Runtime Enforcement of Reactive Systems using Synchronous Enforcers*

Srinivas Pinisetty†
Aalto University, Finland and
University of Gothenburg, Sweden
sripin@chalmers.se

Partha S. Roop
University of Auckland, New Zealand
p.roop@aucklanduni.ac.nz

Steven Smyth
University of Kiel, Germany
ssm@informatik.uni-kiel.de

Stavros Tripakis
Aalto University, Finland and
UC Berkeley, USA
stavros.tripakis@aalto.fi

Reinhard von Hanxleden
University of Kiel, Germany
rvh@informatik.uni-kiel.de

ABSTRACT

Synchronous programming is a paradigm of choice for the design of safety-critical reactive systems. Runtime enforcement is a technique to ensure that the output of a black-box system satisfies some desired properties. This paper deals with the problem of runtime enforcement in the context of synchronous programs. We propose a framework where an enforcer monitors both the inputs and the outputs of a synchronous program and (minimally) edits erroneous inputs/outputs in order to guarantee that a given property holds. We define enforceability conditions, develop an online enforcement algorithm, and prove its correctness. We also report on an implementation of the algorithm on top of the KIELER framework for the SCCharts synchronous language. Experimental results show that enforcement has minimal execution time overhead, which decreases proportionally with larger benchmarks.

CCS CONCEPTS

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

KEYWORDS

Runtime Monitoring, Runtime Enforcement, Safety Properties, Automata, Synchronous Programming, SCCharts

*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), and the Swedish Research Council (grant Nr. 2015-04154, *PolUser: Rich User-Controlled Privacy Policies*).

†This work has been done while the author was at Aalto University. The author is currently at University of Gothenburg since April 2017.

SPIN'17, July 2017, Santa Barbara, CA, USA

ACM ISBN 978-1-4503-5077-8/17/07…$15.00
https://doi.org/10.1145/3092282.3092291

ACM Reference format:
Srinivas Pinisetty, Partha S. Roop, Steven Smyth, Stavros Tripakis, and Reinhard von Hanxleden. 2017. Runtime Enforcement of Reactive Systems using Synchronous Enforcers. In *Proceedings of International SPIN Symposium on Model Checking of Software , Santa Barbara, CA, USA, July 2017 (SPIN'17),* 10 pages.
https://doi.org/10.1145/3092282.3092291

1 INTRODUCTION

Runtime verification (RV) [9] is an active area of research on methods that dynamically verify a set of desirable properties over an execution of a "black-box" system. An alternative to such passive runtime analysis is runtime enforcement (RE) [6, 10, 13, 17]. In RE mechanisms, an *enforcer* is synthesized to observe the executions of a black-box system to ensure that a set of desired properties are satisfied. In the event of a violation, the enforcer performs certain evasive actions so as to prevent the violation. The evasive actions might include blocking the execution [17], modifying input sequence by suppressing and / or inserting actions [10], and buffering input actions until a future time when it could be forwarded [6, 13]. These enforcement mechanisms are not suitable for synchronous reactive systems since delaying the reaction or terminating the system is infeasible. Considering this, there is recent interest in runtime enforcement of synchronous reactive systems [4].

A synchronous reactive system is non-terminating and interacts continuously with the adjoining environment. Hence, the system execution may be considered as a series of steps, where in each step the system reads the inputs from the environment, calls a *reaction function* that computes the outputs for emission. Synchronous programming languages [2] are well suited for the design of synchronous reactive systems. They use observers [7] to express safety properties, which are verified statically (using model checking). There have also been limited attempts to use observers as runtime entities [15], for example for automatic test case generation. More recently Rushby studies applications of observers for the expression of assumptions and axioms in addition to test case generation [16]. However, there have been no studies on the bi-directional RE problem for synchronous reactive systems, which is the focus of the current paper.

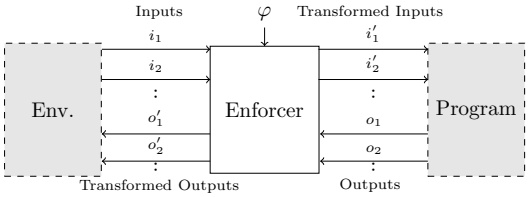


Figure 1: Bi-directional RE for synchronous programs.

We consider bi-directional RE of synchronous programs, and the general context is illustrated in Figure 1. Here, $\{i_1, i_2, \cdots, i_n\}$ are inputs from the environment to the enforcer, $\{i_1', i_2', \cdots, i_n'\}$ are transformed inputs from the enforcer to the program, $\{o_1, o_2, \cdots, o_m\}$ are outputs of the program to the enforcer, and $\{o_1', o_2', \cdots, o_m'\}$ are transformed outputs from the enforcer to the environment. RE for synchronous reactive systems is distinct from the existing RE mechanisms such as [6, 10, 13, 17] since the enforcement mechanism for a synchronous reactive system cannot halt the system or delay events, and must react instantaneously when an error is observed. Moreover, we consider bi-directional enforcement where the enforcer needs to consider the status of the environment and the program in order to enforce the policies. The enforcer must respect the *causality* aspects i.e. every reactive cycle must start with the environment, where the status of the environment inputs must determine the reaction. After the program has reacted, the generated outputs are emitted to the environment. Considering this, the enforcer must act as an intermediary such that it first intercepts the inputs from the environment to validate them relative to the policy and forward the inputs to the program once the policy is satisfied. In the event of any violation, the enforcer may suitably alter the inputs before forwarding to the program. After the program has reacted to these inputs, again the enforcer must ensure that either the policy is satisfied and hence the outputs are forwarded unchanged to the environment or a violation has happened that needs to be handled by altering the outputs to prevent policy violation.

We study the problem of synthesizing an enforcer for any given safety property φ. Similar to enforcement mechanisms in [6, 10, 13, 17], several constraints are required on how an enforcer transforms input-output words. The enforcer cannot delay events, and cannot block execution, but it is allowed to *edit* an event when necessary (i.e., when the event that it receives as input leads to a violation). The notions of *soundness* and *transparency* are similar to the existing enforcement mechanisms [6, 10, 13, 17], where soundness means that the output of the enforcer must satisfy property φ, and transparency expresses that the enforcer should not modify events unnecessarily. In the proposed framework, we also introduce additional requirements called *causality*, and *instantaneity*. These constraints are developed specifically to respect synchronous execution, detailed in Section 3. For any given safety property φ, these constraints also ensure that input/output events are edited *minimally* (to avoid violation of φ) so that other non-safety properties of the system are potentially not affected by the enforcement mechanism.

Contributions. In this paper, we study and formally define, for the first time, the bi-directional enforcer synthesis problem for synchronous reactive systems (expressed as synchronous programs). The main contributions of the paper are (1) We formally define the bi-directional enforcer synthesis problem and characterize the set of safety properties which can be enforced (Section 3), (2) We develop an enforcement algorithm (Section 4) and prove its correctness, (3) We report on an implementation of the algorithm on top of the KIELER framework for the SCCharts synchronous language (Section 5), and (5) We evaluate the approach over a range of synchronous programs in the SCCharts language [20] to illustrate scalability and practicality (Section 5).

2 PRELIMINARIES AND NOTATION

A finite (resp. infinite) word over a finite alphabet Σ 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 Σ. The set of finite (resp. infinite) words over Σ is denoted by Σ^* (resp. Σ^ω). The *length* of a finite word σ is n and is noted $|\sigma|$. The empty word over Σ is denoted by ϵ_Σ, or ϵ when clear from the context. The *concatenation* of two words σ and σ' is denoted as $\sigma \cdot \sigma'$. A word σ' is a *prefix* of a word σ, denoted as $\sigma' \preccurlyeq \sigma$, whenever there exists a word σ'' such that $\sigma = \sigma' \cdot \sigma''$; σ is said to be an *extension* of σ'.

We consider a reactive system with a finite ordered sets of Boolean inputs $I = \{i_1, i_2, \cdots, i_n\}$, and Boolean outputs $O = \{o_1, o_2, \cdots, o_m\}$. 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 $y_i \in \Sigma_O$.

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 σ is $\sigma_I = x_1 \cdot x_2 \cdots x_n \in \Sigma_I$ which is the projection on inputs ignoring outputs. Similarly, the output word obtained from σ is $\sigma_O = y_1 \cdot y_2 \cdots y_n \in \Sigma_O$ is the projection on outputs.

An execution σ 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 $\mathcal{L}(\mathcal{P}) = \{\sigma \in \Sigma^* | \exists \sigma' \in exec(\mathcal{P}) \wedge \sigma \preccurlyeq \sigma'\}$ i.e. $\mathcal{L}(\mathcal{P})$ is the set of all finite prefixes of the sequences in $exec(\mathcal{P})$.

A property φ over Σ defines a set $\mathcal{L}(\varphi) \subseteq \Sigma^*$. A program $\mathcal{P} \models \varphi$ iff $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\varphi)$. Given a word $\sigma \in \Sigma^*$, $\sigma \models \varphi$ iff $\sigma \in \mathcal{L}(\varphi)$. A property φ is *prefix-closed* iff all prefixes of all words from $\mathcal{L}(\varphi)$ are also in $\mathcal{L}(\varphi)$, i.e., $\forall \sigma \in \mathcal{L}(\varphi), \forall \sigma' \in \Sigma^* : \sigma' \preccurlyeq \sigma \implies \sigma' \in \mathcal{L}(\varphi)$. In this paper, we consider prefix-closed properties. Properties are formally expressed as safety automata that we define in the sequel.

Definition 2.1 (Safety Automaton). A *safety automaton* (SA) $\mathcal{A} = (Q, q_0, q_v, \Sigma, \rightarrow)$ is a tuple, 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, $\Sigma = \Sigma_I \times \Sigma_O$ 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. Location q_v is a unique non-accepting (trap) location, and there are no transitions in $\rightarrow$ from q_v to a location in $Q \setminus \{q_v\}$. Whenever there exists $(q, a, q') \in \rightarrow$, we denote it as $q \xrightarrow{a} q'$. Relation $\rightarrow$ is extended to words $\sigma \in \Sigma^*$ by noting $q \xrightarrow{\sigma.a} q'$ whenever there exists q'' such

that $q \xrightarrow{\sigma} q''$ and $q'' \xrightarrow{a} q'$. A location $q \in Q$ is reachable from q_0 if there exists a word $\sigma \in \Sigma^*$ such that $q_0 \xrightarrow{\sigma} q$.

An SA $\mathcal{A} = (Q, q_0, q_v, \Sigma, \rightarrow)$ is *deterministic* if $\forall q \in Q, \forall a \in \Sigma, (q \xrightarrow{a} q' \wedge q \xrightarrow{a} q'') \Rightarrow (q' = q'')$. $\mathcal{A}$ is *complete* if $\forall q \in Q, \forall a \in \Sigma, \exists q' \in Q, q \xrightarrow{a} q'$. A word σ is *accepted* by $\mathcal{A}$ if there exists $q \in Q \setminus \{q_v\}$ such that $q_0 \xrightarrow{\sigma} q$. The set of all words accepted by $\mathcal{A}$ is denoted as $\mathcal{L}(\mathcal{A})$.

Remark 1. *In the rest of this paper, φ is a safety property defined as deterministic and complete SA $\mathcal{A}_\varphi = (Q, q_0, q_v, \Sigma, \rightarrow)$. If the user provides a non-deterministic or incomplete automaton, we determinize and complete it first. We also consider that Q does not contain any (redundant) locations that are unreachable from q_0.*

Due to the causality requirement, the enforcer has to first transform inputs from the environment in each step according to property φ defined as SA $\mathcal{A}_\varphi$. We thus need to consider the input property that we obtain from $\mathcal{A}_\varphi$ by projecting on inputs.

Definition 2.2 (Input SA $\mathcal{A}_{\varphi_I}$). Given $\varphi \subseteq \Sigma^*$, defined as SA $\mathcal{A}_\varphi = (Q, q_0, q_v, \Sigma, \rightarrow)$, input SA $\mathcal{A}_{\varphi_I} = (Q, q_0, q_v, \Sigma_I, \rightarrow_I)$ is obtained from $\mathcal{A}_\varphi$ by ignoring outputs on the transitions, i.e., for every transition $q \xrightarrow{(x,y)} q' \in \rightarrow$ where $(x, y) \in \Sigma$, there is a transition $q \xrightarrow{x} q' \in \rightarrow_I$, where $x \in \Sigma_I$. $\mathcal{L}(\mathcal{A}_{\varphi_I})$ is denoted as $\varphi_I \subseteq \Sigma_I^*$.

Example 2.3 (Example property defined as SA and its input SA). Let $I = \{A, B\}$ and $O = \{R\}$. Consider the following property: S_1*: "A and B cannot happen simultaneously, and also B and R cannot happen simultaneously".* The safety automaton in Figure 2a defines property S_1. Figure 2b presents the input SA for the SA in Figure 2a defining property S_1. Though the SA $\mathcal{A}_\varphi$ is deterministic, the input SA $\mathcal{A}_{\varphi_I}$ might be non-deterministic as is the case in Figure 2b.

Lemma 2.4. *Let $\mathcal{A}_{\varphi_I} = (Q, q_0, q_v, \Sigma_I, \rightarrow_I)$ be the input automaton obtained from $\mathcal{A}_\varphi = (Q, q_0, q_v, \Sigma, \rightarrow)$. We have the following properties:*

1 $\forall (x, y) \in \Sigma, \forall q, q' \in Q : q \xrightarrow{(x,y)} q' \Rightarrow q \xrightarrow{x}_I q'$.

2 $\forall x \in \Sigma_I, \forall q, q' \in Q : q \xrightarrow{x}_I q' \Rightarrow \exists y \in \Sigma_O : q \xrightarrow{(x,y)} q'$.

Intuitively, property 1 of Lemma 2.4 states that if there is a transition from state $q \in Q$ to state $q' \in Q$ upon input-output event $(x, y) \in \Sigma$ in the automaton $\mathcal{A}_\varphi$, then there is also a transition from state q to state q' in the input automaton $\mathcal{A}_{\varphi_I}$ upon the input event $x \in \Sigma_I$. Property 2 of Lemma 2.4 states that if there is a transition from state $q \in Q$ to state $q' \in Q$ 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 state q to state q' upon event (x, y) in the automaton $\mathcal{A}_\varphi$. Lemma 2.4 immediately follows from Definitions 2.1 and 2.2.

Edit Functions. Consider property $\varphi \subseteq \Sigma^*$, defined as SA $\mathcal{A}_\varphi = (Q, q_0, q_v, \Sigma, \rightarrow)$, and SA $\mathcal{A}_{\varphi_I} = (Q, q_0, q_v, \Sigma_I, \rightarrow_I)$ obtained from $\mathcal{A}_\varphi$ by projecting on inputs. We introduce $\text{editI}_{\varphi_\text{I}}$ (resp. editO_φ), that the enforcer uses for editing input (resp. output) events (when necessary), according to input property φ_I (resp. property φ).

- **$\text{editI}_{\varphi_\text{I}}(\sigma_I)$**: Given $\sigma_I \in \Sigma_I^*$, $\text{editI}_{\varphi_\text{I}}(\sigma_I)$ is the set of input events x in Σ_I such that the word obtained by extending σ_I with x satisfies property φ_I. Formally, $\text{editI}_{\varphi_\text{I}}(\sigma_I) = \{x \in \Sigma_I : \sigma_I \cdot x \models \varphi_I\}$.

  Considering the SA $\mathcal{A}_{\varphi_I} = (Q, q_0, q_v, \Sigma_I, \rightarrow_I)$, the set of events in Σ_I that allow to reach a state in $Q \setminus \{q_v\}$ from a state $q \in Q \setminus \{q_v\}$ is defined as:

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

  For example, consider the SA in Figure 2b obtained from the SA in Figure 2a by ignoring outputs. Let $\sigma = (10, 0) \cdot (01, 1)$, and thus $\sigma_I = 10 \cdot 01$. Then, $\text{editI}_{\varphi_\text{I}}(\sigma_I) = \Sigma_I \setminus \{11\}$. Also, $q_0 \xrightarrow{10 \cdot 01}_I q_0$, and $\text{editI}_{\mathcal{A}_{\varphi_\text{I}}}(q_0) = \Sigma_I \setminus \{11\}$.

  If $\text{editI}_{\mathcal{A}_{\varphi_\text{I}}}(q)$ is non-empty, then $\text{nondet-editI}_{\mathcal{A}_{\varphi_\text{I}}}(q)$ returns an element (chosen non-deterministically) from $\text{editI}_{\mathcal{A}_{\varphi_\text{I}}}(q)$, and is undefined if $\text{editI}_{\mathcal{A}_{\varphi_\text{I}}}(q)$ is empty.
- **$\text{editO}_\varphi(\sigma, x)$**: 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 Σ_O s.t. the input-output word obtained by extending σ with (x, y) satisfies property φ. Formally, $\text{editO}_\varphi(\sigma, x) = \{y \in \Sigma_O : \sigma \cdot (x, y) \models \varphi\}$.

  Considering the automaton $\mathcal{A}_\varphi = (Q, q_0, q_v, \Sigma, \rightarrow)$ 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 $q \in Q \setminus \{q_v\}$ with (x, y) is defined as:

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

  For example, consider property S_1 defined by the automaton in Figure 2a. We have $\text{editO}_{\mathcal{A}_\varphi}(q_0, 01) = \{0\}$.

  If $\text{editO}_{\mathcal{A}_\varphi}(q, x)$ is non-empty, $\text{nondet-editO}_{\mathcal{A}_\varphi}(q, x)$ returns an element (chosen non-deterministically) from $\text{editO}_{\mathcal{A}_\varphi}(q, x)$, and is undefined if $\text{editO}_{\mathcal{A}_\varphi}(q, x)$ is empty.

3 PROBLEM DEFINITION

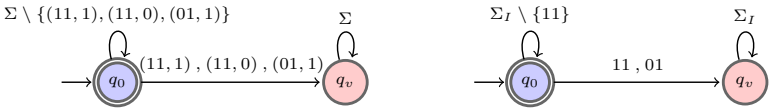
In this section, we formalize the RE problem for synchronous programs. In the setting we consider, as illustrated in Figure 1, an enforcer monitors and corrects both inputs and outputs of a synchronous program according to a given safety property $\varphi \subseteq \Sigma^*$. We assume that the "black-box" synchronous program may be invoked through a special function call called ptick, which is invoked exactly once during each reaction / synchronous step. Formally, ptick is a function from Σ_I to Σ_O that takes a bit vector $x \in \Sigma_I$ and returns a bit vector $y \in \Sigma_O$.

An enforcer for a property φ 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 editO_φ that were introduced in Section 2 that the enforcer for φ uses to edit the current input (respectively output) event according to the property φ. At an abstract level, an enforcer can be seen as a function that transforms input-output words. An enforcement function for a given property φ takes as input an input-output word over Σ and outputs an input-output word over Σ that belongs to φ.

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

Soundness

$$\forall \sigma \in \Sigma^* : E_\varphi(\sigma) \models \varphi. \qquad \textbf{(Snd)}$$



(a) SA $\mathcal{A}_{S_1}$. (b) Input SA obtained from $\mathcal{A}_{S_1}$.

Figure 2: SA (left), and its input SA (right).

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

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

Causality

$$\forall \sigma \in \Sigma^*, \forall x \in \Sigma_I, \forall y \in \Sigma_O, \exists x' \in \text{editI}_{\varphi_1}(E_\varphi(\sigma)_I), \\ \exists y' \in \text{editO}_\varphi(E_\varphi(\sigma), x') : E_\varphi(\sigma \cdot (x,y)) = E_\varphi(\sigma) \cdot (x',y'). \quad \textbf{(Cau)}$$

The input-output sequence released as output by the enforcer upon reading the input-output sequence σ is $E_\varphi(\sigma)$, and $E_\varphi(\sigma)_I \in \Sigma_I^*$ is the projection on the inputs. Note, $\text{editI}_{\varphi_1}(E_\varphi(\sigma)_I)$ returns a set of input events in Σ_I, s.t. $E_\varphi(\sigma)_I$ (which is the projection of input-output word $E_\varphi(\sigma)$ to the input alphabet) followed by any event in $\text{editI}_{\varphi_1}(E_\varphi(\sigma)_I)$ satisfies φ_I. $\text{editO}_\varphi(E_\varphi(\sigma), x')$ returns a set of output events in Σ_O, s.t. for any event y in $\text{editO}_\varphi(E_\varphi(\sigma), x')$, $E_\varphi(\sigma) \cdot (x', y)$ satisfies φ.

- Soundness (**Snd**) means that for any word $\sigma \in \Sigma^*$, the output of the enforcer $E_\varphi(\sigma)$ must satisfy φ.
- Monotonicity (**Mono**) expresses that the output of the enforcer for an extended word σ' of a word σ, extends the output produced by the enforcer for σ. The monotonicity constraint means that the enforcer cannot undo what is already released as output.
- Instantainety (**Inst**) expresses that for any given input-output word σ as input to the enforcer, the output of the enforcer $E_\varphi(\sigma)$ should contain exactly the same number of events that are in σ (i.e., E_φ is length-preserving). This means that the enforcer cannot delay, insert and suppress events. Whenever the enforcer receives a new event, it has to react instantaneously and has to produce an output event immediately.
- Transparency (**Tr**) expresses that for any given word σ and any event (x, y), if the output of the enforcer for σ (i.e., $E_\varphi(\sigma)$) followed by the event (x, y) satisfies the property φ (i.e., $E_\varphi(\sigma) \cdot (x, y) \models \varphi$), then the output that the enforcer produces for input $\sigma \cdot (x, y)$ will be $E_\varphi(\sigma) \cdot (x, y)$. This means that the enforcer makes no change when no change is needed in order to satisfy the property φ.
- Causality (**Cau**) expresses that for every input-output event (x, y) the enforcer produces input-output event (x', y') where the enforcer first processes the input part x, to produce the transformed input x' according to property φ using editI_{φ_1}. The enforcer later reads and transforms output $y \in \Sigma_O$ which is the output of the program after invoking function ptick with the transformed input x', to produce the transformed output y' using editO_φ.

Remark 2. *Let $E_\varphi(\sigma)$ be the input-output sequence released as output by the enforcer for φ after reading input-output sequence $\sigma \in \Sigma^*$. Upon reading a new event (x, y), if what has been already computed as output by the enforcer $E_\varphi(\sigma)$ followed by (x, y) does not allow to satisfy the property φ, then the enforcer edits (x, y) using functions editI_{φ_1} and editO_φ. When the current event (x, y) has to be edited, note that there may be several possible solutions. For example, consider the property S_1 introduced in Example 2.3. Let $\sigma = (10, 1) \cdot (01, 0)$, and the output of the enforcer after processing σ will be $E_\varphi(\sigma) = (10, 1) \cdot (01, 0)$. Let the new event be $(11, 0)$, and $E_\varphi(\sigma) \cdot (11, 0) \not\models \varphi$, and the enforcer has to edit the new event $(11, 0)$. Note that $E_\varphi(\sigma)_I = 10 \cdot 01$, and $\text{editI}_{\varphi_1}(10 \cdot 01) = \{00, 01, 10\}$ and the enforcer can choose any element from $\text{editI}_{\varphi_1}(10 \cdot 01)$ as the transformed input.*

Remark 3 (Enforcing bi-directional properties). *By considering two uni-directional enforcers, where one enforcer checks and transforms inputs from the environment to the program and another enforcer checks and transforms outputs from the program to the environment, bi-directional properties cannot be enforced. For example, bi-directional properties such as the property S_1 introduced in Example 2.3 cannot be enforced using two uni-directional enforcers.*

Remark 4 (When the input word provided to the enforcer satisfies φ). *Constraint* **Tr'** *expresses that when any input-output word $\sigma \in \Sigma^*$ provided as input to the enforcer satisfies the property φ, then the enforcer will not edit any event and will output σ (i.e., $E_\varphi(\sigma) = \sigma$).*

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

Lemma 3.2. **Tr ⇒ Tr'.**

Lemma 3.2 shows that **Tr'** *is a consequence of constraint* **Tr**. *For any φ, for any $\sigma \in \Sigma^*$, proof of this lemma is straightforward using induction on σ.*

Table 1: Example: Tr Vs. Tr'

σ	$E_\varphi(\sigma)$	Tr	Tr'
$(10, 1)$	$(10, 1)$	✓	✓
$(10, 1) \cdot (11, 1)$	$(10, 1) \cdot$ **(10,1)**	✓	✓
$(10, 1) \cdot (11, 1) \cdot (01, 0)$	$(10, 1) \cdot$ **(10,1)** $\cdot$ **(10,0)**	✗	✓

*Example 3.3 (***Tr** *is stronger than* **Tr'**). Via this example, we illustrate that constraint **Tr** is stronger than the alternative transparency constraint **Tr'**. Let us consider the property S_1 introduced in Example 2.3. In Table 1, first column denoted using σ shows input-output words, and the second column denoted using $E_\varphi(\sigma)$ shows the

output of the enforcer for σ, and the next two columns indicate whether $E_\varphi(\sigma)$ satisfies constraints **Tr** and **Tr'** respectively. We can see that there are situations where **Tr'** holds and **Tr** does not hold. When the enforcer reads the third event $(01, 0)$, if it edits this event to $(10, 0)$, then constraint **Tr'** holds, and constraint **Tr** does not hold since $E_\varphi((10, 1) \cdot (11, 1))$ followed by the new event read $(01, 0)$ satisfies the property S_1, and it should not be edited by the enforcer according to constraint **Tr**.

Definition 3.4 (Enforceability). Let $\varphi \subseteq \Sigma^*$ be a property. We say that φ is *enforceable* iff an enforcer E_φ for φ exists according to Definition 3.1.

Not all properties are enforceable, even if we restrict ourselves to prefix-closed safety properties, as the following example shows.

Example 3.5 (Non-enforceable safety property). We illustrate that not all prefix-closed safety properties are enforceable according to Definition 3.1. Consider the automaton in Figure 3 defining the property φ 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 Σ from q_0 leads to a non violating state q_1).

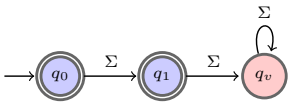


Figure 3: A non-enforceable safety property.

Note that from q_1, every event in Σ only leads to violating state q_v. Thus, when the second event $(1, 0)$ is read, every possible editing of this event will only lead to violation of the property. Upon reading the second event $(1, 0)$, releasing any event in Σ as output will violate soundness, and if no event is released as output, then the instantianety constraint will be violated.

Theorem 3.6 (Condition for enforceability). *Consider a property φ defined as SA $\mathcal{A}_\varphi = (Q, q_0, q_v, \Sigma, \rightarrow)$. Property φ is enforceable iff the following condition holds:*

$$\forall q \in Q, q \neq q_v \Rightarrow \exists (x, y) \in \Sigma : q \xrightarrow{(x,y)} q' \wedge q' \neq q_v \quad \textbf{(EnfCo)}$$

Proof of Theorem 3.6 is given in Appendix A. Note that given any property φ defined as SA $\mathcal{A}_\varphi = (Q, q_0, q_v, \Sigma, \rightarrow)$, it is straightforward to test whether $\mathcal{A}_\varphi$ satisfies condition (**EnfCo**).

Remark 5 (Transforming a non-enforceable property into an enforceable property). *Some non-enforceable properties can be made enforceable by a transformation that excludes some behaviors from the property. We illustrate this with an example. Consider the property defined by the automaton in Figure 4a. This property is not enforceable for the following reason. Suppose that the first input-output event that the enforcer receives is $(1, 1)$. Since there is a transition from q_0 to q_2 upon $(1, 1)$, the enforcer will take this transition (according to transparency constraint). Then, whatever may be the second event that the enforcer receives, note that $\mathsf{editI}_{\varphi_1}$ and editO_φ will be empty, and there is no way to correct the event and avoid reaching q_v. However, we can transform this property into an enforceable property by excluding all the paths/behaviours that are problematic. In particular, we can remove state q_2 from the automaton of Figure 4a and redirect the transition labeled $(1, 1)$ from q_0 to q_v instead. This has the effect of removing the word $(1, 1)$ from the language accepted by this automaton. The resulting automaton (shown in Figure 4b) that we obtain satisfies the condition for enforceability (**EnfCo**) and therefore the resulting new property is enforceable. Note that transforming a non-enforceable property to an enforceable one is not always possible. For instance, the non-enforceable property of Figure 3 cannot be transformed to an enforceable property.*

Transformation of non-enforceable properties. If a given safety property φ defined as automaton $\mathcal{A}_\varphi = (Q, q_0, q_v, \Sigma, \rightarrow)$ does not satisfy the condition for enforceability (**EnfCo**), then we can apply the following transformation process to check whether $\mathcal{A}_\varphi$ can be transformed in to an enforceable property (by discarding some states in $Q \setminus \{q_0\}$ in the automaton $\mathcal{A}_\varphi$). We discuss the algorithm for transformation briefly.

- For every state $q \in Q \setminus \{q_v\}$ if $\forall (x, y) \in \Sigma, q \xrightarrow{(x,y)} q_v$, then merge q with q_v (q is removed from the set of states Q and all the incoming transitions to q go to q_v instead).
- The transformation continues until one of the following two conditions hold:
  - only two states q_0 and q_v remain in Q, i.e., $Q = \{q_0, q_v\}$ such that $\forall (x, y) \in \Sigma, q_0 \xrightarrow{(x,y)} q_v$. In this case, the algorithm returns that $\mathcal{A}_\varphi$ cannot be transformed into an enforceable property.
  - $Q \setminus \{q_0, q_v\}$ is non-empty, and there is no state in $Q \setminus \{q_0, q_v\}$, that has all its outgoing transitions to q_v. In this case, the algorithm returns the resulting transformed automaton which is an enforceable property. Let $\mathsf{sub}(\mathcal{A}_\varphi)$ be the transformed automaton. Note that $\mathcal{L}(\mathsf{sub}(\mathcal{A}_\varphi)) \subseteq \mathcal{L}(\mathcal{A}_\varphi)$.

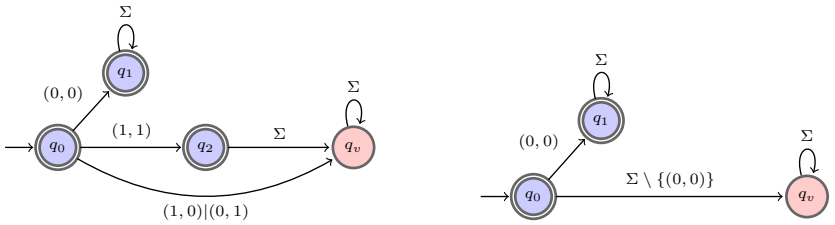
Note that the transformation algorithm is complete i.e., if $\mathcal{A}_\varphi$ can be transformed in to an enforceable property, then the transformation algorithm returns the transformed safety automaton $\mathsf{sub}(\mathcal{A}_\varphi)$. The algorithm excludes only problematic paths (behaviors) from $\mathcal{A}_\varphi$, and all good behaviors will be retained in $\mathsf{sub}(\mathcal{A}_\varphi)$ (i.e., removal of behaviors is done minimally).

4 ALGORITHM

In this section, we provide an algorithm for implementing the bi-directional synchronous enforcement problem defined in Section 3. Let the SA $\mathcal{A}_\varphi = (Q, q_0, q_v, \Sigma, \rightarrow)$ define the property φ that we want to enforce. SA $\mathcal{A}_{\varphi_I} = (Q, q_0, q_v, \Sigma_I, \rightarrow_I)$ is obtained from $\mathcal{A}_\varphi$ by projecting on inputs (see section 2).

We provide an online algorithm that requires automata A_φ and A_{φ_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 [1]. We extend the interface by including the enforcer as an intermediary between the synchronous program and its adjoining environment.

In the algorithm shown below, t keeps track of the time-step (*tick*), initialized with 0. q keeps track of the current state of both the automata $\mathcal{A}_\varphi$ and $\mathcal{A}_{\varphi_I}$. Recall that the automaton $\mathcal{A}_{\varphi_I}$ that we obtain from the automaton $\mathcal{A}_\varphi$ by projecting on inputs (see Section 2) have identical structure, and the only difference is that



(a) A non-enforceable property that can be transformed into an enforceable property.

(b) Enforceable property obtained after transformation.

Figure 4: A non-enforceable property transformed into an enforceable property.

Algorithm 1 Enforcer

1: $t \leftarrow 0$
2: $q \leftarrow q_0$
3: **while** true **do**
4: $x_t \leftarrow$ read_in_chan()
5: **if** $\exists q' \in Q : q \xrightarrow{x_t}_I q' \wedge q' \neq q_v$ **then**
6: $x'_t \leftarrow x_t$
7: **else**
8: $x'_t \leftarrow$ nondet-editI$_{\mathcal{A}_{\varphi_I}}(q)$
9: **end if**
10: ptick(x'_t)
11: $y_t \leftarrow$ read_out_chan()
12: **if** $\exists q' \in Q : q \xrightarrow{(x'_t, y_t)} q' \wedge q' \neq q_v$ **then**
13: $y'_t \leftarrow y_t$
14: **else**
15: $y'_t \leftarrow$ nondet-editO$_{\mathcal{A}_\varphi}(q, x'_t)$
16: **end if**
17: release((x'_t, y'_t))
18: $q \leftarrow q'$ where $q \xrightarrow{(x'_t, y'_t)} q' \wedge q' \neq q_v$
19: $t \leftarrow t + 1$
20: **end while**

the outputs are ignored on the transitions in the automaton $\mathcal{A}_{\varphi_I}$. 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 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 the automaton $\mathcal{A}_\varphi$ upon reading EOut. Similarly, if EOut$_I \in \Sigma_I^*$ is the sequence obtained by projecting on $x'_i s$ from EOut, q also corresponds to the state that we reach in the automaton $\mathcal{A}_{\varphi_I}$ upon reading EOut$_I$.

Functions read_in_chan (resp. read_out_chan) are functions corresponding to reading input (resp. output) channels, and function ptick corresponds to invoking the synchronous program. 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 there exists a transition in $\rightarrow_I$ from the current state q upon x_t to an accepting state in $\mathcal{A}_{\varphi_I}$. 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 nondet-editI$_{\mathcal{A}_{\varphi_I}}(q)$. Let us recall that nondet-editI$_{\mathcal{A}_{\varphi_I}}(q)$ returns an input event that leads to an accepting state in $\mathcal{A}_{\varphi_I}$ 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 there exists a transition in $\rightarrow$ from the current state q upon (x'_t, y_t) to an accepting state in $\mathcal{A}_\varphi$. In case 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 nondet-editO$_{\mathcal{A}_\varphi}(q, x'_t)$. Note that nondet-editO$_{\mathcal{A}_\varphi}(q, x'_t)$ returns an output event y'_t such that (x'_t, y'_t) leads to an accepting state in $\mathcal{A}_\varphi$ from q.

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 the automaton $\mathcal{A}_\varphi$, and the time-step t is incremented. Note that if there exists a transition $q \xrightarrow{(x'_t, y'_t)} q'$ in the SA $\mathcal{A}_\varphi$, then there also exists a transition $q \xrightarrow{x'_t}_I q'$ in the SA $\mathcal{A}_{\varphi_I}$. The current states of both the SA are always synchronized and the same at the beginning of each iteration of the algorithm.

Definition 4.1 (E^*_φ). Consider an enforceable safety property φ. We define the function $E^*_\varphi : \Sigma^* \rightarrow \Sigma^*$, 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 of the enforcement algorithm). *Given any safety property φ defined as SA $\mathcal{A}_\varphi$ that satisfies condition (**EnfCo**), the function E^*_φ defined above is an enforcer for φ, that is, it satisfies (**Snd**), (**Tr**), (**Mono**), (**Inst**), and (**Cau**) constraints of Definition 3.1.*

Proof of Theorem 4.2 is given in Appendix A.

Remark 6 (Determinism of the enforcer). *Since we consider synchronous programs, the enforcer should be deterministic. Regarding determinism, note that though $\mathcal{A}_\varphi$ is deterministic, the enforcer E^*_φ may be non-deterministic, because when the received input x (resp. output y) does not lead to an accepting state from the current state q in $\mathcal{A}_{\varphi_I}$, (resp. $\mathcal{A}_\varphi$), it is edited in step 8 (resp. step 15) of the algorithm.*

Note that editI$_{\mathcal{A}_{\varphi_I}}(q)$ *(resp.* editO$_{\mathcal{A}_{\varphi}}(q,x)$ *where* $x \in$ editI$_{\mathcal{A}_{\varphi_I}}(q)$*), may contain more than one element as illustrated via an example in Remark 2, and* nondet-editI$_{\mathcal{A}_{\varphi_I}}$ *(resp.* nondet-editO$_{\mathcal{A}_{\varphi}}$*) will choose one element from the set* editI$_{\mathcal{A}_{\varphi_I}}$ *(resp.* editO$_{\mathcal{A}_{\varphi}}$*). However, it is straightforward to make the behavior deterministic by computing* editI$_{\mathcal{A}_{\varphi_I}}(q)$ *off-line for all* $q \in Q \setminus \{q_v\}$*, and selecting one element randomly from* editI$_{\mathcal{A}_{\varphi_I}}(q)$ *and remembering the selection for each* q *by storing in a table with size* $|Q|$*. Thus, whenever in some state* q *and when the input read* x *does not lead to an accepting state from* q *(i.e, the condition tested in line 5 evaluates to false), in step 8 we check the element corresponding to the state* q *from the table and assign it to* x'*. Similarly, for event* $q \in Q \setminus \{q_v\}$*, and for all* $x \in$ editI$_{\mathcal{A}_{\varphi_I}}(q)$*, we can compute* editO$_{\mathcal{A}_{\varphi}}(q,x)$ *off-line, select one element randomly and store the selection in a table with size* $|Q \times \Sigma_I|$*. Thus, whenever in some state* q*, when* (x', y) *(where* x' *is the transformed input and* y *is the output read) does not lead to an accepting state, in step 15 we check the element corresponding to* (q, x') *from the table and assign it to* y'*.*

5 APPLICATION TO SCCHARTS

SCCharts is a Statechart dialect that has been designed for safety-critical systems and offers deterministic concurrency [20]. We implemented the algorithm presented in Section 4 in an SCCharts compilation framework[1] according to the *single-pass language-driven incremental compilation* approach [12]. Here, a safety automaton is automatically transformed into a synchronous enforcer using model-to-model transformations. The generated enforcer has three concurrent regions, one for reading and editing the inputs, one for invoking the tick function `ptick` with the edited inputs and a final one for processing and emitting the outputs. The three components exactly match the steps of the algorithm presented in Section 4.

Figure 5 depicts the example safety automaton ABO SA in SCCharts and the automatically generated Enforcer ABO Enf. In this example, A and B serve as input vector, whereas O is the only output. The automaton only has two states, the initial state q_0 and the violation state q_v. The safety property says that A and B and also B and O may not be present at the same time.

In order to evaluate this implementation we used a series of models with increasing sizes as can be seen in Table 2. To generate the mean values we simulated every model 5 times with each run consisting of 1000 ticks. As inputs for each model, a random environment was created. The whole setup was executed for two cases. Firstly, the plain model was simulated within its environment. Secondly, the same environment was used to simulate the model again with an enforcer in between. The number of enforced properties (entry "# Properties" in Table 2) range from 0 to 3 including properties that enforce inputs and also outputs (bi-directional properties). All experiments were conducted on an embedded system equipped with an 1 GHz ARM Cortex-A7 Dual-Core. Depending on the model size and the number of properties enforced, we see an increase of mean execution time between 12%-38% when simulating with an enforcer. Due to the netlist-based code generation of KIELER, there is a constant overhead because of the tick function call. Therefore, the overhead decreases percentage-wise with increasing model size. The Null model test measures the overhead of this black-box call with an enforcer with 0 safety properties. We observe a constant overhead of 0.1μs here.

As a concrete case study, we selected a pacemaker based on [8], which has been implemented in SCCharts. As second experiment we ran the Faulty Heart Model together with the Pacemaker. The results of the close-loop simulation can be seen in the last row of Table 2. Here, the Faulty Heart Model serves as environment for the Pacemaker and generates flawed pulse signals for the heart. We added an enforcer to the pacemaker to make sure that atrial and ventricular signals cannot occur simultaneously, which results in editing the input vector, and also that the pacemaker does not emit pace signals for both in return, which results in editing the output vector. We observe a mean overhead of 12% when using the enforcer.

6 RELATED WORK

Synthesizing enforcers from properties is an active area of research. Security automata proposed by Schneider [17] 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 (EA) [10] allows the enforcer to correct the input sequence by suppressing and (or) inserting events, and the RE mechanisms proposed in [6, 13] allows buffering events and releasing them upon observing a sequence that satisfies the desired property. Recently, compositionality of enforcers has been studied in [14]. These approaches focus on uni-directional RE.

Mandatory Result Automata (MRA) [5] extended EA [10], by considering bi-directional runtime enforcement. Compared to the other RE frameworks such as [6, 10, 13, 17], in MRA 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 [4], which introduces a framework to synthesize enforcers for reactive systems, called as *shields*, from a set of safety properties. In our work, we restrict to prefix-closed safety properties. The approach in [4] seems to consider more than prefix-closed properties (where properties are expressed as automata), but not all regular properties. Also, the approach in [4] has the notion of k-stabilization where the shield allows to deviate from the property for k consecutive steps whenever a property violation is unavoidable. If a second violation occurs within k steps, then the shield enters into a *fail-safe* mode, where it ensures only correctness. So, if two or more errors occur within k-steps, then the shield may generate outputs arbitrarily to satisfy the property being monitored by ignoring outputs from the system being monitored. In our approach, if the input given to the enforcer satisfies the property, then the enforcer does not modify any event. In case if a violation is noticed upon some event, the enforcer corrects it (to avoid violation), and continues to minimize deviation also for the future input events depending on the state of the enforcer and the received input event. Moreover, in [4], the shield is uni-directional,

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

[2] ABRO from [3], ABO from [20], Reactor from [18], Simple Heart Model and Pacemaker are remodeled SCCharts variants from [8], Faulty Heart Model is a variant of the Simple Heart Model with deliberately flawed pulse signals, Traffic Light from [11] (remodeled from Ptolemy Traffic Light)

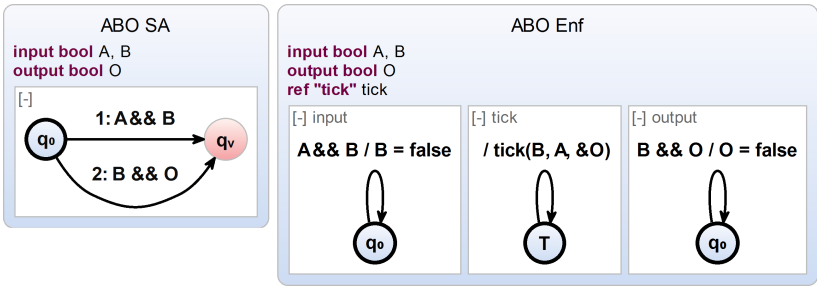


Figure 5: Example safety automaton ABO in SCCharts (left) and its automatically generated enforcer (right).

Table 2: Evaluation results.

Examples[2]	Tick (LoC)	# Properties	Enf. (LoC)	Time (μs)	Time w/ Enf. (μs)	Incr. (%)
Null	0	0	0	0.654	0.752	14.98
ABRO	23	1	21	1.208	1.565	29.55
ABO	28	1	21	0.998	1.368	37.10
Reactor	32	2	32	1.587	2.137	34.61
Faulty Heart Model	43	2	40	1.346	1.869	38.85
Simple Heart Model	76	2	40	2.175	2.825	29.86
Traffic Light	171	3	41	4.039	4.707	16.53
Pacemaker	271	2	35	7.302	8.318	13.91
FHM + Pacemaker	314	2	35	9.195	10.306	12.08

where it observes inputs from the environment and outputs from the system (program), and transforms erroneous outputs. In our work, we consider bi-directional enforcement, as explained and illustrated in Fig. 1.

Our work is also superficially related to repair of reactive programs w.r.t a specification [19], which deals with *white-box* programs. Contrary to [19], in our work, we consider the system (synchronous program) to be a black-box, and we focus on synthesis of enforcers from properties.

7 CONCLUSIONS

Synchronous observers are used to express safety properties for synchronous programs, which may be verified either statically or during runtime. This paper extends observers by proposing the concept of runtime enforcers for synchronous programs. The property to be enforced is modeled as a safety automaton, which is syntactically like an observer (expressed as an automaton with a single violation state) referring to both inputs and outputs of the synchronous program. We formalise, for the first time, the runtime enforcement synthesis problem for synchronous reactive systems. We define enforceability conditions, provide an algorithm, and prove its correctness. The synthesised enforcer interacts with a black-box synchronous program and its adjoining environment to ensure that the property in question holds during program execution. We have implemented the proposed enforcer synthesis algorithm for the SCCharts synchronous language. We highlight the applicability of the proposed approach by enforcing policies over a synchronous pacemaker model. In the near future, we will consider several extensions, including enforcement with valued inputs and outputs (valued signals), non-safety properties, and distributed enforcement.

A APPENDIX: PROOFS

Proof of Theorem 3.6. Let us recall Theorem 3.6. Consider a property φ defined as SA $\mathcal{A}_\varphi = (Q, q_0, q_v, \Sigma, \rightarrow)$[3]. Property φ is enforceable iff the condition (**EnfCo**) holds which is the following condition: $\forall q \in Q, q \neq q_v \Rightarrow \exists (x,y) \in \Sigma : q \xrightarrow{(x,y)} q' \wedge q' \neq q_v$.

We prove that:

- *Sufficient:* If condition (**EnfCo**) holds then E_φ according to Definition 3.1 exists.

  Due to condition (**EnfCo**), whatever may be the current state $q \in Q \setminus \{q_v\}$ of the enforcer, there is at least one possibility to correct the event that it receives when in state q (in case if the received event leads to q_v from q). That is, due to condition (**EnfCo**), $\forall q \in Q \setminus \{q_v\}$, we know for sure that $\text{editI}_{\mathcal{A}_{\varphi_I}}(q)$ will be non-empty, and $\forall q \in Q \setminus \{q_v\}, \forall x \in \text{editI}_{\mathcal{A}_{\varphi_I}}(q) : \text{editO}_{\mathcal{A}_\varphi}(q,x)$ will be also non-empty.

  For any property φ defined as SA $\mathcal{A}_\varphi$, the enforcement function E^*_φ (Definition 4.1) is an enforcer for φ which satisfies all the constraints according to Definition 3.1. Theorem 4.2 shows that for any property φ (defined as SA $\mathcal{A}_\varphi$) that satisfies the condition for enforceability (**EnfCo**), the enforcement function E^*_φ (Definition 4.1) is an enforcer for φ, that is, it satisfies (**Snd**), (**Tr**), (**Mono**), (**Inst**), and (**Cau**) constraints of Definition 3.1.
- *Necessary:* If E_φ according to Definition 3.1 exists, then condition (**EnfCo**) holds.

  Suppose that an enforcer E_φ for φ according to Definition 3.1 exists and assume that condition (**EnfCo**) does not hold for $\mathcal{A}_\varphi$.

[3]Note that we consider that $\mathcal{A}_\varphi$ is deterministic and complete, and Q does not contain any (redundant) locations that are not reachable from q_0 in 1 or more steps.

Since condition (**EnfCo**) does not hold, $\exists q \in Q \setminus \{q_v\} : \forall (x,y) \in \Sigma : q \xrightarrow{(x,y)} q_v$, i.e., there exists a location $q \in Q \setminus \{q_v\}$ such that all the outgoing transitions from q go to q_v.

Since all the locations in Q 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 accepting location q (which has all its outgoing transitions to q_v) from the initial location q_0.

If σ is the input word to the enforcer, then due to constraint (**Tr**), it cannot edit any event in σ, and the enforcer produces σ as output and reaches location q. When in location q, upon receiving any event $(x,y) \in \Sigma$, the enforcer has no possibility to correct it, since every event in Σ leads to q_v from q (i.e., since $\forall (x,y) \in \Sigma : q \xrightarrow{(x,y)} q_v$).

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 $\text{editI}_{\varphi_\text{I}}()$ and $\text{editO}_\varphi()$ from location q will be empty, violating constraints (**Inst**) and (**Cau**). Thus, our assumption is false and condition (**EnfCo**) holds for $\mathcal{A}_\varphi$.

□

Proof of Theorem 4.2. Let us recall the condition for enforceability: A property φ defined as SA $\mathcal{A}_\varphi = (Q, q_0, q_v, \Sigma, \rightarrow)$ is enforceable iff $\forall q \in Q, q \neq q_v \Rightarrow \exists (x,y) \in \Sigma : q \xrightarrow{(x,y)} q' \wedge q' \neq q_v$.

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 the input automaton $\mathcal{A}_{\varphi_I} = (Q, q_0, q_v, \Sigma_I, \rightarrow_I)$ is obtained from $\mathcal{A}_\varphi$ by projecting on inputs (See Definition 2.2, Section 2).

We shall prove that given any safety property φ defined as SA $\mathcal{A}_\varphi$ that satisfies condition (**EnfCo**), the function E^*_φ is an enforcer for φ, that is, it satisfies (**Snd**), (**Tr**), (**Mono**), (**Inst**), and (**Cau**) 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/iterations of 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 σ, i.e., $E^*_\varphi(\sigma)$ satisfies the (**Snd**), (**Tr**), (**Mono**), (**Inst**), and (**Cau**) 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 σ 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 the automaton $\mathcal{A}_{\varphi_I}$ upon $E^*_\varphi(\sigma)_I$. Note that the current state q in Algorithm 1 can never be 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 is a transition in the automaton $\mathcal{A}_\varphi$ from the current state q upon (x_{k+1},y_{k+1}) to an accepting state.

- $\exists q' \in Q : q \xrightarrow{(x_{k+1},y_{k+1})} q' \wedge q' \neq q_v$.

  In Algorithm 1, the condition tested in step 5 will evaluate to true since from Lemma 2.4, in $\mathcal{A}_{\varphi_I}$ we will have $\exists q' \in Q : q \xrightarrow{x_{k+1}}_I q' \wedge q' \neq q_v$, and thus $x'_{k+1} = x_{k+1}$.

  Also, the condition tested in step 12 will evaluate to true in this case since $\exists q' \in Q : q \xrightarrow{(x_{k+1},y_{k+1})} q' \wedge q' \neq q_v$, 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}) satisfies the property φ, and thus constraint (**Snd**) holds. 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 σ 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 φ 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 (**Cau**), in this case from the induction hypothesis, from the definitions of $\text{editI}_{\mathcal{A}_{\varphi_\text{I}}}$ and $\text{editO}_{\mathcal{A}_\varphi}$ we have $x_{k+1} \in \text{editI}_{\mathcal{A}_{\varphi_\text{I}}}(q)$, and also $y_{k+1} \in \text{editO}_{\mathcal{A}_\varphi}(q, x_{k+1})$.

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

- $\nexists q' \in Q : q \xrightarrow{(x_{k+1},y_{k+1})} q' \wedge q' \neq q_v$.

  In this case, we have two sub-cases, based on whether $\exists q' \in Q : q \xrightarrow{x_{k+1}}_I q' \wedge q' \neq q_v$ in $\mathcal{A}_{\varphi_I}$.

  - $\exists q' \in Q : q \xrightarrow{x_{k+1}}_I q' \wedge q' \neq q_v$.

    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 $\nexists q' \in Q : q \xrightarrow{(x_{k+1},y_{k+1})} q' \wedge q' \neq q_v$. 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. It is important to notice that $\text{editO}_{\mathcal{A}_\varphi}(q, x_{k+1})$ will be non-empty in this case since we know for sure that $\exists y'_{k+1} \in \Sigma_O, q' \in Q : q \xrightarrow{(x_{k+1},y'_{k+1})} q' \wedge q' \neq q_v$ (from the condition for enforceability (**EnfCo**), hypothesis ($q \neq q_v$), definition of $\text{editO}_{\mathcal{A}_\varphi}$, and Lemma 2.4). 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}) satisfies property φ, 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 (**Cau**), 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 (**Cau**) holds in this case.

- $\nexists q' \in Q : q \xrightarrow{x_{k+1}}_I q' \wedge q' \neq q_v$.
  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 2.4, we know for sure that $\exists x' \in \Sigma_I, q' \in Q : q \xrightarrow{x'_{k+1}} q' \wedge q' \neq q_v$ in the automaton $\mathcal{A}_{\varphi_I}$. 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 if $\exists q' \in Q : q \xrightarrow{(x'_{k+1}, y_{k+1})} q' \wedge q' \neq q_v$ or not.
  - $\exists q' \in Q : q \xrightarrow{(x'_{k+1}, y_{k+1})} q' \wedge q' \neq q_v$.
    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., $\exists q' \in Q : q \xrightarrow{(x'_{k+1}, y_{k+1})} q' \wedge q' \neq q_v$), we know that $E^*_\varphi(\sigma)$ followed by the new input-output event released as output (x'_{k+1}, y_{k+1}) satisfies the property φ, 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 (**Cau**), 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 $\text{editO}_{\mathcal{A}_\varphi}$.
  - $\nexists q' \in Q : q \xrightarrow{(x'_{k+1}, y_{k+1})} q' \wedge q' \neq q_v$.
    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 $\exists y'_{k+1} \in \Sigma_O, q' \in Q : q \xrightarrow{(x'_{k+1}, y'_{k+1})} q' \wedge q' \neq q_v$ (from the enforceability condition, definitions, and Lemma 2.4). 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 $\text{editI}_{\mathcal{A}_{\varphi_\text{I}}}(q_I)$ 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 $\text{editI}_{\mathcal{A}_{\varphi_\text{I}}}$ and $\text{editO}_{\mathcal{A}_\varphi}$, we know that $E_\varphi(\sigma) \cdot (x'_{k+1}, y'_{k+1})$ satisfies the property φ 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 (**Cau**), we already discussed that $\text{editI}_{\mathcal{A}_{\varphi_\text{I}}}(q)$ is non-empty, $x'_{k+1} \in \text{editI}_{\mathcal{A}_{\varphi_\text{I}}}(q)$, 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 (**Cau**) holds.

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

□

REFERENCES

[1] Charles Andre, Frédéric Boulanger, and Alain Girault. 2001. Software implementation of synchronous programs. In *ACSD, 2001*. IEEE, 133–142.

[2] 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.

[3] Gérard Berry. 2000. *The Esterel v5 Language Primer, Version v5_91.* Centre de Mathématiques Appliquées Ecole des Mines and INRIA, 06565 Sophia-Antipolis.

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

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

[6] 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.

[7] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. 1994. Synchronous observers and the verification of reactive systems. In *Algebraic Methodology and Software Technology (AMAST'93)*. Springer, 83–96.

[8] Zhihao Jiang, Miroslav Pajic, Salar Moarref, Rajeev Alur, and Rahul Mangharam. 2012. Modeling and verification of a dual chamber implantable pacemaker. In *TACAS*. Springer, 188–203.

[9] Martin Leucker and Christian Schallhart. 2009. A Brief Account of Runtime Verification. *Journal of Logic and Algebraic Programming* 78, 5 (2009), 293–303.

[10] 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.

[11] Christian Motika, Hauke Fuhrmann, Reinhard von Hanxleden, and Edward A. Lee. 2012. *Executing Domain-Specific Models in Eclipse.* Technical Report 1214. Christian-Albrechts-Universität zu Kiel. ISSN 2192-6247.

[12] Christian Motika, Steven Smyth, and Reinhard von Hanxleden. 2014. Compiling SCCharts–A Case-Study on Interactive Model-Based Compilation. In *ISoLA (LNCS)*, Vol. 8802. Corfu, Greece, 443–462.

[13] 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.

[14] Srinivas Pinisetty and Stavros Tripakis. 2016. Compositional Runtime Enforcement. In *NASA Formal Methods Symposium, NFM 2016, Minneapolis, MN, USA*. Springer, 82–99.

[15] Pascal Raymond, Xavier Nicollin, Nicolas Halbwachs, and Daniel Weber. 1998. Automatic testing of reactive systems. In *Real-Time Systems Symposium*. IEEE, 200–209.

[16] John Rushby. 2014. The versatile synchronous observer. In *Specification, Algebra, and Software*. Springer, 110–128.

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

[18] Claus Traulsen, Torsten Amende, and Reinhard von Hanxleden. 2011. Compiling SyncCharts to Synchronous C. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE'11)*. IEEE, Grenoble, France, 563–566.

[19] 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

[20] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. 2014. SCCharts: Sequentially Constructive Statecharts for Safety-critical Applications. In *PLDI*. ACM, NY, USA, 372–383.