Comparing Controlled System Synthesis and Suppression Enforcement

Luca Aceto[1,2], Ian Cassar[2,3(✉)], Adrian Francalanza[3], and Anna Ingólfsdóttir[2]

[1] Gran Sasso Science Institute, L'Aquila, Italy
[2] School of Computer Science, Reykjavík University, Reykjavík, Iceland
[3] Department of Computer Science, University of Malta, Msida, Malta
ian.cassar.10@um.edu.mt

Abstract. Runtime enforcement and control system synthesis are two verification techniques that automate the process of transforming an erroneous system into a valid one. As both techniques can modify the behaviour of a system to prevent erroneous executions, they are both ideal for ensuring safety. In this paper, we investigate the interplay between these two techniques and identify control system synthesis as being the static counterpart to suppression-based runtime enforcement, in the context of safety properties.

1 Introduction

Our increasing reliance on software systems is inherently raising the demand for ensuring their reliability and correctness. Several verification techniques help facilitate this task by automating the process of deducing whether the system under scrutiny (SuS) satisfies a predefined set of correctness properties. Properties are either verified pre-deployment (statically), as in the case of model checking (MC) [7,12], or post-deployment (dynamically), as per runtime verification (RV) [11,20,27]. In both cases, any error discovered during the verification serves as guidance for identifying the invalid parts of the system that require adjustment.

Other techniques, such as *runtime enforcement* (RE), additionally attempt to automatically transform the invalid system into a valid one. Runtime enforcement [5,15,26,28] adopts an intrusive monitoring approach by which every observable action executed by the SuS is scrutinized and modified as necessary by a monitor at runtime. Monitors in RE may be described in various ways, such as: transducers [5,8,32], shields [26] and security automata [17,28,34]. They may opt to *replace* the invalid actions by valid ones, or completely *suppress* them,

This work was partly supported by the projects "TheoFoMon: Theoretical Foundations for Monitorability" (nr. 163406-051) and "Developing Theoretical Foundations for Runtime Enforcement" (nr. 184776-051) of the Icelandic Research Fund, by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant agreement nr. 778233, and by the Endeavour Scholarship Scheme (Malta), part-financed by the European Social Fund (ESF) - Operational Programme II – Cohesion Policy 2014–2020.

B. Finkbeiner and L. Mariani (Eds.): RV 2019, LNCS 11757, pp. 148–164, 2019.
https://doi.org/10.1007/978-3-030-32079-9_9

thus rendering them immaterial to the environment interacting with the SuS; in certain cases, monitors can even *insert* actions that may directly affect the environment. Different enforcement strategies are applied depending on the property that needs to be enforced.

A great deal of effort [4,13,22,23,25] has been made to study the interplay between static and dynamic techniques, particularly to understand how the two can be used in unison to minimise their respective weaknesses. It is well established that runtime verification is the *dynamic counterpart* of model checking, which means that a subset of the properties verifiable using MC can also be verified dynamically via RV. In fact, multi-pronged verification approaches often use RV in conjunction with MC. Particularly, MC is used to statically verify the parts of the SuS which cannot be verified dynamically (*e.g.,* inaccessible code, performance constraints, *etc.*), while RV is then used to verify other parts dynamically in order to minimise the state explosion problem inherent to MC.

It is however unclear as to which technique can be considered as the *static counterpart* to runtime enforcement. Identifying such a technique enables the possibility of adopting a multi-pronged enforcement approach. One possible candidate is *controlled system synthesis* (CSS) [9,14,24,30]: it analyses the state space of the SuS and reformulates it pre-deployment by *removing* the system's ability to execute erroneous behaviour. As a result, a restricted (yet valid) version of the SuS is produced; this is known as a *controlled system*.

The primary aim of both RE and CSS is to force the resulting monitored/controlled system adheres to the respective property − this is known as *soundness* in RE and *validity* in CSS. Further guarantees are also generally required to ensure minimal disruption to valid systems − this is ensured via *transparency* in RE and *maximal expressiveness* in CSS. As both techniques may adjust systems by omitting their invalid behaviours, they are ideal for ensuring *safety*. These observations, along with other commonalities, hint at the existence of a relationship between runtime enforcement and controlled system synthesis, in the context of safety properties.

In this paper we conduct a preliminary investigation on the interplay between the above mentioned two techniques with the aim of establishing a static counterpart for runtime enforcement. We intend to identify a set of properties that can be enforced either dynamically, via runtime enforcement, or statically via controlled system synthesis. In this first attempt, we however limit ourselves to study this relationship in the context of *safety properties*. As a vehicle for this comparison, we choose the recent work on CSS by van Hulst *et al.* [24], and compare it to our previous work, presented in [5], on enforcing safety properties via action suppressions. We chose these two bodies of work as they are accurate representations of the two techniques. Moreover, they share a number of commonalities including their choice of specification language, modelling of systems, *etc.* To further simplify our comparison, we formulate both techniques in a core common setting and show that there are subtle differences between them even in that scenario. Specifically, we identify a common core within the work presented in [5,24] by:

– working with respect to the Safe Hennessy Milner Logic with invariance ($\text{sHML}_{\mathbf{inv}}$), that is, the *intersection* of the logics used by both works, namely, the Safe Hennessy Milner Logic with recursion (sHML) in [5] and the Hennessy Milner Logic with invariance and reachability ($\text{HML}^{\mathbf{reach}}_{\mathbf{inv}}$) in [24],
– removing constructs and aspects that are supported by one technique and not by the other, and by
– taking into account the assumptions considered in both bodies of work.

To our knowledge, no one has yet attempted to identify a static counterpart to RE, and an insightful comparison of RE and CSS has not yet been conducted. As part of our main contributions, we thus show that:

(*i*) The monitored system obtained from instrumenting a suppression monitor derived from a formula, and the controlled version of the same system (by the same formula), need not be observationally equivalent, Theorem 2.
(*ii*) In spite of (*i*) we prove that both of the obtained systems are *trace (language) equivalent*, that is, they can execute the same set of traces, Theorem 3.
(*iii*) When restricted to safety properties, controlled system synthesis is the *static counterpart* (Definition 3) to runtime enforcement, Theorem 4.

Although (*i*) entails that an external observer can still tell the difference between these two resultant systems [1], knowing (*ii*) suffices to deduce (*iii*) since it is well known that trace equivalent systems satisfy the exact same set of safety properties, Theorem 1.

Structure of the Paper. Section 2 provides the necessary preliminary material describing how we model systems as labelled transition systems and properties via the chosen logic. In Sect. 3 we give an overview of the equalized and simplified versions of the enforcement model presented in [5] and the controlled system synthesis rules of [24]. Section 4 then compares the differences and similarities between the two models, followed by our first contribution which *disproves* the observational equivalence of the two techniques. Section 5 then presents our second set of contributions consisting of a mapping function that derives enforcement monitors from logic formulas, and the proof that the obtained monitored and controlled versions of a given system are *trace equivalent*. This allows us to establish that controlled system synthesis is the *static counterpart* to enforcement when it comes to safety properties. Section 6 overviews related work, and Sect. 7 concludes.

2 Preliminaries

The Model: We assume systems described as *labelled transition systems* (LTSs), which are triples $\langle \text{Sys}, \text{Act} \cup \{\tau\}, \rightarrow \rangle$ defining a set of *system states*, $s, r, q \in \text{Sys}$, a finite set of *observable actions*, $\alpha, \beta \in \text{Act}$, and a distinguished silent action $\tau \notin \text{Act}$, along with a *transition* relation, $\longrightarrow \subseteq (\text{Sys} \times \text{Act} \cup \{\tau\} \times \text{Sys})$. We let $\mu \in \text{Act} \cup \{\tau\}$ and write $s \xrightarrow{\mu} r$ in lieu of $(s, \mu, r) \in \rightarrow$. We use $s \stackrel{\alpha}{\Longrightarrow} r$ to denote

Syntax

$$\varphi, \psi \in \text{sHML} ::= \mathsf{tt} \text{ (truth)} \mid \mathsf{ff} \text{ (falsehood)} \mid \varphi \wedge \psi \text{ (conjunction)} \mid [\alpha]\varphi \text{ (necessity)} \mid \mathsf{max}\, X.\varphi \text{ (greatest fp.)} \mid X \text{ (fp. variable)}$$

Semantics

$$\llbracket \mathsf{tt}, \rho \rrbracket \stackrel{\text{def}}{=} \text{Sys} \qquad \llbracket [\alpha]\varphi, \rho \rrbracket \stackrel{\text{def}}{=} \{ s \mid \forall r \cdot s \text{ if } s \stackrel{\alpha}{\Rightarrow} r \text{ then } r \in \llbracket \varphi, \rho \rrbracket \}$$

$$\llbracket \mathsf{ff}, \rho \rrbracket \stackrel{\text{def}}{=} \emptyset \qquad \llbracket \varphi \wedge \psi, \rho \rrbracket \stackrel{\text{def}}{=} \llbracket \varphi, \rho \rrbracket \cap \llbracket \psi, \rho \rrbracket$$

$$\llbracket X, \rho \rrbracket \stackrel{\text{def}}{=} \rho(X) \qquad \llbracket \mathsf{max}\, X.\varphi, \rho \rrbracket \stackrel{\text{def}}{=} \bigcup \{ S \mid S \subseteq \llbracket \varphi, \rho[X \mapsto S] \rrbracket \}$$

We also encode $\Box\varphi$ as $\mathsf{max}\, X.\varphi \wedge \bigwedge_{\beta \in \text{Act}} [\beta]X$ where X is a fresh variable.

Fig. 1. The syntax and semantics for sHML.

weak transitions representing $s(\xrightarrow{\tau})^* \cdot \xrightarrow{\alpha} r$ and refer to r as an α-derivative of s. Traces $t, u \in \text{Act}^*$ range over (finite) sequences of observable actions, and we write $s \stackrel{t}{\Rightarrow} r$ for a sequence of weak transitions $s \stackrel{\alpha_1}{\Longrightarrow} \ldots \stackrel{\alpha_n}{\Longrightarrow} r$ where $t = \alpha_1, \ldots, \alpha_n$ for some $n \geq 0$; when $n = 0$, t is the empty trace ε and $s \stackrel{\varepsilon}{\Rightarrow} r$ means $s \xrightarrow{\tau} {}^* r$. For each $\mu \in \text{Act} \cup \{\tau\}$, the notation $\hat{\mu}$ stands for ε if $\mu = \tau$ and for μ otherwise. We write *traces*(s) for the set of traces executable from system state s, that is, $t \in$ *traces*(s) iff $s \stackrel{t}{\Rightarrow} r$ for some r. We use the syntax of the regular fragment of CCS [29] to concisely describe LTSs in our examples. We also assume the classic notions for *trace (language) equivalence* and *observational equivalence*, that is, weak bisimilarity [29,33].

Definition 1 (Trace Equivalence). *Two LTS system states s and r are* trace equivalent *iff they produce the* same *set of traces, i.e., traces(s) = traces(r).* □

Definition 2 (Observational Equivalence). *A relation $\mathcal{R}$ over a set of system states is a* weak bisimulation *iff whenever $(s, r) \in \mathcal{R}$ for every action μ, the following transfer properties are satisfied:*

- *$s \xrightarrow{\mu} s'$ implies there exists a transition $r \stackrel{\hat{\mu}}{\Longrightarrow} r'$ such that $(s', r') \in \mathcal{R}$; and*
- *$r \xrightarrow{\mu} r'$ implies there exists a transition $s \stackrel{\hat{\mu}}{\Longrightarrow} s'$ such that $(s', r') \in \mathcal{R}$.*

Two system states s and r are observationally equivalent*, denoted by $s \approx r$, iff there exists a weak bisimulation that relates them.* □

The Logic: The safety logic sHML [6,7] is defined as the set of formulas generated by the grammar of Fig. 1. It assumes a countably infinite set of logical variables $X, Y \in \text{LVar}$ and provides the standard constructs of truth, tt, falsehood, ff, and conjunctions, $\varphi \wedge \psi$. As a shorthand, we occasionally denote conjunctions as $\bigwedge_{i \in I} \varphi_i$, where I is a finite set of indices, and when $I = \emptyset$, $\bigwedge_{i \in \emptyset} \varphi_i$ is equivalent to tt. The logic is also equipped with the *necessity (universal) modality*, $[\alpha]\varphi$, and allows for defining recursive properties using greatest fixpoints, $\mathsf{max}\, X.\varphi$, which bind free occurrences of X in φ. We additionally encode the *invariance*

operator, $\Box\varphi$, requiring φ to be satisfied by every reachable system state, as the recursive property, $\mathsf{max}\, X.\varphi \wedge \bigwedge_{\beta\in \text{Act}}[\beta]X$, where X is not free in φ.

Formulas in sHML are interpreted over the system powerset domain where $S\in\mathcal{P}(\text{Sys})$. The semantic definition of Fig. 1, $[\![\varphi,\rho]\!]$, is given for *both* open and closed formulas. It employs a valuation from logical variables to sets of states, $\rho \in (\text{LVar} \to \mathcal{P}(\text{Sys}))$, which permits an inductive definition on the structure of the formulas; $\rho' = \rho[X \mapsto S]$ denotes a valuation where $\rho'(X) = S$ and $\rho'(Y) = \rho(Y)$ for all other $Y \neq X$. We assume *closed* formulas, *i.e.,* without free logical variables, and write $[\![\varphi]\!]$ in lieu of $[\![\varphi,\rho]\!]$ since the interpretation of a closed formula φ is independent of the valuation ρ. A system (state) s *satisfies* formula φ whenever $s \in [\![\varphi]\!]$.

It is a well known fact that trace equivalent systems satisfy the same set of safety properties. As the (recursion-free) subset of sHML characterises regular safety properties [21], this means that systems sharing the same traces also satisfy the same sHML formulas.

Theorem 1. *Let s and r be system states in an LTS. Then traces(s) = traces(r) iff s and r satisfy exactly the same* sHML *formulas.* □

Example 1. Consider two systems (a good system, $s_{\mathbf{g}}$, and a bad one, $s_{\mathbf{b}}$) implementing a server that repeatedly accepts *requests* and *answers* them in response, and that only terminates upon accepting a *close* request. Whereas $s_{\mathbf{g}}$ outputs a *single* answer (ans) for every request (req), $s_{\mathbf{b}}$ occasionally produces *multiple* answers for a given request (see the underlined branch in the description of $s_{\mathbf{b}}$ below). Both systems terminate with cls.

$$s_{\mathbf{g}} = \mathsf{rec}\, x.\big(\mathsf{req}.\mathsf{ans}.x + \mathsf{cls}.\mathsf{nil}\big)$$
$$s_{\mathbf{b}} = \mathsf{rec}\, x.\big(\mathsf{req}.(\mathsf{ans}.x \;+\; \underline{\mathsf{ans}.(\mathsf{ans}.x + \mathsf{cls}.\mathsf{nil})}) + \mathsf{cls}.\mathsf{nil}\big)$$

We can specify that a request followed by two consecutive answers indicates invalid behaviour via the sHML formula φ_0.

$$\begin{aligned}\varphi_0 &\stackrel{\text{def}}{=} \Box\,[\mathsf{ans}][\mathsf{ans}]\mathsf{ff}\\ &\stackrel{\text{def}}{=} \mathsf{max}\, X.[\mathsf{ans}][\mathsf{ans}]\mathsf{ff} \wedge \textstyle\bigwedge_{\alpha\in \text{Act}}[\alpha]X\end{aligned}$$

where $\text{Act} \stackrel{\text{def}}{=} \{\mathsf{ans},\mathsf{req},\mathsf{cls}\}$. It defines an invariant property requiring that at every reachable state, whenever the system produces an answer following a request, it cannot produce a subsequent answer, *i.e.,* [ans]ff. Using the semantics in Fig. 1, one can check that $s_{\mathbf{g}}\in[\![\varphi_0]\!]$, whereas $s_{\mathbf{b}}\notin[\![\varphi_0]\!]$ since it exhibits the violating trace $s_{\mathbf{b}} \xrightarrow{\mathsf{req}} \cdot \xrightarrow{\mathsf{ans}} \cdot \xrightarrow{\mathsf{ans}} \ldots$, amongst others. □

3 Controlled System Synthesis and Suppression Enforcement

We present the simplified models for suppression enforcement and controlled system synthesis adapted from [5] and [24] respectively. Both models describe

$$\varphi, \psi \in \text{sHML}_{\mathbf{inv}} ::= \mathsf{tt} \mid \mathsf{ff} \mid \varphi \wedge \psi \mid [\alpha]\varphi \mid \Box\varphi$$

Fig. 2. The syntax for $\text{sHML}_{\mathbf{inv}}$.

the composite behaviour attained by the respective techniques. In suppression enforcement, the composite behaviour describes the *observable behaviour* obtained when the monitor and the SuS interact *at runtime*, while in controlled system synthesis, it describes the *structure* of the resulting controlled system obtained *statically prior to deployment*.

To enable our comparison between both approaches, we standardise the logics used in both works and restrict ourselves to $\text{sHML}_{\mathbf{inv}}$, defined in Fig. 2. $\text{sHML}_{\mathbf{inv}}$ is a strict subset of sHML which results from the intersection of sHML, used for suppression enforcement in [5], and $\text{HML}^{\mathbf{reach}}_{\mathbf{inv}}$, used for controlled system synthesis in [24].

Although the work on CSS in [24] assumes that systems do not perform internal τ actions and that output labels may be associated to system states, the work on RE assumes the converse. We therefore equalise the system models by working with respect to LTSs that do not associate labels to states, and do not perform τ actions. We however assume that the resulting monitored and controlled systems may still perform τ actions.

Since we do not focus on state-based properties, the removal of state labels is not a major limitation as we are only forgoing additional state information from the SuS. Although the removal of τ actions requires the SuS to be fully observable, this does not impose significant drawbacks as the work on CSS can easily be extended to allow such actions.

Despite the fact that controlled system synthesis differentiates between system actions that can be removed (controllable) and those which cannot (uncontrollable), the work on enforcement does not. This is also not a major limitation since enforcement models can easily be adapted to make such a distinction. However, in our first attempt at a comparison, we opt to simplify the models as much as possible, and so to enable our comparison we assume that every system action is controllable and can be removed and suppressed by the respective techniques.

Finally, since we do not liberally introduce constructs that are not present in the original models of [5,24], the simplified models are just *restricted versions* of the original ones. Hence, the results proven with respect to these simplified models should either apply to the original ones or extend easily to the more general setting.

3.1 A Model for Suppression Enforcement

We use a simplified version of the operational model of enforcement presented in [5], which uses the transducers $m, n \in \text{Trn}$ defined in Fig. 3. Transducers define *transformation pairs*, $\{\beta, \mu\}$, consisting of: the *specifying action* β that determines whether or not the transformation should be applied to a system action α, and

Syntax

$$m, n \in \textsc{Trn} ::= \quad \{\alpha, \mu\}.m \quad (\text{where } \mu \in \{\alpha, \tau\}) \quad \mid \textstyle\sum_{i \in I} m_i \quad \mid \mathsf{rec}\, x.m \quad \mid x$$

Dynamics

$$\textsc{eSel}\ \frac{m_j \xrightarrow{\alpha \blacktriangleright \mu} n_j}{\sum_{i \in I} m_i \xrightarrow{\alpha \blacktriangleright \mu} n_j}\ j \in I \qquad \textsc{eRec}\ \frac{m\{\mathsf{rec}\, x.m/x\} \xrightarrow{\alpha \blacktriangleright \mu} n}{\mathsf{rec}\, x.m \xrightarrow{\alpha \blacktriangleright \mu} n}$$

$$\textsc{eTrn}\ \frac{}{\{\alpha, \mu\}.m \xrightarrow{\alpha \blacktriangleright \mu} m}$$

Instrumentation

$$\textsc{iTrn}\ \frac{s \xrightarrow{\alpha} s' \qquad m \xrightarrow{\alpha \blacktriangleright \mu} n}{m[s] \xrightarrow{\mu} n[s']} \qquad \textsc{iDef}\ \frac{s \xrightarrow{\alpha} s' \qquad m \overset{\alpha}{\not\longrightarrow}}{m[s] \xrightarrow{\alpha} \mathsf{id}[s']}$$

where $\mathsf{id} \overset{\text{def}}{=} \mathsf{rec}\, x. \sum_{\beta \in \textsc{Act}} \{\beta\}.x$, $\mathsf{sup} \overset{\text{def}}{=} \mathsf{rec}\, x. \sum_{\beta \in \textsc{Act}} \{\beta, \tau\}.x$ and $m \overset{\alpha}{\not\longrightarrow} \overset{\text{def}}{=} \nexists m', \mu \cdot m \xrightarrow{\alpha \blacktriangleright \mu} m'$.

Fig. 3. A model for transducers.

the *transformation action* μ that specifies whether the matched action α should be suppressed into a τ action, or be left intact. A transformation pair thus acts as a function that takes as input a system action α and transforms it into μ whenever α is equal to specifying action β. As a shorthand, we sometimes write $\{\beta\}$ in lieu of $\{\beta, \beta\}$ to signify that actions equal to β will remain unmodified.

The transition rules in Fig. 3 yield a LTS with labels of the form $\alpha \blacktriangleright \mu$. Intuitively, a transition $m \xrightarrow{\alpha \blacktriangleright \mu} n$ denotes the fact that the transducer in state m *transforms* the visible action α (produced by the system) into action μ and transitions into state n. In this sense, the transducer action $\alpha \blacktriangleright \alpha$ denotes the *identity* transformation, while $\alpha \blacktriangleright \tau$ encodes the *suppression* transformation of action α. The key transition rule is eTrn. It states that the transformation-prefix transducer $\{\alpha, \mu\}.m$ can transform action α into μ, as long as the specifying action α is the same as the action performed by the system. In this case, the transformed action is μ, and the transducer state that is reached is m.

The remaining rules eSel and eRec respectively define the standard selection and recursion operations. A sum of transducers $\sum_{i \in I} m_i$ can reduce via eSel to some n_j over some action $\alpha \blacktriangleright \mu$, whenever there exists a transducer m_j in the summation that reduces to n_j over the same action. Rule eRec enables a recursion transducer $\mathsf{rec}\, x.m$ to reduce to some n when its unfolded instance $m\{\mathsf{rec}\, x.m/x\}$ reduces to n as well. We encode the identity monitor, id, and the suppression monitor, sup, as $\mathsf{rec}\, x. \sum_{\beta \in \textsc{Act}} \{\beta\}.x$ and $\mathsf{rec}\, x. \sum_{\beta \in \textsc{Act}} \{\beta, \tau\}.x$ respectively, *i.e.*, as recursive monitors respectively defining an identity and suppression transformation for every possible action $\beta \in \textsc{Act}$ that can be performed by the system.

Figure 3 also describes an *instrumentation* relation, which composes the behaviour of the SuS s with the transformations of a transducer monitor m

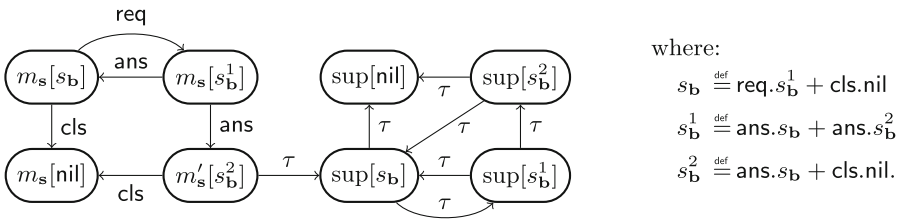

Fig. 4. The runtime execution graph of the monitored system.

that *agrees* with the (observable) actions Act of s. The term $m[s]$ thus denotes the resulting *monitored system* whose behaviour is defined in terms of $\text{Act} \cup \{\tau\}$ from the system's LTS. Concretely, rule iTrn states that when a system s transitions with an observable action α to s' and the transducer m can *transform* this action into μ and transition to n, the instrumented system $m[s]$ transitions with action μ to $n[s']$. Rule iDef is analogous to standard monitor instrumentation rules for premature termination of the transducer [2,18,19,21], and accounts for underspecification of transformations. Thus, if a system s transitions with an observable action α to s', and the transducer m does not specify how to transform it ($m \stackrel{\alpha}{\not\rightarrow}$), the system is still allowed to transition while the transducer defaults to acting like the identity monitor, id, from that point onwards.

Example 2. Consider the suppression transducer $m_{\mathbf{s}}$ below:

$$m_{\mathbf{s}} \stackrel{\text{def}}{=} \mathsf{rec}\, x.(\{\mathsf{ans}\}.m'_{\mathbf{s}}) + \{\mathsf{req}\}.x + \{\mathsf{cls}\}.x$$
$$m'_{\mathbf{s}} \stackrel{\text{def}}{=} (\{\mathsf{ans}, \tau\}.\mathsf{sup} + \{\mathsf{req}\}.x + \{\mathsf{cls}\}.x)$$

where sup recursively suppresses every action $\beta \in \text{Act}$ that can be performed by the system from that point onwards. When instrumented with system $s_{\mathbf{b}}$ from Example 1, the monitor prevents the monitored system $m_{\mathbf{s}}[s_{\mathbf{b}}]$ from answering twice in a row by suppressing the second answer and every subsequent visible action:

$$m_{\mathbf{s}}[s_{\mathbf{b}}] \stackrel{\mathsf{req.ans}}{\Longrightarrow} \cdot \stackrel{\tau}{\longrightarrow} \mathsf{sup}[s_{\mathbf{b}}].$$

When equipped with this dynamic action suppression mechanism, the resulting monitored system $m_{\mathbf{s}}[s_{\mathbf{b}}]$ never violates formula φ_0 at runtime – this is illustrated by the runtime execution graph in Fig. 4. □

We now formalise what we mean by a "static counterpart to suppression enforcement".

Definition 3 (Static Counterpart). *A static verification technique $\mathcal{S}$ is the* static counterpart *for suppression enforcement (in the context of safety properties) when, for every LTS $\langle \text{Sys}, \text{Act}, \rightarrow \rangle$, formula $\varphi \in \text{sHML}_{\boldsymbol{inv}}$ and $s \in \text{Sys}$, there exists a transducer m so that $m[s] \in [\![\varphi]\!]$ iff $\mathcal{S}(s) \in [\![\varphi]\!]$ (where $\mathcal{S}(s)$ is a statically reformulated version of s obtained from applying $\mathcal{S}$).* □

Static Composition

$$\textsc{cBool}\ \frac{s \xrightarrow{\alpha} s' \qquad b \in \{\mathsf{tt}, \mathsf{ff}\}}{(s, b) \overset{\alpha}{\longmapsto} (s', b)} \qquad \textsc{cNec1}\ \frac{s \xrightarrow{\alpha} s'}{(s, [\alpha]\varphi) \overset{\alpha}{\longmapsto} (s', \varphi)}$$

$$\textsc{cNec2}\ \frac{s \xrightarrow{\beta} s' \qquad \beta \neq \alpha}{(s, [\alpha]\varphi) \overset{\beta}{\longmapsto} (s', \mathsf{tt})} \qquad \textsc{cAnd}\ \frac{(s, \varphi) \overset{\alpha}{\longmapsto} (s', \varphi') \qquad (s, \psi) \overset{\alpha}{\longmapsto} (s', \psi')}{(s, \varphi \wedge \psi) \overset{\alpha}{\longmapsto} (s', \mathit{min}(\varphi' \wedge \psi'))}$$

$$\textsc{cMax}\ \frac{(s, \varphi\{\mathsf{max}\, X.\varphi / X\}) \overset{\alpha}{\longmapsto} (s', \psi)}{(s, \mathsf{max}\, X.\varphi) \overset{\alpha}{\longmapsto} (s', \mathit{min}(\psi))}$$

Synthesizability Test

$$\frac{\psi \in \{\mathsf{tt}, X, [\alpha]\varphi\}}{(s, \varphi) \downarrow \psi} \qquad \frac{(s, \varphi) \downarrow \psi_1 \qquad (s, \varphi) \downarrow \psi_2}{(s, \varphi) \downarrow (\psi_1 \wedge \psi_2)} \qquad \frac{(s, \varphi) \downarrow \psi}{(s, \varphi) \downarrow \mathsf{max}\, X.\psi}$$

Invalid Transition Removal

$$\textsc{cTr}\ \frac{(s, \varphi) \overset{\alpha}{\longmapsto} (s', \varphi') \qquad (s', \varphi') \downarrow \varphi'}{(s, \varphi) \xrightarrow{\alpha} (s', \varphi')}$$

Fig. 5. The controlled system synthesis.

3.2 Synthesising Controlled Systems

Figure 5 presents a synthesis function that takes a system $\langle \textsc{Sys}, \textsc{Act}, \rightarrow \rangle$ and a formula φ and constructs a controlled version of the system that satisfies the formula. The new system is synthesised in two stages. In the first stage, a new transition relation $\mapsto \subseteq (\textsc{Sys} \times \textsc{sHML}) \times \textsc{Act} \times (\textsc{Sys} \times \textsc{sHML})$ is constructed over the state-formula product space, $(\textsc{Sys} \times \textsc{sHML})$. Intuitively, this transition relation associates a sHML formula to the initial system state and defines how this changes when the system transitions to other subsequent states. The composite behaviour of the formula and the system is statically computed using the first five rules in Fig. 5.

cBool always adds a transition when the formula is $b \in \{\mathsf{tt}, \mathsf{ff}\}$. Rules cNec1 and cNec2 add a transition from $[\alpha]\varphi$ to φ when s has a transition over α, and to tt if it reduces over $\beta \neq \alpha$. cAnd adds a transition for conjunct formulas, $\varphi \wedge \psi$, when both formulas can reduce independently to some φ' and ψ', with the formula of the end state of the new transition being $\mathit{min}(\varphi' \wedge \psi')$. Finally, cMax adds a fixpoint $\mathsf{max}\, X.\varphi$ transition to $\mathit{min}(\psi)$, when its unfolding can reduce to ψ. In both cAnd and cMax, $\mathit{min}(\varphi)$ stands for a *minimal* logically equivalent formula of φ. This is an oversimplification of the minimisation techniques used in [24] to avoid synthesising an infinite LTS due to invariant formulas and conjunctions, see [24] for more details.

Example 3. Formulas $\varphi' \wedge \mathsf{tt}$, $\varphi' \wedge \mathsf{ff}$ and $\varphi \wedge \psi \wedge \psi$ are *logically equivalent* to (and can thus be minimized into) φ', ff and $\varphi \wedge \psi$ respectively. □

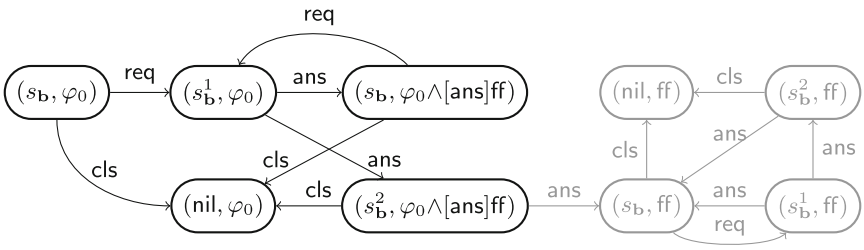

Fig. 6. The LTS obtained from controlling $s_{\mathbf{b}}$ via φ_0.

Instead of defining a rule for fixpoints, the authors of [24] define a synthesis rule directly for invariance stating that when $(s, \varphi) \overset{\alpha}{\longmapsto} (s', \varphi')$, then $(s, \Box\varphi) \overset{\alpha}{\longmapsto} (s', \mathit{min}(\Box\varphi \wedge \varphi'))$. We, however, opted to generalize this rule to fixpoints to simplify our comparison, while still limiting ourselves to $\text{sHML}_{\mathbf{inv}}$ formulas. This is possible since by encoding $\Box\varphi$ as $\mathsf{max}\, X.\varphi \wedge \bigwedge_{\beta \in \text{Act}} [\beta]X$, we get that $(s, \mathsf{max}\, X.\varphi \wedge \bigwedge_{\beta \in \text{Act}} [\beta]X) \overset{\alpha}{\longmapsto} (s', \mathit{min}((\mathsf{max}\, X.\varphi \wedge \bigwedge_{\beta \in \text{Act}} [\beta]X) \wedge \varphi'))$ when $(s, \varphi) \overset{\alpha}{\longmapsto} (s', \varphi')$ where $\mathit{min}((\mathsf{max}\, X.\varphi \wedge \bigwedge_{\beta \in \text{Act}} [\beta]X) \wedge \varphi')$ is the encoded version of $\mathit{min}(\Box\varphi \wedge \varphi')$.

The second stage of the synthesis involves using rule cTr to remove invalid transitions that lead to violating states; this yields the required transition function for the controlled system. This rule relies on the synthesizability test rules to tell whether a controlled state, (s, φ), is valid or not. Intuitively, the test rules fail whenever the current formula φ is semantically equivalent to ff, *e.g.*, formulas $\mathsf{max}\, X.([\alpha]X \wedge \mathsf{ff})$ and $\varphi \wedge \mathsf{ff}$ both fail the synthesizability test rules as they are equivalent to ff. Concretely, the test is vacuously satisfied by truth, tt, logical variables, X, and guarded formulas, $[\alpha]\varphi$, as none of them are logically equivalent to ff. Conjunct formulas, $\psi_1 \wedge \psi_2$, pass the test when both ψ_1 and ψ_2 pass independently. A fixpoint, $\mathsf{max}\, X.\varphi'$, is synthesisable if φ' passes the test.

Transitions that lead to a state that fails the test are therefore removed, and transitions outgoing from failing states become redundant as they are unreachable. The resulting transition function is then used to construct the controlled LTS $\langle (\text{Sys} \times \text{sHML}_{\mathbf{inv}}), \text{Act}, \rightarrow \rangle$.

Example 4. From φ_0 and $s_{\mathbf{b}}$ of Example 1 we can synthesise a controlled system in two stages. In the first stage we compose them together using the composition rules of Fig. 5. We start by generating the composite transition $(s_{\mathbf{b}}, \varphi_0) \overset{\mathsf{req}}{\longmapsto} (s^1_{\mathbf{b}}, \varphi_0)$ via rules cMax and cNec since $s_{\mathbf{b}} \xrightarrow{\mathsf{req}} s^1_{\mathbf{b}}$, and keep on apply the respective rules to the rest of $s_{\mathbf{b}}$'s transitions until we obtain the LTS of Fig. 6. The (grey) ans transition leading to the test failing state, $(s_{\mathbf{b}}, \mathsf{ff})\not\swarrow$, is then removed in the second stage along with its outgoing (grey) transitions, therefore generating the required (black) controlled system. □

4 Discussion and Comparison

We reiterate that controlled system synthesis is a static technique, while suppression enforcement is a dynamic one. Being a dynamic technique, the monitor and the system in suppression enforcement still remain two separate entities, and the instrumentation between them is merely a way for the monitor to interact with the system. In general, the monitor cannot affect the execution of the system itself, but rather modifies its observable trace of actions, such as its inputs and outputs. By contrast, when a controlled system is synthesised, an existing system is paired up with a formula and statically reconstructed into a *new* (valid) system that is incapable of executing the erroneous behaviour.

By removing invalid transitions entirely, controlled system synthesis is more ideal to guarantee the property compliance of the *internal* (less observable) behaviour of a system. For example, this can be useful to ensure that the system does not use a shared resource before locking it. By contrast, the invalid actions are still executed by the system in suppression enforcement, but their effect is rendered invisible to any external observer. This makes suppression enforcement more suitable to ensure that the *external* (observable) behaviour of the system complies with a desired property. For instance, one can ensure that the system does not perform an output that is innocuous to the system itself, but may be providing harmful information to the external environment.

One way of assessing the difference between these two techniques is to use observational equivalence as a yardstick, thus:

$$\forall \varphi \in \text{sHML}, s \in \text{Sys}, \exists m \in \text{Trn} \cdot m[s] \approx (s, \varphi). \tag{1}$$

We show by means of a counter example that (1) is in fact *false* and as a result prove Theorem 2.

Theorem 2 (Non Observational Equivalence). *There exist an* $\text{sHML}_{\boldsymbol{inv}}$ *formula* φ*, an LTS* $\langle \text{Sys}, \text{Act}, \rightarrow \rangle$ *and a system state* $s \in \text{Sys}$ *such that for every monitor* $m \in \text{Trn}$*,* $m[s] \not\approx (s, \varphi)$*.* □

Proof Sketch. Recall the controlled LTS with initial state $(s_{\mathbf{b}}, \varphi_0)$ obtained in Example 4. To prove Theorem 2 we must show that *for every action suppression monitor* m (that can only apply suppression and identity transformations), one *cannot* find a weak bisimulation relation $\mathcal{R}$ so that $(m[s_{\mathbf{b}}], (s_{\mathbf{b}}, \varphi_0)) \in \mathcal{R}$. An elegant way of showing this claim, is by playing the *weak bisimulation games* [7] starting from the pair $(m[s_{\mathbf{b}}], (s_{\mathbf{b}}, \varphi_0))$, for every possible m. The game is played between two players, namely, the attacker and the defender. The attacker wins the game by finding a sequence of moves from the monitored state $m[s_{\mathbf{b}}]$ (or the controlled state $(s_{\mathbf{b}}, \varphi_0)$), which the defender cannot counter, *i.e.,* the move sequence cannot be performed by the controlled state $(s_{\mathbf{b}}, \varphi_0)$ (*resp.* monitored state $m[s_{\mathbf{b}}]$). Note that the attacker is allowed to play a transition from either the current monitored state or the controlled state at each round of the game. A winning strategy for the attacker entails that the composite systems are *not* observationally equivalent.

We start playing the game from the initial pair $(m[s_{\mathbf{b}}], (s_{\mathbf{b}}, \varphi_0))$ for every monitor m. Pick any monitor that suppresses any action other than a second consecutive ans, such as $m_0 \stackrel{\text{def}}{=} \{\mathsf{req}, \tau\}.m'_0$. In this case, it is easy to deduce that the defender always loses the game, that is, if the attacker attacks with $(s_{\mathbf{b}}, \varphi_0) \xrightarrow{\mathsf{req}} (s^1_{\mathbf{b}}, \varphi_0)$ the defender is defenceless since $m_0[s_{\mathbf{b}}] \stackrel{\mathsf{req}}{\Longrightarrow\!\!\!\!\!\!/}$. This remains true regardless of the "depth" at which the suppression of the first req transition occurs.

On the one hand, using the same game characterisation, one can also deduce that by picking a monitor that *fails to suppress* the second consecutive ans action, such as $m_1 \stackrel{\text{def}}{=} \{\mathsf{req}\}.\{\mathsf{ans}\}.\{\mathsf{ans}\}.m'_1$, also prevents the defender from winning. If the attacker plays with $m_1[s_{\mathbf{b}}] \stackrel{\mathsf{req.ans.ans}}{\Longrightarrow} m'_1[s_{\mathbf{b}}]$, the defender loses since it can only counter the first two transitions, *i.e.,* $(s_{\mathbf{b}}, \varphi_0) \stackrel{\mathsf{req.ans}}{\Longrightarrow} \stackrel{\mathsf{ans}}{\Longrightarrow\!\!\!\!\!\!/}$. Again, this holds regardless of the "depth" of the first such failed suppression.

On the other hand, any monitor that actually suppresses the second consecutive ans action, such as $m_{\mathbf{s}}$ from Example 2, still negates a win for the defender. In this case, the attacker can play $(s_{\mathbf{b}}, \varphi_0) \stackrel{\mathsf{req.ans}}{\Longrightarrow} (s^2_{\mathbf{b}}, \varphi_0 \wedge [\mathsf{ans}]\mathsf{ff})$ to which the defender may reply either with $m_{\mathbf{s}}[s_{\mathbf{b}}] \stackrel{\mathsf{req.ans}}{\Longrightarrow} m_{\mathbf{s}}[s_{\mathbf{b}}]$ or $m_{\mathbf{s}}[s_{\mathbf{b}}] \stackrel{\mathsf{req.ans}}{\Longrightarrow} m'_{\mathbf{s}}[s^2_{\mathbf{b}}]$. In the former option, the attacker can subsequently play req in the monitored system, to which the defender cannot reply via the controlled system, *i.e.,* $m_{\mathbf{s}}[s_{\mathbf{b}}] \xrightarrow{\mathsf{req}} m_{\mathbf{s}}[s^1_{\mathbf{b}}]$ but $(s^2_{\mathbf{b}}, \varphi_0 \wedge [\mathsf{ans}]\mathsf{ff}) \xrightarrow{\mathsf{req}}\!\!\!\!\!\!/$. In the latter case, the attacker can now play $m'_{\mathbf{s}}[s^2_{\mathbf{b}}] \xrightarrow{\tau} \mathsf{sup}[s_{\mathbf{b}}]$, which can only be countered by an inaction on behalf of the defender, *i.e.,* the controlled system remains in state $(s^2_{\mathbf{b}}, \varphi_0 \wedge [\mathsf{ans}]\mathsf{ff})$. However, the attacker can subsequently play $(s^2_{\mathbf{b}}, \varphi_0 \wedge [\mathsf{ans}]\mathsf{ff}) \xrightarrow{\mathsf{cls}} (\mathsf{nil}, \varphi_0)$ which is indefensible since $\mathsf{sup}[s_{\mathbf{b}}] \stackrel{\mathsf{cls}}{\Longrightarrow\!\!\!\!\!\!/}$. As in the previous cases, the above reasoning applies.

These cases therefore suffice to deduce that for every possible monitor the attacker always manages to win the game, and hence we conclude that Theorem 2 holds as required. □

This result is important since it proves that powerful external observers, such as the ones presented by Abramsky in [1], can still distinguish between the resulting monitored and controlled systems.

5 Establishing a Static Counterpart to Enforcement

Despite not being observationally equivalent, Examples 2 and 4 provide the intuition that there still exists some level of correspondence between these two techniques. In fact, from the monitored execution graph of Fig. 4 and the controlled LTS in Fig. 6 one can notice that they both execute the *same set of traces*, and are therefore *trace equivalent*. Hence, since trace equivalent systems satisfy the same set of safety properties (Theorem 1), it suffices to conclude that the controlled LTS is statically achieving the same result obtained dynamically by the monitored one, and that it is therefore its static counterpart.

In what follows, we prove that this observation (*i.e.,* trace equivalence) also applies in the general case.

Theorem 3 (Trace Equivalence). *For every LTS* $\langle \textsc{Sys}, \textsc{Act}, \rightarrow \rangle$*, formula* $\varphi \in \text{sHML}_{\boldsymbol{inv}}$ *and* $s \in \textsc{Sys}$*, there exists a monitor* m *such that* $traces(m[s]) = traces((s, \varphi))$. □

To be able to prove this result, we first define a function that maps $\text{sHML}_{\mathbf{inv}}$ formulas to enforcement transducers. We reduce the complexity of this mapping by defining it over the normalised sHML formulas instead.

Definition 4 (sHML normal form). *The set of normalised* sHML *formulas is defined as:*

$$\varphi, \psi \in \text{sHML}_{\boldsymbol{nf}} ::= \mathsf{tt} \quad | \quad \mathsf{ff} \quad | \quad \bigwedge_{i \in I} [\alpha_i]\varphi_i \quad | \quad X \quad | \quad \mathsf{max}\, X.\varphi \ .$$

In addition, a normalised sHML *formula* φ *must satisfy the following conditions:*

1. *In each subformula of* φ *of the form* $\bigwedge_{i \in I} [\alpha_i]\varphi_i$*, the* α_i*'s are* pairwise different*, i.e.,* $\forall i, j \in I \cdot$ if $i \neq j$ then $\alpha_i \neq \alpha_j$.
2. *For every* $\mathsf{max}\, X.\varphi$ *we have* $X \in \boldsymbol{fv}(\varphi)$.
3. *Every logical variable is* guarded *by a modal necessity.* □

In previous work, [3,5] we proved that despite being a syntactic subset of sHML, $\text{sHML}_{\mathbf{nf}}$ is *semantically equivalent* to sHML. Hence, since $\text{sHML}_{\mathbf{inv}}$ is a (strict) subset of sHML, for every $\text{sHML}_{\mathbf{inv}}$ formula we can always find an equivalent $\text{sHML}_{\mathbf{nf}}$ formula. This means that by defining our mapping function in terms of $\text{sHML}_{\mathbf{nf}}$, we can still map every formula in $\text{sHML}_{\mathbf{inv}}$ to the respective monitor.

We proceed to define our mapping function over normalised sHML formulas.

Definition 5. *Recall the definitions of* id *and* sup *from Fig. 3. We define our mapping* $(\!|-|\!) : \text{sHML}_{\boldsymbol{nf}} \mapsto \textsc{Trn}$ *inductively as:*

$$(\!| X |\!) \stackrel{\text{def}}{=} x \qquad (\!| \mathsf{tt} |\!) \stackrel{\text{def}}{=} \mathsf{id} \qquad (\!| \mathsf{ff} |\!) \stackrel{\text{def}}{=} \mathsf{sup} \qquad (\!| \mathsf{max}\, X.\varphi |\!) \stackrel{\text{def}}{=} \mathsf{rec}\, x.(\!| \varphi |\!)$$

$$(\!| \bigwedge_{i \in I} [\{p_i, c_i\}]\varphi_i |\!) \stackrel{\text{def}}{=} \sum_{i \in I} m_i \quad \text{where } m_i \stackrel{\text{def}}{=} \begin{cases} \{\alpha_i, \alpha_i\}.(\!| \varphi_i |\!) & \text{if } \varphi_i \neq \mathsf{ff} \\ \{\alpha_i, \tau\}.(\!| \mathsf{ff} |\!) & \text{otherwise} \end{cases}$$ □

The function is compositional. It assumes a bijective mapping between fixpoint variables and monitor recursion variables and converts logical variables X accordingly, whereas maximal fixpoints, $\mathsf{max}\, X.\varphi$, are converted into the corresponding recursive monitor. The function also converts truth and falsehood formulas, tt and ff, into the identity monitor id and the suppression monitor sup respectively. Normalized conjunctions, $\bigwedge_{i \in I} [\alpha_i]\varphi_i$, are mapped into a *summation* of monitors, $\sum_{i \in I} m_i$, where every branch m_i can be either prefixed by an identity transformation when $\varphi_i \neq \mathsf{ff}$, or by a suppression transformation otherwise. Notice that the requirement that, $\varphi_i \neq \mathsf{ff}$, is in some sense analogous to the synthesisability test applied by the CSS rule cTr of Fig. 5 to retain the valid transitions only. In this mapping function, this requirement is essential to ensure that only the valid actions remain unsuppressed by the resulting monitor.

Example 5. Recall formula φ_0 from Example 1 which can be normalised as:

$$\varphi_0 \stackrel{\text{def}}{=} \mathsf{max}\, X.([\mathsf{ans}]([\mathsf{ans}]\mathsf{ff} \wedge [\mathsf{req}]X \wedge [\mathsf{cls}]X)) \wedge [\mathsf{req}]X \wedge [\mathsf{cls}]X.$$

Using the mapping function defined in Definition 5, we generate monitor

$$(\!|\, \varphi_0 \,|\!) = \mathsf{rec}\, x.(\{\mathsf{ans}\}.(\{\mathsf{ans}, \tau\}.\mathsf{sup} + \{\mathsf{req}\}.x + \{\mathsf{cls}\}.x)) + \{\mathsf{req}\}.x + \{\mathsf{cls}\}.x$$

which is identical to $m_{\mathbf{s}}$ from Example 2. □

With this mapping function in hand, we are able to prove Theorem 3 as a corollary of Proposition 1.

Proposition 1. *For every LTS* $\langle \textsc{Sys}, \textsc{Act}, \rightarrow \rangle$, $\text{sHML}_{\boldsymbol{nf}}$ *formula* φ, $s \in \textsc{Sys}$ *and trace* t, *when* $(\!|\, \varphi \,|\!) = m$ *then* $t \in \mathit{traces}(m[s])$ *iff* $t \in \mathit{traces}((s, \varphi))$. □

Proof Sketch. The if and only-if cases are proven separately and both proofs are conducted by induction on the length of trace t and by case analysis of φ. □

Having concluded the proof of Theorem 3 and knowing Theorem 1, we can finally obtain our main result with respect to Definition 3.

Theorem 4. *Controlled system synthesis is the* static counterpart *of suppression enforcement in the context of safety properties.* □

6 Related Work

Several works comparing formal verification techniques can be found in the literature. In [24] van Hulst *et al.* explore the relationship between their work on controlled system synthesis and the synthesis problem in Ramadge and Wonham's Supervisory Control Theory (SCT) [31]. The aim in SCT is to generate a *supervisor controller* from the SuS and its specification (*e.g.*, a formal property). If successfully generated, the synchronous product of the SuS and the controller is computed to obtain a supervised system. To enable the investigation, van Hulst *et al.* developed language-based notations akin to that used in [31], and proved that Ramadge and Wonham's work can be expressed using their theory.

Ehlers *et al.* in [14] establish a connection between SCT and reactive synthesis – a formal method that attempts to automatically derive a valid reactive system from a given specification. To form this connection, the authors first equalise both fields by using a simplified version of the standard supervisory control problem and focus on a class of reactive synthesis problems that adhere to the requirements imposed by SCT. They then show that the supervisory control synthesis problem can be reduced to a reactive synthesis problem.

Basile *et al.* in [10] explore the gap between SCT and coordination of services, which describe how control and data exchanges are coordinated in distributed systems. This was achieved via a new notion of controllability that allows one to reduce the classical SCT synthesis algorithms to produce orchestrations and choreographies describing the coordination of services as contract automata.

Falcone *et al.* made a brief, comparison between runtime enforcement and SCT in [16] in the context of K-step opacity, but established no formal results that relate these two techniques.

7 Conclusion

We have presented a novel comparison between suppression enforcement and controlled system synthesis – two verification techniques that automate system correction for erroneous systems. Using a counter-example we have proven that those techniques are different modulo observational equivalence, Theorem 2. An Abramsky-type external observer [1] can therefore tell the difference between a monitored and controlled system resulting from the same formula and SuS. However, we were still able to conclude that controlled system synthesis is the static counterpart to suppression enforcement in the context of safety, as defined by Definition 3. This required developing a function that maps logic formulas to suppression monitors, Definition 5, and proving inductively that for every system and formula, one can obtain a monitored and a controlled system that execute the same set of traces at runtime, Theorem 3. As trace equivalent systems satisfy the same safety properties, this result was enough to reach our conclusion, Theorem 4. To our knowledge this is the first formal comparison to be made between these two techniques.

Future Work. Having established a connection between suppression enforcement and control system synthesis with respect to safety properties, it is worth expanding this work at least along two directions and explore how:

(i) runtime enforcement and controlled system synthesis are related with respect to properties other than those representing safety, and how
(ii) suppression enforcement relates to other verification techniques such as supervisory control theory, reactive synthesis, *etc.*

Exploring (i) may entail looking into other work on enforcement and controlled system synthesis that explores a wider set of properties. It might be worth investigating how other enforcement transformations, such as action replacements and insertions, can be used to widen the set of enforceable properties, and how this relates to controlled system synthesis. The connection established by van Hulst *et al.* in [24] between control system synthesis and supervisory control, along with the other relationships reviewed in Sect. 6, may be a starting point for conducting our future investigations on (ii).

References

1. Abramsky, S.: Observation equivalence as a testing equivalence. Theoret. Comput. Sci. **53**, 225–241 (1987). https://doi.org/10.1016/0304-3975(87)90065-X
2. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A.: A framework for parameterized monitorability. In: Baier, C., Dal Lago, U. (eds.) FoSSaCS 2018. LNCS, vol. 10803, pp. 203–220. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89366-2_11
3. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Kjartansson, S.Ö.: Determinizing monitors for HML with recursion. arXiv preprint (2016)

4. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Lehtinen, K.: Adventures in monitorability: from branching to linear time and back again. Proc. ACM Program. Lang. **3**(POPL), 52:1–52:29 (2019). https://doi.org/10.1145/3290365. http://doi.acm.org/10.1145/3290365
5. Aceto, L., Cassar, I., Francalanza, A., Ingólfsdóttir, A.: On runtime enforcement via suppressions. In: 29th International Conference on Concurrency Theory, CONCUR 2018, Beijing, China, 4–7 September 2018, pp. 34:1–34:17 (2018). https://doi.org/10.4230/LIPIcs.CONCUR.2018.34
6. Aceto, L., Ingólfsdóttir, A.: Testing Hennessy-Milner logic with recursion. In: Thomas, W. (ed.) FoSSaCS 1999. LNCS, vol. 1578, pp. 41–55. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49019-1_4
7. Aceto, L., Ingólfsdóttir, A., Larsen, K.G., Srba, J.: Reactive Systems: Modelling, Specification and Verification. Cambridge University Press, New York (2007)
8. Alur, R., Černý, P.: Streaming transducers for algorithmic verification of single-pass list-processing programs. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 599–610. ACM (2011)
9. Arnold, A., Walukiewicz, I.: Nondeterministic controllers of nondeterministic processes. In: Flum, J., Grädel, E., Wilke, T. (eds.) Logic and Automata. Texts in Logic and Games, vol. 2, pp. 29–52. Amsterdam University Press, Amsterdam (2008)
10. Basile, D., ter Beek, M.H., Pugliese, R.: Bridging the gap between supervisory control and coordination of services: synthesis of orchestrations and choreographies. In: Riis Nielson, H., Tuosto, E. (eds.) COORDINATION 2019. LNCS, vol. 11533, pp. 129–147. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-22397-7_8
11. Cassar, I., Francalanza, A., Aceto, L., Ingólfsdóttir, A.: A survey of runtime monitoring instrumentation techniques. In: PrePost 2017, pp. 15–28 (2017)
12. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
13. Desai, A., Dreossi, T., Seshia, S.A.: Combining model checking and runtime verification for safe robotics. In: Lahiri, S., Reger, G. (eds.) RV 2017. LNCS, vol. 10548, pp. 172–189. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67531-2_11
14. Ehlers, R., Lafortune, S., Tripakis, S., Vardi, M.Y.: Bridging the gap between supervisory control and reactive synthesis: case of full observation and centralized control. In: WODES, pp. 222–227. International Federation of Automatic Control (2014)
15. Erlingsson, U., Schneider, F.B.: SASI enforcement of security policies: a retrospective. In: Proceedings of the 1999 Workshop on New Security Paradigms, NSPW 1999, pp. 87–95. ACM, New York (1999)
16. Falcone, Y., Marchand, H.: Runtime enforcement of k-step opacity. In: 52nd IEEE Conference on Decision and Control, pp. 7271–7278, December 2013. https://doi.org/10.1109/CDC.2013.6761043
17. Falcone, Y., Fernandez, J.C., Mounier, L.: What can you verify and enforce at runtime? Int. J. Softw. Tools Technol. Transfer **14**(3), 349 (2012)
18. Francalanza, A.: A theory of monitors. In: Jacobs, B., Löding, C. (eds.) FoSSaCS 2016. LNCS, vol. 9634, pp. 145–161. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49630-5_9

19. Francalanza, A.: Consistently-detecting monitors. In: 28th International Conference on Concurrency Theory (CONCUR 2017). Leibniz International Proceedings in Informatics (LIPIcs), vol. 85, pp. 8:1–8:19. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl (2017)
20. Francalanza, A., et al.: A foundation for runtime monitoring. In: Lahiri, S., Reger, G. (eds.) RV 2017. LNCS, vol. 10548, pp. 8–29. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67531-2_2
21. Francalanza, A., Aceto, L., Ingólfsdóttir, A.: Monitorability for the Hennessy-Milner logic with recursion. Formal Methods Syst. Des. **51**(1), 87–116 (2017)
22. Havelund, K., Pressburger, T.: Model checking Java programs using Java PathFinder. Int. J. Softw. Tools Technol. Transfer **2**(4), 366–381 (2000). https://doi.org/10.1007/s100090050043
23. Havelund, K., Roşu, G.: An overview of the runtime verification tool Java PathExplorer. Formal Methods Syst. Des. **24**(2), 189–215 (2004)
24. van Hulst, A.C., Reniers, M.A., Fokkink, W.J.: Maximally permissive controlled system synthesis for non-determinism and modal logic. Discrete Event Dyn. Syst. **27**(1), 109–142 (2017)
25. Kejstová, K., Ročkai, P., Barnat, J.: From model checking to runtime verification and back. In: Lahiri, S., Reger, G. (eds.) RV 2017. LNCS, vol. 10548, pp. 225–240. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67531-2_14
26. Könighofer, B., et al.: Shield synthesis. Formal Methods Syst. Des. **51**(2), 332–361 (2017)
27. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Logic Algebraic Program. **78**(5), 293–303 (2009)
28. Ligatti, J., Bauer, L., Walker, D.: Edit automata: enforcement mechanisms for run-time security policies. Int. J. Inf. Secur. **4**(1), 2–16 (2005)
29. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, I. Inf. Comput. **100**(1), 1–40 (1992)
30. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1989, pp. 179–190. ACM, New York (1989). https://doi.org/10.1145/75277.75293. http://doi.acm.org/10.1145/75277.75293
31. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete event processes. SIAM J. Control Optim. **25**(1), 206–230 (1987)
32. Sakarovitch, J.: Elements of Automata Theory. Cambridge University Press, New York (2009)
33. Sangiorgi, D.: Introduction to Bisimulation and Coinduction. Cambridge University Press, New York (2011)
34. Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. (TISSEC) **3**(1), 30–50 (2000)