CS 498: Shield Synthesis of Cyber Physical Systems

Runtime Enforcement of Cyber Physical Systems

Presented by:
Samay Varshney (180101097)
Siddhartha Jain (180101078)

Introduction

Here, Heart-pacemaker combination system is taken as a typical example of Cyber Physical system.

Heart model: Run-time parameterisation is used here to exhibit different types of arrhythmia.

Pacemaker model: Timed Automata (TA) is used here for pacemaker model.

Pacemakers are life-saving devices. Ironically, devices designed to save lives have caused serious harm including death of many patients. This called for alternating techniques inspired by formal methods.

Need For Runtime Enforcement (RE)

In Cyber Physical Systems, if any unsafe state is reachable, then the system is stopped. Hence for the safety critical systems, which are reactive, stopping the system may not be feasible and it introduces the need for Run-time Enforcement (RE).

Enforcer: It monitors the inputs from plant and outputs of the controller and edits them when necessary and hence performs enforcement of certain properties.

Closed-loop Run-time Enforcement (RE) of the heart-pacemaker system, interact through an “enforcer” and provides an additional layer of safety by enforcing a set of critical properties. These properties are only enforced when pacemaker fails to guarantee them.

Previous Approaches

There is already a work done related to frameworks to synthesize enforcers for reactive systems, called shields, from a set of safety properties.

Shields are unidirectional, where it observes inputs from the plant and outputs from the controller, and transforms erroneous outputs. but we require bidirectional enforcer, which can edits both inputs and outputs when controller (here pacemaker) fails.

In shields, untimed properties are considered where properties are expressed as automata but we require enforcer to enforce the timed properties, which allow to express how time elapse between events.

Overview of Proposed Approach

The system (synchronous program) is considered to be a black-box, and main focus was on the synthesis of enforcers from properties that can be defined by DTA.

Properties are expressed as discrete time properties using a variant of Discrete Timed Automata (DTA) that is called Synchronous DTA (SDTA).

Bi-directional runtime enforcement of heart-pacemaker cyber physical system.

Boolean Inputs (I) = { AS, VS }
AS = Atrial Sense VS = Ventricular Sense

Boolean Output (O) = { AP, VP }
AP = Atrial Pace VP = Ventricular Pace

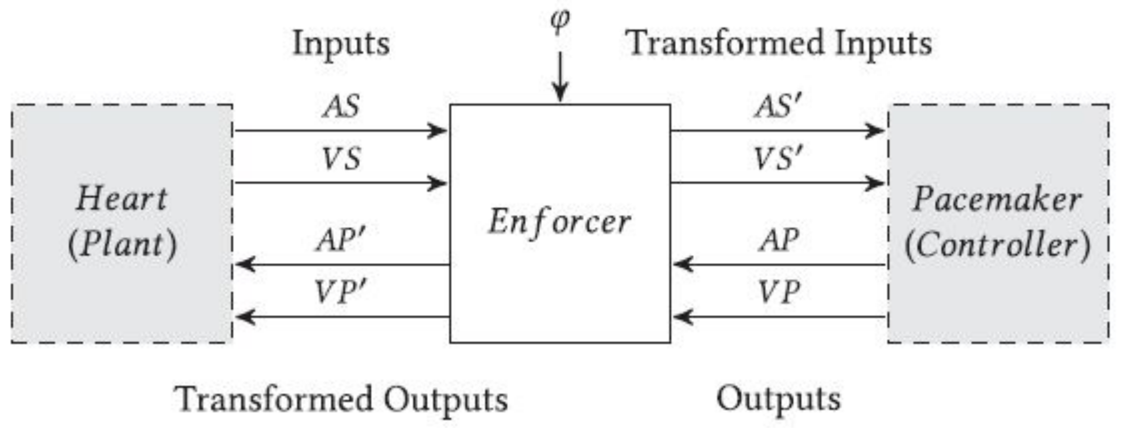


Fig. 1. Enforcer between heart and pacemaker.

Overview of Proposed Approach

At the top of the diagram EGMs for both an atrium and ventricle are shown, while the bottom of the diagram shows the status of various timers during the pacemaker operation.

AVI timer maintains the correct time delay between any atrial event and a subsequent ventricular event.

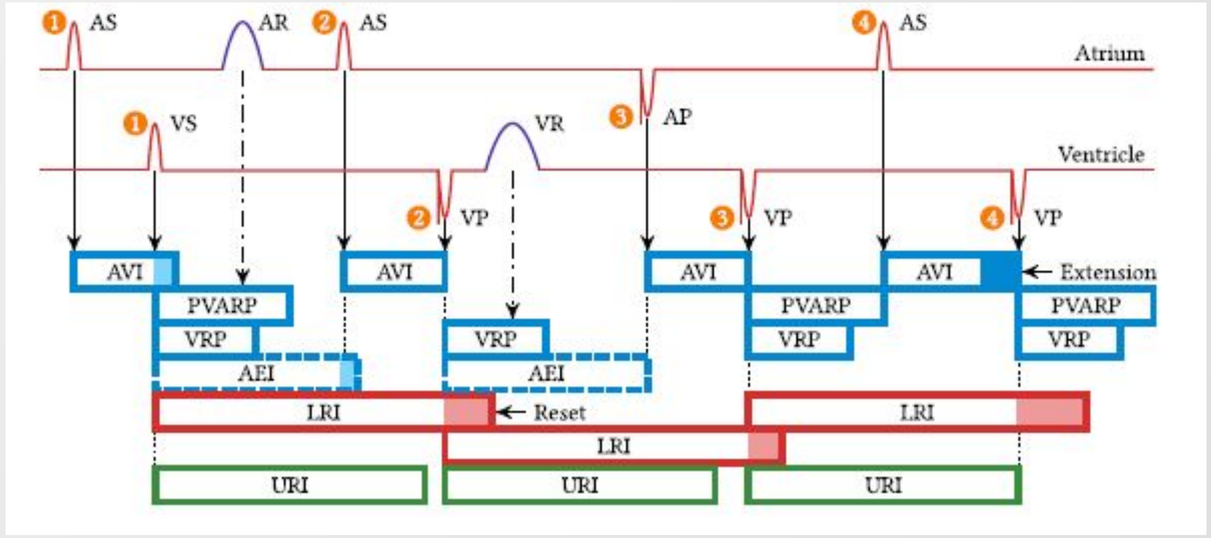
The timing intervals AVI, AEI and LRI are real values denoted as some constant C.

Their discrete counterparts are denoted as AVI_{TICKS}, AEI_{TICKS} etc are denoted as C_{TICKS}.

AS or VS: Sensed natural event
AR or VR: Ignored natural event
AP or VP: Artificial pacing from pacemaker

$C_{TICKS} = \lceil C/WCRT \rceil$ or $C_{TICKS} = \lfloor C/WCRT \rfloor$

WCRT denotes Worst Case Reaction Time



Overview of Proposed Approach

Some of the properties which can be enforced by the enforcer:

P1: AP and VP cannot happen simultaneously.
P2: VS or VP must be true within AVI_{TICKS} after an atrial event AS or AP.
P3: AS or AP must be true within AEI_{TICKS} after an ventricle event VS or VP.
P4: After a ventricle event, another ventricle event can happen only after URI_{TICKS} .
P5: After a ventricle event, another ventricle event should happen within $LRI_{TICKS.}$.

Here properties P1...P5 are discrete time properties, and their dense variants with real-time constraints are denoted as $P1'$ . . . $P5'$. Dense time properties can be expressed as Timed Automata (TA).

By using DTA, all transitions take one tick relative to the ticks of a synchronous global clock inspired by synchronous languages and hence the time will be discretized.

By using products of DTAs, multiple properties can be enforced in a system.

Preliminaries and Notations

Boolean Inputs: I
Boolean Outputs: O
Input alphabet: $\sum_I = 2^I$
Output alphabet: $\sum_O = 2^O$
Input Output alphabet: $\sum = \sum_I \times \sum_O$
Each input (resp. output) will be denoted as a bit-vector.
A reaction (or input output event) is of the form (x_i, y_i), where $x_i \in \sum_I$ and $y_i \in \sum_O$

A finite (resp. infinite) word over a finite alphabet $\sum$ is a finite sequence $\sigma = a_1.a_2...a_n$ (resp. infinite sequence $\sigma = a_1.a_2\ldots$) of elements of $\sum$.
set of finite (infinite) words over $\sum$ is denoted by $\sum^*$ ($\sum^w$).
$\sum^+$ denotes $\sum^* \setminus \{\in\}$ where $\in$ is a empty word.

Length of a finite word σ is denoted by |σ|.
The concatenation of two words σ and σ' is denoted by σ.σ'.
A word σ' is a prefix of a word σ denoted by $\sigma' \preccurlyeq \sigma$.

CPS as a Synchronous System

Controllers of CPSs operate synchronously in a reactive loop that executes once every tick. During a tick, all three components - plant, controller, and enforcer - are executed once.

An execution σ of a synchronous program P is an infinite sequence of input-output events $\sigma \in \Sigma^w$
The behaviour of a synchronous program P is denoted by $exec(P) \subseteq \Sigma^*$

The language of P is denoted by $L(P) = \{\sigma \in \Sigma^* \mid \exists \sigma' \in exec(P) \wedge \sigma \preccurlyeq \sigma'\}$, L(P) is the set of all finite prefixes of the sequence in exec(P)

Properties

Properties that we want to enforce are formally defined using Discrete timed Automata (DTA) that are automata extended with a set of integer variables that are used as discrete clocks for instance to count the number of ticks before a certain event occurs.

A property φ over Σ defines a set $L(\varphi) \subseteq \Sigma^*$. A program $P \models \varphi$ iff $L(P) \subseteq L(\varphi)$.

Definition: A Discrete Timed Automaton is a tuple $A = (L, l_0, l_v, \Sigma, V, \Delta, F)$ where L is the set of locations, $l_0 \in L$ is the initial location, Σ is the alphabet, V is a set of integer clocks, $F \subseteq L$ is the set of accepting locations, and l_v is a unique non-accepting trap location.

The transition relation Δ is $\Delta \subseteq L \times G(V) \times R \times \Sigma \times L$ where G(V) denotes set of guards defined using conjunctions of simple constraints of the form $v \square c$ with $v \in V$, $c \in N$ and $\square \in \{<, \leq, =, \geq, >\}$, and $R \subseteq V$ is a subset of integer clocks that are reset to 0.

Example: Property Defined as a DTA

Let $V = \{v_1, . . . ,v_k\}$ be a finite set of integer clocks. A valuation for v is an element of N, that is a function from v to N.

$\chi \in N^V$ denotes set of valuations for V. χ assigns $\chi(v)$ to each clock variable v of V.

Definition: The semantics of a DTA is a transition system $[[A]] = (Q, q_0, \Sigma, \rightarrow, Q_F, q_v)$ where $Q = L \times N^V$ is the set of states, $q_0 = (l_0, \chi_0)$ is the initial state where χ_0 is the valuation that maps every integer clock variable in V to 0, $Q_F = F \times N^V$ is the set of accepting states, and $q_V = l_V \times N^V$ is the set of trap states. The transition relation $\rightarrow \subseteq Q \times \Sigma \times Q$ is a set of transitions of the form $(l, \chi) \rightarrow (l', \chi')$ with $\chi' = (\chi + 1)[r \leftarrow 0]$ whenever there exists $(l, g, r, a, l') \in \Delta$ such that $\chi \models g$.

For $\chi \in N^V$, $\chi + 1$ is the valuation assigning $\chi(v) + 1$ to each clock variable v of V . Given a set of clock variables $V' \subseteq V$, $\chi[V' \leftarrow 0]$ is the valuation of clock variables χ where all the clock variables in V' are assigned to 0.

Run of a DTA

A run ρ of A from a state $q_1 \in Q$ over a untimed trace $\sigma = a_1 \cdot a_2 \cdot \cdot \cdot a_n \in \Sigma^*$ is a sequence of
moves in [[A]] denoted as $\rho = q_1 \rightarrow q_2 \rightarrow \ldots\ldots \rightarrow q_n$.

Run(A): The set of runs from the initial state $q_0 \in Q$.
Run_{QF}(A): Subset of those runs accepted by A i.e. ending in an accepting state $q_n \in Q_F$.

L(A): The set of untimed traces of runs in Run_{QF}(A).

The product of DTAs is useful when we want to enforce multiple properties. Given two deterministic and complete DTAs A^1 and A^2 the DTA A obtained by computing their product recognizes the language $L(A^1) \cap L(A^2)$, and is also deterministic and complete.

Example: Property Defined as a DTA

S_1: A and B cannot happen simultaneously, A and B alternate starting with an A. B should be true with in 5 ticks after A occurs.

$I = \{A\}$
$O = \{B\}$
input-output alphabet $\Sigma = 2^I \times 2^O$
Set of clock variables $V = \{v_1\}$
Initial location: l_0
Trap location: l_2
Accepting locations: $\{l_0\}$

Initial state: $(l_0, v_1 = 0)$
Input trace $\sigma = (0, 0) \cdot (1, 0) \cdot (0, 0) \cdot (0, 0) \cdot (0, 1)$.

σ is accepted by DTA since the state reached upon run on σ is $(l_0, v_1 = 3)$ which is an accepting state.

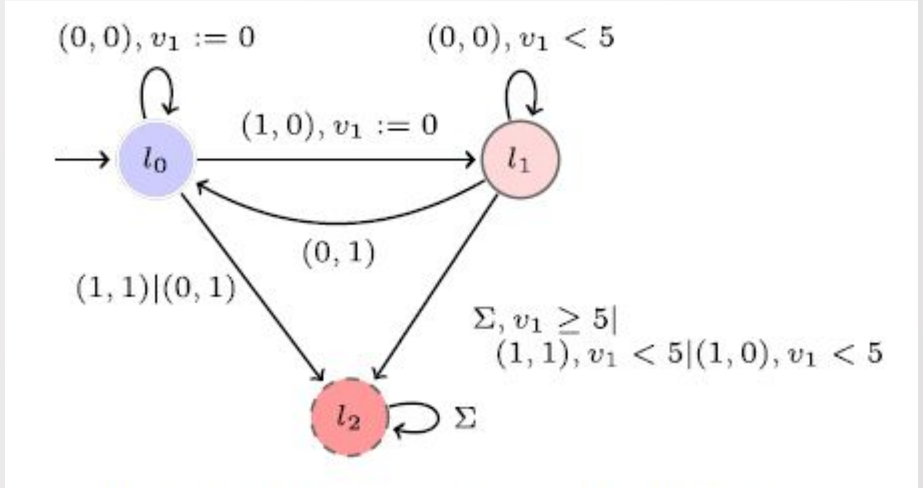


Fig. 3. Property S_1 defined as DTA $\mathcal{A}_{S_1}$.

Input DTA $A\varphi_I$

Input-Output word $\sigma = (x_1, y_1) \cdot (x_2, y_2) \cdot \cdot \cdot (x_n, y_n) \in \Sigma^*$

Input word $\sigma_I = x_1 \cdot x_2 \cdot \cdot \cdot x_n \in \Sigma^*_I$ is the projection on inputs ignoring outputs.

Output word $\sigma_O = y_1 \cdot y_2 \cdot \cdot \cdot y_n \in \Sigma^*_O$ is the projection on outputs.

Input DTA $A\varphi_I$: Given property φ defined as DTA $A\varphi = (L, l_O, l_V, \Sigma, V, \Delta, F)$, input DTA $A\varphi_I = (L, l_O, l_V, \Sigma_I, V, \Delta_I, F)$ is obtained from Aφ by ignoring outputs on the transitions, i.e., for every transition $(l, g, r, (x, y), l') \in \Delta$ in Aφ, there is a transition $(l, g, r, x, l') \in \Delta_I$ in $A\varphi_I$.

$L(A\varphi_I)$ is denoted as $\varphi_I \subseteq \Sigma^*_I$.

The only difference between Aφ and AφI is we ignore outputs in the automaton $A\varphi_I$.

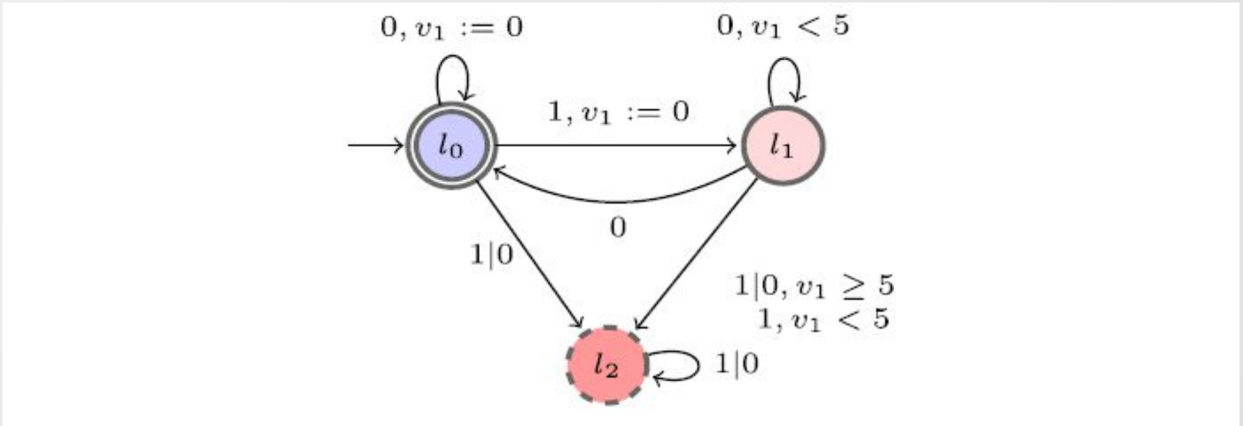


Fig. 5. Automaton obtained from $\mathcal{A}_{S_1}$ in Figure 3 by projecting on inputs.

Edit Functions

Edit Functions are those which enforcer uses to edit input (resp. output) events (whenever necessary), according to input property φ_I (resp.input-output property φ).

In each step, based on the input-output property φ that we want to enforce, the enforcer:

1. First processes the input from the environment, and transforms it using editIφ_I based on the input property φ_I.
2. Later, the output produced by the program is transformed by the enforcer (when necessary) using editOφ.

editI$\varphi_{,I}$(σ_I): Given $\sigma_I \in \Sigma^*_I$, editIφ_I(σ_I) is the set of input events x in Σ_I such that the word obtained by extending σ_I with x can be extended to a sequence that satisfies φ_I.

editIAφ_I(q_I): Let $q_I \in Q_I$ correspond to a state reachable in Aφ_I, editIAφ_I(q_I) is the set of input events x in Σ_I such that the state reached by extending q_I with x can be extended to an accepted state in Q_F.

Edit Functions (Cont.)

editOφ(σ, x): Given an input-output word $\sigma \in \Sigma^*$ and an input event $x \in \Sigma_I$, editOφ(σ, x) is the set of output events y in Σ_O such that the input-output word obtained by extending σ with (x,y) can be extended to a sequence that satisfies the property φ.

rand-editIφ$_I$(σ$_I$): If editIφ$_I$(σ$_I$) is non-empty, then rand-editIφ$_I$(σ$_I$) returns an element (chosen randomly) from editIφ$_I$(σ$_I$), and is undefined if editIφ$_I$(σ$_I$) is empty.

rand-editOφ(σ, x): If editOφ(σ, x) is non-empty, then rand-editOφ(σ, x) returns an element (chosen randomly) from editOφ(σ, x), and is undefined if editOφ(σ, x) is empty.

minD-editIφ$_I$(σ$_I$, x): If editIφ$_I$(σ$_I$) is non-empty, then minD-editIφ$_I$(σ$_I$, x) returns an event from editIφ$_I$(σ$_I$) with minimal distance w.r.t x and is undefined if editIφ$_I$(σ$_I$) is empty.

minD-editOφ(σ, x, y): If editOφ(σ, x) is non-empty, then minD-editOφ(σ, x, y) returns an event from editOφ (σ, x) with minimal distance w.r.t y and is undefined if editOφ(σ, x) is empty.

Every edit function has a alternative with respect to a state $q \in Q_F$ which return set of events such that the state obtained by extending the given state can lie in an accepted states Q_F.

Problem Definition

Digital Controller (here pacemaker) with boolean input and output streams.

Property φ defined as DTA.

Plant (here heart) sampled periodically that result in boolean streams (AS and VS).

Input event (x,y) is a tuple, where x is the input (values of all Boolean inputs), and y is the output (values of all Boolean outputs).

It is assumed that the controller (pacemaker) is a black-box and may be invoked through a special function call called ptick and it's internals are considered to be unknown.

Formally, ptick is a function (with internal state) from Σ_I to Σ_O that takes a bit vector $x \in \Sigma_I$ and returns a bit vector $y \in \Sigma_O$.

Problem Definition: Enforcer for φ

At an abstract level, an enforcer may be viewed as a function that transforms words.

Enforcer for φ : Given property $\varphi \subseteq \Sigma^*$, an enforcer for φ is a function $E\varphi : \Sigma^* \rightarrow \Sigma^*$ satisfying the following constraints:

1. Soundness (Snd): For any input word $\sigma \in \Sigma^*$, the output of the enforcer Eφ(σ) can be extended to a sequence that satisfies φ.

2. Montonicity (Mono): Enforcer cannot undo what is already released as output.

3. Instantaneity (Inst): Enforcer cannot delay, insert and suppress events. Whenever the enforcer receives a new input event, it must react instantaneously and produce an output event immediately.

4. Transparency (Tr): Enforcer will not unnecessarily edit any event. Any new input event (x,y) will be simply forwarded by the enforcer if, output computed earlier by the enforcer followed by (x,y) can be extended to a sequence that satisfies φ in the future.

5. Causality (Ca): For every input event (x,y), the enforcer produces output event (x',y') where it first processes the input part x, to produce the transformed input x' according to property φ using $\text{editI}\varphi_I$ then transforms output $y \in \Sigma_O$ to produce the transformed output y' using editOφ.

Example: Non-enforceable Property

Enforceability: Let $\varphi \subseteq \Sigma^*$ be a property. We say that φ is enforceable if and only if an enforcer $E\varphi$ for φ exists.

$I = \{A\}$, $O = \{B\}$, $\Sigma = \Sigma_I \times \Sigma_O$

Input-output sequence $\sigma = (1, 1) \cdot (1, 0)$.

When the enforcer reads the first event (1, 1), it can output (1, 1) (since every event in Σ from q_0 leads to a non-violating state q_1).

Now from q_1, every event in Σ only leads to the trap state q_V.

Thus, when the second event (1, 0) is read, every possible editing of this event will only lead to violation of the property.

Upon reading the second event (1, 0), releasing any event in Σ as output will violate soundness and if no event is released as output, then the instantaneity constraint will be violated.

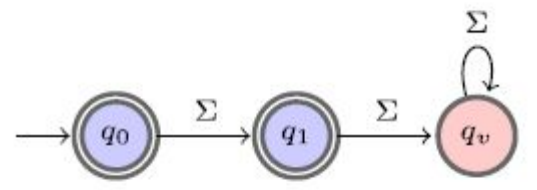


Fig. 6. A non-enforceable property.

Problem Definition: Condition for Enforceability

Condition for Enforceability: Consider a property φ that is defined as DTA $A\varphi = (L, l_O, l_V, \Sigma, V, \Delta, F)$ with semantics $[[A\varphi]] = (Q, q_O, \Sigma, \rightarrow, Q_F, q_V)$. Property φ is enforceable if and only if an accepting state is reachable from every non-violating state (i.e., from every state q not in q_V) in 1 or more steps.

Moreover, Q contain only states that are reachable from q_O.

Algorithm: Bidirectional Sync. Enforcement

Let automaton Aφ = (L, l_O, l_V , Σ, V, Δ, F) , semantics [[Aφ]] = (Q, q_O, Σ,→,Q_F , q_V) define enforceable property φ.
Let input automaton $Aφ_I$ = (L, l_O, l_V , $Σ_I$,V , $Δ_I$, F) with semantics [[$Aφ_I$]] = (Q, q_O, $Σ_I$, → , I ,Q_F ,q_V) is obtained from Aφ by projecting on inputs.

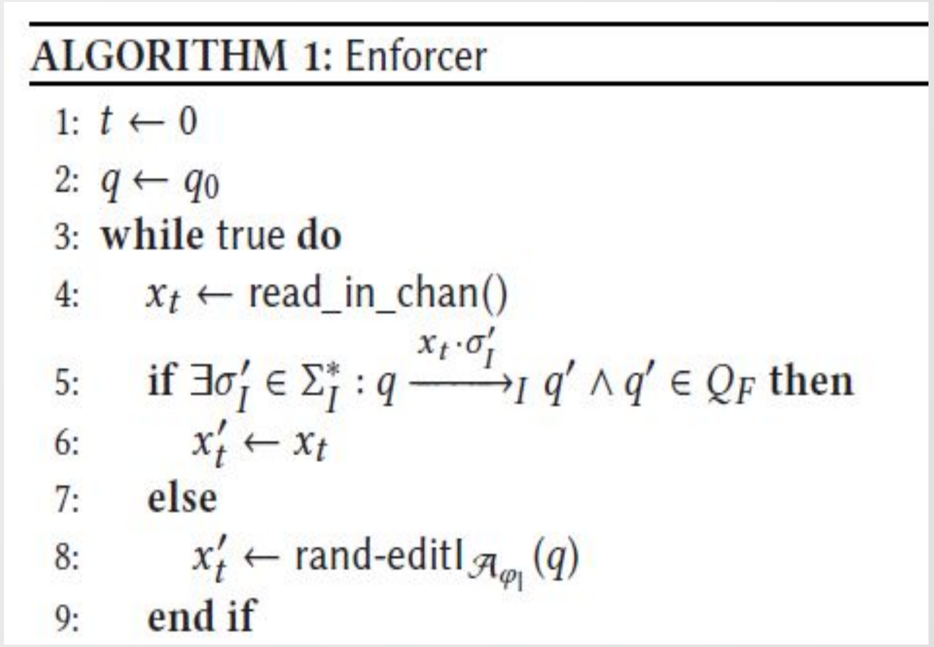
Both Aφ and $Aφ_I$ are treated as inputs.

t keeps track of the time-step (tick), initialized with 0.

q keeps track of the current state of both automata Aφ and $Aφ_I$.

Function **read_in_chan** is a function corresponding to reading input channels and input event is assigned to x_t.

The algorithm tests whether an accepting state is reachable from the current state q upon x_t extended with any sequence $σ_I ∈ Σ^*_I$. If yes, x_t' = x_t otherwise x_t' is assigned with the output of rand-editI$Aφ_I$(q).



ALGORITHM 1: Enforcer

```
$t \leftarrow 0$
$q \leftarrow q_0$
while true do
    $x_t \leftarrow$ read_in_chan()
    if $\exists \sigma'_I \in \Sigma^*_I : q \xrightarrow{x_t \cdot \sigma'_I}_I q' \wedge q' \in Q_F$ then
        $x'_t \leftarrow x_t$
    else
        $x'_t \leftarrow$ rand-editI$_{\mathcal{A}_{\varphi_I}}(q)$
    end if
```

Algorithm: Bidirectional Sync. Enforcement

Function **ptick** corresponds to invoking the synchronous controller execution which takes in input event x_t’ and gives out output event y_t.

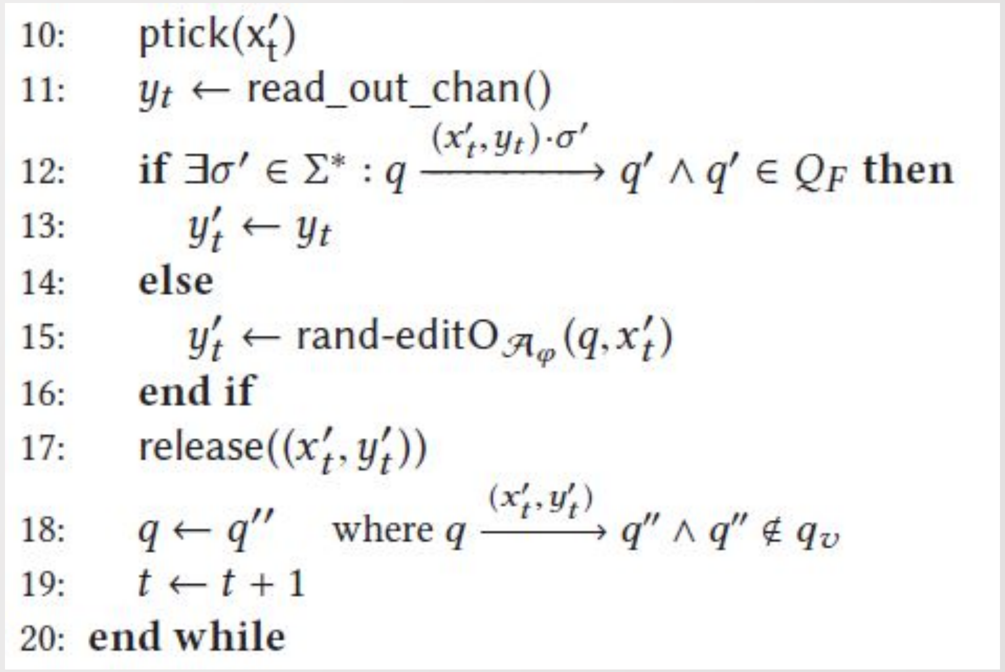
Function **read_out_chan** is a function corresponding to reading output channels and input event is assigned to y_t.

The algorithm tests whether an accepting state is reachable in [[Aφ]] from the current state q upon (x_t ,y_t) followed by any sequence σ ∈ Σ^*. If yes, y_t’ = y_t otherwise y_t is assigned with the output of rand-editOAφ(q, x_t).

Function **release** takes an input-output event, and releases it as output of the enforcer.

Current state q is updated to q which is the state reached upon (x_t ,y_t) from state q in [[Aφ]].

Time-step t is incremented.



```
    ptick($x'_t$)
    $y_t \leftarrow$ read_out_chan()
    if $\exists \sigma' \in \Sigma^* : q \xrightarrow{(x'_t, y_t)\cdot\sigma'} q' \wedge q' \in Q_F$ then
        $y'_t \leftarrow y_t$
    else
        $y'_t \leftarrow$ rand-editO$_{\mathcal{A}_\varphi}(q, x'_t)$
    end if
    release($(x'_t, y'_t)$)
    $q \leftarrow q''$   where $q \xrightarrow{(x'_t, y'_t)} q'' \wedge q'' \notin q_v$
    $t \leftarrow t + 1$
end while
```

Algorithm: Example Input - Output Behaviour

Consider property S_1 defined as DTA in Fig 3 and its corresponding input DTA is presented in Figure 5.

Table 1 shows input output behaviour of the algorithm.

t indicates tick
x (resp. y) indicates input (resp. output) read by the algorithm
x’ (resp. y’) indicates input (resp. output) released by the algorithm.
fwdI (resp fwdO) indicate that the input (resp. output) is simply forwarded
edtI (resp. edtO) indicate that the input (resp. output) is edited.

The input output word read by the enforcer is
(0, 1) · (1, 1) · (1, 0) · (1, 0) · (0, 1)
The input output word released as output by the enforcer
is (0, 0) · (1, 0) · (0, 0) · (0, 0) · (0, 1).

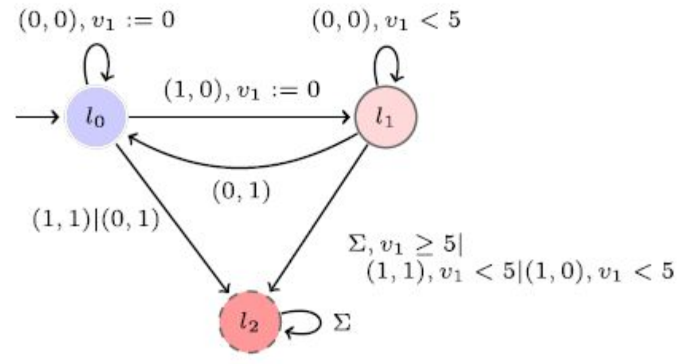


Fig. 3. Property S_1 defined as DTA $\mathcal{A}_{S_1}$.

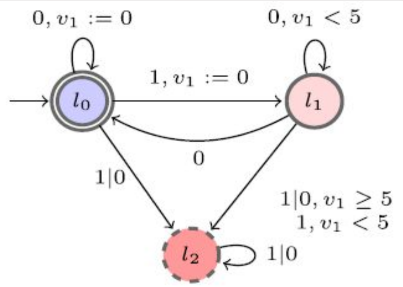
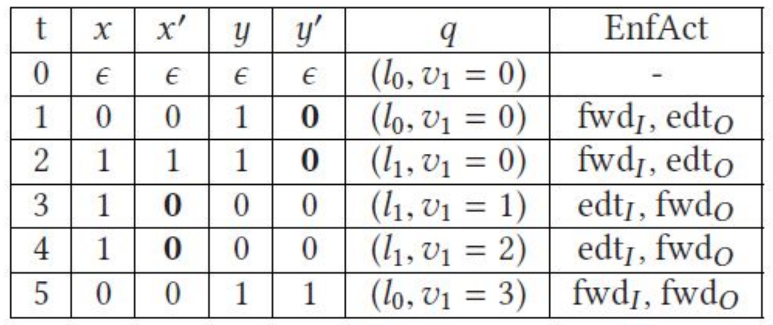


Fig. 5. Automaton obtained from $\mathcal{A}_{S_1}$ in Figure 3 by projecting on inputs.

Table 1. Example Illustrating Algorithm 1



t	x	x'	y	y'	q	EnfAct
0	ϵ	ϵ	ϵ	ϵ	$(l_0, v_1 = 0)$	-
1	0	0	1	**0**	$(l_0, v_1 = 0)$	$\text{fwd}_I, \text{edt}_O$
2	1	1	1	**0**	$(l_1, v_1 = 0)$	$\text{fwd}_I, \text{edt}_O$
3	1	**0**	0	0	$(l_1, v_1 = 1)$	$\text{edt}_I, \text{fwd}_O$
4	1	**0**	0	0	$(l_1, v_1 = 2)$	$\text{edt}_I, \text{fwd}_O$
5	0	0	1	1	$(l_0, v_1 = 3)$	$\text{fwd}_I, \text{fwd}_O$

Comparison with the enforcement of TA

Worst Case Reaction Time (WCRT) is the worst case value of the tick length.

Given any DTA φ, we create a TA φ' as follows:

- Every integer clock $v \in V$ is replaced by a real-valued clock v, $\in V'$, where V is the set of integer clocks and V' is the set of dense time clocks.

- Every transition guard of the form $v <> C_{TICKS}$ where C_{TICKS} is a discrete clock value is replaced by $v' <> C$, where C is the actual real-value. For example, $v1 \leq AVI_{TICKS}$ is replaced by $v1 \leq AVI$. Here, AVI_{TICKS} is an integer approximation of AVI , which can be obtained by $\lceil AVI/WCRT \rceil$ or $\lfloor AVI/WCRT \rfloor$.

Error Bound: The maximum timing error while enforcing any DTA φ relative to the TA φ' is always less than WCRT i.e. $\forall\ v \in V, v' \in V\ \ |(v \times WCRT) - v'| < WCRT$.

Soundness of tick counting

WCRT analysis revealed a value of 583 μs for the non-enforcement case, a value which changed mostly negligibly (less than ~0.5%) with the addition of enforcers. WCRT analysis uses a Monte Carlo simulation approach. So we use WCRT value of 1ms for the system.

Pacemaker specification allows a tolerance value for each timing interval such as AVI, Atrial Escape Interval (AEI) etc. to be ± 5 ms. As the error is bounded in the interval [0, 1) ms, the developed approach is sound.

Optimal Enforcement Function

Optimal Enforcer for φ : Given property $\varphi \subseteq \Sigma^*$, an optimal enforcer for φ is a function $E\varphi\text{-opt} : \Sigma^* \rightarrow \Sigma^*$ satisfying the all the constraints defined previously of Soundness, Montonicity, Instantaneity, Transparency, Causality.

The only difference between Eφ-opt and Eφ is that in Eφ-opt, minD-editIAφ_I is used instead of rand-editIAφ_I, which is used in Eφ i.e., x' = rand-editIAφ_I(q) is replaced with x' = minD-editIAφ_I(q, x).
Thus, instead of picking any random element from editIAφ_I(q), an element in editIφ_I(q) which differs minimally w.r.t the actual input x is selected using minD-editIAφ_I(q, x).

Similarly, minD-editOAφ is used instead of rand-editOAφ i.e., y' = rand-editOAφ (q, x_t') is replaced with y' = minD-editOAφ (q, x_t',y).

Implementation and Results

Algorithm was implemented in the open-source SCCharts framework, a Statechart dialect designed for safety-critical systems.
In order to evaluate the effect of enforcement on the execution time,the enforcers were synthesised for all properties P_1 through P_5 and for $P_2 \land P_3$.
The complete system, including the heart model, pacemaker, and enforcer, was executed on an ARM Cortex-A9 on an Altera DE1-SoC operating at 800 MHz.
Each of these systems were run for 100,000 ticks of execution, corresponding to 100 seconds of simulated time using a 1 ms step size.

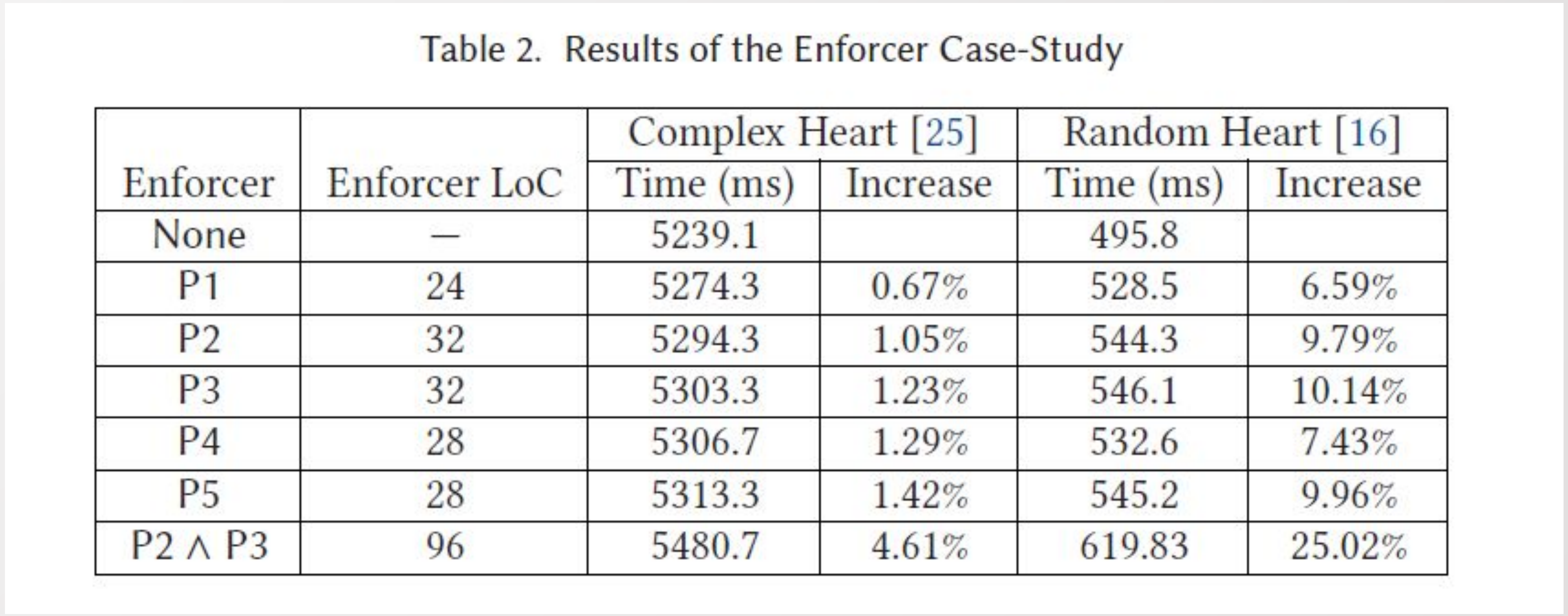


Table 2. Results of the Enforcer Case-Study

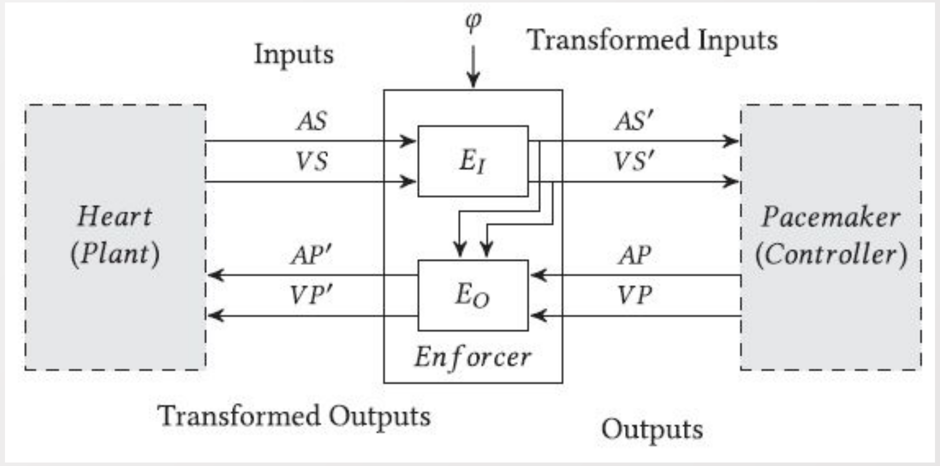
Enforcer	Enforcer LoC	Complex Heart [25]		Random Heart [16]	
		Time (ms)	Increase	Time (ms)	Increase
None	–	5239.1		495.8	
P1	24	5274.3	0.67%	528.5	6.59%
P2	32	5294.3	1.05%	544.3	9.79%
P3	32	5303.3	1.23%	546.1	10.14%
P4	28	5306.7	1.29%	532.6	7.43%
P5	28	5313.3	1.42%	545.2	9.96%
$P2 \land P3$	96	5480.7	4.61%	619.83	25.02%

Implementation and Results

Two experiments were performed, first with the complex heart model and second with a much simpler model named the Random Heart Model.

A total of 10 separate trials were performed for each experiment, with their average execution time taken as the mean of all trials.

In Random Heart Model, we can see faster execution speeds but similar absolute overheads associated with enforcement, and hence higher percentage overheads.



Related Work

Several RE models have been proposed such as:

- Security automata: In this enforcer blocks the execution when it recognizes a sequence of actions that does not satisfy the desired property.
- Edit automata: In this enforcer correct events by suppressing and/or inserting events.
- RE mechanisms which allow buffering events and releasing them upon observing a sequence that satisfies the desired property.
- Synthesis of enforcers for real-time properties expressed as dense TA has also been studied.

All these approaches focus on uni-directional RE (they observe inputs from the plant and outputs from the controller and corrects outputs). Also these approaches didn’t work for reactive systems.

Conclusion and Future Work

The runtime enforcement problem for cyber physical systems (CPS) as a bidirectional runtime enforcement of reactive systems is proposed for the first time. Moreover, the use of discrete timed automata (DTA) in place of timed automata is used to express properties to be enforced.

Error due to the proposed discretisation is bounded and well within the tolerance value of the pacemaker is shown using an on-line algorithm implemented it in the SCCharts tool-chain. Experimental results show that the framework incurs minimal runtime overhead.

One possible avenue for future work is related to an empirical comparison of dense vs discrete run-time enforcement.

Questions?