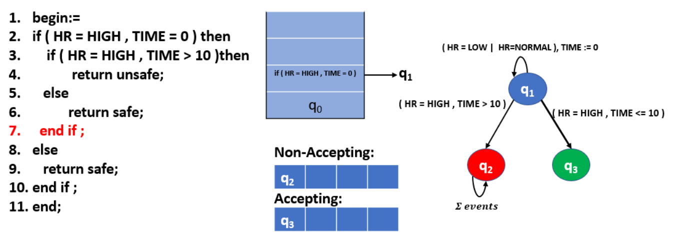


```
1.  begin:=
2.  if ( HR = HIGH , TIME = 0 ) then
3.      if ( HR = HIGH , TIME > 10 )then
4.          return unsafe;
5.      else
6.          return safe;
7.      end if ;
8.  else
9.      return safe;
10. end if ;
11. end;
```

(HR = HIGH , TIME <=10)
q_3
if (HR = HIGH , TIME = 0)
q_1
q_0
Non-Accepting:
q_2
Accepting:
q_3
q_1
(HR = HIGH , TIME > 10)
q_2
Σ events

Figure 17 Step-6

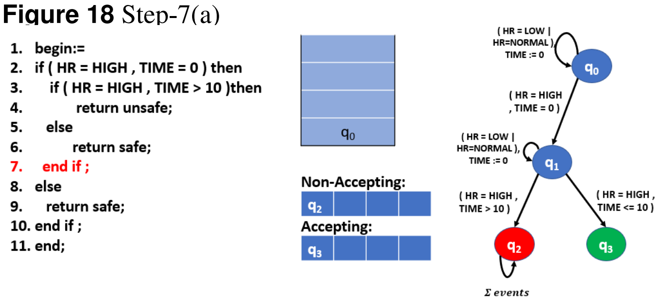
7. When a line starting with an endif statement is encountered, we pop the current stack top and make its parent as the next stack top. Continue this step until the stack top is start node q0, following lines-[23-31] of the algorithm. Also, add events related to HR-high, i.e., HR-low, HR-normal to a self-transition to the parent and reset the clock.



```
1.  begin:=
2.  if ( HR = HIGH , TIME = 0 ) then
3.      if ( HR = HIGH , TIME > 10 )then
4.          return unsafe;
5.      else
6.          return safe;
7.      end if ;
8.  else
9.      return safe;
10. end if ;
11. end;
```

if (HR = HIGH , TIME = 0)
q_1
q_0
Non-Accepting:
q_2
Accepting:
q_3
(HR = LOW | HR=NORMAL), TIME := 0
(HR = HIGH , TIME > 10)
(HR = HIGH , TIME <= 10)
q_2
q_3
Σ events

Figure 18 Step-7(a)

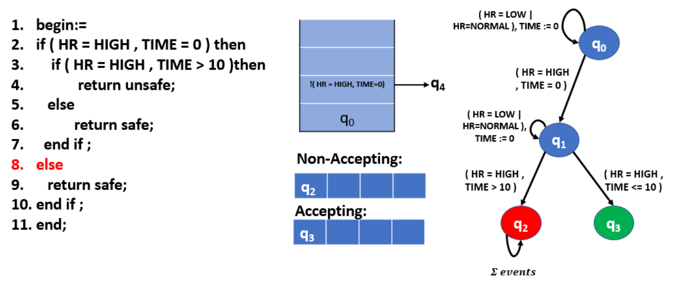


```
1.  begin:=
2.  if ( HR = HIGH , TIME = 0 ) then
3.      if ( HR = HIGH , TIME > 10 )then
4.          return unsafe;
5.      else
6.          return safe;
7.      end if ;
8.  else
9.      return safe;
10. end if ;
11. end;
```

q_0
Non-Accepting:
q_2
Accepting:
q_3
(HR = LOW | HR=NORMAL) TIME := 0
q_0
(HR = HIGH , TIME = 0)
(HR = LOW | HR=NORMAL), TIME := 0
q_1
(HR = HIGH , TIME > 10)
(HR = HIGH , TIME <= 10)
q_2
q_3
Σ events

Figure 19 Step-7(b)

8. When a line starting with the else statement is encountered, and the current stack top is not an if statement, push the else statement into the stack, following line-21 of the algorithm.

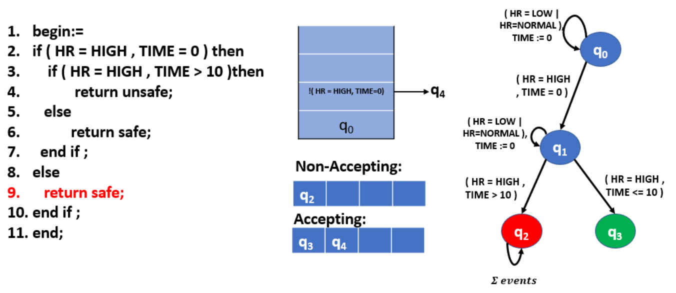


```
1.  begin:=
2.  if ( HR = HIGH , TIME = 0 ) then
3.      if ( HR = HIGH , TIME > 10 )then
4.          return unsafe;
5.      else
6.          return safe;
7.      end if ;
8.  else
9.      return safe;
10. end if ;
11. end;
```

!(HR = HIGH, TIME=0)
q_4
q_0
Non-Accepting:
q_2
Accepting:
q_3
(HR = LOW | HR=NORMAL), TIME := 0
q_0
(HR = HIGH , TIME = 0)
(HR = LOW | HR=NORMAL), TIME := 0
q_1
(HR = HIGH , TIME > 10)
(HR = HIGH , TIME <= 10)
q_2
q_3
Σ events

Figure 20 Step-8

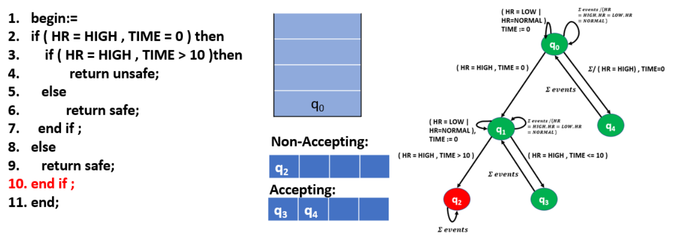
9. When a line starting with return safe statement is encountered mark the state corresponding to the stack top as accepting, following line-33 of the algorithm.



```
1.  begin:=
2.  if ( HR = HIGH , TIME = 0 ) then
3.      if ( HR = HIGH , TIME > 10 )then
4.          return unsafe;
5.      else
6.          return safe;
7.      end if ;
8.  else
9.      return safe;
10. end if ;
11. end;
```

Figure 21 Step-9

10. When a line starting with endif statement is encountered, we pop the current stack top and make its parent as next stack top. Continue this step until the stack top is start node q0, following lines-[23-30] of the algorithm.



```
1.  begin:=
2.  if ( HR = HIGH , TIME = 0 ) then
3.      if ( HR = HIGH , TIME > 10 )then
4.          return unsafe;
5.      else
6.          return safe;
7.      end if ;
8.  else
9.      return safe;
10. end if ;
11. end;
```

Figure 22 Step-10

About the authors

Rahul Bharadwaj Pendyala has completed M.Tech in computer sciences at the Indian Institute of Technology Bhubaneswar, India. You can contact the author at pr22@iitbbs.ac.in.

Abhinandan panda is pursuing PhD. at the Indian Institute of Technology Bhubaneswar, India. You can contact the author at ap53@iitbbs.ac.in.

Srinivas Pinisetty is an assistant professor at School of Electrical Sciences, Indian Institute of Technology Bhubaneswar, India. You can contact the author at spinisetty@iitbbs.ac.in.