# Runtime Enforcement of Cyber-Physical Systems*

SRINIVAS PINISETTY, Aalto University, Finland and University of Gothenburg, Sweden
PARTHA S ROOP, University of Auckland, New Zealand
STEVEN SMYTH, Kiel University, Germany
NATHAN ALLEN, University of Auckland, New Zealand
STAVROS TRIPAKIS, Aalto University, Finland and UC Berkeley, USA
REINHARD VON HANXLEDEN, Kiel University, Germany

Many implantable medical devices, such as pacemakers, have been recalled due to failure of their embedded software. This motivates rethinking their design and certification processes. We propose, for the first time, an additional layer of safety by formalising the problem of run-time enforcement of implantable pacemakers. While recent work has formalised run-time enforcement of reactive systems, the proposed framework generalises existing work along the following directions: (1) we develop bi-directional enforcement, where the enforced policies depend not only on the status of the pacemaker (the controller) but also of the heart (the plant), thus formalising the run-time enforcement problem for cyber-physical systems (2) we express policies using a variant of discrete timed automata (DTA), which can cover all regular properties unlike earlier frameworks limited to safety properties, (3) we are able to ensure the timing safety of implantable devices through the proposed enforcement, and (4) we show that the DTA-based approach is efficient relative to its dense time variant while ensuring that the discretisation error is relatively small and bounded. The developed approach is validated through a prototype system implemented using the open source KIELER framework. The experiments show that the framework incurs minimal runtime overhead.

CCS Concepts: • **General and reference** → **Verification**; • **Theory of computation** → **Program verification**; • **Software and its engineering** → **Software verification**; **Formal software verification**;

Additional Key Words and Phrases: Runtime Monitoring, Runtime Enforcement, Automata, Timed Properties, Cyber-Physical Systems, Synchronous Programming, SCCharts

**ACM Reference format:**
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

*"This article was presented in the International Conference on Embedded Software (EMSOFT) 2017 and appears as part of the ESWEEK-TECS special issue."

This work has been partially supported by the Academy of Finland, the U.S. National Science Foundation (awards #1329759 and #1139138), the Deutsche Forschungsgemeinschaft (PRETSY2 project, award DFG HA 4407/6-2), the Swedish Research Council (grant Nr. 2015-04154, *PolUser: Rich User-Controlled Privacy Policies*) and the University of Auckland Faculty Research Development Fund (grant Nr. 3707500).
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
XXXX-XXXX/2017/7-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Pacemakers are life saving devices that correct bradycardia, a type of arrhythmia when the heart is beating less than 50 beats per minute (bpm). While over a million pacemakers are implanted every year, there have been several adverse events associated with these devices. Ironically, devices designed to save lives have caused serious harm including death of many patients [2]. Such sobering statistics call for alternative techniques inspired by formal methods.

The first known formal model of the cardiac cell based on Hybrid Automata (HA) was developed in 2008 [24]. Subsequently, Boston Scientific released a specification, based on which a formal model of the pacemaker using Timed Automata (TA) [16] was developed. This model was validated using a random heart model also expressed as TA. While this paved the way for model-checking, the limitation of this work is that the associated heart model was simplistic. Subsequently, Oxford university researchers [9] created a 33-node HA model to capture forward conduction of the heart. They then validated a TA-based specification of the pacemaker [16] by creating the closed loop system in Simulink. Recently, this model was extended to capture not only forward, but also backward conduction [25].

In spite of the use of such formal models, model checking remains elusive due to limitations with the underlying HA. For example, recently the SpaceEx model checker for hybrid automata was used in [21]. Here, they showed that the tool could only verify a conduction system of up to 16 cells. To overcome such scalability problems, recent work examined abstraction [14] and statistical model checking [17]. However, we believe that all these ignore a key requirement for pacemaker validation. The human heart is not static but a time varying dynamic system i.e. different types of arrhythmia may happen randomly and also the state of the heart might change randomly. It will be extremely difficult to validate such a closed-loop system formally, albeit recently a mode-based testing approach has been developed. Here, the heart goes through run-time parametrisation to exhibit different types of arrhythmia [1]. This is where we believe that run-time enforcement [20] may provide an alternative that is both formal and scalable.

We propose the closed-loop Run-time Enforcement (RE) of the heart-pacemaker system, where they interact through an "enforcer", which provides an additional layer of safety. It is designed to enforce a set of *critical properties* and does not perform all the tasks of a typical pacemaker, which may include rate adaptation, Electrogram (EGM) processing (signal processing), and efficiency considerations. The enforcement monitor, on the other hand, is much simpler and deals with the run-time enforcement of certain *"life or death"* properties, which are only enforced when the pacemaker fails to guarantee them. The heart-pacemaker combination provides a typical example of a Cyber-Physical System (CPS) [15] and we need to carefully consider the enforcement of such systems based on the current status of related work.

A Pacemaker is a *reactive* system, which must react to the stimulus provided by the heart. While run-time enforcement of *transformational systems* is extensively studied, the enforcement of reactive systems is just emerging [6]. Unlike the enforcers in the former category, which can delay events through buffering, enforcers for reactive systems must operate in the same reactive cycle. While the shield synthesis [6] is relevant in the pacemaker context to some degree, it only performs the enforcement in one direction. While enforcing properties of CPS, bi-directional enforcement (i.e., monitoring both the inputs and the outputs of the controller, and editing erroneous inputs/outputs when necessary) is critical. More importantly, the enforcer must perform the enforcement of timed properties, which is yet to be formalised for bi-directional enforcement of reactive systems. In addition to order of events, timed properties allow to express how time should elapse between events, and occurrence time of events effects satisfaction of the property.

### 1.1 Overview of the proposed approach

In this section, we describe the general principles of bi-directional RE for CPS via examples illustrating the expected input/output behavior of the synthesized enforcer.

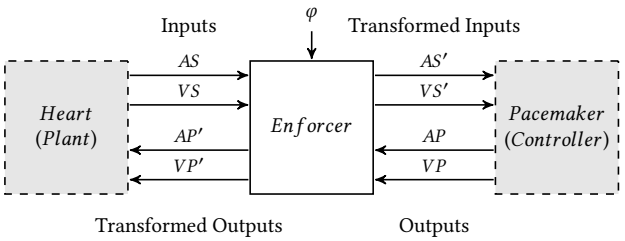



Fig. 1. Enforcer between heart and pacemaker.

Let us consider the framework in the Figure 1, where the heart is the plant, and the pacemaker is the controller. We will consider some requirements of the pacemaker.

The set of Boolean inputs from the heart to the pacemaker are $I = \{AS, VS\}$, where Atrial Sense ($AS$) is the sensor to sense the electrical pulse that contracts the walls of the atria, and Ventricular Sense ($VS$) is the sensor to sense the electrical pulse that contracts the walls of the ventricle. The set of Boolean outputs from the pacemaker are $O = \{AP, VP\}$, where Atrial Pace ($AP$) (resp. Ventricular Pace ($VP$)) is the signal generated by the pacemaker to pace the atrium (resp. ventricle).

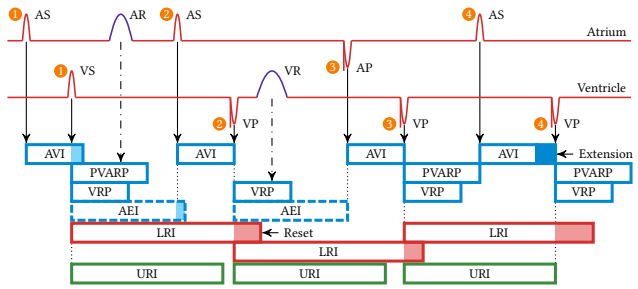



Fig. 2. Timing Diagram for a DDD mode pacemaker

A timing diagram for a DDD mode pacemaker is shown in Figure 2. 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. Labels on the two traces show whether the event is a sensed natural event ($AS$ or $VS$), ignored natural event (Atrial Refractory Sense ($AR$) or Ventricular Refractory Sense ($VR$)), or artificial pacing from the pacemaker ($AP$ or $VP$).

The key timers shown at the bottom of the figure are Post-Ventricular Atrial Refractory Period ($PVARP$), Atrioventricular Interval ($AVI$), Lower Rate Interval ($LRI$) and Upper Rate Interval ($URI$). These maintain specific timing delay i.e. the $AVI$ timer maintains the correct time delay between any atrial event and a subsequent ventricular event.

In this paper we consider the following requirements from [16]. Note that timing intervals such as $AVI$, $AEI$ and $LRI$ are real values denoted as some constant $C$. Their discrete counter parts are denoted as $AVI_{TICKS}$, $AEI_{TICKS}$ etc are denoted as $C_{TICKS}$[1].

$P_1$ $AP$ and $VP$ cannot happen simultaneously.
$P_2$ $VS$ or $VP$ must be true within $AVI_{TICKS}$ after an atrial event $AS$ or $AP$.
$P_3$ $AS$ or $AP$ must be true within $AEI_{TICKS}$ after an ventricle event $VS$ or $VP$.
$P_4$ After a ventricle event, another ventricle event can happen only after $URI_{TICKS}$.
$P_5$ After a ventricle event, another ventricle event should happen within $LRI_{TICKS}$.

Properties $P_1 \dots P_5$ are discrete time properties, and their dense variants with real-time constraints are denoted as $P'_1 \dots P'_5$. Dense time properties can be expressed as Timed Automata (TA) [3].

We express discrete time properties using a variant of Discrete Timed Automata (DTA) [7] that we call Synchronous DTA (SDTA). SDTA have a different timed semantics, where discrete transitions are not possible and all transitions take one *tick* relative to the ticks of a synchronous global clock inspired by synchronous languages [5]. The use of DTA over TA is primarily motivated by the fact that the approach can directly use a formulation similar to synchronous languages, where time is discretized. This makes the overall algorithm simple, where the pacemaker and the enforcer are composed synchronously. From now on when we refer to DTA in this paper, we mean SDTA. We introduce this formally in Section 2.

In this paper, we consider and focus on the heart-pacemaker system to illustrate the proposed RE framework. Our formalization is also applicable in general to other types of CPS, where in addition to enforcing outputs from the system, enforcing inputs from the physical side is also relevant. For instance, in the automotive domain, consider a cruse controller. Consider a property which states that "when the brake and cruise inputs are simultaneously present, the brake is given priority". In this event, the enforcer will forward the brake input while toggling the cruise input. Moreover, when we enforce security policies, an enforcer should be allowed to suppress malicious inputs from an attacker. When we consider systems such as Unmanned Aerial Systems (UAS), an attacker may modify and feed-in bad inputs to take control over the system. An enforcement mechanism may be used to detect and prevent such attacks.

The main contributions of the paper are:

- We introduce the concept of synchronous run-time enforcement of reactive systems, which includes timed properties, using DTA-based specification. A key motivation for using the synchronous framework is that it is a standard framework while designing reactive systems and lends itself nicely to the run-time enforcement problem, which is yet to be studied for synchronous systems involving timed properties.
- The developed synchronous approach is intentional, as compared to the approach based on dense TA, the developed algorithm is simple and does not require region or zone graph construction.
- We formalise a set of constraints for enforcer synthesis, which are novel. Except soundness, all other constraints are new and specific to the developed formulation.
- The developed framework is implemented in the SCCharts tool-chain, which supports synchronous enforcement in addition to synchronous observers.

## 2 PRELIMINARIES AND NOTATIONS

A finite (resp. infinite) word over a finite alphabet $\Sigma$ is a finite sequence $\sigma = a_1 \cdot a_2 \cdots a_n$ (resp. infinite sequence $\sigma = a_1 \cdot a_2 \cdots$) of elements of $\Sigma$. The set of finite (resp. infinite) words over $\Sigma$ is

[1] $C_{TICKS} = \lceil \frac{C}{WCRT} \rceil$ or $C_{TICKS} = \lfloor \frac{C}{WCRT} \rfloor$. $WCRT$ denotes the worst case reaction time and this is detailed in Section 4.2.

denoted by $\Sigma^*$ (resp. $\Sigma^\omega$). The *length* of a finite word $\sigma$ is $n$ and is denoted by $|\sigma|$. The empty word over $\Sigma$ is denoted by $\epsilon_\Sigma$, or $\epsilon$ when clear from the context. $\Sigma^+$ denotes $\Sigma^* \setminus \{\epsilon\}$. The *concatenation* of two words $\sigma$ and $\sigma'$ is denoted by $\sigma \cdot \sigma'$. A word $\sigma'$ is a *prefix* of a word $\sigma$, denoted as $\sigma' \preccurlyeq \sigma$, whenever there exists a word $\sigma''$ such that $\sigma = \sigma' \cdot \sigma''$; conversely $\sigma$ is said to be an *extension* of $\sigma'$.

We focus on systems such as the pacemaker that has Boolean signals as inputs and outputs. We consider a reactive system with a finite ordered sets of Boolean inputs $I = \{i_1, i_2, \cdots, i_n\}$ and Boolean outputs $O = \{o_1, o_2, \cdots, o_n\}$. The input alphabet is $\Sigma_I = 2^I$, and the output alphabet is $\Sigma_O = 2^O$ and the input-output alphabet $\Sigma = \Sigma_I \times \Sigma_O$. Each input (resp. output) event will be denoted as a bit-vector/complete monomial. For example, let $I = \{A, B\}$. Then, the input $\{A\} \in \Sigma_I$ is denoted as 10, while $\{B\} \in \Sigma_I$ is denoted as 01 and $\{A, B\} \in \Sigma_I$ is denoted as 11. A reaction (or input-output event) is of the form $(x_i, y_i)$, where $x_i \in \Sigma_I$ and $y_i \in \Sigma_O$.

### 2.1 CPS as a synchronous system

Controllers of CPSs operate synchronously [5] in a reactive loop that executes once every *tick / reaction*. During a tick, all three components – the plant, the controller, and the enforcer – are executed once. An execution $\sigma$ of a synchronous program $\mathcal{P}$ is an infinite sequence of input-output events $\sigma \in \Sigma^\omega$, and the *behavior* of a synchronous program $\mathcal{P}$ is denoted as $exec(\mathcal{P}) \subseteq \Sigma^\omega$. The *language* of $\mathcal{P}$ is denoted by $\mathcal{L}(\mathcal{P}) = \{\sigma \in \Sigma^* | \exists \sigma' \in exec(\mathcal{P}) \wedge \sigma \preccurlyeq \sigma'\}$. $\mathcal{L}(\mathcal{P})$ is the set of all finite prefixes of the sequences in $exec(\mathcal{P})$.

### 2.2 Properties

A property $\varphi$ over $\Sigma$ defines a set $\mathcal{L}(\varphi) \subseteq \Sigma^*$. A program $\mathcal{P} \models \varphi$ iff $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\varphi)$. In this paper, properties are formally defined as automata extended with a set of integer variables that we define in the sequel.

*2.2.1 Discrete Timed Automata (DTA).* We will now introduce 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. DTA are timed automata with integer valued clocks [7]. In this paper, properties that we want to enforce are defined using a variant of DTA, defined as follows:

*Definition 2.1 (Discrete Timed Automata (DTA)).* A *Discrete Timed Automaton* is a tuple $\mathcal{A} = (L, l_0, l_v, \Sigma, V, \Delta, F)$ where $L$ is the set of *locations*, $l_0 \in L$ is the initial location, $\Sigma$ 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 $\Delta$ is $\Delta \subseteq L \times G(V) \times R \times \Sigma \times L$ where $G(V)$ denotes the set of *guards*, i.e., constraints defined as conjunctions of simple constraints of the form $v \bowtie c$ with $v \in V$, $c \in \mathbb{N}$ and $\bowtie \in \{<, \leq, =, \geq, >\}$, and $R \subseteq V$ is a subset of integer clocks that are reset to 0.

A DTA is a finite automaton extended with a finite set of integer variables (i.e., discrete clocks). Let $V = \{v_1, \ldots, v_k\}$ be a finite set of integer clocks. A *valuation* for $v$ is an element of $\mathbb{N}$, that is a function from $v$ to $\mathbb{N}$. The set of valuations for the set of clocks $V$ is denoted by $\chi$. For $\chi \in \mathbb{N}^V$, $\chi + 1$ 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 $\chi$ where all the clock variables in $V'$ are assigned to 0. Given $g \in G(V)$ and $\chi$, $\chi \models g$ if $g$ holds according to $\chi$.

*Example 2.2 (Example property defined as a DTA).* Let $I = \{A\}$ and $O = \{B\}$, and the input-output alphabet $\Sigma = 2^I \times 2^O$. Event $(1, 0)$ indicates $(A, \overline{B})$[2]. Consider the following property: $S_1$: *"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."*. The DTA in Figure 3 defines property $S_1$, where the set of integer clocks

[2]In some examples, for convenience, we use – to indicate either 0 or 1.

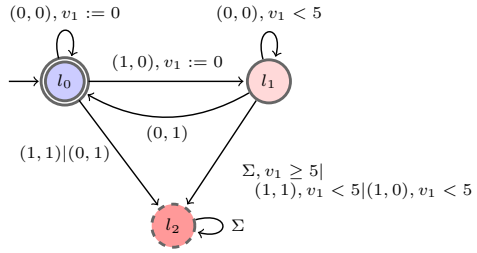



Fig. 3. Property $S_1$ defined as DTA $\mathcal{A}_{S_1}$.

$V = \{v_1\}$. Location $l_0$ is the initial location, location $l_2$ is the non-accepting trap location and the set of accepting locations is $\{l_0\}$. Clock $v_1$ is reset to 0 upon transition from $l_0$ to $l_1$. The transition from location $l_1$ to $l_1$ (self-loop) is taken upon input-output event $(0,0)$ and if the value of the clock $v_1$ is less than 5.

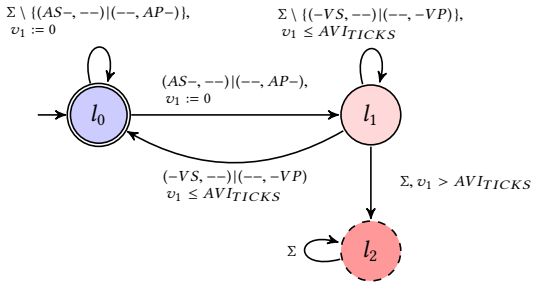



Fig. 4. Property $P_2$ defined as DTA $\mathcal{A}_{P_2}$.

The semantics of a DTA is defined as a transition system where each state consists of the current location and the current values of all the integer clocks. The semantics of a DTA is defined as follows.

*Definition 2.3 (Semantics of DTA).* The *semantics* of a DTA is a transition system $[\![\mathcal{A}]\!] = (Q, q_0, \Sigma, \rightarrow, Q_F, q_v)$ where $Q = L \times \mathbb{N}^V$ is the set of *states*, $q_0 = (l_0, \chi_0)$ is the *initial state* where $\chi_0$ is the valuation that maps every integer clock variable in $V$ to 0, $Q_F = F \times \mathbb{N}^V$ is the set of *accepting states*, and $q_v = l_v \times \mathbb{N}^V$ is the set of trap states. The transition relation $\rightarrow \subseteq Q \times \Sigma \times Q$ is a set of transitions of the form $(l, \chi) \xrightarrow{a} (l', \chi')$ with $\chi' = (\chi + 1)[r \leftarrow 0]$ whenever there exists $(l, g, r, a, l') \in \Delta$ such that $\chi \models g$.

A *run* $\rho$ of $\mathcal{A}$ from a state $q \in Q$ over a *untimed trace* $\sigma = a_1 \cdot a_2 \cdots a_n \in \Sigma^*$ is a sequence of moves in $[\![\mathcal{A}]\!]$ denoted as $\rho = q \xrightarrow{a_1} q_1 \cdots q_{n-1} \xrightarrow{a_n} q_n$, for some $n \in \mathbb{N}$, and is denoted as $q \xrightarrow{\sigma} q_n$. The set of runs from the initial state $q_0 \in Q$, is denoted $\mathrm{Run}(\mathcal{A})$ and $\mathrm{Run}_{Q_F}(\mathcal{A})$ denotes the subset of those runs *accepted* by $\mathcal{A}$, i.e., ending in an accepting state $q_n \in Q_F$. We denote by $\mathcal{L}(\mathcal{A})$ the set of untimed traces of runs in $\mathrm{Run}_{Q_F}(\mathcal{A})$, and we use DTA to define untimed languages. We thus say that a untimed word is *accepted* by $\mathcal{A}$ if it is the trace of an accepted run.

*Definition 2.4 (Deterministic (complete) DTA).* A discrete timed automaton $\mathcal{A} = (L, l_0, l_v, \Sigma, V, \Delta, F)$ with its semantics $[\![\mathcal{A}]\!]$ is said to be a *deterministic* DTA whenever for any location $l$ and any two distinct transitions $(l, g_1, a, Y_1, l_1') \in \Delta$ and $(l, g_2, a, Y_2, l_2') \in \Delta$ with same source $l$, the conjunction of

guards $g_1 \wedge g_2$ is unsatisfiable. $\mathcal{A}$ is *complete* whenever for any location $l \in L$ and any event $a \in \Sigma$, the disjunction of the guards of the transitions leaving $l$ and labelled by $a$ evaluates to *true*.

Remark 1. *In this paper, we focus on enforcement of properties that can be defined by a DTA, i.e., the set of regular discrete time properties. In the rest of this paper, since we consider synchronous programs, we restrict to properties defined as deterministic DTA. Note that deterministic DTA are not time-deterministic, i.e., the time until something happens can be non-deterministic. Regarding completeness, if the user provides an incomplete DTA, we complete it by introducing a unique non-accepting trap location. We also consider that the set of locations L contain only locations that are reachable from the initial location $l_0$.*

*Example 2.5 (Run of a DTA).* Let us consider the DTA in Figure 3. The set of clock variables $V = \{v_1\}$, and initial location is $l_0$. Initial state of this DTA is $(l_0, v_1 = 0)$. Let $\sigma = (0,0) \cdot (1,0) \cdot (0,0) \cdot (0,0) \cdot (0,1)$. Run of the DTA from the initial state $(l_0, v_1 = 0)$ upon $\sigma$ is $(l_0, v_1 = 0) \xrightarrow{(0,0)} (l_0, v_1 = 0) \xrightarrow{(1,0)} (l_1, v_1 = 0) \xrightarrow{(0,0)} (l_1, v_1 = 1) \xrightarrow{(0,0)} (l_1, v_1 = 2) \xrightarrow{(0,1)} (l_0, v_1 = 3)$. $\sigma$ is an accepted by the DTA in Figure 3 since the state reached upon $\sigma$ is $(l_0, v_1 = 3)$ which is an accepting state.

*Definition 2.6 (Product of DTAs).* Given two discrete TAs $\mathcal{A}^1 = (L^1, l_0^1, l_v^1, \Sigma^1, V^1, \Delta^1, F^1)$ and $\mathcal{A}^2 = (L^2, l_0^2, l_v^2, \Sigma^2, V^2, \Delta^2, F^2)$ with disjoint sets of integer clocks, their product is the DTA $\mathcal{A}^1 \times \mathcal{A}^2 = (L, l_0, l_v, \Sigma, V, \Delta, F)$ where $L = L^1 \times L^2$, $l_0 = (l_0^1, l_0^2)$, $l_v = (l_v^1, l_v^2)$, $V = V^1 \cup V^2$ , $F = F^1 \times F^2$, and $\Delta \subseteq L \times \mathcal{G}(V) \times R \times \Sigma \times L$ is the *transition relation*, with $((l^1, l^2), g^1 \wedge g^2, R^1 \cup R^2, a, (l^{1'}, l^{2'})) \in \Delta$ if $(l^1, g^1, R^1, a, l^{1'}) \in \Delta^1$ and $(l^2, g^2, R^2, a, l^{2'}) \in \Delta^2$. In the product DTA, all locations in $(L^1 \times l_v^2) \cup (l_v^1 \times L^2)$ are trap locations, and all the outgoing transitions for these locations can be replaced with self loops. We consider merging all the trap locations into a single location $l_v$ where any outgoing transition from any location in $L \setminus (L^1 \times l_v^2) \cup (l_v^1 \times L^2)$ to a location in $(L^1 \times l_v^2) \cup (l_v^1 \times L^2)$ goes to $l_v$ instead.

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

*2.2.2 Preliminaries to RE.* Given an input-output word $\sigma = (x_1, y_1) \cdot (x_2, y_2) \cdots (x_n, y_n) \in \Sigma^*$, the input word obtained from $\sigma$ is denoted by $\sigma_I$ where $\sigma_I = x_1 \cdot x_2 \cdots x_n \in \Sigma_I^*$ is the projection on inputs ignoring outputs. Similarly, the output word obtained from $\sigma$ is denoted by $\sigma_O$ where $\sigma_O = y_1 \cdot y_2 \cdots y_n \in \Sigma_O^*$ is the projection on outputs.

We consider bi-directional enforcement, where the enforcer has to first transform inputs from the environment in each step according to property $\varphi$ defined as DTA $\mathcal{A}_\varphi$. We thus need to consider the input property that we obtain from $\mathcal{A}_\varphi$ by projecting on inputs.

*Definition 2.7 (Input DTA $\mathcal{A}_{\varphi_I}$).* Given property $\varphi$ defined as DTA $\mathcal{A}_\varphi = (L, l_0, l_v, \Sigma, V, \Delta, F)$, input DTA $\mathcal{A}_{\varphi_I} = (L, l_0, l_v, \Sigma_I, V, \Delta_I, F)$ is obtained from $\mathcal{A}_\varphi$ by ignoring outputs on the transitions, i.e., for every transition $(l, g, r, (x, y), l') \in \Delta$ in $\mathcal{A}_\varphi$, there is a transition $(l, g, r, x, l') \in \Delta_I$ in $\mathcal{A}_{\varphi_I}$. $\mathcal{L}(\mathcal{A}_{\varphi_I})$ is denoted as $\varphi_I \subseteq \Sigma_I^*$.

Automaton $\mathcal{A}_{\varphi_I}$ is identical to automaton $\mathcal{A}_\varphi$ where we ignore outputs when we traverse in the automaton. Locations $L$ in both the automata, initial state $l_0$ and violating state $l_v$ are exactly the same. Moreover, all the clock variables in $\mathcal{A}_\varphi$ also remain in $\mathcal{A}_{\varphi_I}$. Also, the number of transitions, source and destination of each transition, and the guards and resets of clock variables are also the same in both automata. The only difference is we ignore outputs in the automaton $\mathcal{A}_{\varphi_I}$. If $(x, y)$ is in $\Sigma$, then $x \in \Sigma_I$, and every transition $(l, g, r, (x, y), l') \in \Delta$ in $\mathcal{A}_\varphi$, is replaced with transition $(l, g, r, x, l') \in \Delta_I$ in $\mathcal{A}_{\varphi_I}$.

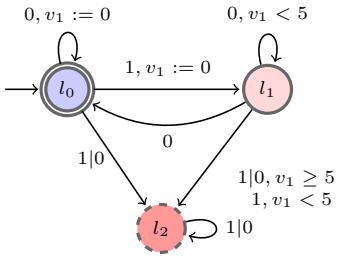



Fig. 5. Automaton obtained from $\mathcal{A}_{S_1}$ in Fig. 3 by projecting on inputs.

*Example 2.8 (Input DTA $\mathcal{A}_{\varphi_I}$ obtained from $\mathcal{A}_\varphi$).* Let us consider the DTA in Figure 3 defining the property $S_1$ introduced in Example 2.2. Figure 5 presents the input DTA obtained from the DTA in Figure 3. Though the DTA $\mathcal{A}_\varphi$ is deterministic, the input DTA $\mathcal{A}_{\varphi_I}$ might be non-deterministic as is the case in Figure 5.

*Edit Functions.* Given property $\varphi \subseteq \Sigma^*$, defined as DTA $\mathcal{A}_\varphi = (L, l_0, l_v, \Sigma, V, \Delta, F)$ with semantics $[\![\mathcal{A}_\varphi]\!] = (Q, q_0, \Sigma, \rightarrow, Q_F, q_v)$, we introduce $\text{editI}_{\varphi_\text{I}}$ (resp. $\text{editO}_\varphi$), which the enforcer uses for editing input (resp. output) events (whenever necessary), according to input property $\varphi_I$ (resp. input-output property $\varphi$). Note that in each step the enforcer first processes the input from the environment, and transforms it using $\text{editI}_{\varphi_\text{I}}$ based on the input property $\varphi_I$ obtained from the input-output property $\varphi$ that we want to enforce. Later, the output produced by the program is transformed by the enforcer (when necessary) using $\text{editO}_\varphi$ based on the input-output property $\varphi$ that we want to enforce.

- **$\text{editI}_{\varphi_\text{I}}(\sigma_I)$**: Given $\sigma_I \in \Sigma_I^*$, $\text{editI}_{\varphi_\text{I}}(\sigma_I)$ is the set of input events $x$ in $\Sigma_I$ such that the word obtained by extending $\sigma_I$ with $x$ can be extended to a sequence that satisfies $\varphi_I$ (i.e., there exists $\sigma' \in \Sigma_I^*$ such that $\sigma_I \cdot x \cdot \sigma'$ satisfies $\varphi_i$). Formally,

$$\text{editI}_{\varphi_\text{I}}(\sigma_I) = \{x \in \Sigma_I : \exists \sigma' \in \Sigma_I^*, \sigma_I \cdot x \cdot \sigma' \models \varphi_I\}.$$

  Consider the automaton $\mathcal{A}_{\varphi_I} = (L, l_0, l_v, \Sigma_I, V, \Delta_I, F)$ with semantics $[\![\mathcal{A}_{\varphi I}]\!] = (Q_I, q_{0_I}, \Sigma_I, \rightarrow_I, Q_{F_I}, q_{v_I})$. Let $q_I \in Q_I$ correspond to a state reachable in $\mathcal{A}_{\varphi_I}$ (i.e., $q_{0_I} \xrightarrow{\sigma_I} q_I$) upon $\sigma_I$. We define $\text{editI}_{\mathcal{A}_{\varphi_\text{I}}}(q_I)$ as follows:

  $\text{editI}_{\mathcal{A}_{\varphi_\text{I}}}(q_I) = \{x \in \Sigma_I : \exists \sigma' \in \Sigma_I^*, q_I \xrightarrow{x \cdot \sigma'}_I q'_I \wedge q'_I \in Q_{F_I}\}$.

  *Example 2.9.* Consider the automaton in Fig. 5 obtained from the automaton in Fig. 3 by ignoring outputs. Let $\sigma = (0,0) \cdot (1,0)$, and thus $\sigma_I = 0 \cdot 1$. Then $(l_0, v_1 = 0) \xrightarrow{0 \cdot 1}_I (l_1, v_1 = 0)$, and $\text{editI}_{\mathcal{A}_{\varphi_\text{I}}}((l_1, v_1 = 0)) = \{0\}$.

- **$\text{editO}_\varphi(\sigma, x)$**: Given an input-output word $\sigma \in \Sigma^*$ and an input event $x \in \Sigma_I$, $\text{editO}_\varphi(\sigma, x)$ is the set of output events $y$ in $\Sigma_O$ such that the input-output word obtained by extending $\sigma$ with $(x, y)$ can be extended to a sequence that satisfies the property $\varphi$ (i.e., $\exists \sigma' \in \Sigma^*$ such that $\sigma \cdot (x, y) \cdot \sigma' \models \varphi$). Formally,

$$\text{editO}_\varphi(\sigma, x) = \{y \in \Sigma_O : \exists \sigma' \in \Sigma^*, \sigma \cdot (x, y) \cdot \sigma' \models \varphi\}.$$

  Consider the DTA $\mathcal{A}_\varphi = (L, l_0, l_v, \Sigma, V, \Delta, F)$ defining property $\varphi$ with semantics $[\![\mathcal{A}_\varphi]\!] = (Q, q_0, \Sigma, \rightarrow, Q_F, q_v)$, and an input event $x \in \Sigma_I$. If $q \in Q$ corresponds to a state reached in

$\mathcal{A}_\varphi$ upon $\sigma_I$ (i.e., $q_0 \xrightarrow{\sigma} q$), $\text{editO}_\varphi(\sigma, x)$ can be alternatively defined as follows:
$\text{editO}_{\mathcal{A}_\varphi}(q, x) = \{y \in \Sigma_O : \exists \sigma' \in \Sigma^*, q \xrightarrow{(x,y)\cdot\sigma'} q' \wedge q' \in Q_F\}$.

*Example 2.10.* Let us consider the property $S_1$ defined by the automaton in Figure 3. We have $\text{editO}_{\mathcal{A}_\varphi}((l_0, v_1 = 0), 0) = \{0\}$.

- **rand-editI$_{\varphi_\text{I}}(\sigma_I)$**: Given $\sigma_I \in \Sigma_I^*$ if $\text{editI}_{\varphi_\text{I}}(\sigma_I)$ is non-empty, then $\text{rand-editI}_{\varphi_\text{I}}(\sigma_I)$ returns an element (chosen randomly) from $\text{editI}_{\varphi_\text{I}}(\sigma_I)$, and is undefined if $\text{editI}_{\varphi_\text{I}}(\sigma_I)$ is empty. Given $q_I \in Q_I$, if $\text{editI}_{\mathcal{A}_{\varphi_\text{I}}}(q_I)$ is non-empty, then $\text{rand-editI}_{\mathcal{A}_{\varphi_\text{I}}}(q_I)$ returns an element (chosen randomly) from $\text{editI}_{\mathcal{A}_{\varphi_\text{I}}}(q_I)$, and is undefined if $\text{editI}_{\mathcal{A}_{\varphi_\text{I}}}(q_I)$ is empty.
- **rand-editO$_\varphi(\sigma, x)$**: Given $\sigma \in \Sigma^*$, and $x \in \Sigma_I$, if $\text{editO}_\varphi(\sigma, x)$ is non-empty, then $\text{rand-editO}_\varphi(\sigma, x)$ returns an element (chosen randomly) from $\text{editO}_\varphi(\sigma, x)$, and is undefined if $\text{editO}_\varphi(\sigma, x)$ is empty. Given $q \in Q$ and $x \in \Sigma_I$, if $\text{editO}_{\mathcal{A}_\varphi}(q, x)$ is non-empty, then $\text{rand-editO}_{\mathcal{A}_\varphi}(q, x)$ returns an element (chosen randomly) from $\text{editO}_{\mathcal{A}_\varphi}(q, x)$, and is undefined if $\text{editO}_{\mathcal{A}_\varphi}(q, x)$ is empty.
- **minD-editI$_{\varphi_\text{I}}(\sigma_I, x)$**: Given $\sigma_I \in \Sigma_I^*$ and $x \in \Sigma_I$, if $\text{editI}_{\varphi_\text{I}}(\sigma_I)$ is non-empty, then $\text{minD-editI}_{\varphi_\text{I}}(\sigma_I, x)$ returns an event from $\text{editI}_{\varphi_\text{I}}(\sigma_I)$ with minimal distance[3] w.r.t $x$, and is undefined if $\text{editI}_{\varphi_\text{I}}(\sigma_I)$ is empty. Given $q_I \in Q_I$ and $x \in \Sigma_I$, if $\text{editI}_{\mathcal{A}_{\varphi_\text{I}}}(q_I)$ is non-empty, then $\text{minD-editI}_{\mathcal{A}_{\varphi_\text{I}}}(q_I, x)$ returns an event from $\text{editI}_{\mathcal{A}_{\varphi_\text{I}}}(q_I)$ with minimal distance w.r.t $x$, and is undefined $\text{editI}_{\mathcal{A}_{\varphi_\text{I}}}(q_I)$ is empty.
- **minD-editO$_\varphi(\sigma, x, y)$**: Given $\sigma \in \Sigma^*$, $x \in \Sigma_I$ and $y \in \Sigma_O$, if $\text{editO}_\varphi(\sigma, x)$ is non-empty, then $\text{minD-editO}_\varphi(\sigma, x, y)$ returns an event from $\text{editO}_\varphi(\sigma, x)$ with minimal distance w.r.t $y$, and is undefined if $\text{editO}_\varphi(\sigma, x)$ is empty. Given $q \in Q$, $x \in \Sigma_I$ and $y \in \Sigma_O$, if $\text{editO}_{\mathcal{A}_\varphi}(q, x)$ is non-empty, then $\text{minD-editO}_{\mathcal{A}_\varphi}(q, x, y)$ returns an event from $\text{editO}_{\mathcal{A}_\varphi}(q, x)$ with minimal distance w.r.t $y$, and is undefined $\text{editO}_{\mathcal{A}_\varphi}(q, x)$ is empty.

## 3 PROBLEM DEFINITION

In this section, we formalize the runtime enforcement problem for CPS. We consider a digital controller (e.g. the pacemaker) with Boolean input and output streams, and enforcement of any regular property $\varphi$, defined as a DTA. We also consider a plant (e.g. the heart) to be a continuous system, which is sampled periodically, that result in Boolean streams (e.g. the Atrial Sense (*AS*) and Ventricular Sense (*VS*) signals sensed by the pacemaker leads). In the setting we consider, as illustrated in Figure 1, an enforcer monitors and corrects both input and output of a synchronous program according to a given correctness property $\varphi$. Let us recall that an 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). At each step, the enforcer consumes an input event $(x, y)$ and produces an output event $(x', y')$ by editing $x$ and $y$ if necessary.

We assume that the controller (pacemaker) may be invoked through a special function call called ptick. Since the pacemaker is considered to be a black-box, internals of the function ptick are considered to be unknown. Formally, ptick is a function (with internal state) from $\Sigma_I$ to $\Sigma_O$ that takes a bit vector $x \in \Sigma_I$ and returns a bit vector $y \in \Sigma_O$.

An enforcer for a property $\varphi$ can only edit an input-output event when necessary, and it cannot block, delay or suppress events. Let us recall the two functions $\text{editI}_{\varphi_\text{I}}$ and $\text{editO}_\varphi$ that were introduced in Section 2 that the enforcer for $\varphi$ uses to edit the current input (respectively output) event according to property $\varphi$.

At an abstract level, an enforcer may be viewed as a function that transforms words. An enforcement function for a given property $\varphi$ takes as input a word over $\Sigma^*$ and outputs a word over $\Sigma^*$ that satisfies $\varphi$, or can be extended to satisfy $\varphi$ in the future.

[3]Distance between two events belonging to the same alphabet is the number of bits that differ in both the events.

*Definition 3.1 (Enforcer for $\varphi$).* Given property $\varphi \subseteq \Sigma^*$, an *enforcer* for $\varphi$ is a function $E_\varphi : \Sigma^* \rightarrow \Sigma^*$ satisfying the following constraints:

**Soundness**

$$\forall \sigma \in \Sigma^*, \exists \sigma' \in \Sigma^* : E_\varphi(\sigma) \cdot \sigma' \models \varphi. \quad \textbf{(Snd)}$$

**Monotonicity**

$$\forall \sigma, \sigma' \in \Sigma^* : \sigma \preccurlyeq \sigma' \Rightarrow E_\varphi(\sigma) \preccurlyeq E_\varphi(\sigma'). \quad \textbf{(Mono)}$$

**Instantaneity**

$$\forall \sigma \in \Sigma^* : |\sigma| = |E_\varphi(\sigma)|. \quad \textbf{(Inst)}$$

**Transparency**

$$\begin{array}{l} \forall \sigma \in \Sigma^*, \forall x \in \Sigma_I, \forall y \in \Sigma_O, \exists \sigma' \in \Sigma^* : \\ E_\varphi(\sigma) \cdot (x,y) \cdot \sigma' \models \varphi \\ \Rightarrow E_\varphi(\sigma \cdot (x,y)) = E_\varphi(\sigma) \cdot (x,y). \end{array} \quad \textbf{(Tr)}$$

**Causality**

$$\begin{array}{l} \forall \sigma \in \Sigma^*, \forall x \in \Sigma_I, \forall y \in \Sigma_O, \\ \exists x' \in \mathsf{editI}_{\varphi_\mathsf{I}}((E_\varphi(\sigma))_I), \exists y' \in \mathsf{editO}_\varphi(E_\varphi(\sigma), x') : \\ E_\varphi(\sigma \cdot (x,y)) = E_\varphi(\sigma) \cdot (x',y'). \end{array} \quad \textbf{(Ca)}$$

*Soundness.* (**Snd**) means that for any input word $\sigma \in \Sigma^*$, the output of the enforcer $E_\varphi(\sigma)$ can be extended to a sequence that satisfies $\varphi$ (i.e., $\exists \sigma' \in \Sigma^* : E_\varphi(\sigma) \cdot \sigma' \models \varphi$).

*Monotonicity.* (**Mono**) expresses that the output of the enforcer for an extended input word $\sigma'$ of an input word $\sigma$, extends the output produced by the enforcer for $\sigma$. The monotonicity constraint means that the enforcer cannot undo what is already released as output.

*Instantaneity.* (**Inst**) expresses that for any given input sequence $\sigma$, the output of the enforcer $E_\varphi(\sigma)$ should contain exactly the same number of events that are in $\sigma$ (i.e., $|\sigma| = |E_\varphi(\sigma)|$). This means that, the 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. This requirement is essential for the enforcement of CPSs, which are reactive in nature.

*Transparency.* Transparency means that the enforcer will not unnecessarily edit any event. Any new input event $(x,y)$ will be simply forwarded by the enforcer if what has been computed as output earlier by the enforcer followed by $(x,y)$ can be extended to a sequence that satisfies $\varphi$ in the future.

(**Tr**) expresses that for any given input sequence $\sigma$ and any input event $(x,y)$, if the output of the enforcer for $\sigma$ (i.e., $E_\varphi(\sigma)$) followed by the input event $(x,y)$ has an extension $\sigma' \in \Sigma^*$ such that $E_\varphi(\sigma) \cdot (x,y) \cdot \sigma'$ satisfies the property $\varphi$, then the output that the enforcer produces for input $\sigma \cdot (x,y)$ will be $E_\varphi(\sigma) \cdot (x,y)$.

*Causality.* (**Ca**) expresses that for every input event $(x,y)$ the enforcer produces output event $(x',y')$ where the enforcer first processes the input part $x$, to produce the transformed input $x'$ according to property $\varphi$ using $\mathsf{editI}_{\varphi_\mathsf{I}}$. The enforcer later reads and transforms output $y \in \Sigma_O$ (output of the program after invoking function ptick with $x'$), to produce the transformed output $y'$ using $\mathsf{editO}_\varphi$.

The input-output sequence released as output by the enforcer upon reading the input-output sequence $\sigma$ is $E_\varphi(\sigma)$ and $(E_\varphi(\sigma))_I \in \Sigma_I^*$ is the projection on the inputs. $\mathsf{editI}_{\varphi_\mathsf{I}}(E_\varphi(\sigma))_I)$ returns a set of input events in $\Sigma_I$, such that $E_\varphi(\sigma))_I$ followed by any event in $\mathsf{editI}_{\varphi_\mathsf{I}}(E_\varphi(\sigma))_I)$ satisfies $\varphi_I$.

$\mathsf{editO}_\varphi(E_\varphi(\sigma)), x')$ returns a set of output events in $\Sigma_O$, such that for any event $y$ in $\mathsf{editO}_\varphi(E_\varphi(\sigma)), x')$, $E_\varphi(\sigma)) \cdot (x',y)$ satisfies $\varphi$.

Lemma 3.2 (Soundness, instantaneity, causality). (Ca) ⇒ (Snd) *and* (Ca) ⇒ (Inst).

For any $\varphi$, for any $\sigma \in \Sigma^*$, proof of Lemma 3.2 is straightforward from the constraints (Definition 3.1) and the definitions of edit functions.

Remark 2. *Our transparency constraint (Tr) is stronger than standard transparency in RE mechanisms (say Tr'), which is the following:*

$$\forall \sigma \in \Sigma^* : \sigma \models \varphi \implies E_\varphi(\sigma) = \sigma. \qquad \textbf{(Tr')}$$

*Constraint (Tr') means that when any given input-output word $\sigma \in \Sigma^*$ provided as input to the enforcer satisfies the property $\varphi$, then the enforcer should not edit any event and thus the output of the enforcer $E_\varphi(\sigma)$ should be $\sigma$. Constraint (Tr) also ensures constraint (Tr').*

Remark 3 (On editing inputs). *Note that our approach is general, and as a special case also works when the user wishes to forbid editing (correcting) the inputs (so the inputs become observable only, but not modifiable). For instance, in the pacemaker running example (introduced in Section 1), where the set of inputs is $I = \{AS, VS\}$, and the set of outputs is $O = \{AP, VP\}$, we can consider enforcing only properties that are concerned with outputs such as property "$P_1$: AP and VP cannot happen simultaneously", and for the automaton defining $P_1$, the set of actions are considered to be $\Sigma = \Sigma_I \times \Sigma_O$, where $\Sigma_I = 2^I, \Sigma_O = 2^O$. Thus, if properties to be enforced are concerned (express constraints) only with outputs (such as property $P_1$) , then the enforcer is guaranteed not to modify inputs.*

*Definition 3.3 (Enforceability).* Let $\varphi \subseteq \Sigma^*$ be a property. We say that $\varphi$ is *enforceable* iff an enforcer $E_\varphi$ for $\varphi$ exists according to Definition 3.1.

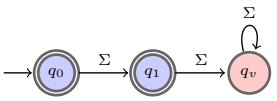



Fig. 6. A non-enforceable property.

*Example 3.4 (Non-enforceable property).* We illustrate that not all regular properties defined as DTA are enforceable according to Definition 3.1. Consider the automaton in Figure 6 defining the property $\varphi$ that we want to enforce, with $I = \{A\}$, $O = \{B\}$ and $\Sigma = \Sigma_I \times \Sigma_O$. Let the input-output sequence provided as input to the enforcer be $\sigma = (1,1) \cdot (1,0)$. When the enforcer reads the first event $(1,1)$, it can output $(1,1)$ (since every event in $\Sigma$ from $q_0$ leads to a non-violating state $q_1$). Note that from $q_1$, every event in $\Sigma$ 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 $\Sigma$ as output will violate soundness (since we will reach the trap state $q_v$, and there will be no possibility to reach an accepting state also in the future), and if no event is released as output, then the instantaneity constraint will be violated.

Theorem 3.5 (Condition for enforceability). *Consider a property $\varphi$ that is defined as DTA $\mathcal{A}_\varphi = (L, l_0, l_v, \Sigma, V, \Delta, F)$, with semantics $[\![\mathcal{A}_\varphi]\!] = (Q, q_0, \Sigma, \to, Q_F, q_v)$. Property $\varphi$ is enforceable iff the following condition holds:*

$$\forall q \in Q, q \notin q_v \Rightarrow \exists \sigma \in \Sigma^+ : q \xrightarrow{\sigma} q' \wedge q' \in Q_F. \qquad \textbf{(EnfCo)}$$

Theorem 3.5 expresses that for a property $\varphi$ defined as DTA $\mathcal{A}_\varphi = (L, l_0, l_v, \Sigma, V, \Delta, F)$ with semantics $[\![\mathcal{A}_\varphi]\!] = (Q, q_0, \Sigma, \rightarrow, Q_F, q_v)$, $E_\varphi$ according to Definition 3.1 exists iff an accepting state is reachable from every non-violating state (i.e., from every state $q \notin q_v$) in 1 or more steps. Let us recall that we consider $Q$ to contain only states that are reachable from $q_0$. Moreover, we also consider that if a state $q \in Q$ is non-accepting and non-violating (i.e., $q \notin \{Q_F \cup q_v\}$), then we consider whether there is a path from $q$ to a state in $Q_F$. State $q$ is merged with $q_v$ otherwise.

## 4 ENFORCER SYNTHESIS

In this section, we provide an algorithm for implementing the bi- directional synchronous enforcement problem defined in Section 3 for properties expressed as DTA. We also compare our approach with an enforcement approach based on (dense) timed automata.

### 4.1 Algorithm

Let the automaton $\mathcal{A}_\varphi = (L, l_0, l_v, \Sigma, V, \Delta, F)$ with semantics $[\![\mathcal{A}_\varphi]\!] = (Q, q_0, \Sigma, \rightarrow, Q_F, q_v)$ define property $\varphi$. Input automaton $\mathcal{A}_{\varphi_I} = (L, l_0, l_v, \Sigma_I, V, \Delta_I, F)$ with semantics $[\![\mathcal{A}_{\varphi I}]\!] = (Q, q_0, \Sigma_I, \rightarrow_I, Q_F, q_v)$ is obtained from $\mathcal{A}_\varphi$ by projecting on inputs (See section 2.2.2).

**ALGORITHM 1:** Enforcer

```
1:  t ← 0
2:  q ← q_0
3:  while true do
4:      x_t ← read_in_chan()
5:      if ∃σ'_I ∈ Σ*_I : q --x_t·σ'_I-->_I q' ∧ q' ∈ Q_F then
6:          x'_t ← x_t
7:      else
8:          x'_t ← rand-editI_{A_φI}(q)
9:      end if
10:     ptick(x'_t)
11:     y_t ← read_out_chan()
12:     if ∃σ' ∈ Σ* : q --(x'_t,y_t)·σ'--> q' ∧ q' ∈ Q_F then
13:         y'_t ← y_t
14:     else
15:         y'_t ← rand-editO_{A_φ}(q, x'_t)
16:     end if
17:     release((x'_t, y'_t))
18:     q ← q'' where q --(x'_t,y'_t)--> q'' ∧ q'' ∉ q_v
19:     t ← t + 1
20: end while
```

We provide an online algorithm that requires automata $A_\varphi$ and $A_{\varphi_I}$ as input. Algorithm 1 is an infinite loop, and an iteration of the algorithm is triggered at every time step. We adapt the *reactive interface* that is used for linking the program to its adjoining environment by following the structure of the interface described in [4]. We extend the interface by including the enforcer as an intermediary between the synchronous program and its adjoining environment.

In Algorithm 1, $t$ keeps track of the time-step (*tick*), initialized with 0, while $q$ keeps track of the current state of both automata $\mathcal{A}_\varphi$ and $\mathcal{A}_{\varphi_I}$. Note that state $q$ contains information about current location and the values of all the variables. Recall that the automaton $\mathcal{A}_{\varphi_I}$ is created from $\mathcal{A}_\varphi$

by projecting on inputs (see Section 2.2.2) and therefore has an identical structure with the only difference being that outputs are ignored on the transitions of $\mathcal{A}_{\varphi_I}$. For each transition, the guards and resets on clock variables are also the same in both automata. Note that at the beginning of each iteration of the algorithm, the current states of both the automata $[\![\mathcal{A}_\varphi]\!]$ and $[\![\mathcal{A}_{\varphi_I}]\!]$ are the same (where both are initialized with $q_0$). At $t$, if $\mathsf{EOut} \in \Sigma^*$ is the input-output sequence obtained by concatenating all the events released as output by the enforcer until time $t$, then $q$ corresponds to the state that we reach in $[\![\mathcal{A}_\varphi]\!]$ upon reading EOut. Similarly, if $\mathsf{EOut}_\mathsf{I} \in \Sigma_I^*$ is the sequence obtained by projecting on $x_i'$ from EOut, $q$ also corresponds to the state that we reach in $[\![\mathcal{A}_{\varphi_I}]\!]$ upon reading $\mathsf{EOut}_\mathsf{I}$.

Function read_in_chan (resp. read_out_chan) is a function corresponding to reading input (resp. output) channels, while function ptick corresponds to invoking the synchronous controller execution. Function release takes an input-output event, and releases it as output of the enforcer. Each iteration of the algorithm proceeds as follows: first all the input channels are read using function read_in_chan and the input event is assigned to $x_t$. Then the algorithm tests whether an accepting state is reachable from the current state $q$ upon $x_t$ extended with any sequence $\sigma_I \in \Sigma_I^*$. In case if this test succeeds, then it is not necessary to edit the input event $x_t$, and the transformed input $x_t'$ is assigned $x_t$. Otherwise, $x_t'$ is assigned with the output of $\text{rand-editI}_{\mathcal{A}_{\varphi_\mathsf{I}}}(q)$. Let us recall that $\text{rand-editI}_{\mathcal{A}_{\varphi_\mathsf{I}}}(q)$ returns an input event that leads to a state $q''$ from $q$ in $[\![\mathcal{A}_{\varphi_I}]\!]$, such that $q''$ is an accepting state, or an accepting state is reachable from $q''$.

After transforming the input $x_t$ according to $\mathcal{A}_{\varphi_I}$, the program is invoked with the transformed input $x_t'$ using function ptick. Afterwards, all the output channels are read using function read_out_chan and the output event is assigned to $y_t$. Then the algorithm tests whether an accepting state is reachable in $[\![\mathcal{A}_\varphi]\!]$ from the current state $q$ upon $(x_t', y_t)$ followed by any sequence $\sigma' \in \Sigma^*$. If this test succeeds, then it is not necessary to edit the output event $y_t$, and the transformed output $y_t'$ is assigned $y_t$. Otherwise, $y_t'$ is assigned with the output of $\text{rand-editO}_{\mathcal{A}_\varphi}(q, x_t')$. Note that $\text{rand-editO}_{\mathcal{A}_\varphi}(q, x_t')$ returns an output event $y_t'$ such that the input-output event $(x_t', y_t')$ leads to a state $q''$ in $[\![\mathcal{A}_\varphi]\!]$ such that $q''$ is an accepting, or an accepting state is reachable from $q''$ in $[\![\mathcal{A}_\varphi]\!]$.

Before proceeding with the next iteration, current state $q$ is updated to $q''$ which is the state reached upon $(x_t', y_t')$ from state $q$ in $[\![\mathcal{A}_\varphi]\!]$, and the time-step $t$ is incremented. Note that if there exists a transition from $q$ to $q''$ upon $(x_t', y_t')$ in $[\![\mathcal{A}_\varphi]\!]$, then there also exists a transition from $q$ to $q''$ upon $x_t'$ in $[\![\mathcal{A}_{\varphi_I}]\!]$. The current states of both the DTA are always synchronized and the same at the beginning of each iteration of the algorithm.

*Definition 4.1 ($E_\varphi^*$).* Consider an enforceable safety property $\varphi$. We define the function $E_\varphi^* : \Sigma^* \to \Sigma^*$, where $\Sigma = \Sigma_I \times \Sigma_O$, as follows. Let $\sigma = (x_1, y_1) \cdots (x_k, y_k) \in \Sigma^*$ be a word received by Algorithm 1. Then we let $E_\varphi^*(\sigma) = (x_1', y_1') \cdots (x_k', y_k')$, where $(x_t', y_t')$ is the pair of events output by Algorithm 1 in Step 17, for $t = 1, \ldots, k$.

Theorem 4.2 (Correctness). *Given any safety property $\varphi$ defined as DTA $\mathcal{A}_\varphi$ that satisfies condition (**EnfCo**), the function $E_\varphi^*$ defined above is an enforcer for $\varphi$, that is, it satisfies (**Snd**), (**Tr**), (**Mono**), (**Inst**), and (**Ca**) constraints of Definition 3.1.*

*Example 4.3.* Consider property $P_1$ introduced in Example 2.2, defined as DTA presented in Figure 3, and its corresponding input DTA is presented in Figure 5. Table 1 illustrates an example input-output behavior of the algorithm. In Table 1, $t$ indicates tick, $x$ (resp. $y$) indicates input (resp. output) read by the algorithm, $x'$ (resp. $y'$) indicates input (resp. output) released by the algorithm. In the last column, $\text{fwd}_I$ (resp $\text{fwd}_O$) indicate that the input (resp. output) is simply forwarded, and $\text{edt}_I$ (resp. $\text{edt}_O$) indicate that the input (resp. output) is edited. The input-output word read by the

Table 1. Example illustrating Algorithm 1.

| t | $x$ | $x'$ | $y$ | $y'$ | $q$ | EnfAct |
|---|---|---|---|---|---|---|
| 0 | $\epsilon$ | $\epsilon$ | $\epsilon$ | $\epsilon$ | $(l_0, v_1 = 0)$ | - |
| 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 | $(l_0, v_1 = 3)$ | $\text{fwd}_I$, $\text{fwd}_O$ |

enforcer is $(0,1) \cdot (1,1) \cdot (1,0) \cdot (1,0) \cdot (0,1)$, and the input-output word released as output by the enforcer is $(0,0) \cdot (1,0) \cdot (0,0) \cdot (0,0) \cdot (0,1)$.

### 4.2 Comparison with the enforcement of TA

In this section, we compare the proposed enforcement based on DTA with dense timed automata TA [3]. First we provide the steps of obtaining a TA from the corresponding DTA. Let $WCRT$, the Worst Case Reaction Time (WCRT), be the worst case value of the tick length of Algorithm 1. Given any DTA $\varphi$, we create a TA $\varphi'$ as follows:

(1) 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 as in [3].
(2) Every transition guard of the form $v \bowtie C_{TICKS}$ where $C_{TICKS}$ is a discrete clock value is replaced by $v' \bowtie C$, where $C$ is the actual real-value. For example, $v1 \leq AVI_{TICKS}$, in the self loop on location $l1$ in Figure 4 is replaced by $v1' \leq AVI$. Here, $AVI_{TICKS}$ is an integer approximation of $AVI$, which can be obtained by $\lfloor \frac{AVI}{WCRT} \rfloor$ or $\lceil \frac{AVI}{WCRT} \rceil$.

Theorem 4.4 (Error Bound). *The maximum timing error while enforcing any DTA $\varphi$ relative to the TA $\varphi'$ is always less than WCRT i.e.* $\forall v \in V, v' \in V' \ \ |(v \times WCRT) - v'| < WCRT$.

Proof. The proof sketch is based on the following observations:

(1) Error never accumulates as clocks are initialised to zero before incrementing every tick from any non-accepting state.They are then reset when we reach an accepting state. This sequence must be followed while enforcing any $\varphi$. So, if the property is correctly enforced, this will hold.
(2) Here $v$ is a discrete clock and $v'$ is its dense counterpart. The maximum value of $v$ is $C_{TICKS} \in \mathbb{N}$ and that of $v'$ is $C \in \mathbb{R}$. Hence, the maximum real value corresponding to $v$ is $(C_{TICKS} \times WCRT)$. Thus the maximum error between the real values computed by the two clocks is $|(C_{TICKS} \times WCRT) - C|$. If $C_{TICKS} = \lfloor \frac{C}{WCRT} \rfloor$, then the maximum error will be $|(\lfloor \frac{C}{WCRT} \rfloor \times WCRT) - C|$, which is always less than $WCRT$. Likewise, if $C_{TICKS} = \lceil \frac{C}{WCRT} \rceil$, the maximum error will also be less than $WCRT$.

□

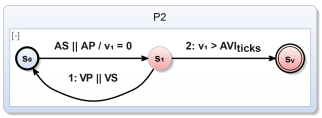
*Soundness of tick counting.* In the current context of pacemaker enforcement, the above result suffices to ensure the soundness of the DTA based discretisation. This is since the WCRT for the 33-node heart model [25], the enforcer, and the pacemaker is shown to be 1 ms (further elaborated in Section 5). 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.

*Complexity of DTA and TA based enforcement.* The complexity of DTA-based enforcement is $O(|\mathcal{A}_\varphi| \times C_1 \times C_2 \times \cdots \times C_n)$, where $C_1, C_2, \ldots, C_n$ represent the maximum values of the clocks

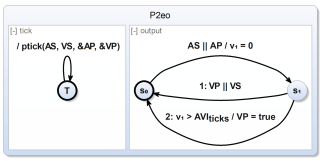
$v_1, v_2, \ldots, v_n$. while the complexity of TA enforcement has been shown to be PSPACE-complete due to the need for reachability computation [11, 19]. We, however, observe that complexity reduction is not the primary motivation of the current work. While we envisage, based on [7], that our approach may scale better than dense TA, an objective comparison between the dense and discrete time variants is considered as future work.

## 5 IMPLEMENTATION AND RESULTS

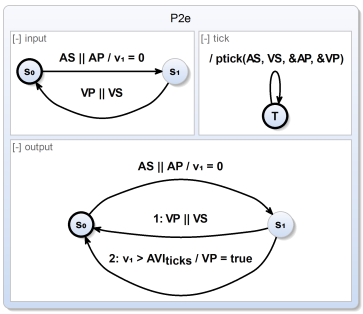
In order to evaluate the proposed enforcement approach, Algorithm 1 was implemented in the open-source SCCharts framework[4], a Statechart dialect designed for safety-critical systems. The generated enforcer was executed alongside multiple heart and pacemaker models in order to measure the overhead associated with enforcement.



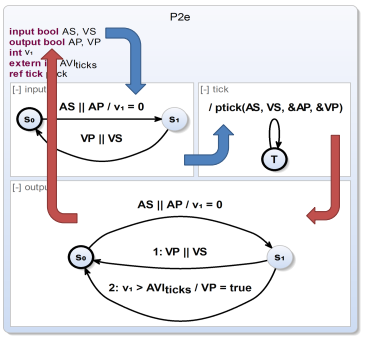

(a) $P_2$ safety property in SCCharts




(c) Autom. generated enforcer (opt.)




(b) Autom. generated enforcer (unopt.)




(d) Runtime scheduling path

Fig. 7. Example safety automaton of the $P_2$ example in SCCharts (a), its automatically generated Enforcer (b), and the optimized version of the enforcer (c). The runtime scheduling is depicted in (d).

| 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 ∧ P3 | 96 | 5480.7 | 4.61 % | 619.83 | 25.02 % |

Table 2. Results of the enforcer case-study

Applying Algorithm 1 allows us to transform property $P_2$ into an SCChart representation of the enforcer $E_\varphi$, as illustrated in Figure 7b. The generated enforcer has three concurrent regions (as in Section 4.1) — one for reading and editing the inputs, one for invoking the tick function ptick, and a final one for processing and emitting the outputs. The rand-edit$O_\varphi$ function can be seen in the output region, where the *VP* signal would be modified if $AVI_{TICKS}$ ticks have passed without any other ventricular event. As $P_2$ is such that the input region does not modify any variables, it can be omitted during an optimization phase, creating an enforcer as shown in Figure 7c.

[4] https://rtsys.informatik.uni-kiel.de/kieler

The scheduling of execution for the model can be seen in Figure 7d. The inputs (*AS* and *VS*) from the heart initially pass through the input region of the enforcer where they may be modified depending on rand-edit$\mathsf{I}_{\varphi_1}$. The transformed inputs are then passed through to the pacemaker via the ptick *effect* in the tick region. Subsequently, the outputs from the pacemaker (*AP* and *VP*) may be transformed in the output region as described by rand-edit$\mathsf{O}_{\varphi}$, before being returned back to the heart.

In order to evaluate the effect of enforcement on the execution time, the enforcers were synthesised for all properties $P_1$ through $P_5$. We also synthesized an enforcer for $P_2 \wedge P_3$. 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. A total of 10 separate trials were performed for each example, with their average execution time taken as the mean of all trials. For each of these tests, the pacemaker under test is a naïve implementation of the Boston Scientific pacemaker specification and University of Pennsylvania (UPenn) TA model in [16].

Table 2 shows the results of these executions for both the un-enforced (None) and enforced (P1 – P5 and P2 ∧ P3) models. In addition, the total Lines of Code (LoC) of the generated code is shown. Here, we can see that the addition of a single enforcer incurs minimal overhead when executed alongside an already complex model of the human heart, such as that of [25]. In such a scenario, overheads range from between 0.67 % to 1.42 % for single enforcers. Additionally, in order to illustrate the ability of our approach to easily test different models, a further experiment was completed using a much simpler model from [16] named the Random Heart Model. Alongside this simpler model we can see faster execution speeds but similar absolute overheads associated with enforcement, and hence higher percentage overheads. For the combined enforcement of $P_2 \wedge P_3$, the LoC increase to 96 from 32 for both $P_2$ and $P_3$, while the timing overhead increases to 4.61 %.

Additionally, WCRT analysis revealed a value of 583 $\mu$s for the non-enforcement case, a value which changed mostly negligibly (less than ~ 0.5 %) with the addition of enforcers. It is of note that due to the black box assumption for the heart and pacemaker models, WCRT analysis must use a Monte Carlo simulation approach, similar to the tests for execution time. This approach is the reason for the prior statements in Section 4.2 using a value of 1 ms.

## 6 RELATED WORK

Online verification of CPS is emerging as a promising avenue to tackle the decidability and complexity issues associated with the static verification. A mechanism for assuring safety properties of CPS via online verification is proposed in [8]. At every time step, the model of the hybrid system is updated/generated online based on the values of observable system parameters, and the reachable state-space for the next time-step is checked. If any unsafe state is reachable, then the system is stopped. We envisage that for safety critical systems, which are reactive, stopping the system may not be feasible and hence consider RE as an alternative paradigm.

Several RE models have been proposed such as Security automata [20] that focus on enforcement of safety properties, where the enforcer blocks the execution when it recognizes a sequence of actions that doses not satisfy the desired property. Edit automata [18] allows the enforcer to correct the input sequence by suppressing and/or inserting events, and the RE mechanisms proposed in [12] 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 been studied [11, 19]. These approaches focus on uni-directional RE. Mandatory Result Automata (MRA) [10] extended edit-automata [18], by considering bi-directional runtime enforcement, where the focus is on handling communication between two parties. However none of the above approaches are

suitable for reactive systems since halting the program and delaying actions is not suitable. This is because for reactive systems the enforcer has to react instantaneously.

Our work is closely related to [6], which introduces a framework to synthesize enforcers for reactive systems, called *shields*, from a set of safety properties. In [6] untimed properties are considered where properties are expressed as automata. In our work we consider timed properties expressed as DTA. Moreover, in [6], the shield is uni-directional, where it observes inputs from the plant and outputs from the controller, and transforms erroneous outputs. In our work, we consider bi-directional enforcement, as explained and illustrated in Fig. 1.

Our work is inspired by the concept of synchronous observers [13], which express safety properties using concurrent threads in a synchronous program. Bounded liveness properties can be expressed using the concept of *tick counting*. These safety properties are then statically verified using model checking. We extend the concept of synchronous observers to the run-time setting by introducing the problem of synchronous run-time enforcement for the first time. While observers are specified by the designer, enforcers are synthesized based on a set of regular policies expressed as DTAs. Repair of reactive programs w.r.t a specification [22] deals with *white-box* programs, synthesizing a repaired program close to the faulty program. Contrary to [22], in our work, we consider the system (synchronous program) to be a black-box, and we focus on synthesis of enforcers from properties.

## 7 CONCLUSION AND FUTURE WORK

There have been wide spread recalls of implantable pacemakers and ICD devices due to safety issues [2]. Considering this, formal methods based solutions have emerged. Most approaches consider hybrid / timed automata models of the cardiac conduction system, which is composed in closed-loop with timed automata models of the pacemaker. Such systems are either analysed through model checking, and when this is not feasible due to model complexity, model-based testing is considered. In both cases, the heart is considered a static system. This is in stark contrast to a real heart that is a highly dynamical system, where arrhythmia is a random event. This paper formalises the runtime enforcement problem, particularly considering a dynamically evolving heart. We formalise the runtime enforcement problem for cyber physical systems (CPS) as a bi-directional runtime enforcement of reactive systems, for the first time. Moreover, we propose the use of discrete timed automata (DTA) in place of timed automata to express properties to be enforced. DTA express regular properties while earlier work on RE of reactive systems considered a subset. We show that the error due to the proposed discretisation is bounded and well within the required tolerance value of the pacemaker. We developed an on-line algorithm, and implemented it in the SCCharts tool-chain. Our 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. In addition, from a practical standpoint, we will consider implementing the enforcer as an external device similar to [23]. We also plan to study the use of RE as a certification aid, and will also consider personalised policies for enforcement and will develop the pacemaker case-study fully with input from physicians. This may involve the introduction of additional features in the framework such as compositionality of the enforcers and priority between properties.

## APPENDIX: PROOFS

Proof of Theorem 3.5. Let us recall Theorem 3.5. Consider a property $\varphi$ defined as DTA $\mathcal{A}_\varphi = (L, l_0, l_v, \Sigma, V, \Delta, F)$ with semantics $[\![\mathcal{A}_\varphi]\!] = (Q, q_0, \Sigma, \rightarrow, Q_F, q_v)$. Property $\varphi$ is enforceable iff the condition (**EnfCo**) holds which is the following condition:

$$\forall q \in Q, q \notin q_v \implies \exists \sigma \in \Sigma^+ : q \xrightarrow{\sigma} q' \wedge q' \in Q_F.$$

DTA $\mathcal{A}_\varphi$ is deterministic and complete (Definition 2.4). From Remark 1, let us recall that $L$ contains only locations that are reachable from the initial location $l_0$. In $[\![\mathcal{A}_\varphi]\!]$, We consider that $Q$ contains only states that are reachable from $q_0$. If a state $q \in Q$ that is reachable from $q_0$ is non-accepting and non-trap (i.e., $q \notin \{Q_F \cup q_v\}$), then we consider that there is a path from $q$ to an accepting state in $Q_F$. State $q$ is merged with $q_v$ otherwise.

We prove that:

- *Sufficient:* If condition (**EnfCo**) holds then $E_\varphi$ according to Definition 3.1 exists.
  Due to condition (**EnfCo**), whatever may be the current state $q \in Q \setminus \{q_v\}$ of the property $\mathcal{A}_\varphi$ being enforced, there is at least one possibility to correct (edit) the event that the enforcer receives when in state $q$ (in case if the received event leads to a state from where only states in $q_v$ are reachable). That is, due to condition (**EnfCo**), $\forall q \in Q \setminus \{q_v\}$, we know for sure that $\text{editI}_{\mathcal{A}_{\varphi_\text{I}}}(q)$ will be non-empty (i.e., $\exists x \in \Sigma_I, \exists \sigma_I \in \Sigma_I^* : q \xrightarrow{x \cdot \sigma_I}_I q' \wedge q' \in Q_{F_I}$). Also, $\forall q \in Q \setminus \{q_v\}, \forall x \in \text{editI}_{\mathcal{A}_{\varphi_\text{I}}}(q) : \text{editO}_{\mathcal{A}_\varphi}(q,x)$ will be also non-empty (i.e., $\exists y \in \Sigma_O, \exists \sigma \in \Sigma^* : q \xrightarrow{(x,y)\cdot\sigma} q' \wedge q' \in Q_F$).
  For any property $\varphi$ defined as DTA $\mathcal{A}_\varphi$, the enforcement function $E^*_\varphi$ (Definition 4.1) is an enforcer for $\varphi$ which satisfies all the constraints according to Definition 3.1. Theorem 4.2 shows that for any property $\varphi$ (defined as DTA $\mathcal{A}_\varphi$) that satisfies the condition for enforceability (**EnfCo**), the enforcement function $E^*_\varphi$ (Definition 4.1) is an enforcer for $\varphi$, that is, it satisfies (**Snd**), (**Tr**), (**Mono**), (**Inst**), and (**Ca**) constraints of Definition 3.1.
- *Necessary:* If $E_\varphi$ according to Definition 3.1 exists, then condition (**EnfCo**) holds.
  Suppose that an enforcer $E_\varphi$ for $\varphi$ according to Definition 3.1 exists and assume that condition (**EnfCo**) does not hold for $\mathcal{A}_\varphi$.

  Since condition (**EnfCo**) does not hold, $\exists q \in Q \setminus \{q_v\} : \forall (x,y) \in \Sigma, \forall \sigma \in \Sigma^*, q \xrightarrow{(x,y)\cdot\sigma} q' \wedge q' \in Q \setminus Q_F$, i.e., there exists a state $q \in Q \setminus \{q_v\}$ such that there is no path that leads to an accepting state in $Q_F$ from $q$.

  Since $Q$ contains only states that are reachable from $q_0$, $\exists \sigma \in \Sigma^* : q_0 \xrightarrow{\sigma} q$, i.e., there certainly exists a word $\sigma \in \Sigma^*$ that leads to the problematic state $q$ from the initial state $q_0$, where $q$ is the initial state $q_0$ itself, or an accepting state (i.e., $q \in Q_F$), or $q$ is a non-violating state from which an accepting state is reachable (i.e., $q \in Q \setminus Q_F \cup q_0$).
  - *$q$ is the initial state $q_0$*:
    In this case, $\sigma = \epsilon$ and there is no path from the initial state $q_0$ to an accepting state in $q_v$. Upon receiving any event as input, the edit functions will be empty and the enforcer cannot produce any event as output. If the enforcer outputs some event, then soundness constraint will be violated, and not releasing any event will violate instantaneity and causality constraints.
  - $q \in Q \setminus Q_F \cup q_0$:
    In this case, $q$ is an non-accepting and non-violating state. When $q$ is non-accepting and non-violating (i.e., $q \in Q \setminus Q_F \cup q_0$), since we consider that an accepting state is reachable from such states (a non-accepting state is considered to be merged with $q_v$ if an accepting state is not reachable form it), our assumption is false and condition (**EnfCo**) holds in this case.
  - $q \in Q_F$:
    There is a word $\sigma \in \Sigma^*$ that leads to state $q \in Q_F$. If that word $\sigma$ is the input word to the enforcer, then due to constraint (**Tr**), it cannot edit any event in $\sigma$, and the enforcer produces $\sigma$ as output and reaches state $q$.

When in state $q$, upon receiving any event $(x,y) \in \Sigma$, the enforcer has no possibility to correct it, since there is no path from $q$ to an accepting state (i.e., since $\forall \sigma' \in \Sigma^* : q \xrightarrow{\sigma'} q_v$). Thus, $\forall (x,y) \in \Sigma$, when the input word given to the enforcer is $\sigma \cdot (x,y)$, the enforcer cannot produce any event as output since $\text{editI}_{\mathcal{A}_{\varphi_I}}()$ and $\text{editO}_{\mathcal{A}_\varphi}()$ from location $q$ will be empty, violating constraints (**Inst**) and (**Ca**).

Thus, our assumption is false and condition (**EnfCo**) holds for $\mathcal{A}_\varphi$.

□

Before we discuss proof of Theorem 4.2, we introduce the following Lemma that is useful in proving Theorem 4.2.

Lemma 1.1. *Let $\mathcal{A}_{\varphi_I} = (L, l_0, l_v, \Sigma_I, V, \Delta_I, F)$ be the input DTA obtained from $\mathcal{A}_\varphi = (L, l_0, l_v, \Sigma, V, \Delta, F)$ according to Definition 2.7.*

*We have the following properties:*

1 $\forall (x,y) \in \Sigma, \forall l, l' \in L : (l,g,r,(x,y),l') \in \Delta \Rightarrow (l,g,r,x,l') \in \Delta_I$ .

2 $\forall x \in \Sigma_I, \forall l, l' \in L : (l,g,r,x,l') \in \Delta_I \Rightarrow \exists y \in \Sigma_O : (l,g,r,(x,y),l') \in \Delta$ .

Intuitively, property 1 of Lemma 1.1 states that if there is a transition from location $l \in L$ to location $l' \in L$ upon input-output event $(x,y) \in \Sigma$ in $\mathcal{A}_\varphi$, then there is also a transition from location $l$ to location $l'$ in the input automaton $\mathcal{A}_{\varphi_I}$ upon the input event $x \in \Sigma_I$. Property 2 of Lemma 1.1 states that if there is a transition from location $l \in L$ to location $l' \in Q$ upon input event $x \in \Sigma_I$, then there certainly exists an output event $y \in \Sigma_O$ s.t. there is a transition from location $l$ to location $l'$ upon event $(x,y)$ in the automaton $\mathcal{A}_\varphi$. Lemma 1.1 immediately follows from Definitions 2.1 and 2.7.

Proof of Theorem 4.2. Let us recall the condition for enforceability:

A property $\varphi$ that is defined as DTA $\mathcal{A}_\varphi = (L, l_0, l_v, \Sigma, V, \Delta, F)$, with semantics $[\![\mathcal{A}_\varphi]\!] = (Q, q_0, \Sigma, \rightarrow, Q_F, q_v)$ is enforceable iff the following condition holds:

$$\forall q \in Q, q \notin q_v \implies \exists \sigma \in \Sigma^+ : q \xrightarrow{\sigma} q' \wedge q' \in Q_F.$$

Let us also recall the definition of function $E^*_\varphi : \Sigma^* \rightarrow \Sigma^*$ (Definition 4.1). Let $\sigma = (x_1,y_1)\cdots(x_k,y_k) \in \Sigma^*$ be a word received by Algorithm 1. Then we let $E^*_\varphi(\sigma) = (x'_1,y'_1)\cdots(x'_k,y'_k)$, where $(x'_t,y'_t)$ is the pair of events output by Algorithm 1 in Step 17, for $t = 1,\ldots,k$.

Note that input DTA $\mathcal{A}_{\varphi_I} = (L, l_0, l_v, \Sigma_I, V, \Delta_I, F)$ (with semantics $[\![\mathcal{A}_{\varphi I}]\!] = (Q, q_0, \Sigma_I, \rightarrow_I, Q_F, q_v)$) is obtained from $\mathcal{A}_\varphi = (L, l_0, l_v, \Sigma, V, \Delta, F)$ (with semantics $[\![\mathcal{A}_\varphi]\!] = (Q, q_0, \Sigma, \rightarrow, Q_F, q_v)$) by projecting on inputs (See Definition 2.7, Section 2).

We shall prove that given any property $\varphi$ defined as DTA $\mathcal{A}_\varphi$ that satisfies condition (**EnfCo**), the function $E^*_\varphi$ is an enforcer for $\varphi$, that is, it satisfies (**Snd**), (**Tr**), (**Mono**), (**Inst**), and (**Ca**) constraints of Definition 3.1.

Let us prove this theorem using induction on the length of the input sequence $\sigma \in \Sigma^*$ (which also corresponds to the number of ticks ($t$ in Algorithm 1).

*Induction basis.* Theorem 4.2 holds trivially for $\sigma = \epsilon$ since the algorithm will not release any input-output event as output and thus $E^*_\varphi(\epsilon) = \epsilon$.

*Induction step.* Assume that for every $\sigma = (x_1,y_1)\cdots(x_k,y_k) \in \Sigma^*$ of some length $k \in \mathbb{N}$, let $E^*_\varphi(\sigma) = (x'_1,y'_1)\cdots(x'_k,y'_k) \in \Sigma^*$, for $t = 1,\ldots,k$, and Theorem 4.2 holds for $\sigma$, i.e., $E^*_\varphi(\sigma)$ satisfies the (**Snd**), (**Tr**), (**Mono**), (**Inst**), and (**Ca**) constraints. Let $q \in Q \setminus \{q_v\}$ be the current state of both the automata $\mathcal{A}_\varphi$ and $\mathcal{A}_{\varphi_I}$ after processing input $\sigma$ of length $k$, i.e., $q$ corresponds to the state that we reach upon $E^*_\varphi(\sigma)$ in $[\![\mathcal{A}_\varphi]\!]$, and the state that we reach in $[\![\mathcal{A}_{\varphi_I}]\!]$ upon $E^*_\varphi(\sigma)_I$. Note that the

current state $q$ in Algorithm 1 can never belong to the set of trap states $q_v$ ($q$ is initialized to $q_0$ and it is updated in step 18 to a state $q' \in Q \setminus \{q_v\}$).

We now prove that for any event $(x_{k+1}, y_{k+1}) \in \Sigma$, Theorem 4.2 holds for $\sigma \cdot (x_{k+1}, y_{k+1})$, where $x_{k+1} \in \Sigma_I$ is the input event read by Algorithm 1, and $y_{k+1} \in \Sigma_O$ is the output event read by Algorithm 1 in $k+1^{th}$ iteration (i.e., when $t = k+1$).

We have the following two possible cases based on whether there exists a word $\sigma' \in \Sigma^*$ s.t. an accepting state is reachable in $[\![\mathcal{A}_\varphi]\!]$ from $q$ upon $(x_{k+1}, y_{k+1}) \cdot \sigma'$.

- $\exists \sigma' \in \Sigma^* : q \xrightarrow{(x_{k+1}, y_{k+1}) \cdot \sigma'} q' \wedge q' \in Q_F$.
  In Algorithm 1, the condition tested in step 5 will evaluate to true since from Lemma 1.1, in $[\![\mathcal{A}_{\varphi_I}]\!]$ we will have $q \xrightarrow{x_{k+1} \cdot \sigma'_I}_I q' \wedge q' \in Q_{F_I}$, and thus $x'_{k+1} = x_{k+1}$.
  Also, the condition tested in step 12 will evaluate to true in this case since $\exists \sigma' \in \Sigma^* : q \xrightarrow{(x_{k+1}, y_{k+1}) \cdot \sigma'} q' \wedge q' \in Q_F$, and thus $y'_{k+1} = y_{k+1}$. At the end of the $k+1^{th}$ iteration, the input-output event released as output by the algorithm in step 17 is $(x_{k+1}, y_{k+1})$. The output of the algorithm after completing the $k+1^{th}$ iteration is $E^*_\varphi(\sigma \cdot (x_{k+1}, y_{k+1})) = E^*_\varphi(\sigma) \cdot (x_{k+1}, y_{k+1})$.
  Regarding constraint (**Snd**), in this case, what has been already released as output by the algorithm earlier before reading event $(x_{k+1}, y_{k+1})$ (i.e., $E^*_\varphi(\sigma)$) followed by the new input-output event released as output $(x_{k+1}, y_{k+1})$ can be extended to a sequence that satisfies $\varphi$ (since $q$ is the state reached upon $E^*_\varphi(\sigma)$, and there exists a word $\sigma'$ s.t. an accepting state is reachable from $q$ upon $(x_{k+1}, y_{k+1}) \cdot \sigma'$ ). Thus constraint (**Snd**) holds in this case.
  Regarding constraint (**Mono**), it holds since $\sigma \preccurlyeq \sigma \cdot (x_{k+1}, y_{k+1})$ and also $E^*_\varphi(\sigma) \preccurlyeq E^*_\varphi(\sigma) \cdot (x_{k+1}, y_{k+1})$.
  Regarding constraint (**Inst**) from the induction hypothesis, we have for $\sigma$ of some length $k$, $|\sigma| = |E_\varphi(\sigma)|$. We also have $E^*_\varphi(\sigma \cdot (x_{k+1}, y_{k+1})) = E^*_\varphi(\sigma) \cdot (x_{k+1}, y_{k+1})$. Thus, $|\sigma \cdot (x_{k+1}, y_{k+1})| = |E^*_\varphi(\sigma \cdot (x_{k+1}, y_{k+1}))| = k+1$, and constraint (**Inst**) holds. Constraint (**Tr**) holds in this case since the output of the enforcer before reading $(x_{k+1}, y_{k+1})$ i.e., $E^*_\varphi(\sigma)$ followed by the new input-output event read $(x_{k+1}, y_{k+1})$ satisfies the property $\varphi$ and we already saw that the output event released by the algorithm after reading $(x_{k+1}, y_{k+1})$ is $E^*_\varphi(\sigma) \cdot (x_{k+1}, y_{k+1})$.
  Regarding constraint (**Ca**), in this case from the induction hypothesis, from the definitions of $\mathsf{editI}_{\mathcal{A}_{\varphi_I}}$ and $\mathsf{editO}_{\mathcal{A}_\varphi}$ we have $x_{k+1} \in \mathsf{editI}_{\mathcal{A}_{\varphi_I}}(q)$, and also $y_{k+1} \in \mathsf{editO}_{\mathcal{A}_\varphi}(q, x_{k+1})$.
  Theorem 4.2 thus holds for $\sigma \cdot (x_{k+1}, y_{k+1})$ in this case.
- $\nexists \sigma' \in \Sigma^* : q \xrightarrow{(x_{k+1}, y_{k+1}) \cdot \sigma'} q' \wedge q' \in Q_F$.
  In this case, it is not possible to reach an accepting state in $[\![\mathcal{A}_\varphi]\!]$ from the current state $q$ upon the new event $(x_{k+1}, y_{k+1})$ followed by any extension it.
  In this case, we have two sub-cases, based on whether $\exists \sigma'_I \in \Sigma^*_I : q \xrightarrow{x_{k+1} \cdot \sigma'_I}_I q' \wedge q' \in Q_F$.
  - $\exists \sigma'_I \in \Sigma^*_I : q \xrightarrow{x_{k+1} \cdot \sigma'_I}_I q' \wedge q' \in Q_F$.
    In Algorithm 1, the condition tested in step 5 will evaluate to true and thus $x'_{k+1} = x_{k+1}$.
    In this case, the condition tested in step 12 will evaluate to false since $\nexists \sigma' \in \Sigma^* : q \xrightarrow{(x_{k+1}, y_{k+1}) \cdot \sigma'} q' \wedge q' \in Q_F$. $y'_{k+1}$ will thus be an element belonging to the set $\mathsf{editO}_{\mathcal{A}_\varphi}(q, x_{k+1})$ if $\mathsf{editO}_{\mathcal{A}_\varphi}(q, x_{k+1})$ is non-empty.
    It is important to notice that $\mathsf{editO}_{\mathcal{A}_\varphi}(q, x_{k+1})$ will be non-empty in this case since we know for sure that $\exists y'_{k+1} \in \Sigma_O, q' \in Q, \sigma' \in \Sigma^* : q \xrightarrow{(x_{k+1}, y'_{k+1}) \cdot \sigma'} q' \wedge q' \in Q_F$ (from the condition for enforceability (**EnfCo**), hypothesis ($q \notin q_v$), definition of $\mathsf{editO}_{\mathcal{A}_\varphi}$, and Lemma 1.1). Thus $y'_{k+1}$

is an element belonging to $\text{editO}_{\mathcal{A}_\varphi}(q, x_{k+1})$. The output of the algorithm after completing the $k+1^{th}$ iteration is $E^*_\varphi(\sigma \cdot (x_{k+1}, y_{k+1})) = E^*_\varphi(\sigma) \cdot (x_{k+1}, y'_{k+1})$.
Regarding constraint (**Snd**), from the definition of $\text{editO}_{\mathcal{A}_\varphi}$, we know that $E^*_\varphi(\sigma)$ followed by the new input-output event released as output $(x_{k+1}, y'_{k+1})$ can be extended to satisfy property $\varphi$ (i.e., $\exists \sigma' \in \Sigma^* : E^*_\varphi(\sigma) \cdot (x_{k+1}, y'_{k+1}) \cdot \sigma' \models \varphi$), and thus constraint (**Snd**) holds.
The reasoning for constraints (**Mono**) and (**Inst**) are similar to the previous cases since we saw that Algorithm 1 releases a new event $(x_{k+1}, y'_{k+1})$ as output after reading event $(x_{k+1}, y_{k+1})$ after completing $k+1^{th}$ iteration.
Constraint (**Tr**) holds trivially in this case since $E^*_\varphi(\sigma) \cdot (x_{k+1}, y_{k+1}) \not\models \varphi$.
Regarding constraint (**Ca**), in this case from the induction hypothesis, from the definitions of $\text{editI}_{\mathcal{A}_{\varphi_\text{I}}}$ we have $x_{k+1} \in \text{editI}_{\mathcal{A}_{\varphi_\text{I}}}(q)$, and we already discussed that $\text{editO}_{\mathcal{A}_\varphi}(q, x_{k+1})$ will be non-empty and thus constraint (**Ca**) holds in this case.

- $\nexists \sigma'_I \in \Sigma_I^* : q \xrightarrow{x_{k+1} \cdot \sigma'_I}_I q' \wedge q' \in Q_F$.
In Algorithm 1, the condition tested in step 5 will evaluate to false in this case. It is important to notice that $\text{editI}_{\mathcal{A}_{\varphi_\text{I}}}(q)$ will be non-empty since from the condition for enforceability and Lemma 1.1, we know for sure that $\exists x'_{k+1} \in \Sigma_I, q' \in Q, \sigma'_I \in \Sigma_I : q \xrightarrow{x'_{k+1} \cdot \sigma'_I} q' \wedge q' \in Q_F$ in $[\![\mathcal{A}_{\varphi_I}]\!]$. Thus, $x'_{k+1}$ will be an element belonging to $\text{editI}_{\mathcal{A}_{\varphi_\text{I}}}(q)$.
We have two sub-cases based on whether
$\exists \sigma' \in \Sigma^* : q \xrightarrow{(x'_{k+1}, y_{k+1}) \cdot \sigma'} q' \wedge q' \in Q_F$ in $[\![\mathcal{A}_\varphi]\!]$ or not.
  - $\exists \sigma' \in \Sigma^* : q \xrightarrow{(x'_{k+1}, y_{k+1}) \cdot \sigma'} q' \wedge q' \in Q_F$ in $[\![\mathcal{A}_\varphi]\!]$.
  In Algorithm 1, the condition tested in step 12 will evaluate to true in this case. Thus, $y'_{k+1} = y_{k+1}$ in this case and the event released as output by the algorithm at the end of $k+1^{th}$ iteration is $(x'_{k+1}, y_{k+1})$. We have $E^*_\varphi(\sigma \cdot (x_{k+1}, y_{k+1})) = E^*_\varphi(\sigma) \cdot (x'_{k+1}, y_{k+1})$.
  Regarding constraint( **Snd**), from the condition of this case (i.e., $\exists \sigma' \in \Sigma^* : q \xrightarrow{(x'_{k+1}, y_{k+1}) \cdot \sigma'} q' \wedge q' \in Q_F$), we know that $E^*_\varphi(\sigma)$ followed by the new input-output event released as output $(x'_{k+1}, y_{k+1})$ can be extended to satisfy the property $\varphi$, and thus constraint (**Snd**) holds.
  The reasoning for constraints (**Mono**) and (**Inst**) are similar to the previous cases since we saw that the algorithm releases a new event $(x'_{k+1}, y_{k+1})$ as output after reading event $(x_{k+1}, y_{k+1})$ at the end of $k+1^{th}$ iteration.
  Constraint (**Tr**) holds trivially in this case since $E^*_\varphi(\sigma) \cdot (x_{k+1}, y_{k+1}) \not\models \varphi$.
  Regarding constraint (**Ca**), we already discussed that $\text{editI}_{\mathcal{A}_{\varphi_\text{I}}}(q)$ is non-empty and $x'_{k+1} \in \text{editI}_{\mathcal{A}_{\varphi_\text{I}}}(q)$, and $y_{k+1} \in \text{editO}_{\mathcal{A}_\varphi}(q, x'_{k+1})$ from the condition of this case and definitions of $\text{editI}_{\mathcal{A}_{\varphi_\text{I}}}$ and $\text{editO}_{\mathcal{A}_\varphi}$.
  - $\nexists \sigma' \in \Sigma^* : q \xrightarrow{(x'_{k+1}, y_{k+1}) \cdot \sigma'} q' \wedge q' \in Q_F$ in $[\![\mathcal{A}_\varphi]\!]$.
  In the algorithm, the condition tested in step 12 will evaluate to false in this case. $y'_{k+1}$ will thus be an element belonging to the set $\text{editO}_{\mathcal{A}_\varphi}(q, x'_{k+1})$ if $\text{editO}_{\mathcal{A}_\varphi}(q, x'_{k+1})$ is non-empty. Note that $\text{editO}_{\mathcal{A}_\varphi}(q, x'_{k+1})$ will be non-empty in this case since we know for sure that $\exists y'_{k+1} \in \Sigma_O, \sigma' \in \Sigma^* : q \xrightarrow{(x'_{k+1}, y'_{k+1}) \cdot \sigma'} q' \wedge q' \in Q_F$ in $[\![\mathcal{A}_\varphi]\!]$ (from the enforceability condition, definitions, and Lemma 1.1). Thus $y'_{k+1}$ is an element belonging to $\text{editO}_{\mathcal{A}_\varphi}(q, x'_{k+1})$. The output of the algorithm after completing the $k+1^{th}$ iteration is $E^*_\varphi(\sigma \cdot (x_{k+1}, y_{k+1})) =$

$E^*_\varphi(\sigma)\cdot(x'_{k+1},y'_{k+1})$ where $x'_{k+1}$ is an element belonging to $\text{editI}_{\mathcal{A}_{\varphi_I}}(q_I)$ and $y'_{k+1}$ is an element belonging to $\text{editO}_{\mathcal{A}_\varphi}(q,x'_{k+1})$.

Regarding constraint (**Snd**), from the definitions of $\text{editI}_{\mathcal{A}_{\varphi_I}}$ and $\text{editO}_{\mathcal{A}_\varphi}$, we know that $E_\varphi(\sigma)\cdot(x'_{k+1},y'_{k+1})$ can be extended to a sequence that satisfies the property $\varphi$ and thus constraint (**Snd**) holds. The reasoning for constraints (**Mono**) and (**Inst**) are similar to the previous cases since we saw that the algorithm releases a new event $(x'_{k+1},y'_{k+1})$ as output after reading event $(x_{k+1},y_{k+1})$. In this case, constraint (**Tr**) holds trivially since $E_\varphi(\sigma)\cdot(x_{k+1},y_{k+1})\not\models\varphi$. Regarding constraint (**Ca**), we already discussed that $\text{editI}_{\mathcal{A}_{\varphi_I}}(q)$ is non-empty and $x'_{k+1}\in\text{editI}_{\mathcal{A}_{\varphi_I}}(q)$, and also that $\text{editO}_{\mathcal{A}_\varphi}(q,x'_{k+1})$ is non-empty and $y'_{k+1}\in\text{editO}_{\mathcal{A}_\varphi}(q,x'_{k+1})$ and thus constraint (**Ca**) holds.

Theorem 4.2 thus holds for $\sigma\cdot(x_{k+1},y_{k+1})$ in this case.

Thus Theorem 4.2 holds for $\sigma\cdot(x_{k+1},y_{k+1})$. □

## 2 APPENDIX: FUNCTIONAL DEFINITION

Recall that an input-output event (reaction) is a pair $(x,y)$, where $x\in\Sigma_I$ is the input and $y\in\Sigma_O$ is the output. Upon consuming an input-output event $(x,y)$ as input, the enforcer produces an input-output event $(x',y')$ as output immediately.

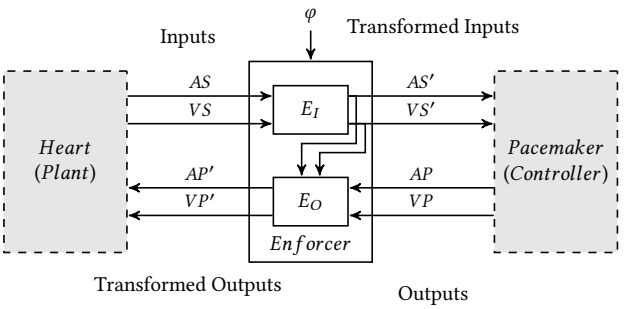



Fig. 8. Enforcement function

The enforcer first processes the input $x$ (i.e., input from the environment) and produces transformed input $x'$, and later it processes the output $y$ (i.e., output produced by the program), to finally produce the transformed input-output event $(x',y')$. As illustrated in Figure 8, the enforcement function $E_\varphi$ is defined by composing two functions $E_I$ and $E_O$, where function $E_I$ reads input from the environment $x$ (ignoring output) and produces transformed input $x'$, and $E_O$ reads transformed input $x'$ (output of $E_I$) and output $y$ (output produced by the program upon input $x'$) and produces/adds transformed input-output event $(x',y')$ to the output of the enforcer.

*Definition 2.1 (Enforcement function).* Given property $\varphi\subseteq\Sigma^*$ defined as DTA $\mathcal{A}_\varphi$, that we want to enforce, the enforcement function $E_\varphi:\Sigma^*\to\Sigma^*$ is defined as:

$$E_\varphi(\sigma) \quad = E_O(E_I(\sigma_I),\sigma_O)$$

where:

$E_I:\Sigma^*_I\to\Sigma^*_I$ is defined as:

$$E_I(\epsilon_{\Sigma_I}) = \epsilon_{\Sigma_I}$$
$$E_I(\sigma_I \cdot x) = \begin{cases} E_I(\sigma_I) \cdot x & \text{if } \exists \sigma'_I \in \Sigma_I^* : E_I(\sigma_I) \cdot x \cdot \sigma'_I \models \varphi_I, \\ E_I(\sigma_I) \cdot x' & \text{otherwise} \end{cases}$$

where $x' = \text{rand-editI}_{\mathcal{A}_{\varphi_I}}(q)$.

$E_O : \Sigma_I^* \times \Sigma_O^* \to (\Sigma_I \times \Sigma_O)^*$ is defined as:

$$E_O(\epsilon_{\Sigma_I}, \epsilon_{\Sigma_O}) = \epsilon_\Sigma$$
$$E_O(\sigma_I \cdot x, \sigma_O \cdot y) = \begin{cases} E_O(\sigma_I, \sigma_O) \cdot (x, y) & \text{if } \exists \sigma' \in \Sigma^* : E_O(\sigma_I, \sigma_O) \cdot (x, y) \cdot \sigma' \models \varphi, \\ E_O(\sigma_I, \sigma_O) \cdot (x, y') & \text{otherwise} \end{cases}$$

where $y' = \text{rand-editO}_{\mathcal{A}_\varphi}(q, x)$, and $q$ is the state reached in $[\![\mathcal{A}_\varphi]\!]$ upon $E_O(\sigma_I, \sigma_O)$[5].

Let us understand Definition 2.1 further. Function $E_\varphi$ takes a word over $\Sigma^*$ and returns a word over $\Sigma^*$ as output. For a word $\sigma \in \Sigma^*$, $\sigma_I \in \Sigma_I^*$ is the projection of $\sigma$ on inputs, and $\sigma_O \in \Sigma_O^*$ is the projection of $\sigma$ on outputs. Function $E_\varphi$ is defined using two functions $E_I$ and $E_O$, and the output of function $E_O$ is the output of the enforcement function $E_\varphi$.

*Function $E_I$.* For a given word $\sigma \in \Sigma^*$, function $E_I$ takes the word obtained by projecting on the inputs ($\sigma_I \in \Sigma_I^*$) as input and returns a word in $\Sigma_I^*$ as output. Function $E_I$ is defined inductively. It returns $\epsilon_{\Sigma_I}$ when the input $\sigma_I = \epsilon_{\Sigma_I}$. If $\sigma_I$ is read as input $E_I(\sigma_I)$ is returned as output, and when another new input $x \in \Sigma_I$ is observed, there are two possible cases based on whether $E_I(\sigma_I) \cdot x$ can be extended to a sequence that satisfies the property $\varphi_I$ or not.

- If $E_I(\sigma_I)$ followed by the new input $x$ can be extended to a sequence that satisfies the property $\varphi_I$ (property obtained from $\varphi$ by projecting on inputs) in the future, then the new input $x$ is appended to the previous output of function $E_I$ (i.e., $E_I(\sigma_I \cdot x) = E_I(\sigma_I) \cdot x$).
- If the previous case does not hold, then $E_I(\sigma_I) \cdot x$ cannot be extended to satisfy property $\varphi_I$ in the future (i.e., $\nexists \sigma'_I \in \Sigma_I^*$ such that $E_I(\sigma_I) \cdot x \cdot \sigma'_I \models \varphi_I$). Thus, input $x$ is edited using $\text{rand-editI}_{\mathcal{A}_{\varphi_I}}(q)$ to obtain transformed input $x'$, and this transformed input $x'$ is appended to the previous output of function $E_I$ (i.e., $E_I(\sigma_I \cdot x) = E_I(\sigma_I) \cdot x'$). $\text{rand-editI}_{\mathcal{A}_{\varphi_I}}(q)$ returns $x' \in \Sigma_I$, such that $E_I(\sigma_I) \cdot x'$ can be extended to a sequence that satisfies $\varphi_I$.

*Function $E_O$.* Function $E_O$ takes an input word belonging to $\Sigma_I^*$ and an output word belonging to $\Sigma_O^*$ as input, and it returns an input-output word belonging to $\Sigma^*$ which is a sequence of tuples, where each event contains an input and an output.

Function $E_O$ is defined inductively. When both input word and output word are empty, the output of $E_O$ is $\epsilon$. If $\sigma_I \in \Sigma_I^*$, and $\sigma_O \in \Sigma_O^*$ is read as input, its output will be $E_O(\sigma_I, \sigma_O)$, and when another new input event $x$ and output event $y$ are observed, there are two possible cases based on whether $E_O(\sigma_I, \sigma_O) \cdot (x, y)$ can be extended to a sequence that satisfies property $\varphi$ or not.

- If $E_O(\sigma_I, \sigma_O)$ followed by $(x, y)$ can be extended to a sequence that satisfies $\varphi$, then $(x, y)$ is appended to the previous output of function $E_O$ (i.e., $E_O(\sigma_I \cdot x, \sigma_O \cdot y) = E_O(\sigma_I, \sigma_O) \cdot (x, y)$).
- If the above case does not hold, then $E_O(\sigma_I, \sigma_O) \cdot (x, y)$ does not satisfy $\varphi$ and there is no extension $\sigma' \in \Sigma^*$ such that $E_O(\sigma_I, \sigma_O) \cdot (x, y) \cdot \sigma' \models \varphi$. Thus, output $y$ is edited using

[5]The automaton $\mathcal{A}_{\varphi_I}$ is created from $\mathcal{A}_\varphi$ by projecting on inputs (see Section 2.2.2) and therefore has an identical structure with the only difference being that outputs are ignored on the transitions of $\mathcal{A}_{\varphi_I}$. The set of states in $[\![\mathcal{A}_\varphi]\!]$ and $[\![\mathcal{A}_{\varphi_I}]\!]$ are the same. Note that if there exists a path to state $q$ from the initial state upon $E_O(\sigma_I, \sigma_O)$ in $[\![\mathcal{A}_\varphi]\!]$, then there also exists a path to state $q$ upon $E_O(\sigma_I, \sigma_O)_I$ (which is $E_I(\sigma_I)$) in $[\![\mathcal{A}_{\varphi_I}]\!]$.

rand-editO$_{\mathcal{A}_\varphi}(q,x)$ to obtain transformed output $y'$, and the event $(x,y')$ is appended to the previous output of the function $E_O$ (i.e., $E_O(\sigma_I \cdot x, \sigma_O \cdot y) = E_O(\sigma_I,\sigma_O) \cdot (x,y')$). Function rand-editO$_{\mathcal{A}_\varphi}(q,x)$ returns $y' \in \Sigma_O$ such that $E_O(\sigma_I,\sigma_O) \cdot (x,y')$ can be extended to a sequence that satisfies $\varphi$.

Regarding state $q$ in definitions $E_O$ and $E_I$, atate $q$ is the state reachable in $[\![\mathcal{A}_\varphi]\!]$ upon $E_O(\sigma_I,\sigma_O)$. $[\![\mathcal{A}_\varphi]\!]$ and $[\![\mathcal{A}_{\varphi_I}]\!]$ have the same set of states. In definition of $E_I$, $q$ correspond to the same state in $[\![\mathcal{A}_{\varphi_I}]\!]$, which is a state reachable in $[\![\mathcal{A}_{\varphi_I}]\!]$ upon $E_I(\sigma_I)$.

*Definition 2.2 (Optimal enforcement function).* Given property $\varphi \subseteq \Sigma^*$, that we want to enforce, the *optimal* enforcement function $E_\varphi$-opt : $\Sigma^* \to \Sigma^*$ is defined as:

$$E_\varphi\text{-opt}(\sigma) \quad = E_O\text{-opt}(E_I\text{-opt}(\sigma_I),\sigma_O).$$

The only difference in $E_I$-opt compared to $E_I$ in Definition 2.1 is that in minD-editI$_{\mathcal{A}_{\varphi_\mathsf{I}}}$ is used instead of rand-editI$_{\mathcal{A}_{\varphi_\mathsf{I}}}$. That is, $x' =$ rand-editI$_{\mathcal{A}_{\varphi_\mathsf{I}}}(q)$ is replaced with $x' =$ minD-editI$_{\mathcal{A}_{\varphi_\mathsf{I}}}(q,x)$. Thus, instead of picking any random element from editI$_{\mathcal{A}_{\varphi_\mathsf{I}}}(q)$, an element in editI$_{\varphi_\mathsf{I}}(q)$ which differs minimally w.r.t the actual input $x$ is selected using minD-editI$_{\mathcal{A}_{\varphi_\mathsf{I}}}(q,x)$.

Similarly, the only difference in $E_O$-opt compared to $E_O$ in Definition 2.1 is that in $E_O$-opt, minD-editO$_{\mathcal{A}_\varphi}$ is used instead of rand-editO$_{\mathcal{A}_\varphi}$ (i.e., $y' =$ rand-editO$_{\mathcal{A}_\varphi}(q,x)$ is replaced with $y' =$ minD-editO$_{\mathcal{A}_\varphi}(q,x,y)$).

Theorem 2.3 ($E_\varphi$-opt exists iff $E_\varphi$ exists). *Given some property $\varphi$, an optimal enforcer $E_\varphi$-opt (according to Definition 2.2) exists iff an enforcement function $E_\varphi$ (according to Definition 2.1) exists.*

## REFERENCES


[1] Weiwei Ai, Nitish Patel, and Partha Roop. 2016. Requirements-centric closed-loop validation of implantable cardiac devices. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*. IEEE, 846–849.

[2] Homa Alemzadeh, Ravishankar K Iyer, Zbigniew Kalbarczyk, and Jai Raman. 2013. Analysis of safety-critical computer failures in medical devices. *Security & Privacy, IEEE* 11, 4 (2013), 14–26.

[3] Rajeev Alur and David L. Dill. 1994. A theory of timed automata. *Theoretical Computer Science* 126, 2 (1994), 183 – 235. https://doi.org/10.1016/0304-3975(94)90010-8

[4] Charles Andre, Frédéric Boulanger, and Alain Girault. 2001. Software implementation of synchronous programs. In *Application of Concurrency to System Design, 2001. Proceedings. 2001 International Conference on*. IEEE, 133–142.

[5] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. 2003. The synchronous languages 12 years later. *Proc. IEEE* 91, 1 (Jan 2003), 64–83. https://doi.org/10.1109/JPROC.2002.805826

[6] Roderick Bloem, Bettina Könighofer, Robert Könighofer, and Chao Wang. 2015. Shield Synthesis: Runtime Enforcement for Reactive Systems. In *TACAS (LNCS)*, Vol. 9035. Springer.

[7] Marius Bozga, Oded Maler, and Stavros Tripakis. 1999. Efficient Verification of Timed Automata Using Dense and Discrete Time Semantics. In *Correct Hardware Design and Verification Methods: 10th IFIP WG10.5 Advanced Research Working Conference, CHARME'99 BadHerrenalb,Germany,September 27–29, 1999 Proceedings*, Laurence Pierre and Thomas Kropf (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 125–141. https://doi.org/10.1007/3-540-48153-2_11

[8] Lei Bu, Qixin Wang, Xin Chen, Linzhang Wang, Tian Zhang, Jianhua Zhao, and Xuandong Li. 2011. Toward Online Hybrid Systems Model Checking of Cyber-physical Systems' Time-bounded Short-run Behavior. *SIGBED Rev.* 8, 2 (June 2011), 7–10. https://doi.org/10.1145/2000367.2000368

[9] Taolue Chen, Marco Diciolla, Marta Kwiatkowska, and Alexandru Mereacre. 2013. A simulink hybrid heart model for quantitative verification of cardiac pacemakers. In *Proceedings of the 16th international conference on Hybrid systems: computation and control*. ACM, 131–136.

[10] Egor Dolzhenko, Jay Ligatti, and Srikar Reddy. 2015. Modeling runtime enforcement with mandatory results automata. *Int. J. Inf. Sec.* 14, 1 (2015), 47–60.

[11] Ylies Falcone, Thierry Jéron, Hervé Marchand, and Srinivas Pinisetty. 2016. Runtime enforcement of regular timed properties by suppressing and delaying events. *Science of Computer Programming* 123 (2016), 2–41.

[12] Yliès Falcone, Laurent Mounier, Jean-Claude Fernandez, and Jean-Luc Richier. 2011. Runtime enforcement monitors: composition, synthesis, and enforcement abilities. *FMSD* 38, 3 (2011), 223–262.

[13] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. 1994. Synchronous observers and the verification of reactive systems. In *Algebraic Methodology and Software Technology (AMAST'â ÁŹ93)*. Springer, 83–96.

[14] Zhihao Jiang, Miroslav Pajic, Rajeev Alur, and Rahul Mangharam. 2014. Closed-loop verification of medical devices with model abstraction and refinement. *International Journal on Software Tools for Technology Transfer* 16, 2 (2014), 191–213. https://doi.org/10.1007/s10009-013-0289-7

[15] Z. Jiang, M. Pajic, and R. Mangharam. 2012. Cyber-Physical Modeling of Implantable Cardiac Medical Devices. *Proc. IEEE* 100, 1 (Jan 2012), 122–137. https://doi.org/10.1109/JPROC.2011.2161241

[16] Zhihao Jiang, Miroslav Pajic, Salar Moarref, Rajeev Alur, and Rahul Mangharam. 2012. Modeling and Verification of a Dual Chamber Implantable Pacemaker. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'12)*. Springer-Verlag, Berlin, Heidelberg, 188–203. https://doi.org/10.1007/978-3-642-28756-5_14

[17] Marta Kwiatkowska, Harriet Lea-Banks, Alexandru Mereacre, and Nicola Paoletti. 2014. Formal modelling and validation of rate-adaptive pacemakers. In *Healthcare Informatics (ICHI), 2014 IEEE International Conference on*. IEEE, 23–32.

[18] Jay Ligatti, Lujo Bauer, and David Walker. 2009. Run-Time Enforcement of Nonsafety Policies. *ACM Trans. Inf. Syst. Secur.* 12, 3, Article 19 (Jan. 2009), 19:1–19:41 pages.

[19] Srinivas Pinisetty, Yliès Falcone, Thierry Jéron, Hervé Marchand, Antoine Rollet, and Omer Nguena Timo. 2014. Runtime enforcement of timed properties revisited. *FMSD* 45, 3 (2014), 381–422. https://doi.org/10.1007/s10703-014-0215-y

[20] Fred B. Schneider. 2000. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.* 3, 1 (2000), 30–50.

[21] Partha S. Roop Sidharta Andalam, Avinash Malik and Mark Trew. 2016. Hybrid Automata Model of the Heart for Formal Verification of Pacemakers. In *Applied Verification for Continuous and Hybrid Systems (ARCH'16)*. Vienna, Austria.

[22] Christian von Essen and Barbara Jobstmann. 2013. *Program Repair without Regret*. Springer, Berlin, Heidelberg, 896–911. https://doi.org/10.1007/978-3-642-39799-8_64

[23] F. Xu, Z. Qin, C. C. Tan, B. Wang, and Q. Li. 2011. IMDGuard: Securing implantable medical devices with the external wearable guardian. In *2011 Proceedings IEEE INFOCOM*. 1862–1870. https://doi.org/10.1109/INFCOM.2011.5934987

[24] P Ye, E Entcheva, SA Smolka, and R Grosu. 2008. Modelling excitable cells using cycle-linear hybrid automata. *IET systems biology* 2, 1 (2008), 24–32.

[25] Eugene Yip, Sidharta Andalam, Partha S Roop, Avinash Malik, Mark Trew, Weiwei Ai, and Nitish Patel. 2016. Towards the Emulation of the Cardiac Conduction System for Pacemaker Testing. *arXiv preprint arXiv:1603.05315* (2016).