Department of Electrical and Electronics Engineering

ELECTENG 789 Project Z,

Research Project Report

Security of Pacemakers with The External Wearable Device using Runtime Verification Approach

Prepared by Vidula Sawant

Student ID – 712746152

Supervised by Parthasarathi Roop

A research project report presented to the University of Auckland in partial fulfilment of the requirements of the degree of Master of Engineering Studies.

Declaration of Originality

I hereby declare that this report is my own original work and was not copied from anyone or anywhere nor written in collaboration with any other person. To the best of my knowledge, it contains no material which to a substantial extent has been submitted for the award of any other degree or diploma of a university or other institution of higher learning.

Vidula Sawant

Date 20th November, 2017

Security of Pacemakers with The External Wearable Device using Runtime Verification Approach

Vidula Sawant
University of Auckland, New Zealand
vsaw985@aucklanduni.ac.nz

Abstract—Recent studies have driven our attention to security vulnerabilities in cardiac pacemakers. Recall of 500,000 pacemakers was recorded because of hacking risks, which may lead to patient death fears. Hence, it is the need of the hour that such security threats be monitored, because in the interim, patient safety is under threat. In this project, we introduce an external wearable device which can counter such threats. The device monitors the heart-pacemaker actions via surface ECG signals. After processing and feature extraction from the ECG, it checks for violation of certain safety properties by feeding the extracted data to a verification monitor. Properties to be verified are formally expressed as timed automata, from which runtime verification monitors are generated. We consider monitor synthesis approaches for both dense and discrete time properties, and report on performance results comparing both these approaches.

Keywords—heart, pacemaker, wearable device, runtime verification monitor, dense timed automata, discrete timed automata, timed properties.

I. INTRODUCTION

A cardiac pacemaker is essentially an electrical device that helps manage irregular heart rhythms [1]. Technical advances in biomedical engineering have resulted in a boom of wirelessly accessible pacemakers. Recent investigations on pacemakers revealed security vulnerabilities on existing pacemakers available commercially [3], [4]. Millions of patients are able to lead a normal and prolonged life because of the invention of pacemakers. In the future, pacemakers are expected to be aware of Internet, and become a vital component in prevalent systems such as smart homes, android phones and hospitals, making pacemaker's security crucial. These artificial pacemakers are fitted with tiny radio components so they can be controlled and updated without having to cut them out and replace them each time. This means someone with the right technical knowledge and in the vicinity of the patient, could take control over the pacemaker, making it pace inappropriately. The hacker can make the pacemaker pace either too slow or too fast or may even drain the battery of the pacemaker.

These security risks are life threatening, which can make a life saving device a life killer. Thus, in this project we propose an externally wearable device which continuously monitors the ECG signals of the body for verifying critical safety properties defined for the heart-pacemaker activity using runtime verification techniques, giving additional layer of security as well as safety. Runtime verification mechanism deals with system analysis and execution approach based on extracting information from a running system (in this case heart-pacemaker system) and using it to detect and perhaps react to observed behaviors satisfying or violating certain properties [15]. Using RV mechanisms such as [9], RV monitors are generated from properties expressed as timed automata.

To this end, Section II presents the related work, Section III focuses on the background of the overall heart pacemaker system and briefly discusses about runtime verification, while Section IV gives an insight on the overview of the proposed approach. Section V details about the processing of the ECG signals and Section VI gives an introduction to timed automata. Section VII describes the implementation of the framework while the results are shown in Section VIII. We conclude and talk about future work in Section IX.

II. RELATED WORK

The growing use of cardiac pacemakers has encouraged researchers to study the security issues on pacemakers [11], [12]. Though their proposed solution is secure, it does not address what happens in an emergency situation when certain timers which the pacemaker uses to control the heart rate are violated.

The work by [13] explored the concept of security, and proposed the design of fail-open, a property to physically evade the pacemaker's security protection in an extremity, through the use of an external device. Another solution includes a spoofing attack resistant mechanism related to jamming. However, such jamming protocols do not consider the properties of the pacemaker, and cannot be directly implemented, since it requires complex hardware. In [5] they introduce IMDGuard which holds the external wearable device to coordinate interactions between the IMD (implantable medical device) and the doctor in such a way that it provides security in a regular condition, and safely allows access in an emergency. The patient's ECG signals are exploited to extract keys shared exclusively between the IMD and Guardian (security device). This solution, while secure does not consider the current design of the pacemaker, which does not allow pacemaker to send signals wirelessly. Our security device does not rely on the wireless communication between the pacemaker and the device. It is essentially an external observer.

III. BACKGROUND

III. A. CARDIAC PACEMAKERS

The pacemaker is implanted under the skin of the patient's chest, just under the collarbone, hooked up to the heart with small wires. Pacemakers are used to treat arrhythmias. Arrhythmias are problems with the rate of the heartbeat. During an arrhythmia, the heart can beat too fast, which is called Tachycardia, or too slow, which is called Bradycardia. The pacemaker senses intrinsic events (Atrial and Ventricular events) of the heart and gives electrical pacing pulses (either an atrial pulse or a ventricular pulse) whenever necessary [2].

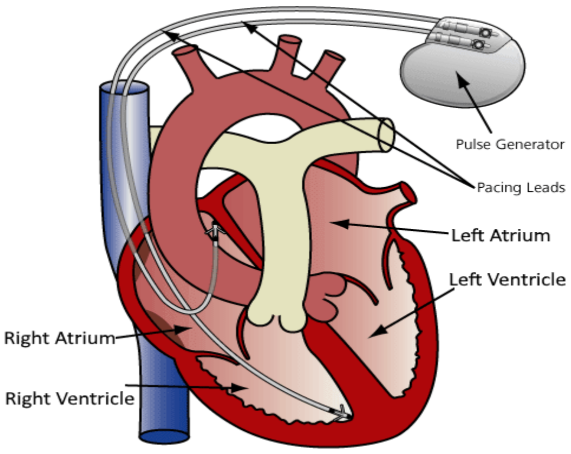


Fig.1: The heart-pacemaker system showing the atrial and the ventricular chambers with pacing leads [20]

The impulse starts in a small bundle of dedicated cells located in the right atrium, called the SA (Sinoatrial node) node. The electrical activity spreads through the walls of the atria and causes them to contract. This forces blood into the ventricles. The SA node sets the rate and rhythm of your heartbeat. This activity of the heart is reflected in its ECG signals as shown in Figure 2.

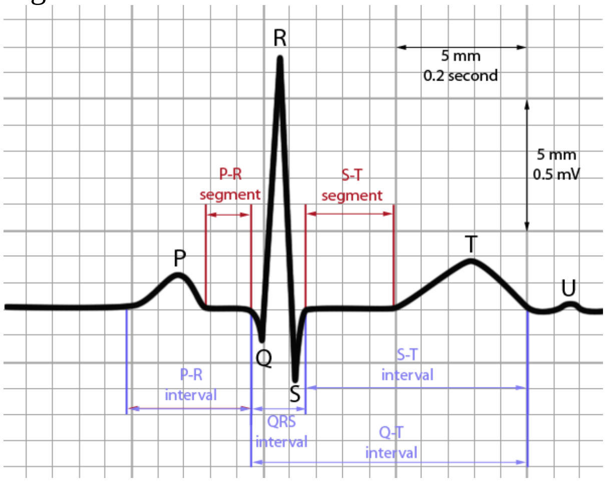


Fig.2: A typical ECG signal [20]

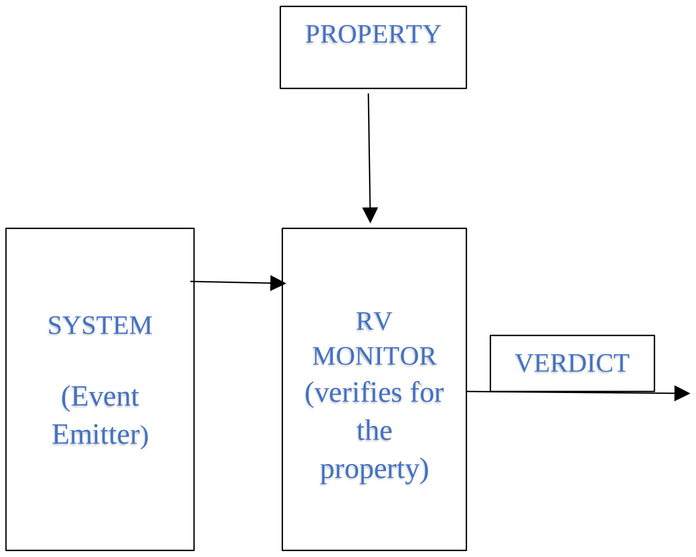
The P-R interval symbolizes the atrial depolarization (indicating an atrial event has occurred). The QRS complex denote the ventricular contraction (indicating ventricular event has occurred), atrial repolarization also happens during this interval. The S-T interval marks ventricular repolarization. Thus, a lot of information regarding the heart pacemaker activity can be extracted from the ECG signals of the patients, which is exactly upon which our device is based.

III. B. RUNTIME VERIFICATION

Runtime verification refers to the theories, techniques and tools to continuously check whether the system under scrutiny satisfies or violates a desired property [15]. It complements other verification techniques such as model checking and testing. A verification monitor does not affect or alter the program execution. Such a monitor can be used to check the current execution of a system (online) or a stored execution of a system (offline) [19]. RV Monitor takes a stream of events (execution of system being monitored) and emits verdicts. In RV approaches [9], verification monitors are automatically obtained from high-level formal description of properties.

As illustrated in Figure 3, a runtime verification monitor takes a sequence of events from an event emitter (system being monitored) as input and produces a verdict as output that provides information whether the current execution of the system satisfies the required property or not.

Fig. 3: Block Diagram of Runtime verification monitor



IV. OVERVIEW OF THE PROPOSED APPROACH

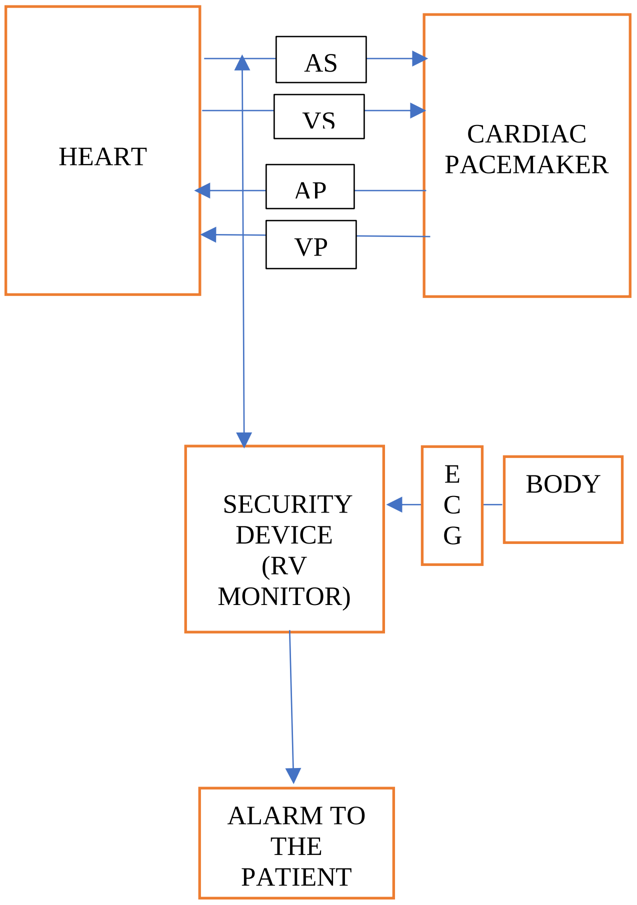
IV.A. ARCHITECTURE

In this section, we outline the heart-pacemaker-security device architecture and the basic principles of runtime verification.

Pacemakers come in various modes depending upon the patient’s requirement example VVI, DDI, VDD, DDD etc. The mode of our interest is the DDD mode in which both atria and ventricles are sensed as well as paced.

In Figure 4, AS and VS stand for atrial and ventricular sense respectively (activity of the heart sensed by the pacemaker). AP and VP stand for atrial and ventricular pulse respectively (pacing pulses given by the pacemaker). Our wearable security device (RV monitor) is essentially an external examiner which takes ECG signals from the body, processes it to extract required data for verification, checks for violation of properties and gives output to the patient in an event an anomaly has been detected.

Fig.4: Block Diagram of the architecture



IV. B. TIMING INTERVALS AND PROPERTIES

The function of a pacemaker is to manage the timing relationship between the atrial and ventricular events. Thus, Timed Automata is suitable for the runtime verification of heart-pacemaker activity from ECG signals. In automata theory, a timed automata is a finite automata enriched with a finite set of real-valued clocks. During a run of a timed automaton, clock values increase all with same speed. Along the transitions of the automaton, clock values can be compared to integers [7], [19].

In this paper we test the pacemaker (precisely ECGs) for the following properties:

Property1: Atrial (P-peak) and Ventricular (Q-peak) events cannot happen simultaneously.

Property2: Ventricular event (Q-peak) must be true within Ppeak-Qpeak interval after an atrial event (P-peak).

Property3: Atrial event (P-peak) must be true within Qpeak-Ppeak interval after a ventricle event (Q-peak).

Property4: After a ventricle event (Qpeak), another ventricle event (Qpeak) can happen only after ST interval.

Property5: After a ventricle event (Qpeak), another ventricle event (Qpeak) should happen within Qpeak-Qpeak interval.

It is ensured that these properties do not interfere with the pacemaker timers (LRI, AVI, etc) [8].

These timing intervals are real-time values. We have also considered discrete-time variants of the above properties, and examined the verification monitor synthesis approaches for both dense and discrete time properties. We obtain discrete time values of the intervals by considering TICKS. Properties are defined as dense (discrete) timed automata, for which RV monitors are obtained. The security device checks for violation of these properties and alerts the user if any anomaly has occurred.

IV. C. CONFIGURATION OF THE EXTERNALLY WEARABLE SECURITY DEVICE

The pacemaker, once implanted, is expected to remain inside the body for an extended period of time. Before implanting it is programmed by the doctor with the help of a programming unit (outside controller) with direct connection. After implantation, if the pacemaker has to be reprogrammed, that should be done wirelessly. The programming unit provides doctors an interface to interact with pacemaker through radio frequency transmission for adjusting running parameters (timers), changing operation modes, or retrieving stored data [16]. The pacemaker and the security device are standard wireless programmable medical instruments. The externally wearable device is assumed to have more power and computational resources than the pacemaker. It is capable of measuring ECG signals.

The US Food and Drug Administration (FDA) recently recalled approximately 465,000 pacemakers made by the company Abbott's (formerly St. Jude Medical) that were vulnerable to hacking, but the situation points to an ongoing security problem. Most pacemakers don't currently have the memory, processing power or battery life to support proper cryptographic security, encryption or access control. The vulnerability of the pacemaker appears to be that someone with "commercially-available equipment" could send commands to the pacemaker, changing its settings and software [17].

Whenever the pacemaker is implanted, the doctor programs the pacemaker's computer with an external programming unit. The doctor doesn't have to use needles or have direct contact with the pacemaker. The two main types of programming for pacemakers are demand pacing and rate-responsive pacing. Doctor will work with the patient to decide which type of pacemaker is best for the patient. While programming the doctor essentially sets the pacing mode (eg. DDD), sets the threshold voltage value of the pacing pulse, sensitivity of the pacemaker and most importantly the timers (AVI, AEI, LRI, etc) [8]. If the hacker hacks the pacemaker he/she may try to increase or decrease any of these timers. Hence, the properties in Section IV. B are formulated keeping in mind this security vulnerability.

According to our proposed work, the patient is given the security device once the pacemaker is implanted. The device is also programmed by the doctor, who initially sets the authentication details. The next time when the device is to be programmed, it first has to go through an authentication process. So, the chances of any malicious entity hacking the security device is lowered. Essentially, all the values of the set timers are stored inside the security device's memory. It also knows the normal heartrate at which the pacemaker is set to pace (eg. 60-120 BPM). The device also has information of the attributes of the pacing pulses like voltage, current, impedance. The device has built-in accelerometer which monitors the activity of the body.

The device has access to the surface ECG signals. It continuously monitors these signals. After appropriate processing, it extracts the peaks of P, Q, R, S, T waves and the pacing pulses. Depending on the time instances at which the peaks are occurring, it checks for any violation of the mentioned properties in Section IV.B using runtime verification algorithm. If any of the timing intervals are violated, it gives an alarm to the user.

V. PROCESSING OF THE ECG SIGNALS

The signals processing on ECG signals is done in MATLAB. The database of ECG signals with pacing artifacts is obtained from [18]. It essentially contains different ECGs combined with artificially superimposed pacing pulses, corresponding to various pacemaker modes. The database is publicly available for development and testing purpose.

Generally, a raw ECG signal (Figure 5) is corrupted by Baseline wander, Power line interference (50 Hz or 60), Electromyographic (EMG) or muscle noise, Artifacts due to electrode motion and Electrode Contact Noise [14]. Hence, pre-processing of the signal is important to remove unwanted data and to be able to extract vital information from the signals. Savitzky-Golay filtering is used to remove noise from the signal as shown in Figure 6.

After the filtering of the signal, the waves of our interest are detected using peak analysis [14] as shown in Figure 6. X-axis denotes time in seconds and Y-axis denotes amplitude in millivolts.

Figure 8 shows a closer look at the processed signal. The P, Q, R, S, T waves and the pacing pulses are extracted using the function “findpeaks” in MATLAB (refer Appendix A). The signal begins with a pacing pulse (atrial pacing pulse), followed by P-wave, then another pacing

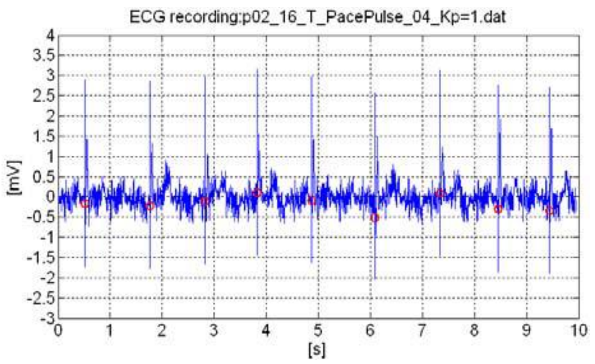


Fig. 5: Raw ECG signal [18]

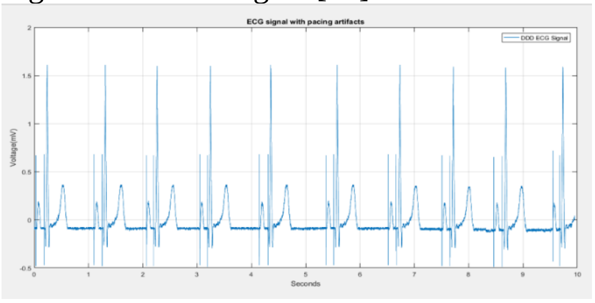


Fig. 6: Filtered ECG signal

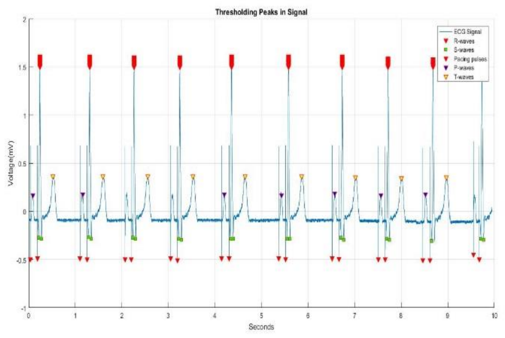


Fig. 7: ECG after feature extraction

pulse (ventricular pacing pulse), followed by QRS complex and T-wave.

After processing the code returns arrays of P, Q, R, S T, waves and the pacing pulses (basically the time in seconds at which the peaks are occurring). Thus, a stream of timed events from the ECG data is obtained.

For verification purpose, the time instances at which the P peaks occur are considered as atrial events arrival and the time instances at which Q peaks occur are considered as ventricular events arrival. R waves detection help in calculating the heartbeat every minute. Hence, the security device can constantly check for the normal heartrate (beats per minute) too.

Also, detection of pacing pulses gives crucial information about the amplitude of the pulses. Based on this information the condition of the leads can be found out, which will help in examining any malfunctions in the lead.

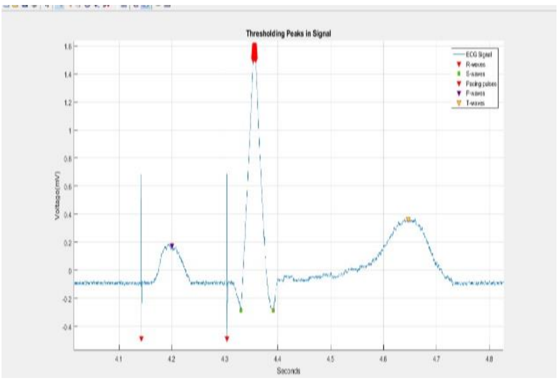


Fig.8: Closer view

VI. TIMED AUTOMATA

In order to obtain RV monitors from properties mentioned in Section IV.B, we need to formally express properties as timed automata. In this section we briefly discuss about dense and discrete timed automata via examples.

VI. A. DENSE TIMED AUTOMATA

Definition (Dense Timed automata). A timed automaton (TA) is a tuple *A* = (L, l0, X, Σ, Δ, G), such that L is a finite set of locations with l0 ∈ L the initial location, X is a finite set of clocks, Σ is a finite set of actions, Δ ⊆L x G(X) x Σ x 2^X x L is the transition relation. G is a set of accepting locations [7].

Consider the following property: P1: “p and q cannot happen simultaneously. p and q alternate starting with a p. There should be a delay of at least 10 time units between p and q.”

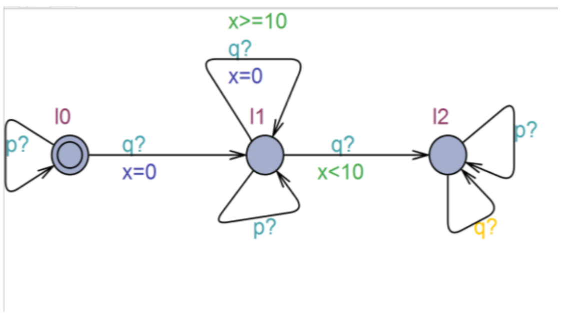


Fig. 9: TA defining property P1

The TA in figure 9 defines property P1. Location l_0 is the initial location, location l_2 is the non-accepting trap location and location l_1 is an intermediate location. Clock x is reset to 0 upon transition from l_0 to l_1 when q occurs. The transition from l_1 to l_1 (self-loop) is taken when p occurs or when q occurs after or at 10 time units (i.e. the property is satisfied). The transition from l_1 to l_2 is taken when q comes before 10 time units (i.e. the property is violated).

VI. B. DISCRETE TIMED AUTOMATA

DTA are timed automata extended with a set of integer variables that are used as discrete clocks for example to count the number of ticks before a certain event occurs. Time passage is modelled through a special action TICK [8].

Definition (Discrete Timed Automata). A Discrete Timed Automaton is a tuple A = (L, l0, lv, Σ , V, Δ, F) where

L => set of locations,

L0 => is the initial location,

Σ => is the alphabet,

V => set of integer clocks,

F => set of accepting locations,

lv => non-accepting trap location.

Δ => transition relation ($\Delta \subseteq L \times G(V) \times R \times \Sigma \times L$ where G(V) denotes the set of guards, i.e., constraints of the form $\{<, \leq, =, \geq, >\}$ and $R \subseteq V$ is a subset of integer clocks that are reset to 0.

Consider the same property as expressed in Section VI. A and Figure 9. In DTA instead of time units there will be TICKS, time is discretized. Hence, whenever an event occurs, the TICK is incremented by 1. So here, we check the property for 10 TICKS. Thus, we are checking for real time, instead we observe for TICKS.

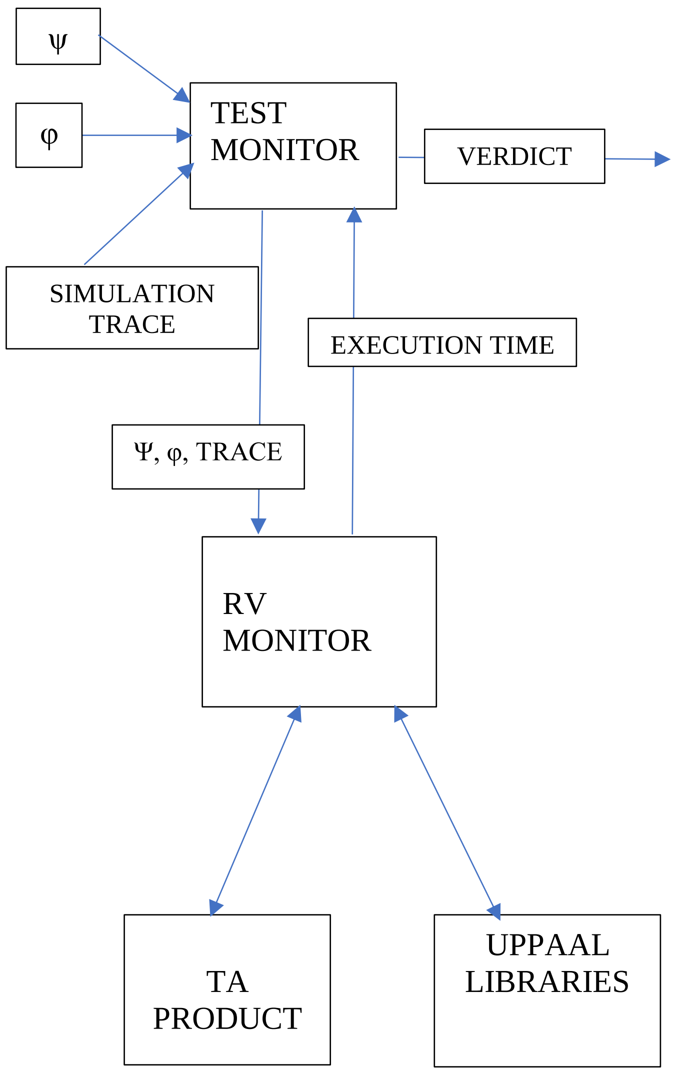
VII. RV MONITORS FROM PROPERTIES DEFINED AS TIMED AUTOMATA

In this section we discuss the implementation of the runtime verification monitor from properties defined as dense and discrete timed automata.

VII. A. IMPLEMENTATION OF RV MONITORS FROM PROPERTIES DEFINED AS DENSE TIMED AUTOMATA

In [9] they have proposed tools to generate RV monitors from properties expressed as TAs. The implementation of RV algorithm as discussed in [9] is shown as below.

Fig. 10: Implementation of RV algorithm



In Figure 10, ψ denotes the input property (property of the system) and φ denotes the property to be verified. Both the properties are expressed as TAs and represented in UPPAAL .XML format. The implementation uses publicly available libraries and tools for timed automata based on the UPPAAL model-checker. The Algorithm [9] is implemented in 700 lines of code in Python (excluding libraries) (see Appendix B).
The Test Monitor is invoked with two TA defining ψ and φ, and a simulation trace which is a timed word that belongs to the property of the system ψ. The Test monitor then invokes the RV monitor multiple times for the given inputs, computes average values over multiple executions and stores the performance results. The Test Monitor then emits verdicts obtained from the RV monitor. The TA Product module contains functionality to compute the product of two TAs (ψ and φ).The RV Monitor module also uses pyuppaal and UPPAAL DBM libraries.
According to our case study, the tool requires as input property all the actions which the automaton can take. For simplification and to stick to the properties mentioned in Section IV.B, we only consider the p peaks, q peaks and pacing pulses as events where p-peaks denote atrial events, q-peaks denote ventricular events. Our input property is as shown in Figure 11.

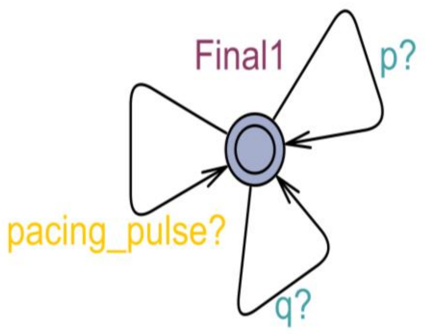


Fig. 11: Input property

Let the Test Monitor take Property 4 (Section IV.B) as φ (property to be verified).

The TA defining Property4 is as shown in Figure 12. ST interval to be checked is set to 900 milliseconds for demonstration. Y is the clock. The Test Monitor is invoked passing the input property (Figure 11), property to verify (Figure 12) and a simulation trace to check whether the property is satisfied. Note that the simulation trace is given manually and not directly from MATLAB. But it is mimicked from the ECG signal processing. An example of a simulation trace could be [(‘p’,89),(‘q’,212),(‘p’,160),(‘q’,650)] (where each event is associated with a delay, indicating the time elapsed after the previous event or the system initialization for the first event.) Here, the monitor receives the first action ‘p’ at t=89. Since, the property is not violated the monitor will output “Verdict True”. The second action ‘q’ comes at t=301 and the monitor will again emit verdict true. The third action ‘p’ comes at t= 461, again the monitor will emit true. The fourth action ‘q’ comes at t=1111, here the property is violated because there should be a minimum interval of 900 between two q’s (ventricular events), therefore the monitor will output “Conclusive Verdict False”, indicating property is violated and the security device will display an alert message. The monitor also returns the time taken for execution and the size of memory required.

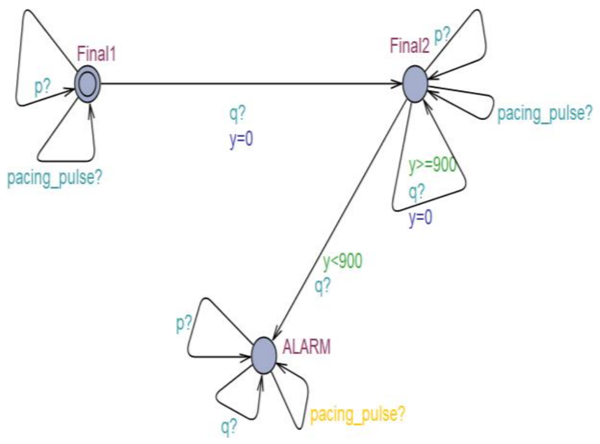


Fig. 12: Property 4 (ST- interval)

Thus, the approach can indeed detect violation of properties ultimately checking for any hack.

VII. B. IMPLEMENTATION OF RV MONITORS FROM PROPERTIES DEFINED AS DISCRETE TIMED AUTOMATA

This implementation is essentially a modification of the work presented in [10], [20]. The implementation is performed in python.

The required libraries for runtime verification necessary for this case study are taken from [10], [20]. Unlike Section VII. A, here we only need the property to be verified and not the input property (property defining the system). Also, we need not give a user defined simulation trace. The simulation trace is directly imported from MATLAB in the main python code. The DTAs defining the properties are generated using the class “class DTA” in python. The transitions in the DTA are obtained using the transition function “delta_ta”. (see Appendix C).

The trace obtained from MATLAB is real time, but the function verifying the property takes only Boolean inputs as events. Hence, initially the trace is converted in to Boolean inputs before passing it to the verifier. In main, the property, DTA and clocks are invoked. The verifier codes are imported in the beginning of the code. All the properties are generated and tested for violation, also the time taken for the execution is calculated at every iteration. The monitor works correctly (perfectly checks for violation of properties) for all the properties without any errors.

VIII. RESULTS

In order to evaluate and compare the performance of the runtime verification approach, it was implemented in both Dense Timed Automata as well as Discrete Timed Automata.

The comparison results are as shown in the Table 1.

Table 1 shows the Line of codes (LoC) and time taken for the execution by all the properties in TA and DTA. Note that the time taken varies with the length of the simulation trace in TA and Maximum ticks in DTA. In both the implementation, the simulation trace taken to execute the codes was same for comparison purposes. It is clear from the table that DTA takes lesser time and lesser LoC than TA. But, implementation for longer traces (as long as 1k time units) is not feasible in DTA as done in TA. Hence, both the implementations have their own pros and cons.

Property	LoC (TA)	LoC (DTA)	Time(s) TA	Time(s) (DTA)
P1	700	152	0.1	2.75e-07
P2	700	185	0.2	7.55e-07
P3	700	185	0.2	7.55e-07
P4	700	182	0.4	3.77e-07
P5	700	182	0.4	3.77e-07

Table. 1: Results of the implementation

IX. CONCLUSION AND FUTURE WORK

This report presents a solution to the security issues of the pacemaker by introducing an external wearable security device. The required task is performed using runtime verification techniques. The implementation is done expressing the essential properties in Dense Timed Automata and Discrete Timed Automata. Along with providing security of the pacemaker the device can also be used to notify any malfunctions of the pacemaker and its leads which comes as an added advantage.

In future, the project can be extended to having real time feature extraction in Matlab (Simulink) and integrating that with Python, by calling Python functions directly from MATLAB. Also, the security device can be trained to be smarter, by which it will verify properties considering the activity of the patient.

ACKNOWLEDGMENTS

The author expresses gratitude to Dr. Parthasarthi Roop for his valuable guidance throughout the project. The author would like to especially thank Dr. Srinivas Pinisetty for making her available the required resources for implementation and essential inputs and feedback. Lastly, the author thanks Weiwei Ai, Kalman Zambo, Bevan Krause, Anuj Gangan and Mihir Ranade for providing timely help and support.

REFERENCES

1) https://www.nhlbi.nih.gov/health/health-topics/topics/pace
2) https://www.heart.org/idc/groups/heart-public/@wcm/@hcm/documents/downloadable/ucm_300451.pdf
3) https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4516335/
4) http://www.bbc.com/news/technology-41099867
5) Fengyuan Xu, Zhengrui Qin, Chiu C. Tan, Baosheng Wang, and Qun Li. 2011. IMDGuard: Securing Implantable Medical Devices with the External Wearable Guardian.
6) Patricia AH Williams, Andrew J Woodward. Cybersecurity vulnerabilities in medical devices:a complex environment and multifaceted problem. Medical Devices: Evidence and Research 2015:8 305–316
7) Srinivas Pinisetty, Yliès Falcone, T Jéron, Hervé Marchand, Antoine Rollet, et al.. Runtime enforcement of timed properties revisited. Formal Methods in System Design, Springer Verlag, 2014, 45 (3), pp.381-422.
8) Srinivas Pinisetty, Partha S Roop, Steven Smyth, Nathan Allen, Stavros Tripakis, and Reinhard von Hanxleden. 2017. Runtime Enforcement of Cyber-Physical Systems. 1, 1, Article 1 (July 2017), 25 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn
9) Srinivas Pinisetty, Thierry J´eronb, Stavros Tripakisc, Yli`es Falconed, Herv´e Marchandb, Viorel Preoteasaa. Predictive Runtime Verification of Timed Properties. Journal of Systems and Software. June 23, 2017
10) S. S. S. T. R. v. H. S. Pinisetty, P. S. Roop, "Runtime enforcement of reactive systems using synchronous enforcers," April 2017, submitted.
11) T. Denning, Y. Matsuoka, and T. Kohno. Neurosecurity: security and privacy for neural devices. *Neurosurgical Focus*, 2009.
12) D. Halperin, T. Heydt-Benjamin, K. Fu, T. Kohno, and W. Maisel.Security and privacy for implantable medical devices. *IEEE Pervasive Computing 2008*.
13) T. Denning, K. Fu, and T. Kohno. Absence makes the heart grow fonder: new directions for implantable medical device security. In *HotSec 2008*.
14) https://au.mathworks.com/help/signal/examples/peak-analysis.html
15) https://en.wikipedia.org/wiki/Runtime_verification
16) https://www.youtube.com/watch?v=iWZoOeEtdfg
17) http://theconversation.com/three-reasons-why-pacemakers-are-vulnerable-to-hacking-83362
18) Irena Jekova1*, Valentin Tsibulko2 , Ivo Iliev2. ECG Database Applicable for Development and Testing of Pace Detection Algorithms. INT.J. BIOAUTOMATION, 2014, 18(4), 377-388
19) Srinivas Pinisetty, Viorel Preoteasa, Stavros Tripakis, Thierry Jéron, Yliès Falcone, et al.. Predictive Runtime Enforcement *. SAC 2016 31st ACM Symposium on Applied Computing, Apr 2016, Pisa, Italy. ACM, pp.6, 2016.
20) https://www.google.co.nz

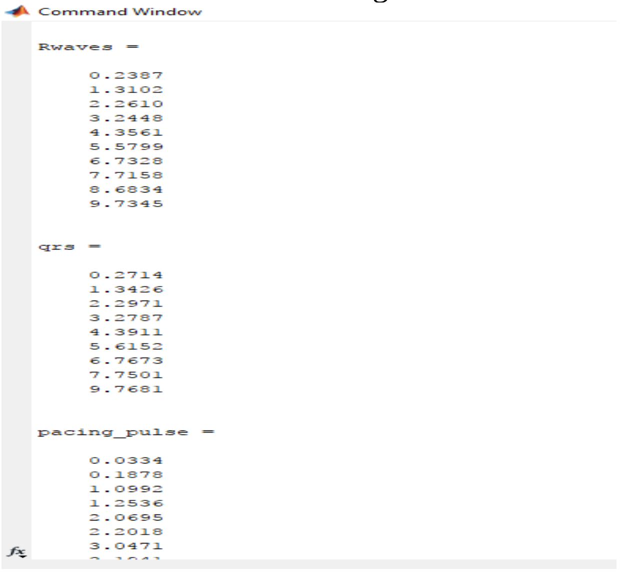
APPENDIX A

MATLAB code for ECG signal processing

```
clear all; clc; close all;
load pacing

Fs = 128000;

t = 1/Fs:1/Fs:length(ECG)/Fs; % seconds
plot(t,ECG)
title('ECG signal with pacing artifacts')
xlabel('Seconds');
ylabel('Voltage(mV)')
legend('DDD ECG Signal')
grid on
```

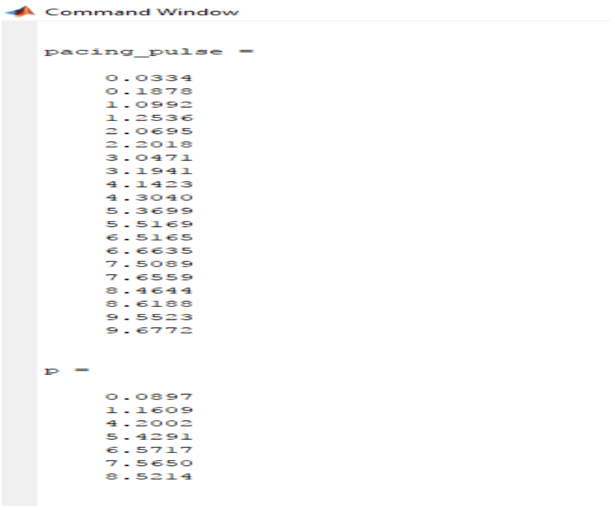
```
%Savitzky-Golay filtering is used to remove noise
in the signal.
ECG = sgolayfilt(ECG,7,21);

%R peaks detection
[~,locs_Rwave] =
findpeaks(ECG,'MinPeakHeight',1.5,...
                        'MinPeakDistance',100000);
Rwaves = locs_Rwave/Fs

[~,min_locs] = findpeaks(-
ECG,'MinPeakDistance',10000);

%Q and S peak detection
locs_Swave = min_locs(ECG(min_locs)>-0.315 &
ECG(min_locs)<-0.25);
qrs = locs_Swave/Fs

%Pacing pulse detection
[~,locs_pulse] = findpeaks(-
ECG,'MinPeakHeight',0.4,...
                        'MinPeakDistance',10000);

pacing_pulse = locs_pulse/Fs

%P peak detection
[~,min_locs2] =
findpeaks(ECG,'MinPeakDistance',7000);
locs_Pwave = min_locs2(ECG(min_locs2)>0.15 &
ECG(min_locs2)<0.2);
p = locs_Pwave/128000

%T peak detection
[~,min_locs3] =
findpeaks(ECG,'MinPeakDistance',7000);
locs_Twave = min_locs3(ECG(min_locs2)>0.34 &
ECG(min_locs2)<0.4);
Twaves = locs_Twave/128000

figure
hold on
plot(t,ECG)
plot(Rwaves,ECG(locs_Rwave),'rv','MarkerFaceCol
or','r');
plot(qrs,ECG(locs_Swave),'rs','MarkerFaceColor','g'
);
plot(pacing_pulse,ECG(locs_pulse),'rv','MarkerFac
eColor','r');
plot(p,ECG(locs_Pwave),'rv','MarkerFaceColor','b');
plot(Twaves,ECG(locs_Twave),'rv','MarkerFaceCol
or','y');
axis([0 10 -1 2])
grid on
title('Thresholding Peaks in Signal')
legend('ECG Signal','R-waves','S-waves','Pacing
pulses','P-waves','T-waves')
xlabel('Seconds')
ylabel('Voltage(mV)')

%Calculation of beats per minute
beat_count = length(Rwaves);
N = length(ECG)/Fs
dur_in_min = N/60
BPM = beat_count/dur_in_min
```

Results obtained after running the code.



```
Command Window

Rwaves =

    0.2387
    1.3102
    2.2610
    3.2448
    4.3561
    5.5799
    6.7328
    7.7158
    8.6834
    9.7345


qrs =

    0.2714
    1.3426
    2.2971
    3.2787
    4.3911
    5.6152
    6.7673
    7.7501
    9.7681


pacing_pulse =

    0.0334
    0.1878
    1.0992
    1.2536
    2.0695
    2.2018
    3.0471
```

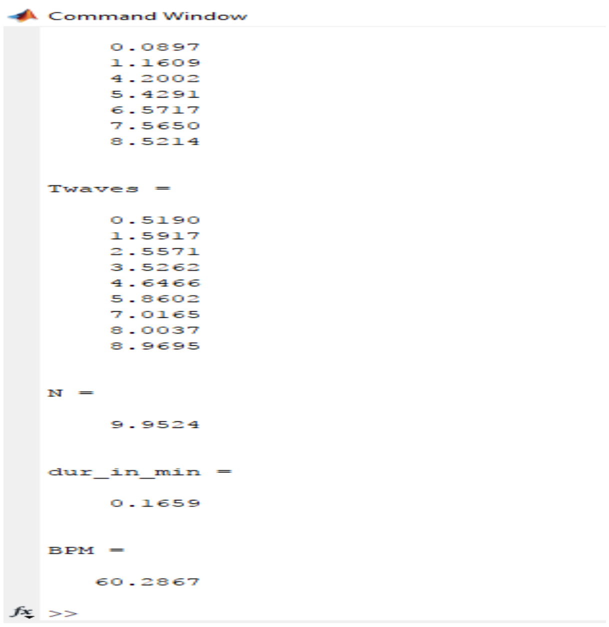
Fig. 13: Arrays of R,Q,S waves and pacing pulses



```
Command Window

pacing_pulse =

    0.0334
    0.1878
    1.0992
    1.2536
    2.0695
    2.2018
    3.0471
    3.1941
    4.1423
    4.3040
    5.3699
    5.5169
    6.5165
    6.6635
    7.5089
    7.6559
    8.4644
    8.6188
    9.5523
    9.6772


p =

    0.0897
    1.1609
    4.2002
    5.4291
    6.5717
    7.5650
    8.5214
```

Fig. 14: Arrays of p waves and pacing pulses

Figure 13, 14 and 15 show the arrays obtained for the peaks of the waves. In Figure 15 one can see how beats per minute is calculated.



```
        0.0897
        1.1609
        4.2002
        5.4291
        6.5717
        7.5650
        8.5214


Twaves =

        0.5190
        1.5917
        2.5571
        3.5262
        4.6466
        5.8602
        7.0165
        8.0037
        8.9695


N =

        9.9524


dur_in_min =

        0.1659


BPM =

       60.2867

>>
```

Fig. 15: Array of Twave and BPM

APPENDIX B
Python implementation of Runtime Verification Algorithm in Dense TA

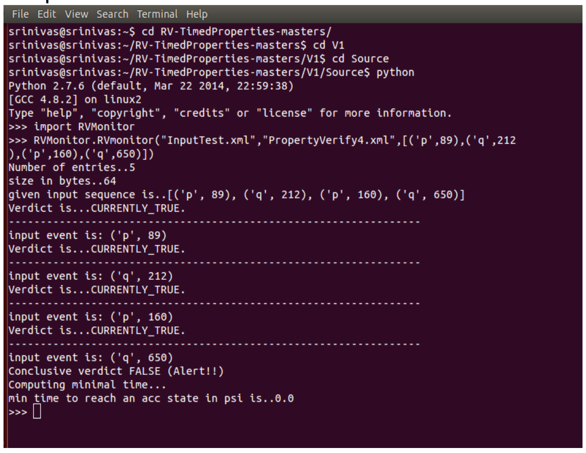
```
# Imports
import sys
import os.path
import pyuppaal
import dbmpyuppaal
import pydbm.udbm
import subprocess
import re
import time
import copy
import productTA
import shutil
import xml.etree.ElementTree as ET
import cPickle

#######  Runtime Verification Monitor#######
INPUT parameters #####
##########--Input property psi--UPPAAL model as
xml, containing automaton representing the input
property psi
##########--Property to verify-UPPAAL model as
xml, containing automaton representing the
property to verify varphi.
##########################
-inputTrace--a sample input timed word belonging
to the input property psi, where each event
consists of an action and a delay. ####
def RVmonitor(inputProp, prop, inputTrace):
    t1 = time.time()
    # PYUPPAAL
    psi =
pyuppaal.NTA.from_xml(open(inputProp)).templat
es[0] # input property psi.
    varphi =
pyuppaal.NTA.from_xml(open(prop)).templates[0]
# property to verify varphi.

    ## Accepting locations in  psi and in varphi ##
    accPsi = getAccLoc(psi)
    accVarphi=getAccLoc(varphi)
    ##################
    # DBMPYUPPAAL
    Parsed_psi_DbmPyUppaal =
dbmpyuppaal.parse_xml(inputProp)
    Parsed_varphi_DbmPyUppaal =
dbmpyuppaal.parse_xml(prop)
    Psi_Automaton =
Parsed_psi_DbmPyUppaal.getTemplateByName("P
roperty")
    Varphi_Automaton =
Parsed_varphi_DbmPyUppaal.getTemplateByNam
e("Property")
    psiCurrentState =
dbmpyuppaal.NTAState(Parsed_psi_DbmPyUppaal
).delay().extrapolateMaxBounds()
    varphiCurrentState =
dbmpyuppaal.NTAState(Parsed_varphi_DbmPyUp
paal).delay().extrapolateMaxBounds()
    ###Clocks#########
    # Clocks in the psi and varphi and initializing all
the clock values with 0.0.
    clksPsi=[]
    clksVarphi=[]
    totalDuration=0.0
    for clock in Parsed_psi_DbmPyUppaal.clocks:
        clksPsi.append((str(clock),0))
    for clock in
Parsed_varphi_DbmPyUppaal.clocks:
        clksVarphi.append((str(clock),0))

    #pyuppaal.
    ######################
    ## Compute product automaton B (product of
psi and varphi), and the final location sets F and
Fneg. ####
    ### first parameter is the input property, and
the second parameter should be the property to
verify.##
    result= productTA.Product(psi, varphi)
    ## product aut B.
    automatonB = result[0]
    ## Final loc F (i.e. F_psi*F_varphi).
    locF = result[1]
    ## Final loc Fneg (i.e., F_psi*neg(F_varphi)).
    locFneg = result[2]

    #######################
    tree = ET.parse("baseData.xml")
    root = tree.getroot()
    root.find('declaration').text =
Parsed_psi_DbmPyUppaal.declaration+"\n"+getCl
ockStr(clksVarphi)
    tree.write("baseData.xml")
    #######################

    ### Storing the product automaton as an
UPPAAL model in xml format.###
    shutil.copy("baseData.xml", "productTA.xml")
    product =
pyuppaal.NTA.from_xml(open("productTA.xml"))

    #product.templates[0] = automatonB
    product.add_template(automatonB)
    ####File operations (writing the updated model
to file).###
    f = open("productTA.xml", "r+")
    f.writelines(pyuppaal.NTA.to_xml(product))
    f.close()
    ######################
    Parsed_product_DbmPyUppaal =
dbmpyuppaal.parse_xml("productTA.xml")
    productCurrentState =
dbmpyuppaal.NTAState(Parsed_product_DbmPyU
ppaal).delay().extrapolateMaxBounds()
    autB_DBMpy =
Parsed_product_DbmPyUppaal.getTemplateByNa
me("TA_Product")
    ######################
    ## Pre-compute all reachable (symbolic) states,
and for each state compute if locations "locF" and
if locations "locFneg" are reachable from that state
##
    preComputeReach =
symbolicStatesAut(autB_DBMpy,
productCurrentState, locF, locFneg)

    print "Number of
entries.."+str(preComputeReach[1].__len__())

    #size =
sys.getsizeof(cPickle.dumps(preComputeReach[1])
)
    size = sys.getsizeof(preComputeReach[1])
    print "size in bytes.."+str(size)
    t2 = time.time()

    #### Online monitoring starts here.
Read/process events in the given sample input
trace ############

    print "given input sequence is.."+str(inputTrace)
    while True:
        id =
preComputeReach[0].index(productCurrentState)
        reachCurrState = preComputeReach[1][id]
        reachFneg= reachCurrState[1]
        reachF = reachCurrState[2]


        if not reachFneg:
            print "Conclusive verdict TRUE (Non-
accepting loc is not reachable in future!!)"
            print "Computing minimal time..."
            res=computeReachPaths(Psi_Automaton,
psiCurrentState,accPsi)
            minTime = getOptimalDelays(res[1],  clksPsi)
            print "min time to reach an accepting state
in psi is.."+str(minTime)
            break
        elif not reachF:
            print "Conclusive verdict FALSE (Violation is
unavoidable!!)"
            print "Computing minimal time..."
            res=computeReachPaths(Psi_Automaton,
psiCurrentState,accPsi)
            minTime = getOptimalDelays(res[1],  clksPsi)
            print "min time to reach an acc state in psi
is.."+str(minTime)
            break
        elif
str(psiCurrentState.locations["Property"].name) in
accPsi and
str(varphiCurrentState.locations["Property"].name
) in accVarphi:
            print "Verdict is...CURRENTLY_TRUE."
        elif
str(psiCurrentState.locations["Property"].name) in
accPsi and not
str(varphiCurrentState.locations["Property"].name
) in accVarphi:
            print "Verdict is...CURRENTLY_FALSE."
        else:
            print "Verdict is..UNKNOWN."
        print "---------------------------------------------------
------------"
        ##
        if inputTrace.__len__()==0:
            break
                # Remove and take first event from the
trace ##
        ## event is a tuple (action, delay) ###
        event = inputTrace.pop(0)
        print "input event is: " + str(event)
        totalDuration = totalDuration+event[1]

        ## Increment all clock values with the delay
corresponding to the current event. ##
        for i in range(0,clksPsi.__len__()):

            clksPsi[i] = (clksPsi[i][0],
clksPsi[i][1]+event[1])
```

```
    for i in range(0,clksVarphi.__len__()):

        clksVarphi[i] = (clksVarphi[i][0],
clksVarphi[i][1]+event[1])

    ## Move in the automata psi, varphi and the
product.##
    movePsiRes =
moveAut(Psi_Automaton,Parsed_psi_DbmPyUppa
al, psiCurrentState, event[0], clksPsi)
    psiCurrentState = movePsiRes[0]
    moveVarphiRes =
moveAut(Varphi_Automaton,
Parsed_varphi_DbmPyUppaal, varphiCurrentState,
event[0], clksVarphi)
    varphiCurrentState = moveVarphiRes[0]
    moveProdB =
moveAut(autB_DBMpy,Parsed_product_DbmPyUp
paal, productCurrentState, event[0],
clksPsi+clksVarphi)
    productCurrentState = moveProdB[0]

    # Consider resets in the transition that is
taken in the automata and reset clocks. ###
    for (clock,value) in movePsiRes[1]:
        for clk in clksPsi:
            #print clksPsi.index(clk)
            if clk[0]== str(clock):
                clksPsi[clksPsi.index(clk)] = (clk[0],0)

    for (clock,value) in moveVarphiRes[1]:
        for clk in clksVarphi:
            if clk[0]== str(clock):
                clksVarphi[clksVarphi.index(clk)] =
(clk[0],0)
```

Calling RV monitor with arguments

```
#RVmonitor('InputProp.xml',
'PropertyVerify4.xml',
[('p',89),('q',212),('p',160),('q',650)])
```

Output



```
File  Edit  View  Search  Terminal  Help
srinivas@srinivas:~$ cd RV-TimedProperties-masters/
srinivas@srinivas:~/RV-TimedProperties-masters$ cd V1
srinivas@srinivas:~/RV-TimedProperties-masters/V1$ cd Source
srinivas@srinivas:~/RV-TimedProperties-masters/V1/Source$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:38)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import RVMonitor
>>> RVMonitor.RVmonitor("InputTest.xml","PropertyVerify4.xml",[('p',89),('q',212
),('p',160),('q',650)])
Number of entries..5
size in bytes..64
given input sequence is..[('p', 89), ('q', 212), ('p', 160), ('q', 650)]
Verdict is...CURRENTLY_TRUE.
-----------------------------------------------------------------
input event is: ('p', 89)
Verdict is...CURRENTLY_TRUE.
-----------------------------------------------------------------
input event is: ('q', 212)
Verdict is...CURRENTLY_TRUE.
-----------------------------------------------------------------
input event is: ('p', 160)
Verdict is...CURRENTLY_TRUE.
-----------------------------------------------------------------
input event is: ('q', 650)
Conclusive verdict FALSE (Alert!!)
Computing minimal time...
min time to reach an acc state in psi is..0.0
>>> 
```

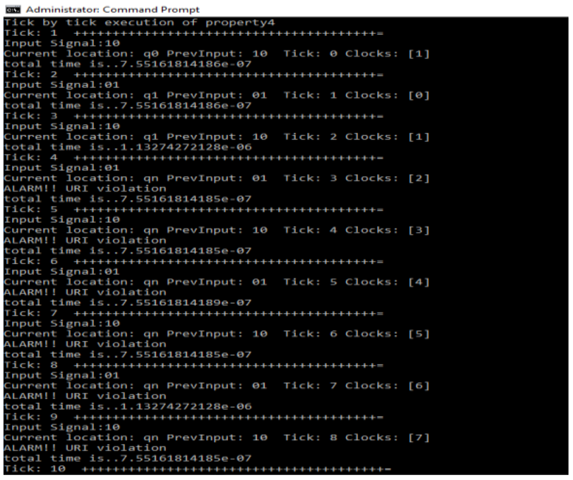
Fig. 16: Simulation output for the given trace

APPENDIX C
Python implementation of Runtime Verification Algorithm in Discrete TA

```
import sys
import random
sys.path.append('../')
import Automata
import SyncRE
import SyncEnforcer
import EnvSimulator
import enforcer_runner
import time
import matlab.engine
eng = matlab.engine.start_matlab()

eng.ecgpro(nargout=0)
p = eng.workspace['p']
qrs = eng.workspace['qrs']


def format_bin(value, size):
    if value < 0:
        value = 2**size + value
    return bin(value)[2:].rjust(size, '0')


def generate_actions(bits):
    actions = []
    output_index = bits/2

    for i in range(2**bits):
        i_formatted = format_bin(i, bits)
        action = (i_formatted[:output_index],
i_formatted[output_index:])
        actions.append(action)

    return actions



def delta_uri(q, a, v, do_clks = True):
    '''
    delta_ta is a function that is used as a transition
function of our discrete timed automata
    Inputs: q - location you are in,
            a - action you are checking,
            v - set of integer clock variables
            do_clks - boolean to indicate whether to
advance the DTA or peek at the transition
            e = empty, used to indicate if something is
empty
    Outputs: qj - returns all next locations that
match the criteria
```

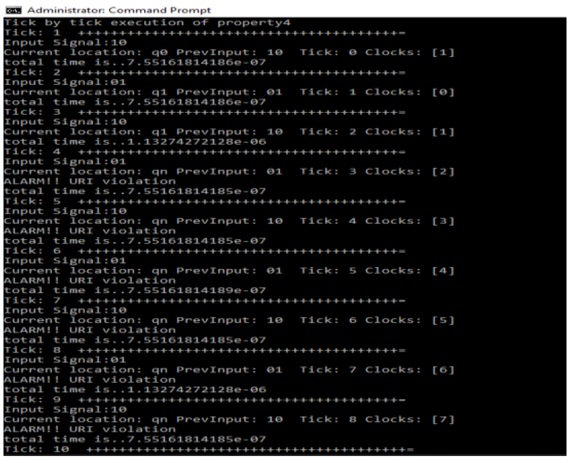
```
    '''

    qj = ''
    return_clocks = v
    reset_clocks = [0] * len(v)

    if q == 'q0':
        #Atrial and Ventricular event cannot happen
simultaneously
        if (((a[0])[1] == '1') and ((a[0])[0] == '1')):
            qj = 'qn'

        # Ventricular event
        elif ((((a[0])[1] == '1'))):
            qj = 'q1'
            # reset condition on transition
            if do_clks:
                reset_clocks[0] = 1

        else:
            qj = 'q0'

    elif q == 'q1':
        #Atrial and Ventricular event cannot happen
simultaneously
        if (((a[0])[1] == '1') and ((a[0])[0] == '1')):
            qj = 'qn'

        # Ventricular event
        elif (((a[0])[1] == '1')):

            # if threshold elapsed
            if (v[0] > 2):
                qj = 'q0'

            # if under threshold
            else:
                qj = 'qn'

        else:
            qj = 'q1'

    elif q == 'qn':
        qj = 'qn'


    else:
        print('DEBUG URI: incorrect format for q = {}, a
= {}, v = {}, do_clks = {},'.format(q, a, v, do_clks))

    if do_clks:
        for i in range(len(v)):
            if (reset_clocks[i] == 1):
                return_clocks[i] = 0

            else:
                return_clocks[i] += 1
    if do_clks:
        return qj, return_clocks

    else:
        return qj


def generate_uri():
    new_DTA = Automata.DTA(
    #actions#
    generate_actions(bits=4),
    #locations#
    ['q0', 'qv', 'qn'],
    #initial location#
    'q0',
    #accepting locations#
    lambda q: q in ['q0'],
    #transitions#
    delta_uri,
    #integer clocks#
    [0,],
    #non-accepting location#
    'qn'
    )
    return new_DTA



def main():

    i = 0
    continue_running = True

    input_trace = ''
    output_trace = ''

    print "Tick by tick execution of property4"

    '''
    Example trace
    01 (state q0 --> state q1)
    00 (state q1)
    01 (state qn, violation)

    '''

    possible_inputs = ["00", "10", "01", "11"]

    if p[0] < qrs[0]:
        x=1
    else:
        x=0
    phi = generate_uri()
    phi.q = phi.q0
    MAX_TICKS =  len(p)+ len(qrs)

    while i < MAX_TICKS:
        print 'Tick: ' + str(i+1) + '
++++++++++++++++++++++++++++++++++++++++='

        if x ==1:
            input_index=1
            x=0;
        else:
            x=1
            input_index=2

        output_trace = "00"

        #input_index = random.randint(0, 3)

        (phi.q, phi.V) =  phi.d(phi.q,
(possible_inputs[input_index], output_trace),
phi.V, True)


        print("Input
Signal:{}".format(possible_inputs[input_index]))


        print("Current location: {} PrevInput: {}  Tick: {}
Clocks: {}".format(phi.q,
possible_inputs[input_index],   i, phi.V))

        if phi.q == 'qn':
            print("ALARM!! URI violation")

        i += 1
        t1 = time.clock()
        t2 = time.clock()
        print "total time is.." + str(t2-t1)



if __name__ == '__main__':
    main()
```

Note that we have named the property as URI here (as earlier, while writing the code it was named that way), it is actually Property4. Also, q here denotes the locations in DTA and not Q-waves. The peaks of the Q-waves are denoted here as "qrs" to avoid confusion between the two.



```
Administrator: Command Prompt
Tick by tick execution of property4
Tick: 1  ++++++++++++++++++++++++++++++++++++++++=
Input Signal:10
Current location: q0 PrevInput: 10  Tick: 0 Clocks: [1]
total time is..7.55161814186e-07
Tick: 2  ++++++++++++++++++++++++++++++++++++++++=
Input Signal:01
Current location: q1 PrevInput: 01  Tick: 1 Clocks: [0]
total time is..7.55161814186e-07
Tick: 3  ++++++++++++++++++++++++++++++++++++++++=
Input Signal:10
Current location: q1 PrevInput: 10  Tick: 2 Clocks: [1]
total time is..1.13274272128e-06
Tick: 4  ++++++++++++++++++++++++++++++++++++++++=
Input Signal:01
Current location: qn PrevInput: 01  Tick: 3 Clocks: [2]
ALARM!! URI violation
total time is..7.55161814185e-07
Tick: 5  ++++++++++++++++++++++++++++++++++++++++=
Input Signal:10
Current location: qn PrevInput: 10  Tick: 4 Clocks: [3]
ALARM!! URI violation
total time is..7.55161814185e-07
Tick: 6  ++++++++++++++++++++++++++++++++++++++++=
Input Signal:01
Current location: qn PrevInput: 01  Tick: 5 Clocks: [4]
ALARM!! URI violation
total time is..7.55161814189e-07
Tick: 7  ++++++++++++++++++++++++++++++++++++++++=
Input Signal:10
Current location: qn PrevInput: 10  Tick: 6 Clocks: [5]
ALARM!! URI violation
total time is..7.55161814185e-07
Tick: 8  ++++++++++++++++++++++++++++++++++++++++=
Input Signal:01
Current location: qn PrevInput: 01  Tick: 7 Clocks: [6]
ALARM!! URI violation
total time is..1.13274272128e-06
Tick: 9  ++++++++++++++++++++++++++++++++++++++++=
Input Signal:10
Current location: qn PrevInput: 10  Tick: 8 Clocks: [7]
ALARM!! URI violation
total time is..7.55161814185e-07
Tick: 10  ++++++++++++++++++++++++++++++++++++++++=
```

Fig. 17: Tick by Tick execution of the property and checking for violation



```
Administrator: Command Prompt
Tick by tick execution of property4
Tick: 1  ++++++++++++++++++++++++++++++++++++++++=
Input Signal:10
Current location: q0 PrevInput: 10  Tick: 0 Clocks: [1]
total time is..7.55161814186e-07
Tick: 2  ++++++++++++++++++++++++++++++++++++++++=
Input Signal:01
Current location: q1 PrevInput: 01  Tick: 1 Clocks: [0]
total time is..7.55161814186e-07
Tick: 3  ++++++++++++++++++++++++++++++++++++++++=
Input Signal:10
Current location: q1 PrevInput: 10  Tick: 2 Clocks: [1]
total time is..1.13274272128e-06
Tick: 4  ++++++++++++++++++++++++++++++++++++++++=
Input Signal:01
Current location: qn PrevInput: 01  Tick: 3 Clocks: [2]
ALARM!! URI violation
total time is..7.55161814185e-07
Tick: 5  ++++++++++++++++++++++++++++++++++++++++=
Input Signal:10
Current location: qn PrevInput: 10  Tick: 4 Clocks: [3]
ALARM!! URI violation
total time is..7.55161814185e-07
Tick: 6  ++++++++++++++++++++++++++++++++++++++++=
Input Signal:01
Current location: qn PrevInput: 01  Tick: 5 Clocks: [4]
ALARM!! URI violation
total time is..7.55161814189e-07
Tick: 7  ++++++++++++++++++++++++++++++++++++++++=
Input Signal:10
Current location: qn PrevInput: 10  Tick: 6 Clocks: [5]
ALARM!! URI violation
total time is..7.55161814185e-07
Tick: 8  ++++++++++++++++++++++++++++++++++++++++=
Input Signal:01
Current location: qn PrevInput: 01  Tick: 7 Clocks: [6]
ALARM!! URI violation
total time is..1.13274272128e-06
Tick: 9  ++++++++++++++++++++++++++++++++++++++++=
Input Signal:10
Current location: qn PrevInput: 10  Tick: 8 Clocks: [7]
ALARM!! URI violation
total time is..7.55161814185e-07
Tick: 10  ++++++++++++++++++++++++++++++++++++++++=
```

Fig. 18: Tick by Tick execution

The comparison in the result section is done taking the same simulation trace in both the approaches. Here, for demonstration purpose the trace in the TA part is minimised.