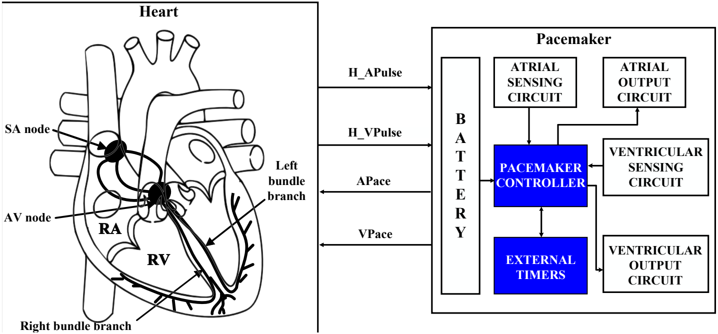
Runtime Enforcement of Reactive Systems using Synchronous Enforcers

Srinivas Pinisetty[1], Partha Roop[3], Steven Smyth[4], Stavros Tripakis[1,2], Reinhard von Hanxleden[4]

Aalto University, Finland
University of California, Berkeley
University of Auckland, New Zealand
Kiel University, Germany

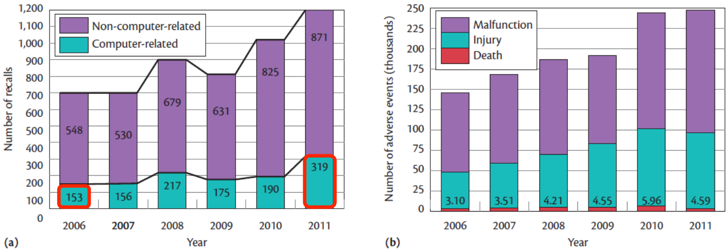
Implantable pacemakers



[a]

[a] Zhao and Roop, "Model Driven Design of Cardiac Pacemakers using IEC61499, CRC Press, 2015".

Adverse events



	Class I: high risk	Class II: medium risk	Class III: low risk	Total recalls	Number of devices
Software	14 (33.3%)	718 (65.6%)	46 (75.3%)	778 (64.3%)	2,303,441 (19.2%)
Hardware	8 (19.0%)	158 (14.4%)	13 (27.4%)	179 (14.8%)	4,228,133 (35.2%)
Other	10 (23.8%)	124 (11.3 %)	8 (12.3%)	142 (11.7%)	2,831,048 (23.5%)
Battery	8 (19.0%)	57 (5.2%)	5 (6.8%)	70 (5.8%)	2,385,613(19.8%)
I/O	2 (4.8%)	38 (3.5%)	1 (2.7%)	41 (3.4%)	276,601 (2.3%)
Total recalls	42 (3.5%)	1,095 (90.5%)	73 (6.0%)	1,210	12,024,836

[Ref.]: Alemzadeh, H., Iyer, R.K., Kalbarczyk, Z., Raman, J., "Analysis of Safety-Critical Computer Failures in Medical Devices", *Security and Privacy* , IEEE , vol.11, no.4. pp.14,26. July-Aug, 2013. [a]

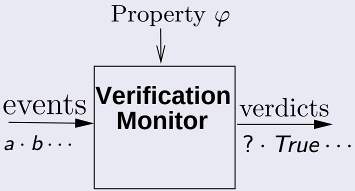
[a] This figure is reproduced from the reference above.

Approaches to enhance pacemaker software

- Two key CS related initiatives: http://cybercardia.cs.stonybrook.edu, and Marta Kwiatkowska's group in Oxford.
- Model-based approach: Modeling and verification of a dual chamber implantable pacemaker, *Jiang, Pajic, Moarref, Alur, Mangaram.* TACAS 2012
- Testing: Heart-on-a-chip: A closed-loop testing platform for implantable pacemakers *Jiang, Radhakrishnan, Sampath, Sarode, Mangharam.* CyPhy 2013
- Requirements-Centric Closed-Loop Validation of Implantable Cardiac Devices. *Weiwei Ai, Nitish Patel and Partha Roop.* DATE '16.
- Except the work of Ai et al., others consider a static model of the heart during closed-loop testing / model checking.
- Focus of the current work is on *run-time enforcement*, where a dynamically evolving heart model and a pacemaker can be used for run-time verification and enforcement.

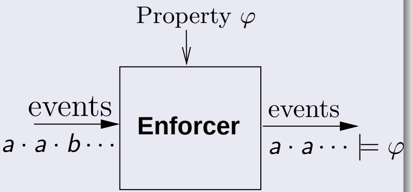
Runtime verification and enforcement

Runtime verification



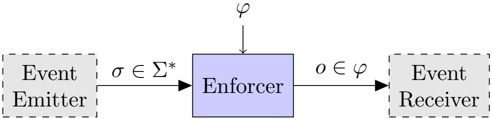
- Does σ satisfy φ ?
- Output: stream of **verdicts**.

Runtime enforcement

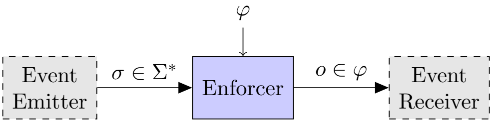


- Input: stream of events.
- Modified to satisfy the property.
- Output: stream of **events**.

Runtime enforcement (previous work)



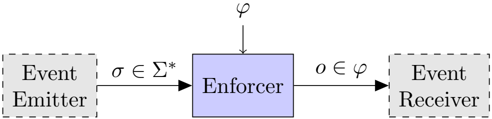
Runtime enforcement (previous work)



Enforcer for φ operating at runtime

- φ: any regular property (defined as automaton).

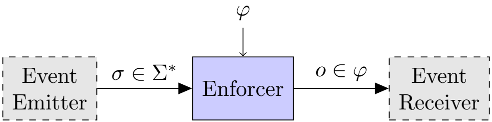
Runtime enforcement (previous work)



Enforcer for φ operating at runtime

- φ: any regular property (defined as automaton).
- An enforcer behaves as a function $E : \Sigma^* \to \Sigma^*$.
  - Input ($\sigma \in \Sigma^*$): any sequence of events over Σ (Event emitter is a black-box).

Runtime enforcement (previous work)

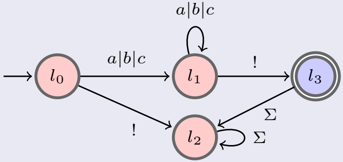


Enforcer for φ operating at runtime

- φ: any regular property (defined as automaton).
- An enforcer behaves as a function $E : \Sigma^* \rightarrow \Sigma^*$.
  - Input ($\sigma \in \Sigma^*$): any sequence of events over Σ (Event emitter is a black-box).
  - Output ($o \in \Sigma^*$): a sequence of events such that $o \models \varphi$.

Example: EM

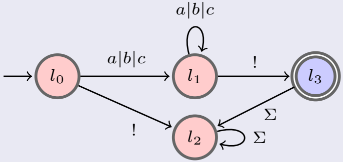
Property φ



- $\Sigma = \{a, b, c, !\}$

Example: EM

Property φ

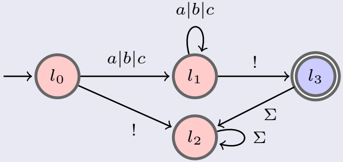


- $\Sigma = \{a, b, c, !\}$

INPUT	MEMORY	OUTPUT
a $\notin \varphi$	**a**	ϵ

Example: EM

Property φ

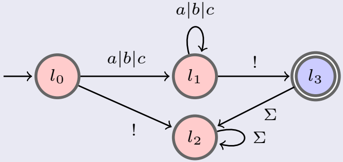


- $\Sigma = \{a, b, c, !\}$

INPUT	MEMORY	OUTPUT
$\mathbf{a} \notin \varphi$	$\mathbf{a}$	ϵ
$a \cdot \mathbf{b} \notin \varphi$	$a \cdot \mathbf{b}$	ϵ
$a \cdot b \cdot \mathbf{c} \notin \varphi$	$a \cdot b \cdot \mathbf{c}$	ϵ

Example: EM

Property φ

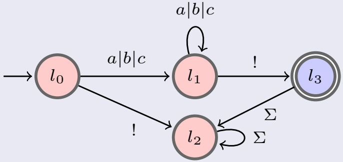


- $\Sigma = \{a, b, c, !\}$

INPUT	MEMORY	OUTPUT
$\mathbf{a} \notin \varphi$	$\mathbf{a}$	ϵ
$a \cdot \mathbf{b} \notin \varphi$	$a \cdot \mathbf{b}$	ϵ
$a \cdot b \cdot \mathbf{c} \notin \varphi$	$a \cdot b \cdot \mathbf{c}$	ϵ
$a \cdot b \cdot c \cdot \mathbf{!} \in \varphi$	ϵ	$a \cdot b \cdot c \cdot !$

Example: EM

Property φ



- $\Sigma = \{a, b, c, !\}$

INPUT	MEMORY	OUTPUT
$\mathbf{a} \notin \varphi$	$\mathbf{a}$	ϵ
$a \cdot \mathbf{b} \notin \varphi$	$a \cdot \mathbf{b}$	ϵ
$a \cdot b \cdot \mathbf{c} \notin \varphi$	$a \cdot b \cdot \mathbf{c}$	ϵ
$a \cdot b \cdot c \cdot \mathbf{!} \in \varphi$	ϵ	$a \cdot b \cdot c \cdot !$

Remarks

- Store events in the memory until observing input sequence that satisfies φ.

Shield Synthesis[1]

- Designed for reactive systems.
- Shield must "act upon erroneous outputs on the fly", without knowledge of the future.
- Has multiple input streams to deal with.

[1] Bloem et al., TACAS, 2015

Synchronous Languages

- The reactive system operates "infinitely fast" relative to the environment. This is known as the synchrony hypothesis.
- All concurrent components progress in "lock-step" relative to the ticks of a logical clock.
- Concurrency is usually "compiled away" to produce sequential code.

Synchronous observers

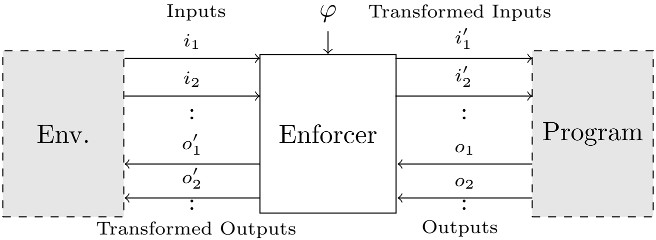
```
module BeatObserver:
  input AS; VS
   output  beatViolation;
    loop
        present AS and VS then
                emit beatViolation;
        end;
        pause;
    end loop
end module
```

Figure: BeatObserver in Esterel

Problem Statement

- Observers are usually static entities.
- Run-time observers may be considered as run-time verifiers but these are not enforcers.
- Observers are specified by the designers while monitors / enforcers are automatically synthesized from the specification of properties.
- Shield synthesis: this is the closest to our framework. Has two limitations. First, it performs no enforcement on the environment, which is very important for reactive systems. Second, it lacks causality and performs uni-directional enforcement. For synchronous reactive systems, enhanced bi-directional enforcement is essential.

Runtime enforcement in the synchronous setting



- Two-way enforcement like MRA with additional capability.
- Similar to a shield but supports enforcement of both the environment and the program. Also, has a notion of causality.

Execution of a synchronous program

- Execution of a program $\mathcal{P}$ is an infinite sequence of reactions.
- During each reaction, the program reacts to a set of inputs received from the environment to produce a set of outputs.
- I, O denote ordered sets of inputs and outputs respectively.
- The input alphabet $\Sigma_I = 2^I$ and the output alphabet $\Sigma_O = 2^O$ and $\Sigma = \Sigma_I \times \Sigma_O$. Each input/output will be denoted as a bit-vector / complete monomial e.g. 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 is of the form (x_i, y_i), where $x_i \in \Sigma_I$ and $y_i \in \Sigma_O$.
- A trace is a sequence of reactions of the form $\sigma = (x_0, y_0).(x_1, y_1).(x_2, y_2)... \in \Sigma^\omega$.
- We use the shorthand $\sigma = r_0.r_1.r_2... \in \Sigma^\omega$, where r_i denotes the i-th reaction.
- The behaviour of the program $\mathcal{P}$ is $exec(\mathcal{P}) \subseteq \Sigma^\omega$.
- $\mathcal{L}(\mathcal{P})=\{\sigma \in \Sigma^* | \exists \sigma' \in exec(\mathcal{P}) \wedge \sigma \preccurlyeq \sigma'\}$.

Properties

- A property φ defines a set of valid executions, where $\mathcal{L}(\varphi) \subseteq \Sigma^*$.
- We consider *prefix-closed* properties (all prefixes of all words in $\mathcal{L}(\varphi)$ are also in $\mathcal{L}(\varphi)$).
- A property φ is defined as a safety automaton $A^\varphi = (Q, q_0, q_v, \Sigma, \rightarrow)$, where Q is the set of states, called *locations*, $q_0 \in Q$ is an unique initial location, $q_v \in Q$ is a unique violating (non-accepting) location, Σ is the alphabet, and $\rightarrow \subseteq Q \times \Sigma \times Q$ is the transition relation. All the locations in Q except q_v (i.e., $Q \setminus \{q_v\}$) are accepting locations.

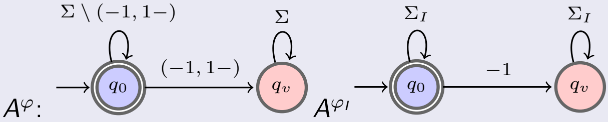
A property and its input projection

Projection over inputs

Given property $\varphi \subseteq \Sigma^*$, defined as automaton $\mathcal{A}_\varphi = (Q, q_0, q_v, \Sigma, \rightarrow)$, we define and use the following: $\boldsymbol{\varphi_I}$ and $\boldsymbol{\mathcal{A}_{\varphi_I}}$: Input automaton $\mathcal{A}_{\varphi_I} = (Q, q_0, q_v, \Sigma_I, \rightarrow_I)$ is obtained from $\mathcal{A}_\varphi = (Q, q_0, q_v, \Sigma, \rightarrow)$ by ignoring outputs. If (x, y) is in Σ, then $x \in \Sigma_I$, and every transition $q \xrightarrow{(x,y)} q'$ in $\mathcal{A}_\varphi$ is replaced with transition $q \xrightarrow{x}_I q'$ in $\mathcal{A}_{\varphi_I}$.
$\mathcal{L}(\mathcal{A}_{\varphi_I})$ is denoted as $\varphi_I \subseteq \Sigma_I^*$.

Example property defined as SA

- $I = \{A, B\}$, and $O = \{R, W\}$.
- "B and R cannot happen simultaneously".



Synchronous RE Preliminaries (1)

- σ_I: Given $\sigma = (x_1, y_1) \cdot (x_2, y_2) \cdots (x_n, y_n)$, the projection on inputs is $\sigma_I = x_1 \cdot x_2 \cdots x_n \in \Sigma_I$.
- σ_O: Given $\sigma = (x_1, y_1) \cdot (x_2, y_2) \cdots (x_n, y_n)$, the projection on outputs is $\sigma_O = y_1 \cdot y_2 \cdots y_n \in \Sigma_O$.
- $\mathcal{A}_{\varphi_I}$: From $\mathcal{A}_\varphi$, $\mathcal{A}_{\varphi_I}$ is obtained by ignoring outputs on the transitions.

Synchronous RE Preliminaries (2)

editI_{φ_I} (resp. editO_{φ}), that the enforcer uses for editing input (resp. output) event (whenever necessary).

- $\text{editI}_{\varphi_I}(\sigma_I)$: $\text{editI}_{\varphi_I}(\sigma_I) = \{x \in \Sigma_I : \sigma_I \cdot x \models \varphi_I\}$.
- Considering $\mathcal{A}_{\varphi_I} = (Q, q_0, q_v, \Sigma_I, \rightarrow_I)$, and $q \in Q$,

$$\text{editI}_{\mathcal{A}_{\varphi_I}}(q) = \{x \in \Sigma_I : q \xrightarrow{x}_I q' \wedge q' \neq q_v\}.$$

- $\text{editO}_{\varphi}(\sigma, x)$: $\text{editO}_{\varphi}(\sigma, x) = \{y \in \Sigma_O : \sigma \cdot (x, y) \models \varphi\}$.
- Considering $\mathcal{A}_{\varphi} = (Q, q_0, q_v, \Sigma, \rightarrow)$ defining property φ, and an input event $x \in \Sigma_I$, the set of output events y in Σ_O that allow to reach a state in $Q \setminus \{q_v\}$ from a state $q \in Q$ with (x, y) is defined as:

$$\text{editO}_{\mathcal{A}_{\varphi}}(q, x) = \{y \in \Sigma_O : q \xrightarrow{(x,y)} q' \wedge q' \neq q_v\}.$$

- $\text{rand–editI}_{\varphi_I}(\sigma_I)$: An element chosen randomly from $\text{editI}_{\varphi_I}(\sigma_I)$.
- $\text{rand–editO}_{\varphi}(\sigma, x)$: An element chosen randomly from $\text{editO}_{\varphi}(\sigma, x)$.

Enforcer synthesis problem– Assumptions

- We assume that the synchronous program may be invoked as a "black box" system through a special function call called `ptick`. This function takes a bit vector x and returns a bit vector y. Formally, $ptick : \Sigma_I \rightarrow \Sigma_O$.
- Recall functions $\mathsf{editI}_{\varphi_I}$ and editO_{φ} that were introduce for editing inputs (respectively outputs). These are used by the enforcer to edit the input/output bit vectors.

Enforcer synthesis problem–constraints

Preliminaries (recall)

- I: set of inputs, O: set of outputs.
- $\Sigma_I = 2^I$, $\Sigma_O = 2^O$, and $\Sigma = \Sigma_I \times \Sigma_O$.
- Event (reaction): (x_i, y_i) where $x_i \in \Sigma_i$ and $y_i \in \Sigma_O$.
- Word σ: $(x_0, y_0) \cdot (x_1, y_1) \cdots \in \Sigma^*$.
- Property φ: $\varphi \subseteq \Sigma^*$.

Given φ, synthesize an enforcer $E_\varphi : \Sigma^* \rightarrow \Sigma^*$ that satisfies:

- Soundness: $\forall \sigma \in \Sigma^* : E_\varphi(\sigma) \models \varphi$.
- Monotonicity: $\forall \sigma, \sigma' \in \Sigma^* : \sigma \preccurlyeq \sigma' \Rightarrow E_\varphi(\sigma) \preccurlyeq E_\varphi(\sigma')$.
- Instantaneity: $\forall \sigma \in \Sigma^* : |\sigma| = |E_\varphi(\sigma)|$.
- Transparency: $\forall \sigma \in \Sigma^*, \forall x \in \Sigma_I, \forall y \in \Sigma_O$:
$$E_\varphi(\sigma) \cdot (x, y) \models \varphi \implies E_\varphi(\sigma \cdot (x, y)) = E_\varphi(\sigma) \cdot (x, y).$$
- Causality: $\forall \sigma \in \Sigma^*, \forall x \in \Sigma_I, \forall y \in \Sigma_O$,
$$\exists x' \in \mathrm{editI}_{\varphi_I}(E_\varphi(\sigma)_I), \exists y' \in \mathrm{editO}_\varphi(E_\varphi(\sigma), x') :$$
$$E_\varphi(\sigma \cdot (x, y)) = E_\varphi(\sigma) \cdot (x', y').$$

When input σ satisfies φ

Transparency': $\forall \sigma \in \Sigma^* : \sigma \in \varphi \Rightarrow E_\varphi(\sigma) = \sigma$

Transparency' means that when the input sequence σ satisfies φ, then σ will be the output of the enforcer.

Lemma (*Transparency* $\implies$ *Transparency'*)

$(\forall \sigma \in \Sigma^*, \forall x \in \Sigma_I, \forall y \in \Sigma_O : E_\varphi(\sigma) \cdot (x, y) \models \varphi \implies E_\varphi(\sigma \cdot (x, y)) = E_\varphi(\sigma) \cdot (x, y))$
$\implies$
$(\forall \sigma \in \Sigma^* : \sigma \in \varphi \Rightarrow E_\varphi(\sigma) = \sigma)$.

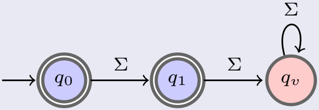
Example (Transparency is stronger)

- $I = \{A, B\}$, $O = \{O\}$, Property φ: A and B cannot happen simultaneously.

σ	$E_\varphi(\sigma)$	TR	TR'
$01-$	$01-$	✓	✓
$01-\cdot 11-$	$01-\cdot 10-$	✓	✓
$01-\cdot 11-\cdot 01-$	$01-\cdot 10-\cdot 10-$	✗	✓
$01-\cdot 11-\cdot 01-$	$01-\cdot 10-\cdot 01-$	✓	✓

Enforceable safety properties

A non-enforceable safety property



Enforceability condition

A property φ defined as automaton $A^{\varphi} = (Q, q_0, q_v, \Sigma, \rightarrow)$ is enforceable (i.e., E_{φ} according to our problem def. exists) if

$$\forall q \in Q, q \neq q_v \implies \exists (x, y) \in \Sigma : q \xrightarrow{(x,y)} q' \in \delta \wedge q' \neq q_v$$

Functional Definition (1)

Preliminaries (recall)

- I: set of inputs, O: set of outputs.
- $\Sigma_I = 2^I$, $\Sigma_O = 2^O$, and $\Sigma = \Sigma_I \times \Sigma_O$.
- Event (reaction): (x_i, y_i) where $x_i \in \Sigma_i$ and $y_i \in \Sigma_O$.
- Word σ: $(x_0, y_0) \cdot (x_1, y_1) \cdots (x_n, y_n) \in \Sigma^*$.
  - σ_I: $x_0 \cdot x_i \cdots x_n \in \Sigma_I$ (projection of $x_i's$ from σ).
  - σ_O: $y_0 \cdot y_i \cdots y_n \in \Sigma_O$ (projection of $y_i's$ from σ).
- Property φ: $\varphi \subseteq \Sigma^*$, Automaton $\mathcal{A}_\varphi$.
  - Property φ_I, automaton $\mathcal{A}_{\varphi_I}$ (from $\mathcal{A}_\varphi$ considering only $x_i's$.)

$E_\varphi : \Sigma^* \rightarrow \Sigma^*$

$E_\varphi(\sigma \cdot (x, y)) = E_O(E_I(\sigma_I \cdot x), \sigma_o \cdot y)$.

- σ_I: projection of $x_i's$ from σ, σ_O: projection of $y_i's$ from σ.
- $E_I : \Sigma_I^* \rightarrow \Sigma_I^*$, $E_O : \Sigma_I^* \times \Sigma_O^* \rightarrow (\Sigma_I \times \Sigma_O)^*$.

Definition of E_I and E_O (next slide).

Functional Definition (2)

$E_\varphi : \Sigma^* \rightarrow \Sigma^*$

$E_\varphi(\sigma \cdot (x, y)) = E_O(E_I(\sigma_I \cdot x), \sigma_o \cdot y)$.

$E_I : \Sigma_I^* \rightarrow \Sigma_I^*$

$$E_I(\sigma_I \cdot x) = \begin{cases} E_I(\sigma_I) \cdot x & \text{if } E_I(\sigma_I) \cdot x \models \varphi_I, \\ E_I(\sigma_I) \cdot edit_I(x) & \text{otherwise} \end{cases}$$

$E_O : \Sigma_I^* \times \Sigma_O^* \rightarrow (\Sigma_I \times \Sigma_O)^*$

$$E_O(\sigma_I \cdot x, \sigma_O \cdot y) = \begin{cases} E_O(\sigma_I, \sigma_O) \cdot (x, y) & \text{if } E_O(\sigma_I, \sigma_O) \cdot (x, y) \models \varphi, \\ E_O(\sigma_I, \sigma_O) \cdot (x, edit_O(y)) & \text{otherwise} \end{cases}$$

$edit_I()$, and $edit_O()$ in next slide.

Functional Definition (3): $Edit_I()$ function

- I: set of inputs, O: set of outputs, $\Sigma_I = 2^I$, $\Sigma_O = 2^O$, and $\Sigma = \Sigma_I \times \Sigma_O$.
- Event (reaction): (x_i, y_i) where $x_i \in \Sigma_i$ and $y_i \in \Sigma_O$.
- Word σ: $(x_0, y_0) \cdot (x_1, y_1) \cdots (x_n, y_n) \in \Sigma^*$, σ_I: $x_0 \cdot x_i \cdots x_n \in \Sigma_I$, and σ_O: $y_0 \cdot y_i \cdots y_n \in \Sigma_O$.
- Property φ: $\varphi \subseteq \Sigma^*$, Automaton $\mathcal{A}_\varphi$.
  - Property φ_I, automaton $\mathcal{A}_{\varphi_I}$ (from $\mathcal{A}_\varphi$ considering only $x_i's$.)

$edit_I$

- INPUT:$\mathcal{A}_{\varphi_I} = (Q, q_0, q_v, \Sigma, \delta)$, $q \in Q$ (state reached upon $E_I(\sigma)$), new input $x \in \Sigma_I$.
- OUTPUT: $x' \in \Sigma_I$.
- $OK\_solutions\_I(\mathcal{A}_{\varphi_I}, q, x) = \{x' \in \Sigma_I : q \xrightarrow{x'} q' \in \delta \wedge q' \neq q_v\}$.
- $edit_I$ (different possible solutions).
  1. $edit_I(\mathcal{A}_{\varphi_I}, q, x) = rand(OK\_solutions\_I(\mathcal{A}_{\varphi_I}, q, x))$ "random selection from $OK\_solutions\_I()$".
  2. Element from $OK\_solutions_I(\mathcal{A}_{\varphi_I}, q, x)$ that differs MINIMALLY w.r.t the actual input x.
     - $edit_I(\mathcal{A}_{\varphi_I}, q, x) = min_{dist}(OK\_solutions\_I(\mathcal{A}_{\varphi_I}, q, x))$.
     - "$min_{dist}(OK\_solutions\_I(\mathcal{A}_{\varphi_I}, q, x))$": pick an element from $OK\_solutions\_I()$ that has minimal distance w.r.t x.
  3. ...

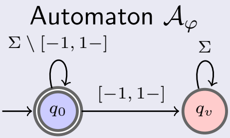
Functional Definition (4): $Edit_O()$ function

- I: set of inputs, O: set of outputs, $\Sigma_I = 2^I$, $\Sigma_O = 2^O$, and $\Sigma = \Sigma_I \times \Sigma_O$.
- Event (reaction): (x_i, y_i) where $x_i \in \Sigma_I$ and $y_i \in \Sigma_O$.
- Word σ: $(x_0, y_0) \cdot (x_1, y_1) \cdots (x_n, y_n) \in \Sigma^*$, σ_I: $x_0 \cdot x_i \cdots x_n \in \Sigma_I$, and σ_O: $y_0 \cdot y_i \cdots y_n \in \Sigma_O$.
- Property φ: $\varphi \subseteq \Sigma^*$, Automaton $\mathcal{A}_\varphi$.

$edit_O$

- INPUT: $\mathcal{A}_\varphi = (Q, q_0, q_v, \Sigma, \delta)$, $q \in Q$ (state reached upon $E_\varphi(\sigma)$), new input event (x, y) where $x \in \Sigma_I$ and $y \in \Sigma_O$.
- OUTPUT: y' where $y' \in \Sigma_O$.
- $OK\_solutions\_O(\mathcal{A}_\varphi, q, (x, y)) = \{y' \in \Sigma_O : q \xrightarrow{(x,y')} q' \in \delta \wedge q' \neq q_v\}$.
- $edit_O$ (different possible solutions).
  1. $edit_O(\mathcal{A}_\varphi, q, (x, y)) = rand(OK\_solutions\_O(\mathcal{A}_\varphi, q, (x, y))$ "random selection from $OK\_solutions\_O()$".
  2. Element from $OK\_solutions\_O(\mathcal{A}_\varphi, q, (x, y))$ that differs MINIMALLY w.r.t y.
     - $edit_O(\mathcal{A}_\varphi, q, (x, y)) = min_{dist}(OK\_solutions\_O(\mathcal{A}_\varphi, q, (x, y)))$.
     - "$min_{dist}(OK\_solutions\_O(\mathcal{A}_\varphi, q, (x, y)))$": pick an element from $OK\_solutions\_O()$ that has minimal distance w.r.t y.
  3. ...

Functional Definition (5): Example



$\sigma_I = \epsilon_i$
$E_I(\epsilon_I) = \epsilon_i, \quad q_i = q_{o_i}$
$\sigma_O = \epsilon_o$
$E_O(\epsilon_I, \epsilon_o) = \epsilon, \quad q = q_o$

$\sigma_I = \mathbf{10}$
$E_I(\mathbf{10}) = \mathbf{10}, \quad q_i = q_{o_i}$
$\sigma_O = \mathbf{11}$
$E_O(\mathbf{10}, \mathbf{11}) = (\mathbf{10,11}), \quad q = q_o$

$\sigma_I = 10 \cdot \mathbf{11}$
$E_I(10 \cdot \mathbf{11}) = 10 \cdot \mathbf{11}, \quad q_i = q_{o_i}$
$\sigma_O = 11 \cdot \mathbf{10}$
$E_O(10 \cdot \mathbf{11}, 11 \cdot \mathbf{10}) = (10, 11) \cdot (11, edit_o(\mathcal{A}_\varphi, q, (11, 10)))$
$= (10, 11) \cdot (\mathbf{11,00}), \quad q = q_o$

- $I = \{A, B\}$, $O = \{R, W\}$.
- Property: B and R cannot happen simultaneously.
- Initially $\sigma = \epsilon$, $\sigma_I = \epsilon_i$, $\sigma_o = \epsilon_o$.
- q : state in $\mathcal{A}_\varphi$ upon $E_\varphi(\sigma)$, q_i : state in $\mathcal{A}_{\varphi_I}$ upon $E_I(\sigma_I)$.
- $OK\_solutions\_O(\mathcal{A}_\varphi, q_0, (11, 10)) = \{00, 01\}$.
- $min_{dist}(OK\_Solutions\_O(\mathcal{A}_\varphi, q_0, (11, 10))) = 00$.

Enforcement algorithm (1)

Input: $A^{\varphi} = (Q, q_0, q_v, \Sigma, \rightarrow)$.
$A^{\varphi_I} = (Q_I, q_{0_I}, q_{v_I}, \Sigma_I, \rightarrow_I)$ (Obtained from A^{φ} by ignoring outputs.)

Enforcement algorithm

initialize tick/time, automata current states;
while *True* **do**
    READ-input-channels;
    EDIT-input-if-necessary;
    READ-output-channels (after invoking program);
    EDIT-output-if-necessary;
    UPDATE-automata-current-states;
end

Enforcement algorithm (2)

Enforcer

$t \leftarrow 0$
$(q, q_I) \leftarrow (q_0, q_{0_I})$
while true **do**
  $x_t \leftarrow \text{read\_in\_chan}()$
  if $q_I \xrightarrow{x_t}_I q'_I \wedge q'_I \neq q_{v_I}$ **then**
    $x'_t \leftarrow x_t$
  else
    $x'_t \leftarrow \text{rand–editI}_{\mathcal{A}_{\varphi_I}}(q_I)$
  end if
  $\text{ptick}(x'_t)$
  $y_t \leftarrow \text{read\_out\_chan}()$
  if $q \xrightarrow{(x'_t, y_t)} q' \wedge q' \neq q_v$ **then**
    $y'_t \leftarrow y_t$
  else
    $y'_t \leftarrow \text{rand–editO}_{\mathcal{A}_{\varphi}}(q, x'_t)$
  end if
  $\text{release}((x'_t, y'_t))$
  $q \leftarrow q'$ where $q \xrightarrow{(x'_t, y'_t)} q'$
  $q_I \leftarrow q'_I$ where $q_I \xrightarrow{x'_t}_I q'_I$
  $t \leftarrow t + 1$
end while

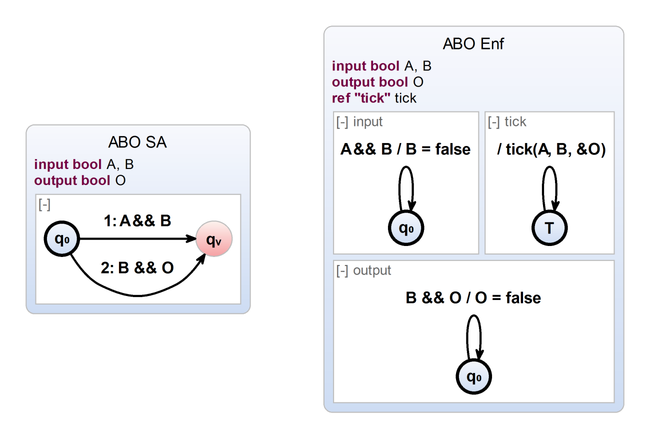
Enforcement algorithm (3)

Theorem

Given any safety property φ that is enforceable, for any $t > 0$, let $\sigma = (x_1, y_1) \cdots (x_t, y_t) \in \Sigma^*$ be the input-output word obtained by concatenating input-output events read by the algorithm. Let the sequence obtained by concatenating input-output events released as output by the algorithm be $E_\varphi(\sigma) = (x'_1, y'_1) \cdots (x'_t, y'_t) \in \Sigma^*$. The enforcement algorithm satisfies the soundness, transparency, monotonicity, instantainety, and causality constraints.

Application to the SCCharts synchronous language

Example: Property and its enforcer



Results

Examples	Tick (LoC)	φ: in-out	Enf. (LoC)	Time (ms)	Time w/ Enf. (ms)	Incr. (%)
Null	0	0-0	0	0.000654	0.000752	14.98
ABRO	23	1-0	21	0.001208	0.001565	29.55
ABO	28	1-0	21	0.000998	0.001368	37.10
Reactor	32	1-1	32	0.001587	0.002137	34.61
Faulty Heart Model	43	1-1	40	0.001346	0.001869	38.85
Simple Heart Model	76	1-1	40	0.002175	0.002825	29.86
Traffic Light	171	0-3	41	0.004039	0.004707	16.53
Pacemaker	271	1-1	35	0.007302	0.008318	13.91
FHM + Pacemaker	314	1-1	35	0.009195	0.010306	12.08

Conclusions and Future Work

- We formulated the problem of run-time enforcement of reactive systems modelled using the synchronous approach.
- We formalise the run-time enforcement problem as a bi-directional enforcement of prefix-closed safety properties.
- The concept of observers in synchronous languages is extended to the concept of enforcers and this approach has been developed for the SCCharts language.
- We have started extending the formulation to the practical setting of implantable pacemakers, where we have to enforce regular properties.