Noname manuscript No.
(will be inserted by the editor)

Comparing Controlled System Synthesis and Suppression Enforcement

Luca Aceto · Ian Cassar · Adrian Francalanza · Anna Ingólfsdóttir

Received: date / Accepted: date

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.

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.

L. Aceto
Gran Sasso Science Institute, L'Aquila, Italy
Department of Computer Science, Reykjavík University, Iceland
E-mail: luca@ru.is

I. Cassar
Department of Computer Science, University of Malta, Malta
Department of Computer Science, Reykjavík University, Iceland
E-mail: ian.cassar.10@um.edu.mt

A. Francalanza
Department of Computer Science, University of Malta, Malta
E-mail: adrian.francalanza@um.edu.mt

A. Ingólfsdóttir
Department of Computer Science, Reykjavík University, Iceland
E-mail: annai@ru.is

1 Introduction

Our increasing reliance on software systems is 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), using techniques such as model checking (MC) [3, 13], or post-deployment (dynamically), as per runtime verification (RV) [12, 21, 28]. In both cases, any error discovered during the verification serves as guidance for identifying the invalid parts of the system that require adjustment.

Other post-deployment techniques, such as *runtime enforcement* (RE), additionally attempt to automatically transform the invalid system into a valid one. Runtime enforcement [6, 16, 27, 29] 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 [6, 9, 33], shields [27] and security automata [18, 29, 35]. They may opt to *replace* the invalid actions by valid ones, or completely *suppress* them, 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 [7, 14, 23, 24, 26] has been devoted to studying 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. For instance, system verification based on MC is often carried out with respect to a model of the environment in which the studied system operates. This makes the sole use of model-checking techniques problematic in settings, such as mobile robotics, where the precise conditions in which systems operate are often known only at runtime and may change over time. Here RV can be used to check post-deployment the environmental conditions used to validate systems at design time [14]. Sometimes, MC is used to statically verify the parts of the SuS which cannot be verified dynamically (*e.g.,* due to inaccessible code or high impact on the performance of the SuS), while RV is then used to verify other parts dynamically in order to minimise the state explosion problem inherent to MC.

A natural question to ask is which technique can be considered as the *static counterpart* to runtime enforcement, *i.e.,* a technique that can statically achieve the same (or equivalent) results as per runtime enforcement. Identifying such a technique is quite desirable as it would allow for properties to be enforced statically when dynamic verification is not ideal, *e.g.,* when the monitor's runtime overheads are infeasible, and vice versa *e.g.,* to mitigate state explosions during static analysis. Such a technique therefore enables the possibility of adopting a multi-pronged enforcement approach that is similar to the one used for verification with RV and MC, but from an enforcement perspective. One promising static technique that has several aspects in common with runtime enforcement is *controlled system synthesis* (CSS) [10, 15, 25, 31]. This approach analyses the state space of the SuS and reformulates it pre-deployment to *remove* the system's ability of executing 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 to adhere 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 permissiveness* 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.* [25], and compare it to our previous work, presented in [6], 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 [6, 25] 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 [6] and the Hennessy Milner Logic with invariance and reachability ($\text{HML}^{\mathbf{reach}}_{\mathbf{inv}}$) in [25],
- 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 investigation we offer the following contributions:

(*i*) We prove that 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) are *trace (language) equivalent*, that is, they can execute the same set of traces, Theorem 2.

(*ii*) When restricted to safety properties, controlled system synthesis is the *static counterpart* (Definition 3) to runtime enforcement, Theorem 3.

(*iii*) In spite of (*i*) and (*ii*), both of the obtained systems need not be observationally equivalent, Theorem 4.

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

Structure of the paper. This article is an extended version of [8]; it includes improved and more expanded explanations and examples, along with the complete proofs for our theorems.

The rest of the paper is structured as follows. Section 2 provides the necessary preliminary material describing how we model systems as labelled transition systems and properties via the chosen logic. In Section 3 we give an overview of the simplified versions of the enforcement model presented in [6] and the controlled system synthesis rules of [25]. In Section 4 we then present our first 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.* Section 5 then presents a deeper comparison of the differences and similarities between the two models, followed by our second contribution which *disproves* the observational equivalence of the two techniques. 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 Section 7 concludes.

2 Preliminaries

The Model: We assume systems described as *labelled transition systems* (LTSs), which are triples $\langle \textsc{Sys}, (\textsc{Act} \cup \{\tau\}), \rightarrow \rangle$ defining a set of *system states*, $s, r, q \in \textsc{Sys}$, a finite set of *observable actions*, $\alpha, \beta \in \textsc{Act}$, and a distinguished silent action $\tau \notin \textsc{Act}$, along with a *transition* relation, $\longrightarrow \subseteq (\textsc{Sys} \times \textsc{Act} \cup \{\tau\} \times \textsc{Sys})$. We let $\mu \in \textsc{Act} \cup \{\tau\}$ and write $s \xrightarrow{\mu} r$ in lieu of $(s, \mu, r) \in \rightarrow$. We use $s \overset{\alpha}{\Longrightarrow} r$ to denote weak transitions representing $s(\xrightarrow{\tau})^{*} \cdot \xrightarrow{\alpha} r$ and refer to r as an α-derivative of s. Traces $t, u \in \textsc{Act}^{*}$ range over (finite) sequences of observable actions, and we write $s \overset{t}{\Longrightarrow} r$ for a sequence of weak transitions $s \overset{\alpha_1}{\Longrightarrow} \ldots \overset{\alpha_n}{\Longrightarrow} r$ where $t = \alpha_1, \ldots, \alpha_n$ for some $n \geq 0$; when $n = 0$, t is the empty trace ε and $s \overset{\varepsilon}{\Longrightarrow} r$ means $s \xrightarrow{\tau}{}^{*} r$. For each $\mu \in \textsc{Act} \cup \{\tau\}$, 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 $s \overset{t}{\Longrightarrow} r$ for some r. We use the syntax of the regular fragment of CCS [30] to concisely describe LTSs in our examples. We also assume the classic notions for *trace (language) equivalence* and *observational equivalence*, that is, weak bisimilarity [30, 34].

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). $\square$

Definition 2 (Observational Equivalence) A relation $\mathcal{R}$ over a set of system states is a *weak bisimulation*

Syntax

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

Semantics

$$[\![\mathsf{tt}, \rho]\!] \overset{\text{def}}{=} \textsc{Sys} \qquad [\![\mathsf{ff}, \rho]\!] \overset{\text{def}}{=} \emptyset \qquad [\![X, \rho]\!] \overset{\text{def}}{=} \rho(X)$$

$$[\![\, [\alpha]\varphi, \rho]\!] \overset{\text{def}}{=} \{s \mid \forall r \cdot s \textit{ if } s \overset{\alpha}{\Longrightarrow} r \textit{ then } r \in [\![\varphi, \rho]\!]\}$$

$$[\![\varphi \wedge \psi, \rho]\!] \overset{\text{def}}{=} [\![\varphi, \rho]\!] \cap [\![\psi, \rho]\!]$$

$$[\![\mathsf{max}\, X.\varphi, \rho]\!] \overset{\text{def}}{=} \bigcup \{S \mid S \subseteq [\![\varphi, \rho[X \mapsto S]]\!]\}$$

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

Fig. 1 The syntax and semantics for sHML.

iff whenever $(s, r) \in \mathcal{R}$ the following transfer properties are satisfied, for every action μ:

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

The Logic: The safety logic sHML [2, 3] is defined as the set of formulas generated by the grammar of Figure 1. It assumes a countably infinite set of logical variables $X, Y \in \textsc{LVar}$ 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 $I = \emptyset$, $\bigwedge_{i \in \emptyset} \varphi_i$ stands for 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, $\mathsf{max}\, X.\varphi \wedge \bigwedge_{\beta \in \textsc{Act}} [\beta]X$, where X is not free in φ.

Formulas in sHML are interpreted over the system powerset domain where $S \in \mathcal{P}(\textsc{Sys})$. The semantic definition of Figure 1, $[\![\varphi, \rho]\!]$, is given for *both* open and closed formulas. It employs a valuation from logical variables to sets of states, $\rho \in (\textsc{LVar} \rightarrow \mathcal{P}(\textsc{Sys}))$, which permits an inductive definition on the structure of the formulas. In that definition, $\rho' = \rho[X \mapsto S]$ denotes the 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 $s \in [\![\varphi]\!]$.

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

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

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*, *answers* them in response, and *logs* the serviced request. It also 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.

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

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

$$\varphi_0 \overset{\text{def}}{=} \Box[\mathsf{ans}][\mathsf{ans}]\mathsf{ff}$$
$$\overset{\text{def}}{=} \max X.[\mathsf{ans}][\mathsf{ans}]\mathsf{ff} \wedge \textstyle\bigwedge_{\alpha\in\mathrm{ACT}} [\alpha]X$$

where $\mathrm{ACT} \overset{\text{def}}{=} \{\mathsf{ans}, \mathsf{req}, \mathsf{cls}\}$. The above formula defines an invariant property requiring that, at every reachable state, whenever the system produces an answer, it cannot produce a subsequent answer *i.e.*, [ans]ff. Using the semantics in Figure 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 [6] and [25], respectively. Both models describe 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 $\mathrm{sHML}_{\mathbf{inv}}$, defined in Figure 2. $\mathrm{sHML}_{\mathbf{inv}}$ is a strict subset of sHML which results from the intersection of sHML, used for suppression enforcement in [6], and $\mathrm{HML}^{\mathbf{reach}}_{\mathbf{inv}}$, used for controlled system synthesis in [25].

Although the work on CSS in [25] 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 [6, 25], 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 can be extended 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 [6], which uses the transducers $m, n \in \mathrm{TRN}$ defined in Figure 3. Transducers define *transformation pairs*, $\{\beta, \mu\}$, which intuitively state that β actions performed by the system should be transformed into μ actions. A transformation pair thus acts as a function that takes as input a system action α and transforms it into μ whenever $\alpha = \beta$. 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 Figure 3 yield an 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

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

$$\textsc{eSel}\,\frac{m_j \xrightarrow{\alpha\blacktriangleright\mu} n_j}{\sum_{i\in I} m_i \xrightarrow{\alpha\blacktriangleright\mu} n_j}\; j\in I$$

$$\textsc{eRec}\,\frac{m\{\mathsf{rec}\, x.m/x\} \xrightarrow{\alpha\blacktriangleright\mu} n}{\mathsf{rec}\, x.m \xrightarrow{\alpha\blacktriangleright\mu} n}$$

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

Instrumentation

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

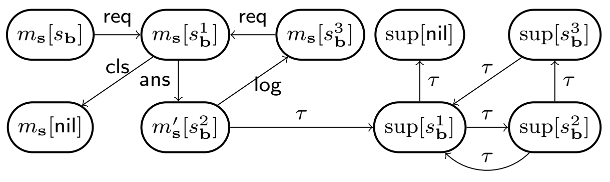
where $\mathsf{id} \stackrel{\text{def}}{=} \mathsf{rec}\, x. \sum_{\beta\in\textsc{Act}} \{\beta\}.x$ and $m \overset{\alpha}{\not\rightarrow} \stackrel{\text{def}}{=} \nexists m', \mu \cdot m \xrightarrow{\alpha\blacktriangleright\mu} m'$.

Fig. 3 A model for transducers.

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 $\sum_{i\in I} m_i$ (where I is a finite set) 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 that *agrees* with the (observable) actions $\textsc{Act}$ of s. The term $m[s]$ thus denotes the resulting *monitored system* whose transitions are labelled with actions in $\textsc{Act} \cup \{\tau\}$ 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


$$\begin{tikzcd}
m_{\mathbf{s}}[s_{\mathbf{b}}] \arrow[r, "\mathsf{req}"] & m_{\mathbf{s}}[s^1_{\mathbf{b}}] \arrow[dl, "\mathsf{cls}"'] \arrow[d, "\mathsf{ans}"] & m_{\mathbf{s}}[s^3_{\mathbf{b}}] \arrow[l, "\mathsf{req}"'] & \mathrm{sup}[\mathsf{nil}] & \mathrm{sup}[s^3_{\mathbf{b}}] \arrow[dl, "\tau"'] \\
m_{\mathbf{s}}[\mathsf{nil}] & m'_{\mathbf{s}}[s^2_{\mathbf{b}}] \arrow[ur, "\mathsf{log}"'] \arrow[rr, "\tau"] & & \mathrm{sup}[s^1_{\mathbf{b}}] \arrow[u, "\tau"'] \arrow[r, "\tau"] & \mathrm{sup}[s^2_{\mathbf{b}}] \arrow[u, "\tau"'] \arrow[l, bend left, "\tau"]
\end{tikzcd}$$

where $s_{\mathbf{b}} \stackrel{\text{def}}{=} \mathsf{req}.s^1_{\mathbf{b}} \qquad s^1_{\mathbf{b}} \stackrel{\text{def}}{=} \mathsf{ans}.s^2_{\mathbf{b}} + \mathsf{cls}.\mathsf{nil}$

$s^2_{\mathbf{b}} \stackrel{\text{def}}{=} \mathsf{ans}.s^1_{\mathbf{b}} + \mathsf{log}.s^3_{\mathbf{b}} \quad s^3_{\mathbf{b}} \stackrel{\text{def}}{=} \mathsf{req}.s^1_{\mathbf{b}}$.

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

premature termination of the transducer [5, 19, 20, 22], 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 that action ($m \overset{\alpha}{\not\rightarrow}$), 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:

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

where sup recursively suppresses every action $\beta \in \textsc{Act}$ 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:

$$m_{\mathbf{s}}[s_{\mathbf{b}}] \xRightarrow{\mathsf{req.ans}} \cdot \xrightarrow{\tau} \mathsf{sup}[s_{\mathbf{b}}].$$

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 Figure 4. $\square$

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 $\varphi \in \text{sHML}$ 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 $\mapsto \subseteq (\textsc{Sys} \times \text{sHML}) \times \textsc{Act} \times (\textsc{Sys} \times \text{sHML})$ is constructed over the state-formula product space, $(\textsc{Sys} \times \text{sHML})$. Intuitively, a state (s, φ) represents a version of state s that is controlled according to the constraints required by φ. A μ-transition $(s, \varphi) \xrightarrow{\mu} (s', \varphi')$ represents a reduction from one controlled state to another. The 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

Static Composition

$$\textsc{cBool}\ \dfrac{s \xrightarrow{\alpha} s' \qquad b \in \{\mathsf{tt},\mathsf{ff}\}}{(s,b) \xmapsto{\alpha} (s',b)}$$

$$\textsc{cNec1}\ \dfrac{s \xrightarrow{\alpha} s'}{(s,[\alpha]\varphi) \xmapsto{\alpha} (s',\varphi)}$$

$$\textsc{cNec2}\ \dfrac{s \xrightarrow{\beta} s' \qquad \beta \neq \alpha}{(s,[\alpha]\varphi) \xmapsto{\beta} (s',\mathsf{tt})}$$

$$\textsc{cAnd}\ \dfrac{(s,\varphi) \xmapsto{\alpha} (s',\varphi') \qquad (s,\psi) \xmapsto{\alpha} (s',\psi')}{(s,\varphi\wedge\psi) \xmapsto{\alpha} (s',\mathit{min}(\varphi'\wedge\psi'))}$$

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

Synthesizability Test

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

$$\dfrac{(s,\varphi) \downarrow \psi}{(s,\varphi) \downarrow \mathsf{max}\,X.\psi}$$

Invalid Transition Removal

$$\textsc{cTr}\ \dfrac{(s,\varphi) \xmapsto{\alpha} (s',\varphi') \qquad (s',\varphi') \downarrow \varphi'}{(s,\varphi) \xrightarrow{\alpha} (s',\varphi')}$$

Fig. 5 The Controlled System Synthesis.

composite behaviour of the formula and the system is statically computed using the first five rules in Figure 5.

cBool adds a transition from (s, b) when the formula is $b \in \{\mathsf{tt}, \mathsf{ff}\}$ if that transition is possible in s. 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 syntactic manipulation techniques used in [25] to avoid synthesising an infinite LTS due to invariant formulas and conjunctions, see [25] 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. □

Instead of defining a rule for fixpoints, the authors of [25] define a synthesis rule directly for invariance stating that when $(s,\varphi) \xmapsto{\alpha} (s',\varphi')$, then $(s, \Box\,\varphi) \xmapsto{\alpha} (s', \mathit{min}(\Box\,\varphi\wedge\varphi'))$. We, however, opted to generalize this rule to fixpoints to simplify our comparison, while still limiting ourselves to sHML$_{\mathbf{inv}}$ formulas. This is possible since by encoding $\Box\,\varphi$ as $\mathsf{max}\,X.\varphi \wedge \bigwedge_{\beta\in\textsc{Act}}[\beta]X$, we get that $(s, \mathsf{max}\,X.\varphi \wedge \bigwedge_{\beta\in\textsc{Act}}[\beta]X) \xmapsto{\alpha} (s', \mathit{min}(\psi))$ when $(s,\varphi)\xmapsto{\alpha}(s',\varphi')$ where $\psi \overset{\text{def}}{=} (\mathsf{max}\,X.\varphi \wedge \bigwedge_{\beta\in\textsc{Act}}[\beta]X)\wedge\varphi'$ and $\mathit{min}(\psi)$ 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 $\langle(\textsc{Sys} \times \text{sHML}_{\mathbf{inv}}), \textsc{Act}, \rightarrow\rangle$.

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 Figure 5. We start by generating the composite transition $(s_{\mathbf{b}}, \varphi_0) \xmapsto{\mathsf{req}} (s^1_{\mathbf{b}}, \varphi_0)$ via rules cMax and cNec since $s_{\mathbf{b}} \xrightarrow{\mathsf{req}} s^1_{\mathbf{b}}$, and keep on applying the respective rules to the rest of $s_{\mathbf{b}}$'s transitions until we obtain the LTS of Figure 6. The (grey) ans transition leading to the test failing state, $(s_{\mathbf{b}}, \mathsf{ff})\not\downarrow$, is then removed in the second stage along with its outgoing (grey) transitions, therefore generating the required (black) controlled system. □

4 Establishing a static counterpart to enforcement

To be able to establish whether CSS is a static counterpart to suppression enforcement, we must first formalise the meaning of a "static counterpart". We thus define it as Definition 3.

Definition 3 (Static Counterpart) A static verification technique $\mathcal{S}$ is the *static counterpart* of suppression enforcement (in the context of safety properties) when, for every LTS $\langle\textsc{Sys}, \textsc{Act}, \rightarrow\rangle$, formula

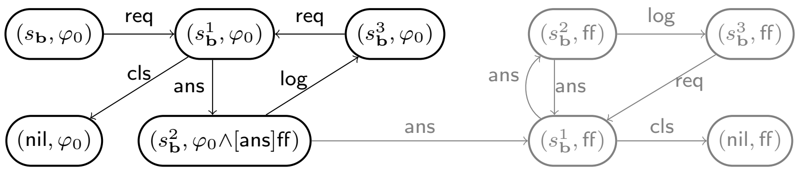

req
req
cls
ans
log
ans
log
ans
ans
req
cls
$(s_{\mathbf{b}}, \varphi_0)$
$(s^1_{\mathbf{b}}, \varphi_0)$
$(s^3_{\mathbf{b}}, \varphi_0)$
$(\mathsf{nil}, \varphi_0)$
$(s^2_{\mathbf{b}}, \varphi_0 \wedge [\mathsf{ans}]\mathsf{ff})$
$(s^2_{\mathbf{b}}, \mathsf{ff})$
$(s^3_{\mathbf{b}}, \mathsf{ff})$
$(s^1_{\mathbf{b}}, \mathsf{ff})$
$(\mathsf{nil}, \mathsf{ff})$

Fig. 6 The LTS obtained from controlling $s_{\mathbf{b}}$ via φ_5.

$\varphi \in \text{sHML}_{\mathbf{inv}}$ 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 $\mathcal{S}$). $\square$

Determining that CSS is the static counterpart of suppression enforcement (as stated by Definition 3) is inherently difficult as it requires showing that every $\text{sHML}_{\mathbf{inv}}$ formula (of which there is an infinite amount) can be enforced using both techniques. Despite this, Examples 2 and 4 already provide the intuition that there exists some level of correspondence between these two techniques. In fact, from the monitored execution graph of Figure 4 and the controlled LTS in Figure 6 one can notice that they both execute the *same set of traces*, and are therefore *trace equivalent.* Since trace equivalent systems satisfy the same set of safety properties (Theorem 1), establishing trace equivalence 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.

We thus begin by showing that trace equivalent systems satisfy the same set of *safety properties.* As the (recursion-free) subset of sHML characterises regular safety properties [22], 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.* $\square$

The proof for this theorem we present relies on the work on detection (runtime verification) monitors by Francalanza *et al.* in [22]. Detection monitors d in [22] can reject a trace t at runtime by issuing the verdict no whenever they detect that an sHML formula has been violated by t, *i.e.,* $d \overset{t}{\Longrightarrow} \mathsf{no}$. In respect to these detection monitors, Francalanza *et al.* prove the following results.

– **Detection Soundness:** For every sHML formula φ, system s, trace t and detection monitor d, if $s \overset{t}{\Longrightarrow}$ and $d \overset{t}{\Longrightarrow} \mathsf{no}$ then $s \notin [\![\varphi]\!]$.
– **Detection Completeness:** For every sHML formula φ, system s, if $s \notin [\![\varphi]\!]$ then there exists a trace t and a detection monitor d such that $s \overset{t}{\Longrightarrow}$ and $d \overset{t}{\Longrightarrow} \mathsf{no}$.

Detection soundness states that if a system state s executes a trace t that gets rejected by a detection monitor d, then s violates φ. Completeness states the converse. Using this framework we can now easily prove Theorem 1 as follows.

Proof for Theorem 1. Assume that

$$traces(s) \subseteq traces(r) \quad \text{and that} \tag{1}$$
$$s \notin [\![\varphi]\!]. \tag{2}$$

Knowing (2) and *Detection Completeness* from [22], we can infer that there exists a trace t, and detection monitor d, such that when s executes t, d *rejects* it for being invalid, *i.e.,* $d \overset{t}{\Longrightarrow} \mathsf{no}$. Hence, since from (1) we know that the invalid trace t can also be executed by r, by *Detection Soundness* from [22] we can also conclude that $r \notin [\![\varphi]\!]$ as required, and we are done. $\square$

Hence, since trace equivalent systems satisfy the same set of safety properties (Theorem 1), it suffices to conclude that the controlled LTS can produce the same set of traces as that generated by a monitored one at runtime.

Theorem 2 (Trace Equivalence) *For every formula $\varphi \in \text{sHML}_{\mathbf{inv}}$, there exists a monitor m such that for every $s \in \text{Sys}$, $traces(m[s]) = traces((s, \varphi))$.* $\square$

The existential quantification on the monitor m in Theorem 2 entails the need of using some sort of mapping that maps $\text{sHML}_{\mathbf{inv}}$ formulas to suppression monitors. To be able to prove this result, we thus 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:

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

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*, denoted as $\#_{i \in I}\, \alpha_i$, *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 $X \in \mathbf{fv}(\varphi)$.
3. Every logical variable is *guarded* by a modal necessity.

$\square$

In previous work [4, 6] we proved that, despite being a syntactic subset of sHML, sHML$_{\mathbf{nf}}$ is *semantically equivalent* to sHML. Hence, since sHML$_{\mathbf{inv}}$ is a (strict) subset of sHML, for every sHML$_{\mathbf{inv}}$ formula we can always find an equivalent sHML$_{\mathbf{nf}}$ formula. This means that by defining our mapping function in terms of sHML$_{\mathbf{nf}}$, we can still map every formula in sHML$_{\mathbf{inv}}$ to the respective monitor.

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

Definition 5 We define our mapping $(\!| - |\!) : \text{sHML}_{\mathbf{nf}} \mapsto \textsc{Trn}$ inductively as:

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

$$(\!| \mathsf{max}\, X.\varphi |\!) \stackrel{\text{def}}{=} \mathsf{rec}\, x.(\!| \varphi |\!) \quad (\!| \bigwedge_{i\in I} [\alpha_i]\varphi_i |\!) \stackrel{\text{def}}{=} \sum_{i\in I} m_i$$

$$\textit{where } m_i \stackrel{\text{def}}{=} \begin{cases} \{\alpha_i, \alpha_i\}.(\!| \varphi_i |\!) & \text{if } \varphi_i \neq \mathsf{ff} \\ \{\alpha_i, \tau\}.(\!| \mathsf{ff} |\!) & \text{otherwise} \end{cases} \qquad \square$$

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 $\varphi_i \neq \mathsf{ff}$ in Definition 5 is in some sense analogous to the pruning of transitions applied by the CSS rule cTr of Figure 5 to retain the valid transitions only. In this mapping function, this requirement is essential to ensure that only the actions that do not lead to violations of the input formula remain unsuppressed by the resulting monitor.

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

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

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

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

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

With this mapping function in hand, we are able to prove Theorem 2 as a corollary of Proposition 1.

Proposition 1 *For every LTS $\langle \textsc{Sys}, \textsc{Act}, \rightarrow \rangle$, sHML$_{\mathbf{nf}}$ formula φ, $s \in \textsc{Sys}$ and trace t, it holds that $t \in$ traces$(m[s])$ iff $t \in$ traces$((s, \varphi))$ where $(\!| \varphi |\!) = m$.* $\square$

In our proof we rely on Lemma 1.

Lemma 1 *For every system s and r, sHML$_{\mathbf{nf}}$ formula φ and action α, if $(\!| \varphi |\!)[s] \xRightarrow{\alpha} r$ then $(\!| \varphi |\!)[s] \xrightarrow{\alpha} r$.* $\square$

The proof for this lemma is provided after that of Proposition 1. We now proceed to prove the *if* and *only-if* cases separately as follows.

Proof for the only-if case. We proceed by induction on the length of t.

Case $t = \varepsilon$. This case holds vacuously since the empty trace can be executed by every system, that is, $\varepsilon \in$ *traces*$((s, \varphi))$ as required.

Case $t = \alpha t'$. Assume that $\alpha t' \in$ *traces*$((\!| \varphi |\!)[s])$ and so by the definition of *traces* we know that there exists a system r such that

$$t' \in \textit{traces}(r) \tag{3}$$

and that $(\!| \varphi |\!)[s] \xRightarrow{\alpha} r$. Hence, by Lemma 1 we get that

$$(\!| \varphi |\!)[s] \xrightarrow{\alpha} r. \tag{4}$$

We now proceed by case analysis on φ.

- $\varphi \in \{X, \mathsf{ff}\}$: These cases do not apply since they contradict assumption (4), namely since $\nexists m \cdot (\!| X |\!) = m$, and since $\forall t \in \textsc{Act} \cdot (\!| \mathsf{ff} |\!)[s] \stackrel{t}{\not\Rightarrow}$ where $(\!| \mathsf{ff} |\!) = \mathsf{sup}$.
- $\varphi = \mathsf{tt}$: Since $(\!| \mathsf{tt} |\!) = \mathsf{id}$, by rule iTrn, from (4) we get that

  $$s \xrightarrow{\alpha} s' \tag{5}$$

  and that $r = \mathsf{id}[s'] = (\!| \mathsf{tt} |\!)[s']$ which in conjunction with (3) and the *inductive hypothesis* we can deduce that

  $$t' \in \textit{traces}((s', \mathsf{tt})). \tag{6}$$

  Since $\varphi = \mathsf{tt}$ and knowing (5) we can synthesise the controlled transition $(s, \mathsf{tt}) \xrightarrow{\alpha} (s', \mathsf{tt})$ so that from (6) we conclude that $\alpha t' \in$ *traces*$((s, \mathsf{tt}))$ as required.
- $\varphi = \bigwedge_{i\in I} [\alpha_i]\varphi_i$: Since $(\!| \bigwedge_{i\in I} [\alpha_i]\varphi_i |\!)$ synthesises the monitor $\left(\sum_{i\in I} \begin{cases} \{\alpha_i, \alpha_i\}.(\!| \varphi_i |\!) & \textit{if } \varphi_i \neq \mathsf{ff} \\ \{\alpha_i, \tau\}.(\!| \mathsf{ff} |\!) & \textit{otherwise} \end{cases} \right)$, we must explore the instrumentation rules that permit for the α-reduction in (4).
  - iTrn: By applying rule iTrn to (4) we have that

    $$s \xrightarrow{\alpha} s' \tag{7}$$
    $$r = m[s'] \tag{8}$$

and that $\left(\sum_{i\in I}\left\{\begin{array}{ll}\{\alpha_i,\alpha_i\}.(\!|\varphi_i|\!) & \textit{if } \varphi_i \neq \mathsf{ff}\\ \{\alpha_i,\tau\}.(\!|\mathsf{ff}|\!) & \textit{otherwise}\end{array}\right.\right) \xrightarrow{\alpha\blacktriangleright\alpha}$ m so that by rules eSel and eTrn we know that

$$\exists j \in I \cdot \alpha_j = \alpha \tag{9}$$

$$m = (\!|\varphi_j|\!) \text{ (where } \varphi_j \neq \mathsf{ff}). \tag{10}$$

Knowing (7), (9) and that $\varphi_j \neq \mathsf{ff}$ we can synthesise the controlled transition

$$(s,[\alpha_j]\varphi_j) \overset{\alpha}{\longmapsto} (s',\varphi_j). \tag{11}$$

Moreover, since the conjunct modal necessities are *pairwise disjoint*, from (9) we infer that for every $i \in I \setminus \{j\}$, $\alpha_i \neq \alpha$ and so we can synthesise the controlled transition $(s,[\alpha_i]\varphi_i) \overset{\alpha}{\longmapsto} (s',\mathsf{tt})$ which in conjunction with (11) can be synthesised as the transition

$$(s,\textstyle\bigwedge_{i\in I}[\alpha_i]\varphi_i) \xrightarrow{\alpha} (s',\varphi_j) \tag{12}$$

since $\mathit{min}(\varphi_j \wedge \bigwedge_{i\in I\setminus\{j\}}\mathsf{tt}) = \varphi_j$. Finally, since by (3), (8), (10) and the *inductive hypothesis* we have that $t' \in \mathit{traces}((s',\varphi_j))$, from (12) we conclude that $\alpha t' \in \mathit{traces}((s,\bigwedge_{i\in I}[\alpha_i]\varphi_i))$ as required.

– iDef: By rule iDef we get that

$$s \xrightarrow{\alpha} s' \tag{13}$$

$$r = \mathsf{id}[s'] \tag{14}$$

and that $\left(\sum_{i\in I}\left\{\begin{array}{ll}\{\alpha_i,\alpha_i\}.(\!|\varphi_i|\!) & \textit{if } \varphi_i \neq \mathsf{ff}\\ \{\alpha_i,\tau\}.(\!|\mathsf{ff}|\!) & \textit{otherwise}\end{array}\right.\right) \overset{\alpha}{\not\longrightarrow}$ from which we can infer that for every $i \in I$, $\alpha_i \neq \alpha$. With this result, from (13) we can thus synthesise the controlled transition $(s,\bigwedge_{i\in I}[\alpha_i]\varphi_i) \xrightarrow{\alpha} (s',\mathsf{tt})$ and so since $(\!|\mathsf{tt}|\!) = \mathsf{id}$ and by (3), (14) and the *inductive hypothesis* we have that $t' \in \mathit{traces}((s',\mathsf{tt}))$. Hence, we can conclude that $\alpha t' \in \mathit{traces}((s,\bigwedge_{i\in I}[\alpha_i]\varphi_i))$ as required.

– $\varphi = \mathsf{max}\,X.\varphi'$: Since $X \in \mathbf{fv}(\varphi')$ we can deduce that $\varphi' \notin \{\mathsf{tt},\mathsf{ff}\}$, and also that $\varphi' \neq X$ since logical variables are required to be guarded in sHML$_{\mathbf{nf}}$. We can thus infer that φ' adheres to a specific structure, that is, $\mathsf{max}\,Y_0...Y_n\cdot\bigwedge_{i\in I}[\alpha_i]\varphi_i$ (where $\mathsf{max}\,Y_0...Y_n.$ is an arbitrary number of fixpoint declarations, possibly none). Hence, since from (4) we can also infer that $(\!|\mathsf{max}\,X.\mathsf{max}\,Y_0...Y_n\cdot\bigwedge_{i\in I}[\alpha_i]\varphi_i|\!)[s] \xrightarrow{\alpha} r$, and since fixpoint unfolding preserves semantics, we get that

$$(\!|\textstyle\bigwedge_{i\in I}[\alpha_i]\varphi_i\{..\}|\!)[s] \xrightarrow{\alpha} r$$

$$\text{where } \{..\} \overset{\text{def}}{=} \{{}^{\mathsf{max}\,XY_0...Y_n\cdot\bigwedge_{i\in I}[\alpha_i]\varphi_i}/_X,\ldots\} \tag{15}$$

After reaching the point where we know (15), the proof proceeds as per the previous case (*i.e.*, when $\varphi = \bigwedge_{i\in I}[\alpha_i]\varphi_i$). We thus skip this part of the proof and simply deduce that $\alpha t' \in \mathit{traces}((s,\bigwedge_{i\in I}[\alpha_i]\varphi_i\{..\}))$. Hence, since unfolded recursive formulas are equivalent to their folded versions, and since φ' is defined as $\mathsf{max}\,Y_0...Y_n\cdot\bigwedge_{i\in I}[\alpha_i]\varphi_i$, we can thus deduce that $\alpha t' \in \mathit{traces}((s,\mathsf{max}\,X.\varphi'))$ as required, and so we are done.

□

Proof for the if case. We again proceed by induction on the structure of t.

Case $t = \varepsilon$. This case holds vacuously since $\varepsilon \in \mathit{traces}((\!|\varphi|\!)[s])$ as required.

Case $t = \alpha t'$. Assume that $\alpha t' \in \mathit{traces}((s,\varphi))$ and so by the definition of *traces* we know that there exists a system r such that

$$(s,\varphi) \xrightarrow{\alpha} r \tag{16}$$

$$t' \in \mathit{traces}(r). \tag{17}$$

We proceed by case analysis on φ.

– $\varphi \in \{\mathsf{ff}, X\}$: These cases do not apply because state (s,φ) is invalid.

– $\varphi = \mathsf{tt}$: Since $(s,\mathsf{tt}) \xrightarrow{\alpha} (s',\mathsf{tt})$ this case holds trivially since $r = (s',\mathsf{tt})$ and so by (17) and the *inductive hypothesis* we get that $t' \in \mathit{traces}((\!|\mathsf{tt}|\!)[s'])$ and since $(\!|\mathsf{tt}|\!) = \mathsf{id}$, by rules iTrn and eTrn we have that $(\!|\mathsf{tt}|\!)[s] \xrightarrow{\alpha} (\!|\mathsf{tt}|\!)[s']$ which allows us to conclude that $\alpha t' \in \mathit{traces}((\!|\mathsf{tt}|\!)[s])$.

– $\varphi = \bigwedge_{i\in I}[\alpha_i]\varphi_i$ and $\#_{i\in I}\,\alpha_i$: In this case we have that

$$(s,\textstyle\bigwedge_{i\in I}[\alpha_i]\varphi_i) \overset{\alpha}{\longmapsto} r \tag{18}$$

$$\exists\psi\cdot r = (s',\mathit{min}(\psi)) \tag{19}$$

and so since the branches of the conjunction are disjoint, we only need to further investigate the following cases:

– $\forall i \in I\cdot\alpha_i \neq \alpha$: Hence, from (18) we can infer that for every $i \in I$ we have that $(s,[\alpha_i]\varphi_i) \overset{\alpha}{\longmapsto} (s',\mathsf{tt})$ and that

$$s \xrightarrow{\alpha} s' \tag{20}$$

$$\mathit{min}(\psi) = \mathsf{tt} \text{ (since } \psi = \textstyle\bigwedge_{i\in I}\mathsf{tt}). \tag{21}$$

Therefore, as we know (20) and that for every $i \in I$ then $\alpha_i \neq \alpha$, by rules eTrn and eSel we can infer that $\left(\sum_{i\in I}\left\{\begin{array}{ll}\{\alpha_i,\alpha_i\}.(\!|\varphi_i|\!) & \textit{if } \varphi_i \neq \mathsf{ff}\\ \{\alpha_i,\tau\}.(\!|\mathsf{ff}|\!) & \textit{otherwise}\end{array}\right.\right) \overset{\alpha}{\not\longrightarrow}$ and so by the definition of $(\!|-|\!)$ and rule iDef we conclude that

$$(\!|\textstyle\bigwedge_{i\in I}[\alpha_i]\varphi_i|\!)[s] \xrightarrow{\alpha} (\!|\mathsf{tt}|\!)[s']. \tag{22}$$

Finally, from (17), (19), (21) and by the *inductive hypothesis* we have that $t' \in \mathit{traces}((\!|\,\mathsf{tt}\,|\!)[s'])$ and so from (22), we infer that $\alpha t' \in \mathit{traces}((\!|\bigwedge_{i\in I}[\alpha_i]\varphi_i|\!)[s])$.

– $\exists j \in I \cdot \eta_j = \alpha$ but $\forall i \in I \setminus \{j\} \cdot \alpha_i \neq \alpha$: In this case, from the controlled synthesis rules and from (18) and (19) we can infer that $\exists j \in I \cdot (s, [\alpha_j]\varphi_j) \overset{\alpha}{\longmapsto} (s', \varphi_j)$ and that $\forall i \in I \setminus \{j\} \cdot (s, [\alpha_i]\varphi_i) \overset{\alpha}{\longmapsto} (s', \mathsf{tt})$ and finally that

$$s \xrightarrow{\alpha} s' \tag{23}$$

$$\mathit{min}(\psi) = \mathit{min}(\varphi_j \wedge \textstyle\bigwedge_{i\in I}\mathsf{tt}) = \varphi_j \tag{24}$$

where $\varphi_j \neq \mathsf{ff}$, as otherwise, if $\varphi_j = \mathsf{ff}$ the resulting state $(s', \mathit{min}(\mathsf{ff}))$ would be invalid and thus removed by the synthesis along with any transitions leading to it (including (18)). Knowing that there exists $j \in I$ so that $\alpha_j \neq \alpha$ and by rule eTrn we can also deduce that $\{\alpha_j, \alpha_j\}.(\!|\varphi_j|\!) \xrightarrow{\alpha\blacktriangleright\alpha} (\!|\varphi_j|\!)$ and so by rule eSel we have that $\Big(\sum_{i\in I}\begin{cases}\{\alpha_i,\alpha_i\}.(\!|\varphi_i|\!) & \textit{if } \varphi_i \neq \mathsf{ff}\\ \{\alpha_i,\tau\}.(\!|\mathsf{ff}|\!) & \textit{otherwise}\end{cases}\Big) \xrightarrow{\alpha\blacktriangleright\alpha}$ $(\!|\varphi_j|\!)$. This means that by (23), rule iTrn and the definition of $(\!|-|\!)$ we conclude that

$$(\!|\textstyle\bigwedge_{i\in I}[\alpha_i]\varphi_i|\!)[s] \xrightarrow{\alpha} (\!|\varphi_j|\!)[s']. \tag{25}$$

Finally, since from (17), (19), (24) and the *inductive hypothesis* we know that $t' \in \mathit{traces}((\!|\varphi_j|\!)[s'])$, from (25) we can infer that $\alpha t' \in \mathit{traces}((\!|\bigwedge_{i\in I}[\alpha_i]\varphi_i|\!)[s])$ as required.

– $\varphi = \mathsf{max}\,X.\varphi'$ and $X \in \mathbf{fv}(\varphi')$: We now have that $(s, \mathsf{max}\,X.\varphi') \xrightarrow{\alpha} (s', \psi)$ because

$$(s, \varphi'\{\mathsf{max}\,X.\varphi'/X\}) \overset{\alpha}{\longmapsto} (s', \psi) \tag{26}$$

and so since φ' can neither be X (since sHML$_{\mathbf{nf}}$ requires fixpoint variables to be guarded) nor ff or tt (since $X \in \mathbf{fv}(\varphi')$) we can deduce that φ' must have the form $\mathsf{max}\,Y_0...Y_n.\bigwedge_{i\in I}[\alpha_i]\varphi_i$ and so since fixpoint unfolding preserves formula semantics, from (26) we can subsequently deduce that $(s, \bigwedge_{i\in I}[\alpha_i]\varphi_i\{..\}) \overset{\alpha}{\longmapsto} (s', \psi)$ where $\{..\} \overset{\text{def}}{=} \{\mathsf{max}\,XY_0...Y_n.\bigwedge_{i\in I}[\alpha_i]\varphi_i/X, \ldots\}$. From this point onwards the proof proceeds as per the previous case ($\varphi = \bigwedge_{i\in I}[\alpha_i]\varphi_i$), we thus skip this part of the proof and safely conclude that

$$\alpha t' \in \mathit{traces}((\!|\textstyle\bigwedge_{i\in I}[\alpha_i]\varphi_i\{..\}|\!)[s]). \tag{27}$$

Since fixpoint folding preserves semantics and $\varphi' = \mathsf{max}\,Y_0...Y_n.\bigwedge_{i\in I}[\alpha_i]\varphi_i$, from (27) we thus conclude that $\alpha t' \in \mathit{traces}((\!|\mathsf{max}\,X.\varphi'|\!)[s])$ as required, and so we are done. □

We now prove the auxiliary lemma Lemma 1 as follows. The reader may safely skip the following proofs upon first reading and proceed from 12.

Proof for Lemma 1. We must prove that for every system state s and r, sHML$_{\mathbf{nf}}$ formula φ and action α, if $(\!|\varphi|\!)[s] \overset{\alpha}{\Longrightarrow} r$ then $(\!|\varphi|\!)[s] \xrightarrow{\alpha} r$.

Since we assume that the SuS s does not perform τ actions, by the rules in our enforcement model we know that the only case when a τ reduction is part of a monitored execution occurs when the monitor suppresses a (visible) action of s. We proceed by case analysis on φ.

Case $\varphi \in \{X, \mathsf{ff}\}$. These cases do not apply since $\nexists m \cdot (\!|X|\!) = m$ and since $(\!|\mathsf{ff}|\!) = \mathsf{sup}$ and so $\nexists \beta \in$ Act $\cdot\, \mathsf{sup}[s] \overset{\beta}{\Longrightarrow}$.

Case $\varphi = \mathsf{tt}$. Since $(\!|\mathsf{tt}|\!) = \mathsf{id}$ cannot suppress any action, we deduce that the weak transition in $((\!|\mathsf{tt}|\!), s) \overset{\alpha}{\Longrightarrow} r$ is in fact a strong one and so that $((\!|\mathsf{tt}|\!), s) \xrightarrow{\alpha} r$ as required.

Case $\varphi = \bigwedge_{i\in I}[\alpha_i]\varphi_i$. Assume that

$$\Big(\sum_{i\in I}\begin{cases}\{\alpha_i\}.(\!|\varphi_i|\!) & \textit{if } \varphi_i \neq \mathsf{ff}\\ \{\alpha_i,\tau\}.(\!|\mathsf{ff}|\!) & \textit{otherwise}\end{cases}\Big)[s] \overset{\alpha}{\Longrightarrow} r. \tag{28}$$

From the weak reduction in (28) we infer that the system must perform some action β which is then suppressed by one of the monitor's branches, and so there must exist an index $j \in I$ so that $\alpha_j = \beta$ and $\{\alpha_j, \tau\}.(\!|\mathsf{ff}|\!) \xrightarrow{\beta\blacktriangleright\tau}$ $(\!|\mathsf{ff}|\!)$. However, since $(\!|\mathsf{ff}|\!) = \mathsf{sup}$, we know that if any invalid action β were to be executed by s and, as a consequence, suppressed by the monitor, any subsequent action (including α) would also be suppressed by sup, in which case the instrumented system in (28) would be unable to eventually execute α and thus yield a contradiction. Therefore, the only way that transition (28) can happen is when the monitor *does not* suppress any action prior to executing α, which thus means that the weak reduction in (28) is in fact a strong one, *i.e.,* $\Big(\sum_{i\in I}\begin{cases}\{\alpha_i\}.(\!|\varphi_i|\!) & \textit{if } \varphi_i \neq \mathsf{ff}\\ \{\alpha_i,\tau\}.(\!|\mathsf{ff}|\!) & \textit{otherwise}\end{cases}\Big)[s] \xrightarrow{\alpha} r$ as required.

Case $\varphi = \mathsf{max}\,X.\varphi'$ *where* $X \in \boldsymbol{fv}(\varphi')$. Assume that $(\!|\mathsf{max}\,X.\varphi'|\!)[s] \overset{\alpha}{\Longrightarrow} r$ and so since $[\![\mathsf{max}\,X.\varphi']\!]$ is logically equivalent to $[\![\varphi'\{\mathsf{max}\,X.\varphi'/X\}]\!]$ we can deduce that

$$(\!|\varphi'\{\mathsf{max}\,X.\varphi'/X\}|\!)[s] \overset{\alpha}{\Longrightarrow} r. \tag{29}$$

Since $\varphi'\{\mathsf{max}\,X.\varphi'/X\} \in$ sHML$_{\mathbf{nf}}$, by the restrictions imposed by sHML$_{\mathbf{nf}}$ we know that φ' cannot be X because (bound) logical variables are required to be guarded, and it also cannot be tt or ff since X is required to be defined in φ, *i.e.,* $X \in \mathbf{fv}(\varphi')$. Hence, we know that φ' can only have the form of

$$\varphi' = \mathsf{max}\,Y_0...Y_n.\textstyle\bigwedge_{i\in I}[\alpha_i]\varphi_i \tag{30}$$

where $\mathsf{max}\,Y_0 \ldots Y_n.\ldots$ represents an arbitrary number of fixpoint declarations, possibly none. Hence, since $[\![\varphi']\!]$ is logically equivalent to $[\![\bigwedge_{i\in I}[\alpha_i]\varphi_i\{..\}]\!]$ where $\{..\} \overset{\text{def}}{=} \{{}^{\mathsf{max}\,XY_0\ldots Y_n.\bigwedge_{i\in I}[\alpha_i]\varphi_i}/_X, \ldots\}$, from (29) and (30) we have that

$$(\!| \bigwedge_{i\in I}[\alpha_i]\varphi_i\{..\} |\!)[s] \overset{\alpha}{\Longrightarrow} r. \tag{31}$$

Having reached the point where we know (31), the proof becomes identical as per the previous case ($\varphi = \bigwedge_{i\in I}[\alpha_i]\varphi_i$), we thus skip this part of the proof and safely conclude that $(\!| \bigwedge_{i\in I}[\alpha_i]\varphi_i\{..\} |\!)[s] \xrightarrow{\alpha} r$. Hence, knowing (30) and that $[\![\varphi']\!] = [\![\bigwedge_{i\in I}[\alpha_i]\varphi_i\{..\}]\!]$, from (29) and (30) we conclude that $(\!| \mathsf{max}\,X.\varphi' |\!)[s] \xrightarrow{\alpha} r$ as required, and so we are done. □

Having concluded the proof of Theorem 2 and knowing Theorem 1, we can finally obtain our main result with respect to Definition 3.

Theorem 3 *Controlled system synthesis is the* static counterpart *of suppression enforcement in the context of safety properties.* □

5 Distinguishing between Suppression Enforcement and CSS

Despite concluding that CSS is the static counterpart to suppression enforcement, there are still a number of subtle differences between these two techniques. For one, since suppression enforcement is a dynamic technique, the monitor and the system still remain two separate entities, and the instrumentation between them is merely a way for the monitor to interact with the SuS. In general, the monitor does affect the execution of the SuS 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* (correct) 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.

Moreover, it turns out that although both techniques produce composite systems that are trace equivalent to each other, an external observer may still be able to tell them apart by merely observing them. One way of formally assessing this is to use observational equivalence (characterised by weak bisimilarity) as a yardstick, thus:

$$\forall \varphi \in \text{sHML}, s \in \textsc{Sys}, \exists m \in \textsc{Trn} \cdot m[s] \approx (s, \varphi). \tag{32}$$

We show by means of a counter example that (32) is in fact *false* and as a result prove Theorem 4.

Theorem 4 (Observational Difference) *There exist an* sHML$_{\boldsymbol{inv}}$ *formula* φ*, an LTS* $\langle \textsc{Sys}, \textsc{Act}, \rightarrow \rangle$ *and a system state* $s \in \textsc{Sys}$ *such that for all monitors* $m \in \textsc{Trn}$*,* $m[s] \not\approx (s, \varphi)$*.* □

Proof sketch. Recall the controlled LTS with initial state $(s_{\mathbf{b}}, \varphi_0)$ obtained in Example 4. To prove Theorem 4 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* [3] 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 $m_0 \overset{\text{def}}{=} \{\mathsf{req}, \tau\}.m'_0$. 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 $m_0[s_{\mathbf{b}}] \overset{\mathsf{req}}{\not\Longrightarrow}$. 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 $m_1 \overset{\text{def}}{=} \{\mathsf{req}\}.\{\mathsf{ans}\}.\{\mathsf{ans}\}.m'_1$, also prevents the defender from winning. If the attacker plays with $m_1[s_{\mathbf{b}}] \overset{\mathsf{req.ans.ans}}{\Longrightarrow} m'_1[s_{\mathbf{b}}]$, the defender loses since it can only counter the first two transitions, *i.e.*, $(s_{\mathbf{b}}, \varphi_0) \overset{\mathsf{req.ans}}{\Longrightarrow}$

$\overset{\mathsf{ans}}{\not\Longrightarrow}$. 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 $m_2 \overset{\mathrm{def}}{=} \{\mathsf{req}\}.\{\mathsf{ans}\}.\{\mathsf{ans}, \bullet\}.m_2'$, still negates a win for the defender. In this case, the attacker can play $(s_{\mathbf{b}}, \varphi_0) \xRightarrow{\mathsf{req.ans}} (s^2_{\mathbf{b}}, \varphi_0 \wedge [\mathsf{ans}]\mathsf{ff})$ to which the defender must reply with $m_2[s_{\mathbf{b}}] \xRightarrow{\mathsf{req.ans}} \{\mathsf{ans}, \bullet\}.m_2'[s^2_{\mathbf{b}}]$. The attacker can subsequently play $\{\mathsf{ans}, \bullet\}.m_2'[s^2_{\mathbf{b}}] \xrightarrow{\tau} m_2'[s^1_{\mathbf{b}}]$, which can only be countered by an inaction on behalf of the defender, *i.e.,* the controlled system remains in state $(s^2_{\mathbf{b}}, \varphi_0 \wedge [\mathsf{ans}]\mathsf{ff})$.

Since we do not know the form of m_2', we consider the following two cases, namely, when m_2' suppresses cls, and the case when it does not. In the first case, the attacker can attack with $m_2'[s^1_{\mathbf{b}}] \xrightarrow{\tau} m_2''[\mathsf{nil}]$ (for some m_2'') where τ represents the suppression of the cls action. Once again the defender can only counter with an inaction and stay in state $(s^2_{\mathbf{b}}, \varphi_0 \wedge [\mathsf{ans}]\mathsf{ff})$. At this point the attacker wins the play by attacking with $(s^2_{\mathbf{b}}, \varphi_0 \wedge [\mathsf{ans}]\mathsf{ff}) \xrightarrow{\mathsf{log}} (s^3_{\mathbf{b}}, \varphi_0)$ since $m_2''[\mathsf{nil}] \overset{\mathsf{log}}{\not\Longrightarrow}$. In the second case, the attacker also wins by attacking with $m_2'[s^1_{\mathbf{b}}] \xrightarrow{\mathsf{cls}} m_2'''[\mathsf{nil}]$ (for some m_2''') since $(s^2_{\mathbf{b}}, \varphi_0 \wedge [\mathsf{ans}]\mathsf{ff}) \overset{\mathsf{cls}}{\not\Longrightarrow}$.

The same result can be obtained using monitor $m_{\mathbf{s}}$ from Example 2. In this case, the attacker can play $(s_{\mathbf{b}}, \varphi_0) \xRightarrow{\mathsf{req.ans}} (s^2_{\mathbf{b}}, \varphi_0 \wedge [\mathsf{ans}]\mathsf{ff})$ to which the defender can only reply with $m_{\mathbf{s}}[s_{\mathbf{b}}] \xRightarrow{\mathsf{req.ans}} m_{\mathbf{s}}'[s^2_{\mathbf{b}}]$. The attacker can subsequently play $m_{\mathbf{s}}'[s^2_{\mathbf{b}}] \xrightarrow{\tau} \mathsf{sup}[s^1_{\mathbf{b}}]$, 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 $(s^2_{\mathbf{b}}, \varphi_0 \wedge [\mathsf{ans}]\mathsf{ff}) \xrightarrow{\mathsf{log}} (\mathsf{nil}, \varphi_0)$ which is indefensible since $\mathsf{sup}[s_{\mathbf{b}}] \overset{\mathsf{log}}{\not\Longrightarrow}$.

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 4 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.

6 Related Work

Several works comparing formal verification techniques can be found in the literature. In [25] 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) [32]. 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 [32], and proved that Ramadge and Wonham's work can be expressed using their theory.

Ehlers *et al.* in [15] 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 [11] 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 [17] 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. We were 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 2. As trace equivalent systems satisfy the same safety properties, Theorem 1, this result was enough to reach our conclusion, Theorem 3. Using a counter-example we however deduced that these two techniques are different modulo observational equivalence, Theorem 4. An Abramsky-type external observer [1] can therefore tell the difference between a monitored and controlled system resulting from the same formula and SuS. 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 [25] between control system synthesis and supervisory control, along with the other relationships reviewed in Section 6, may be a starting point for conducting our future investigations on (ii).

References

1. Abramsky S (1987) Observation equivalence as a testing equivalence. Theoretical Computer Science 53:225–241, URL https://doi.org/10.1016/0304-3975(87)90065-X
2. Aceto L, Ingólfsdóttir A (1999) Testing hennessy-milner logic with recursion. In: Thomas W (ed) Foundations of Software Science and Computation Structures, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 41–55
3. Aceto L, Ingólfsdóttir A, Larsen KG, Srba J (2007) Reactive Systems: Modelling, Specification and Verification. Cambridge University Press, New York, NY, USA
4. Aceto L, Achilleos A, Francalanza A, Ingólfsdóttir A, Kjartansson SÖ (2016) Determinizing monitors for HML with recursion. arXiv preprint
5. Aceto L, Achilleos A, Francalanza A, Ingólfsdóttir A (2018) A framework for parameterized monitorability. In: Foundations of Software Science and Computation Structures, Springer International Publishing, Cham, pp 203–220
6. Aceto L, Cassar I, Francalanza A, Ingólfsdóttir A (2018) On runtime enforcement via suppressions. In: 29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China, pp 34:1–34:17, DOI 10.4230/LIPIcs.CONCUR.2018.34
7. Aceto L, Achilleos A, Francalanza A, Ingólfsdóttir A, Lehtinen K (2019) Adventures in monitorability: From branching to linear time and back again. Proc ACM Program Lang 3(POPL):52:1–52:29, DOI 10.1145/3290365, URL http://doi.acm.org/10.1145/3290365
8. Aceto L, Cassar I, Francalanza A, Ingólfsdóttir A (2019) Comparing controlled system synthesis and suppression enforcement. In: Runtime Verification, Springer International Publishing, Cham, pp 148–164
9. Alur R, Černý P (2011) 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, ACM, pp 599–610
10. Arnold A, Walukiewicz I (2008) Nondeterministic controllers of nondeterministic processes. In: Logic and Automata, Amsterdam University Press, Texts in Logic and Games, vol 2, pp 29–52
11. Basile D, ter Beek MH, Pugliese R (2019) Bridging the gap between supervisory control and coordination of services: Synthesis of orchestrations and choreographies. In: COORDINATION 2019 - 21st International Conference on Coordination Models and Languages, (To appear)
12. Cassar I, Francalanza A, Aceto L, Ingólfsdóttir A (2017) A survey of runtime monitoring instrumentation techniques. In: PrePost2017, pp 15–28
13. Clarke EM, Grumberg O, Peled D (1999) Model Checking. MIT press
14. Desai A, Dreossi T, Seshia SA (2017) Combining model checking and runtime verification for safe robotics. In: Runtime Verfication (RV), Springer International Publishing, Cham, LNCS, pp 172–189
15. Ehlers R, Lafortune S, Tripakis S, Vardi M (2014) Bridging the gap between supervisory control and reactive synthesis: Case of full observation and centralized control. IFAC Proceedings Volumes 47(2):222 – 227, 12th IFAC International Workshop on Discrete Event Systems (2014)
16. Erlingsson U, Schneider FB (1999) Sasi enforcement of security policies: A retrospective. In: Proceedings of the 1999 Workshop on New Security Paradigms, ACM, New York, NY, USA, NSPW '99, pp 87–95
17. Falcone Y, Marchand H (2013) Runtime enforcement of k-step opacity. In: 52nd IEEE Conference on Decision and Control, pp 7271–7278, DOI 10.1109/CDC.2013.6761043
18. Falcone Y, Fernandez JC, Mounier L (2012) What can you verify and enforce at runtime? International Journal on Software Tools for Technology Transfer 14(3):349

19. Francalanza A (2016) A Theory of Monitors. In: International Conference on Foundations of Software Science and Computation Structures, Springer, pp 145–161
20. Francalanza A (2017) Consistently-Detecting Monitors. In: 28th International Conference on Concurrency Theory (CONCUR 2017), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, Leibniz International Proceedings in Informatics (LIPIcs), vol 85, pp 8:1–8:19
21. Francalanza A, Aceto L, Achilleos A, Attard DP, Cassar I, Della Monica D, Ingólfsdóttir A (2017) A foundation for runtime monitoring. In: Runtime Verification, Springer International Publishing, Cham, pp 8–29
22. Francalanza A, Aceto L, Ingólfsdóttir A (2017) Monitorability for the Hennessy-Milner logic with recursion. Formal Methods in System Design 51(1):87–116
23. Havelund K, Pressburger T (2000) Model checking java programs using java pathfinder. International Journal on Software Tools for Technology Transfer 2(4):366–381, DOI 10.1007/s100090050043, URL https://doi.org/10.1007/s100090050043
24. Havelund K, Roşu G (2004) An overview of the runtime verification tool java pathexplorer. Formal methods in system design 24(2):189–215
25. van Hulst AC, Reniers MA, Fokkink WJ (2017) Maximally permissive controlled system synthesis for non-determinism and modal logic. Discrete Event Dynamic Systems 27(1):109–142
26. Kejstová K, Ročkai P, Barnat J (2017) From Model Checking to Runtime Verification and Back. In: RV, Springer
27. Könighofer B, Alshiekh M, Bloem R, Humphrey L, Könighofer R, Topcu U, Wang C (2017) Shield synthesis. Formal Methods in System Design 51(2):332–361
28. Leucker M, Schallhart C (2009) A brief account of runtime verification. The Journal of Logic and Algebraic Programming 78(5):293–303
29. Ligatti J, Bauer L, Walker D (2005) Edit automata: enforcement mechanisms for run-time security policies. International Journal of Information Security 4(1):2–16
30. Milner R, Parrow J, Walker D (1992) A calculus of mobile processes, I. Information and computation 100(1):1–40
31. Pnueli A, Rosner R (1989) On the synthesis of a reactive module. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, New York, NY, USA, POPL '89, pp 179–190, DOI 10.1145/75277.75293, URL http://doi.acm.org/10.1145/75277.75293
32. Ramadge PJ, Wonham WM (1987) Supervisory control of a class of discrete event processes. SIAM J Control Optim 25(1):206–230
33. Sakarovitch J (2009) Elements of Automata Theory. Cambridge University Press, New York, NY, USA
34. Sangiorgi D (2011) Introduction to Bisimulation and Coinduction. Cambridge University Press, New York, NY, USA
35. Schneider FB (2000) Enforceable security policies. ACM Transactions on Information and System Security (TISSEC) 3(1):30–50