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

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.

*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.
We consider bi-directional RE of synchronous programs, and the general context is illustrated in Figure 1. Here, \{i_1, i_2, \ldots, i_n\} are inputs from the environment to the enforcer, \{i'_1, i'_2, \ldots, i'_m\} are transformed inputs from the enforcer to the program, \{o_1, o_2, \ldots, o_m\} are outputs of the program to the enforcer, and \{o'_1, o'_2, \ldots, o'_n\} 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 \( \phi \). 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 \( \phi \), 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 \( \phi \), these constraints also ensure that input/output events are edited minimally (to avoid violation of \( \phi \)) 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).
that \( q \xrightarrow{a} q'' \) 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 \( A = (Q, q_0, q_f, \Sigma, \rightarrow) \) is deterministic if \( \forall q \in Q, \forall a \in \Sigma, (q \xrightarrow{a} q' \land q \xrightarrow{a} q'') \Rightarrow (q' = q'') \). \( A \) is complete if \( \forall q \in Q, \forall a \in \Sigma, \exists q' \in Q, q \xrightarrow{a} q' \). A word \( \sigma \) is accepted by \( A \) if there exists \( q \in Q \setminus \{q_f\} \) such that \( q_0 \xrightarrow{\sigma} q \). The set of all words accepted by \( A \) is denoted by \( L(A) \).

Remark 1. In the rest of this paper, \( \varphi \) is a safety property defined as deterministic and complete SA \( A_\varphi = (Q, q_0, q_f, \Sigma, \rightarrow) \). If the user provides a non-deterministic or incomplete automaton, we determine 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 \( \varphi \) defined as SA \( A_\varphi \). We thus need to consider the input property that we obtain from \( A_\varphi \) by projecting on inputs.

Definition 2.2 (Input SA, \( A_{\varphi I} \)). Given \( \Sigma \subseteq \Sigma^* \), defined as SA \( A_\varphi = (Q, q_0, q_f, \Sigma, \rightarrow) \), input SA \( A_{\varphi I} = (Q, q_0, q_f, \Sigma_I, \rightarrow I) \) is obtained from \( A_\varphi \) 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 \( x \in \Sigma_I \). \( L(A_{\varphi I}) \) 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 \( O = \{\} \). Consider the following property: \( S_1: \text{"}A \text{ and } B \text{ cannot happen simultaneously, and also } B \text{ and } R \text{ cannot happen simultaneously\"} \). 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 \( A_{\varphi I} \) is deterministic, the input SA \( A_{\varphi I} \) might be non-deterministic as is the case in Figure 2b.

Lemma 2.4. Let \( A_{\varphi I} = (Q, q_0, q_f, \Sigma_I, \rightarrow I) \) be the input automaton obtained from \( A_\varphi = (Q, q_0, q_f, \Sigma, \rightarrow) \) we have the following properties:

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

Intuitively, property 1 of Lemma 2.4 states that if there is a transition from state \( q \in Q \) to state \( q' \in Q \), the input \( (x, y) \in \Sigma \) is accepted in the automaton \( A_\varphi \), then there is also a transition from state \( q \) to state \( q' \) in the input automaton \( A_{\varphi I} \), 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 \( y \in \Sigma_O \) such that it is a transition from state \( q \) to state \( q' \) upon event \( (x, y) \) in the automaton \( A_\varphi \).

Lemma 2.4 immediately follows from Definitions 2.1 and 2.2.

Definition 3.1 (Enforcer for \( \varphi \)). Given \( \varphi \subseteq \Sigma^* \), defined as SA \( A_\varphi = (Q, q_0, q_f, \Sigma, \rightarrow) \), and SA \( A_{\varphi I} = (Q, q_0, q_f, \Sigma_I, \rightarrow I) \) obtained from \( A_\varphi \) by projecting on inputs. We introduce edit\( _{\varphi I}(\sigma_I) \) (resp. edit\( _{\varphi O}(\sigma_O) \)), that the enforcer uses for editing input (resp. output) events (when necessary), according to input property \( \varphi_I \) (resp. property \( \varphi \)).

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, edit\( _{\varphi_I}(\sigma_I) = \Sigma_I \setminus \{11\} \). Also, \( q_0 \xrightarrow{10 \cdot 01} q_0 \), and edit\( _{\varphi_I}(q_0) = \Sigma_I \setminus \{11\} \).

If \( \text{edit}_{\varphi_I}(\varphi) \) is non-empty, then nondet-edit\( _{\varphi_I}(\varphi) \) returns an element (chosen non-deterministically) from edit\( _{\varphi_I}(\varphi) \), and is undefined if edit\( _{\varphi_I}(\varphi) \) is empty.

• edit\( _{\varphi I}(\sigma_I) \): Given an input-output word \( \sigma \in \Sigma^* \) and an input event \( x \in \Sigma_I \), edit\( _{\varphi I}(\sigma, x) \) is the set of output events \( y \in \Sigma_O \) s.t. the input-output word obtained by extending \( \sigma \) with \( (x, y) \) satisfies property \( \varphi \). Formally, edit\( _{\varphi I}(\sigma, x) = \{ y \in \Sigma_O : \sigma \cdot (x, y) \models \varphi \} \).

Considering the automaton \( A_\varphi = (Q, q_0, q_f, \Sigma, \rightarrow) \) defining property \( \varphi \), and an input event \( x \in \Sigma_I \), the set of output events \( y \in \Sigma_O \) that allow to reach a state in \( Q \setminus \{q_f\} \) from a state \( q \in Q \setminus \{q_f\} \) with \((x, y)\) is defined as:

\[ \text{edit}_{\varphi I}(\sigma, x) = \{ y \in \Sigma_O : \sigma \cdot (x, y) \models \varphi \} \]

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

If \( \text{edit}_{\varphi O}(\sigma, x) \) is non-empty, nondet-edit\( _{\varphi I}(\varphi, x) \) returns an element (chosen non-deterministically) from edit\( _{\varphi I}(\varphi, x) \), and is undefined if edit\( _{\varphi I}(\varphi, x) \) 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 tick, which is invoked exactly once during each reaction / synchronous step. Formally, tick is a function from \( \Sigma_I \) to \( \Sigma_O \) that takes a bit vector \( x \in \Sigma_I \) and returns a bit vector \( y \in \Sigma_O \).

An enforcer for a property \( \varphi \) can only edit an input-output event when necessary, and it cannot block, delay or suppress events. Let us recall the two functions edit\( _{\varphi I} \) and edit\( _{\varphi O} \) that were introduced in Section 2 that the enforcer for \( \varphi \) uses to edit the current input (respectively output) event according to the property \( \varphi \). At an abstract level, an enforcer can be seen as a function that transforms input-output words. An enforcement function for a given property \( \varphi \) takes as input an input-output word over \( \Sigma \) and outputs an input-output word over \( \Sigma \) that belongs to \( \varphi \).

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

**Soundness**

\[ \forall \sigma \in \Sigma^* : E_{\varphi}(\sigma) \models \varphi. \] (Snd)
Monotonicity

\[ \forall \sigma, \sigma' \in \Sigma^* : \sigma \preceq \sigma' \Rightarrow E_\varphi(\sigma) \preceq E_\varphi(\sigma'). \]  
(Mono)

Instantaneity

\[ \forall \sigma \in \Sigma^* : |\sigma| = |E_\varphi(\sigma)|. \]  
(Inst)

Transparency

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

Causality

\[ \forall \sigma \in \Sigma^*, \forall x \in \Sigma_I, \forall y \in \Sigma_O, \exists x' \in \text{edit}_I(\varphi(E_\varphi(\sigma))) \ni \exists y' \in \text{edit}_O(\varphi(E_\varphi(\sigma)), x') : E_\varphi(\sigma \cdot (x, y)) = E_\varphi(\sigma \cdot (x', y')). \]  
(Cau)

The input-output sequence released as output by the enforcer upon reading the input-output sequence \( \sigma \) is \( E_\varphi(\sigma) \), and \( E_\varphi(\sigma) \) is the projection on the inputs. Note, \( \text{edit}_\varphi(E_\varphi(\sigma)) \) returns a set of input events in \( \Sigma_I \), s.t. \( E_\varphi(\sigma) \) (which is the projection of input-output word \( E_\varphi(\sigma) \) to the input alphabet) followed by any event in \( \text{edit}_\varphi(E_\varphi(\sigma)) \) satisfies \( \varphi \). \( \text{edit}_O(\varphi(E_\varphi(\sigma)), x') \) returns a set of output events in \( \Sigma_O \), s.t. for any event \( y \) in \( \text{edit}_\varphi(E_\varphi(\sigma), x') \), \( E_\varphi(\sigma) \cdot (x', y) \) satisfies \( \varphi \).

- **Soundness (Snd)** means that for any word \( \sigma \in \Sigma^* \), the output of the enforcer \( E_\varphi(\sigma) \) must satisfy \( \varphi \).
- **Monotonicity (Mono)** expresses that the output of the enforcer for an extended word \( \sigma' \) of a word \( \sigma \), extends the output produced by the enforcer for \( \sigma \). The monotonicity constraint means that the enforcer cannot undo what is already released as output.
- **Instantaneity (Inst)** expresses that for any given input-output word \( \sigma \) as input to the enforcer, the output of the enforcer \( E_\varphi(\sigma) \) should contain exactly the same number of events that are in \( \sigma \) (i.e., \( E_\varphi \) 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 \( \sigma \) and any event \( (x, y) \), if the output of the enforcer for \( \sigma \) (i.e., \( E_\varphi(\sigma) \)) followed by the event \( (x, y) \) satisfies the property \( \varphi \) (i.e., \( E_\varphi(\sigma \cdot (x, y)) = \varphi \)), 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 \( \varphi \).
- **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 \(\varphi\) using \(\text{edit}_I(\varphi)\). The enforcer later reads and transforms output \(y \in \Sigma_O\) which is the output of the program after invoking function \text{ptick} with the transformed input \(x'\), to produce the transformed output \(y'\) using \(\text{edit}_O(\varphi)\).

**Remark 2.** Let \( E_\varphi(\sigma) \) be the input-output sequence released as output by the enforcer for \( \varphi \) 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 \( \varphi \), then the enforcer edits \((x, y)\) using functions \(\text{edit}_I(\varphi)\) and \(\text{edit}_O(\varphi)\). 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 \( \sigma = (10, 1) \cdot (10, 0) \), and the output of the enforcer after processing \( \sigma \) will be \( E_\varphi(\sigma) = (10, 1) \cdot (10, 0) \). Let the new event be \((11, 0)\), and \( E_\varphi(\sigma \cdot (11, 0)) \) does not satisfy \( \varphi \), and the enforcer has to edit the new event \((11, 0)\). Note that \( E_\varphi(\sigma \cdot (11, 0)) = (10, 0, 11) \), and the enforcer can choose any element from \(\text{edit}_O(10, 0, 11)\) 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 \( \varphi \).)** Constraint \( \text{Tr}' \) expresses that when any input-output word \( \sigma \in \Sigma^* \) provided as input to the enforcer satisfies the property \( \varphi \), then the enforcer will not edit any event and will output \( \sigma \) (i.e., \( E_\varphi(\sigma) = \sigma \)).

\[ \forall \sigma \in \Sigma^* : E_\varphi(\sigma) \models \varphi \Rightarrow E_\varphi(\sigma) = \sigma. \]  
(Tr')

**Lemma 3.2.** \( \text{Tr}' \) is a consequence of constraint \( \text{Tr} \). For any \( \varphi \), for any \( \sigma \in \Sigma^* \), proof of this lemma is straightforward using induction on \( \sigma \).

**Table 1: Example: Tr Vs. Tr'**

<table>
<thead>
<tr>
<th>( \sigma )</th>
<th>( E_\varphi(\sigma) )</th>
<th>( \text{Tr} )</th>
<th>( \text{Tr}' )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (10, 1) )</td>
<td>(10, 1)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>( (11, 1) )</td>
<td>(10, 1)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>( (10, 1) \cdot (10, 0) )</td>
<td>(10, 1)</td>
<td>(10, 0)</td>
<td>✓</td>
</tr>
<tr>
<td>( (10, 1) \cdot (11, 1) )</td>
<td>(10, 1)</td>
<td>(10, 1)</td>
<td>✓</td>
</tr>
<tr>
<td>( (10, 1) \cdot (10, 0) )</td>
<td>(10, 1)</td>
<td>(10, 0)</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Example 3.3 (Tr is stronger than Tr').** Via this example, we illustrate that constraint \( \text{Tr} \) is stronger than the alternative transparency constraint \( \text{Tr}' \). Let us consider the property \( S_1 \) introduced in Example 2.3. In Table 1, first column denoted using \( \sigma \) shows input-output words, and the second column denoted using \( E_\varphi(\sigma) \) shows the
output of the enforcer for $\sigma$, and the next two columns indicate whether $E_{\Phi}(\sigma)$ satisfies constraints $Tr$ and $Tr'$ respectively. We can see that there are situations where $Tr'$ holds and $Tr$ does not. When the enforcer reads the third event (0,1), if it edits this event to (0,0), then constraint $Tr'$ holds, and constraint $Tr$ does not hold since $E_{\Phi}(10,1) \cdot (11,1)$ 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 $\Phi \subseteq \Sigma^*$ be a property. We say that $\Phi$ is enforceable iff an enforcer $E_{\Phi}$ for $\Phi$ 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 $\Phi$ that we want to enforce, with $I = \{A\}$, $O = \{B\}$ and $\Sigma = \Sigma_Y \times \Sigma_O$. Let the input-output sequence provided as input to the enforcer be $\sigma = (1,1) \cdot (1,0)$. When the enforcer reads the first event (1,1), it can output (1,1) (since every event in $\Sigma$ from $q_0$ leads to a non-violating state $q_1$).

![Figure 3: A non-enforceable safety property.](image)

Note that from $q_1$, every event in $\Sigma$ only leads to violating state $q_2$. Thus, when the second event (1,0) is read, every possible editing of this event will only lead to violation of the property. Upon reading the second event (1,0), releasing any event in $\Sigma$ as output will violate soundness, and if no event is released as output, then the instantaneous constraint will be violated.

**Theorem 3.6 (Condition for enforceability).** Consider a property $\Phi$ defined as $SA A_\Phi = (\Sigma, q_0, q_\Phi, \Sigma, \rightarrow)$. Property $\Phi$ is enforceable iff the following condition holds:

$$\forall q \in Q, q \neq q_\Phi \Rightarrow \exists (x, y) \in \Sigma : q \xrightarrow{(x, y)} q' \land q' \neq q_\Phi \quad (\text{EnfCo})$$

Proof of Theorem 3.6 is given in Appendix A. Note that given any property $\Phi$ defined as $SA A_\Phi = (Q, q_0, q_\Phi, \Sigma, \rightarrow)$, it is straightforward to test whether $A_\Phi$ 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 edit$_{in}$ and edit$_{out}$ will be empty, and there is no way to correct the event and avoid reaching $q_2$. However, we can transform this property into an enforceable property by excluding all the paths/behaviors that are problematic. In particular, we can remove state $q_0$ from the automaton of Figure 4a and redirect the transition labeled (1,1) from $q_0$ to $q_1$ 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.

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

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

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

## 4 ALGORITHM

In this section, we provide an algorithm for implementing the bidirectional synchronous enforcement problem defined in Section 3. Let the SA $A_\Phi = (Q, q_0, q_\Phi, \Sigma, \rightarrow)$ define the property $\Phi$ that we want to enforce. SA $A_{\Phi_l} = (Q, q_0, q_\Phi, \Sigma_1, \rightarrow_l)$ is obtained from $A_\Phi$ by projecting on inputs (see section 2).

We provide an online algorithm that requires automata $A_\Phi$ and $A_{\Phi_l}$ 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 $A_\Phi$ and $A_{\Phi_l}$. Recall that the automaton $A_{\Phi_l}$ that we obtain from the automaton $A_\Phi$ by projecting on inputs (see Section 2) have identical structure, and the only difference is that
Figure 4: A non-enforceable property transformed into an enforceable property.

Algorithm 1 Enforcer

1: t ← 0
2: q ← q₀
3: while true do
4:    xₜ ← read_in_chan()
5:    if ∃q' ∈ Q : q →ₜ xₜ q' ∧ q' ≠ qᵥ then
6:       xₜ' ← xₜ
7:    else
8:       xₜ' ← nondet-edit₁ₐₚ(q)
9:    end if
10:   ptick(xₜ')
11:   yₜ ← read_out_chan()
12:   if ∃q' ∈ Q : (xₜ', yₜ) →ₜ q' ∧ q' ≠ qᵥ then
13:      yₜ' ← yₜ
14:   else
15:      yₜ' ← nondet-edit₀₁ₐₚ(q, xₜ')
16:   end if
17:   release((xₜ', yₜ'))
18:   q ← q' where (xₜ', yₜ') →ₜ q' ∧ q' ≠ qᵥ
19:   t ← t + 1
20: end while

the inputs and outputs are independent of the current state of the automaton A₁ₚ.
Note that at the beginning of each iteration of the algorithm, the current states of both the automata Aₚ and A₁ₚ are the same (where both are initialized with q₀). At t, if EOut ∈ Σ* 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 Aₚ upon reading EOut. Similarly, if EOut₁ ∈ Σ* is the sequence obtained by projecting on xₜ's from EOut, q also corresponds to the state that we reach in the automaton A₁ₚ upon reading EOut₁.

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 xₜ. Then the algorithm tests whether there exists a transition in →₁ from the current state q upon xₜ to an accepting state in A₁ₚ. In case if this test succeeds, then it is not necessary to edit the input event xₜ, and the transformed input xₜ' is assigned to xₜ. Otherwise, xₜ' is assigned with the output of nondet-edit₁ₐₚ(q). Let us recall that nondet-edit₁ₐₚ(q) returns an input event that leads to an accepting state in A₁ₚ, from q.

After transforming the input xₜ according to A₁ₚ, the program is invoked with the transformed input xₜ' using function ptick. Afterwards, all the output channels are read using function read_out_chan and the output event is assigned to yₜ. Then the algorithm tests whether there exists a transition in → from the current state q upon (xₜ', yₜ) to an accepting state in Aₚ. In case if this test succeeds, then it is not necessary to edit the output event yₜ, and the transformed output yₜ' is assigned to yₜ. Otherwise, yₜ' is assigned with the output of nondet-edit₀₁ₐₚ(q, xₜ'). Note that nondet-edit₀₁ₐₚ(q, xₜ') returns an output event yₜ' such that (xₜ', yₜ') leads to an accepting state in Aₚ, from q.

Before proceeding with the next iteration, current state q is updated to q' which is the state reached upon (xₜ', yₜ') from state q in the automaton Aₚ, and the time-step t is incremented. Note that if there exists a transition (xₜ', yₜ') in the SA Aₚ, then there also exists a transition q →ₜ₁ q' in the SA A₁ₚ. 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 (Eₚ). Consider an enforceable safety property φ. We define the function Eₚ(φ) : Σ* → Σ*, where Σ = SA × SA, as follows. Let σ = (x₁, y₁) · · · (xk, yk) ∈ Σ* be a word received by Algorithm 1. Then we let Eₚ(σ) = (xₐ₁, yₐ₁) · · · (xₐk, yₐk), where (xₐ₁, yₐ₁) is the pair of events output by Algorithm 1 in Step 17, for t = 1, . . . , k.

Theorem 4.2 (Correctness of the Enforcement Algorithm). Given any safety property φ defined as SA Aₚ that satisfies condition (EnfCo), the function Eₚ defined above is an enforce 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 Aₚ is deterministic, the enforcer Eₚ 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 A₁ₚ, (resp. Aₚ), it is edited in step 8 (resp. step 15) of the algorithm.
Note that $\text{edit}_{\mathcal{P}}(\mathcal{N}_{\mathcal{P}}(q), x)$ (resp. $\text{edit}O_{\mathcal{P}}(q, x)$) where $x \in \text{edit}_{\mathcal{P}}(\mathcal{N}_{\mathcal{P}}(q))$, may contain more than one element as illustrated via an example in Remark 2, and nonet-$\text{edit}_{\mathcal{P}}(\mathcal{N}_{\mathcal{P}}(q))$ (resp. nonet-$\text{edit}O_{\mathcal{P}}(q)$) will choose one element from the set $\text{edit}_{\mathcal{P}}(\mathcal{N}_{\mathcal{P}}(q))$ (resp. $\text{edit}O_{\mathcal{P}}(q)$). However, it is straightforward to make the behavior deterministic by computing $\text{edit}_{\mathcal{P}}(\mathcal{N}_{\mathcal{P}}(q))$ off-line for all $q \in Q \setminus \{q_s\}$, and selecting one element randomly from $\text{edit}_{\mathcal{P}}(\mathcal{N}_{\mathcal{P}}(q))$ 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 $e \in Q \setminus \{q_s\}$, and for all $x \in \text{edit}_{\mathcal{P}}(\mathcal{N}_{\mathcal{P}}(q))$, we can compute $\text{edit}O_{\mathcal{P}}(q, x)$ 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 framework1 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 $\text{tick}$ 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 $\text{ABO} \text{SA}$ in SCCharts and the automatically generated Enforcer $\text{ABO} \text{Enf}$. 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 does 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.

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

2ABRO 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 Polenky Traffic Light)
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 concept of runtime enforcers for synchronous programs. The property to be enforced is 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 $\varphi$ defined as $\mathcal{A}_{\varphi} = (Q, q_0, q_V, \Sigma, \rightarrow)^7$. Property $\varphi$ is enforceable if the condition (EnfCo) holds which is the following condition: $\forall q \in Q, q \neq q_V \Rightarrow \exists (x, y) \in \Sigma : q \xrightarrow{\varphi} q' \wedge q' \neq q_V$.

We prove that:

- Sufficient: If condition (EnfCo) holds then $E_{\varphi}$ according to Definition 3.1 exists.

Due to condition (EnfCo), whatever may be the current state $q \in Q \setminus \{q_V\}$ of the 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 edit$_I\mathcal{A}_{\varphi}(q)$ will be non-empty, and $\forall q \in Q \setminus \{q_V\}, \forall x \in \text{edit}_I\mathcal{A}_{\varphi}(q), \text{edit}_O\mathcal{A}_{\varphi}(q, x)$ will be also non-empty.

For any property $\varphi$ defined as $\mathcal{A}_{\varphi}$, the enforcement function $E_{\varphi}^w$ (Definition 4.1) is an enforcer for $\varphi$ which satisfies all the constraints according to Definition 3.1. Theorem 4.2 shows that for any property $\varphi$ (defined as $\mathcal{A}_{\varphi}$) that satisfies the condition for enforceability (EnfCo), the enforcement function $E_{\varphi}$ (Definition 4.1) is an enforcer for $\varphi$, that is, it satisfies (Snd), (Tr), (Mono), and (Cau) constraints of Definition 3.1.

- Necessary: If $E_{\varphi}$ according to Definition 3.1 exists, then condition (EnfCo) holds.

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

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.

---

Table 2: Evaluation results.

<table>
<thead>
<tr>
<th>Examples</th>
<th>Tick (LoC)</th>
<th># Properties</th>
<th>Enf. (LoC)</th>
<th>Time ((\mu s))</th>
<th>Time w/ Enf. ((\mu s))</th>
<th>Incr. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.654</td>
<td>0.752</td>
<td>14.98</td>
</tr>
<tr>
<td>ABRO</td>
<td>23</td>
<td>1</td>
<td>21</td>
<td>1.208</td>
<td>1.565</td>
<td>29.35</td>
</tr>
<tr>
<td>ABO</td>
<td>28</td>
<td>1</td>
<td>21</td>
<td>0.998</td>
<td>1.368</td>
<td>37.10</td>
</tr>
<tr>
<td>Reactor</td>
<td>32</td>
<td>2</td>
<td>32</td>
<td>1.587</td>
<td>2.137</td>
<td>34.61</td>
</tr>
<tr>
<td>Faulty Heart Model</td>
<td>43</td>
<td>2</td>
<td>40</td>
<td>1.346</td>
<td>1.869</td>
<td>38.85</td>
</tr>
<tr>
<td>Simple Heart Model</td>
<td>76</td>
<td>2</td>
<td>40</td>
<td>2.175</td>
<td>2.825</td>
<td>29.86</td>
</tr>
<tr>
<td>Traffic Light</td>
<td>171</td>
<td>3</td>
<td>41</td>
<td>4.039</td>
<td>4.707</td>
<td>16.53</td>
</tr>
<tr>
<td>Pacemaker</td>
<td>271</td>
<td>2</td>
<td>35</td>
<td>7.302</td>
<td>8.318</td>
<td>13.91</td>
</tr>
<tr>
<td>FHM + Pacemaker</td>
<td>314</td>
<td>2</td>
<td>35</td>
<td>9.195</td>
<td>10.306</td>
<td>12.08</td>
</tr>
</tbody>
</table>
Since condition (EnfCo) does not hold, \( \exists q' \in Q \setminus \{q_0\} : \forall (x, y) \in \Sigma : q \xrightarrow{(x, y)} q', \text{i.e., there exists a location } q \in Q \setminus \{q_0\} \text{ such that all the outgoing transitions from } q \text{ go to } q'. \)

Since all the locations in \( Q \) are reachable from \( q_0 \), \( \exists \sigma \in \Sigma^* : q_0 \xrightarrow{\sigma} q, \text{i.e., there certainly exists a word } \sigma \in \Sigma^* \text{ that leads to the problematic accepting location } q \text{ (which has all its outgoing transitions to } q) \text{ from the initial location } q_0. \)

If \( \sigma \) is the input word to the enforcer, then due to constraint (Tr), it cannot edit any event in \( \sigma \), and the enforcer produces \( \sigma \) as output and reaches 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 \( \Sigma \) leads to \( q \) (i.e., since \( \forall (x, y) \in \Sigma : q \xrightarrow{(x, y)} q \)).

Thus, \( \forall (x, y) \in \Sigma \), when the input word given to the enforcer is \( \sigma \), the enforcer cannot produce any event as output since \( \text{edit}_1(\sigma) \) and \( \text{edit}_2(q) \) from location \( q \) will be empty, violating constraints (Inst) and (Cau). Thus, our assumption is false and condition (EnfCo) holds for \( A_\sigma \).

Proof of Theorem 4.2. Let us recall the condition for enforceability: A property \( \phi \) defined as \( \text{SA}_\sigma = (Q, q_0, q_\text{f}, \Sigma, \rightarrow) \) is enforceable iff \( \forall q \in Q, q \neq q_0 \Rightarrow \exists (x, y) \in \Sigma : q \xrightarrow{(x, y)} q' \land q' \neq q \).

Let us also recall the definition of function \( E_\sigma^t : \Sigma^* \rightarrow \Sigma^* \) (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 \( E_\sigma^t(x) = (x_1', y_1') \cdots (x_k', y_k') \), where \( (x_i', y_i') \) is the pair of events output by Algorithm 1 in Step 17, for \( t = 1, \ldots, k \). Note that the input automaton \( A_\sigma = (Q, q_0, q_\text{f}, \Sigma, \rightarrow) \) is obtained from \( A_\sigma \) by projecting on inputs (See Definition 2.2, Section 2).

We shall prove that given any safety property \( \phi \) defined as \( \text{SA}_\sigma \) that satisfies condition (EnfCo), the function \( E_\sigma^t \) is an enforcer for \( \phi \), 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 \( E_\sigma^t(\epsilon) = \epsilon \).

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 \( E_\sigma^t(\sigma) = (x_1', y_1') \cdots (x_k', y_k') \in \Sigma^* \), for \( t = 1, \ldots, k \), and Theorem 4.2 holds for \( \sigma, i.e., E_\sigma^t(\sigma) \) satisfies the (Snd), (Tr), (Mono), (Inst), and (Cau) constraints. Let \( q \in Q \setminus \{q_0\} \) be the current state of both the automata \( A_\sigma \) and \( A_\sigma' \), after processing input \( \sigma \) of length \( k \), i.e., \( q \) corresponds to the state that we reach upon \( E_\sigma^t(\sigma) \) in \( A_\sigma \), and the state that we reach in the automaton \( A_\sigma' \) upon \( E_\sigma^t(\sigma) \). Note that the current state \( q \) in Theorem 1 cannot ever be \( q_0 \) (\( q_0 \) is initialized to \( q_0 \)) and it is updated in step 18 to a state \( q' \in Q \setminus \{q_0\} \).

We now prove that for any event \( (x_{k+1}, y_{k+1}) \in \Sigma \), Theorem 4.2 holds for \( \sigma = (x_{k+1}, y_{k+1}) \), where \( x_{k+1} \in \Sigma \) is the input event read by Algorithm 1, and \( y_{k+1} \in \Sigma \) 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 \( A_\sigma' \) 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' \land q' \neq q_0.**

  In Algorithm 1, the condition tested in step 5 will evaluate to true since from Lemma 2.4, in \( A_{\sigma_1} \), we will have \( \exists q' \in Q : q \xrightarrow{(x_{k+1}, y_{k+1})} q' \land q' \neq q_0 \), and thus \( x_{k+1} = y_{k+1} = x_{k+1} \).

  Also, the condition tested in step 12 will evaluate to true in this case since \( \exists q' \in Q : q \xrightarrow{(x_{k+1}, y_{k+1})} q' \land q' \neq q_0 \), and thus \( y_{k+1} = 1 \). 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 \( E_\sigma^t(\sigma \cdot (x_{k+1}, y_{k+1})) = E_\sigma^t(\sigma) \cdot (x_{k+1}, y_{k+1}) \).

  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., \( E_\sigma^t(\sigma) \)) followed by the new input-output event released as output \( (x_{k+1}, y_{k+1}) \) satisfies the property \( \phi \), and thus constraint (Snd) holds. Regarding constraint (Mono), it holds since \( \sigma \equiv \sigma \cdot (x_{k+1}, y_{k+1}) \) and also \( E_\sigma^t(x_{k+1}, y_{k+1}) = E_\sigma^t(x_{k+1}, y_{k+1}) \). Regarding constraint (Inst) from the induction hypothesis, we have for \( \sigma \) of some length \( k \), \( |\sigma| = |E_\sigma^t(\sigma)| \). We also have \( E_\sigma^t(x_{k+1}, y_{k+1}) = E_\sigma^t(\sigma) \cdot (x_{k+1}, y_{k+1}) \). Thus, \( |\sigma \cdot (x_{k+1}, y_{k+1})| = |E_\sigma^t(\sigma \cdot (x_{k+1}, y_{k+1}))| = k + 1 \), 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., \( E_\sigma^t(\sigma) \) followed by the new input-output event read \( (x_{k+1}, y_{k+1}) \) satisfies the property \( \phi \) and we already saw that the output event released by the algorithm after reading \( (x_{k+1}, y_{k+1}) \) is \( E_\sigma^t(x_{k+1}, y_{k+1}) \). Regarding constraint (Cau), in this case from the induction hypothesis, from the definitions of \( \text{edit}_A \) and \( \text{edit}_A \), we have \( x_{k+1} \in \text{edit}_A(q, x_{k+1}) \), and also \( y_{k+1} \in \text{edit}_A(q, x_{k+1}) \).

Theorem 4.2 thus holds for \( \sigma \cdot (x_{k+1}, y_{k+1}) \).
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 \(edit_{\varphi}(q)\), we have \(x_{k+1} \in edit_{\varphi}(q)\), and we already discussed that \(editO_{\varphi}(q, x_{k+1})\) will be non-empty and thus constraint (Cau) holds in this case.

- \(\exists q' \in Q : q \xrightarrow{s} q' \land q' \not\models q_o\).

In Algorithm 1, the condition tested in step 5 will evaluate to false in this case. It is important to notice that \(edit_{\varphi}(q)\) will be non-empty since from the condition for enforceability and Lemma 2.4, we know for sure that \(\exists q' \in \Sigma, q' \not\models q_o\). We have two sub-cases based on if \(\exists q' \in Q : q \xrightarrow{(x_{k+1}, y_{k+1})} q' \land q' \not\models q_o\), or not.

- \(\exists q' \in Q : q \xrightarrow{(x_{k+1}, y_{k+1})} q' \land q' \not\models q_o\).

In Algorithm 1, the condition tested in step 12 will evaluate to true in this case. Thus, \(y_{k+1} \equiv y_{k+1}\) 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., \(\exists q' \in Q : q \xrightarrow{(x_{k+1}, y_{k+1})} q' \land q' \not\models q_o\)), 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 \(\varphi\), and thus constraint (Snd) holds. The reasoning for constraints (Mono) and (Inst) are similar to the previous cases since we saw that the algorithm releases a new event \((x_{k+1}, y_{k+1})\) as output after reading event \((x_{k+1}, y_{k+1})\) at the end of \(k + 1\)th iteration. Constraint (Tr) holds trivially in this case since \(E_{\varphi}(\sigma) \cdot (x_{k+1}, y_{k+1}) \not\models \varphi\). Regarding constraint (Cau), we already discussed that \(edit_{\varphi}(q)\) is non-empty and \(x_{k+1} \in edit_{\varphi}(q)\), and \(y_{k+1} \in editO_{\varphi}(q, x_{k+1})\) from the condition of this case and definitions of \(edit_{\varphi}\) and \(editO_{\varphi}\).

- \(\exists q' \in Q : q \xrightarrow{(x_{k+1}, y_{k+1})} q' \land q' \not\models q_o\).

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 \(editO_{\varphi}(q, x_{k+1})\) if \(editO_{\varphi}(q, x_{k+1})\) is non-empty. Note that \(editO_{\varphi}(q, x_{k+1})\) will be non-empty in this case since we know for sure that \(\exists q' \in \Sigma, q' \not\models q_o\). We have \(E_{\varphi}(\sigma \cdot (x_{k+1}, y_{k+1})) = E_{\varphi}(\sigma) \cdot (x_{k+1}, y_{k+1})\) where \(x_{k+1}\) is an element belonging to \(edit_{\varphi}(q)\) and \(y_{k+1}\) is an element belonging to \(editO_{\varphi}(q, x_{k+1})\).

Regarding constraint (Snd), from the definitions of \(edit_{\varphi}\) and \(editO_{\varphi}\), we know that \(E_{\varphi}(\sigma) \cdot (x_{k+1}, y_{k+1})\) satisfies the property \(\varphi\) and thus constraint (Snd) holds. The reasoning for constraints (Mono) and (Inst) are similar to the previous cases since we saw that the algorithm releases a new event \((x_{k+1}, y_{k+1})\) as output after reading event \((x_{k+1}, y_{k+1})\).

In this case, constraint (Tr) holds trivially since \(E_{\varphi}(\sigma) \cdot (x_{k+1}, y_{k+1}) \not\models \varphi\). Regarding constraint (Cau), we already discussed that \(edit_{\varphi}(q)\) is non-empty, \(x_{k+1} \in edit_{\varphi}(q)\), and also that \(editO_{\varphi}(q, x_{k+1})\) is non-empty and \(y_{k+1} \in editO_{\varphi}(q, x_{k+1})\) and thus constraint (Cau) holds.

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

REFERENCES