








On Bidirectional Runtime Enforcement

Luca Aceto^{1,2} , Ian Cassar^{2,3} , Adrian Francalanza³  ,
and Anna Ingólfssdóttir² 

¹ Gran Sasso Science Institute, L'Aquila, Italy

² Department of Computer Science, ICE-TCS,
Reykjavík University, Reykjavík, Iceland

³ Department of Computer Science, University of Malta, Msida, Malta
`adrian.francalanza@um.edu.mt`

Abstract. Runtime enforcement is a dynamic analysis technique that instruments a monitor with a system in order to ensure its correctness as specified by some property. This paper explores *bidirectional* enforcement strategies for properties describing the input and output behaviour of a system. We develop an operational framework for bidirectional enforcement and use it to study the enforceability of the safety fragment of Hennessy-Milner logic with recursion (sHML). We provide an automated synthesis function that generates correct monitors from sHML formulas, and show that this logic is enforceable via a specific type of bidirectional enforcement monitors called action disabling monitors.

1 Introduction

Runtime enforcement (RE) [18, 32] is a dynamic verification technique that uses *monitors* to analyse the runtime behaviour of a system-under-scrutiny (SuS) and transform it in order to conform to some correctness *specification*. The seminal work in RE [11, 27, 32, 33, 37] models the behaviour of the SuS as a *trace of arbitrary* actions. Crucially, it assumes that the monitor can either *suppress* or *replace* any trace action and, whenever possible, *insert* additional actions into the trace. This work has been effectively used to implement *unidirectional* enforcement approaches [5, 9, 19, 28] that monitor the trace of *outputs* produced by the SuS as illustrated by Fig. 1(a). In this setup, the monitor is instrumented with the SuS to form a *composite system* (represented by the dashed enclosure in Fig. 1) and is tasked with transforming the output behaviour of the SuS to ensure its correctness. For instance, an erroneous output β of the SuS is intercepted by

This work was partly supported by the projects “TheoFoMon: Theoretical Foundations for Monitorability” (nr.163406-051), “Developing Theoretical Foundations for Runtime Enforcement” (nr.184776-051) and “MoVeMnt: Mode(l)s of Verification and Monitorability” (nr.217987-051) of the Icelandic Research Fund, by the Italian MIUR project PRIN 2017FTXR7S IT MATTERS “Methods and Tools for Trustworthy Smart Systems”, 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 – 2014–2020.

© IFIP International Federation for Information Processing 2021

Published by Springer Nature Switzerland AG 2021

K. Peters and T. A. C. Willemse (Eds.): FORTE 2021, LNCS 12719, pp. 3–21, 2021.

https://doi.org/10.1007/978-3-030-78089-0_1

the monitor and transformed into β' , to stop the error from propagating to the surrounding environment.

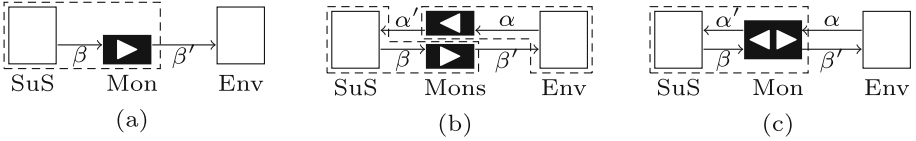


Fig. 1. Enforcement instrumentation setups.

Despite its merits, unidirectional enforcement lacks the power to enforce properties involving the *input* behaviour of the SuS. Arguably, these properties are harder to enforce: unlike outputs, inputs are instigated by the environment and not the SuS itself, meaning that the SuS possesses only partial control over them. Moreover, even when the SuS can control when certain inputs can be supplied (*e.g.*, by opening a communication port, or by reading a record from a database *etc.*), the environment still determines the provided payload.

Broadly, there are two approaches to enforce bidirectional properties at runtime. Several bodies of work employ two monitors attached at the output side of each (diadic) interacting party [12, 17, 26]. As shown in Fig. 1(b), the extra monitor is attached to the *environment* to analyse its outputs before they are passed on as *inputs* to the SuS. While this approach is effective, it assumes that a monitor can actually be attached to the environment (which is often inaccessible).

By contrast, Fig. 1(c) presents a less explored *bidirectional enforcement* approach where the monitor analyses the entire behaviour of the SuS without the need to instrument the environment. The main downside of this alternative setup is that it enjoys limited control over the SuS’s inputs. As we already argued, the monitor may be unable to enforce a property that could be violated by an input action with an invalid payload value. In other cases, the monitor might need to adopt a different enforcement strategy to the ones that are conventionally used for enforcing output behaviour in a unidirectional one.

This paper explores how existing monitor transformations—namely, suppressions, insertions and replacements—can be repurposed to work for bidirectional enforcement, *i.e.*, the setup in Fig. 1(c). Since inputs and outputs must be enforced differently, we find it essential to distinguish between the monitor’s transformations and their resulting effect on the visible behaviour of the composite system. This permits us to study the enforceability of properties defined via the safety subset sHML of the well-studied branching-time logic μ HML [8, 31, 36] (a reformulation of the modal μ -calculus [29]). Our contributions are:

- (i) A general instrumentation framework for bidirectional enforcement (Fig. 4) that is parametrisable by any system whose behaviour can be modelled as a labelled transition system. The framework subsumes the one presented in previous work [5] and differentiates between input and output actions.

Syntax

$$\varphi, \psi \in \text{SHML} ::= \text{tt} \quad (\text{truth}) \quad | \quad \text{ff} \quad (\text{falsehood}) \quad | \quad \bigwedge_{i \in I} \varphi_i \quad (\text{conjunction})$$

$$| [p, c]\varphi \quad (\text{necessity}) \quad | \quad \max X.\varphi \quad (\text{greatest fp.}) \quad | \quad X \quad (\text{fp. variable})$$
Semantics

$$\llbracket \text{tt}, \rho \rrbracket \stackrel{\text{def}}{=} \text{SYS} \qquad \llbracket \text{ff}, \rho \rrbracket \stackrel{\text{def}}{=} \emptyset \qquad \llbracket X, \rho \rrbracket \stackrel{\text{def}}{=} \rho(X)$$

$$\llbracket \bigwedge_{i \in I} \varphi_i, \rho \rrbracket \stackrel{\text{def}}{=} \bigcap_{i \in I} \llbracket \varphi_i, \rho \rrbracket \qquad \llbracket \max X.\varphi, \rho \rrbracket \stackrel{\text{def}}{=} \bigcup \{S \mid S \subseteq \llbracket \varphi, \rho[X \mapsto S] \rrbracket\}$$

$$\llbracket [p, c]\varphi, \rho \rrbracket \stackrel{\text{def}}{=} \{s \mid \forall \alpha, r, \sigma \cdot (s \xrightarrow{\alpha} r \text{ and } \text{match}(p, \alpha) = \sigma \text{ and } c\sigma \Downarrow \text{true}) \text{ implies } r \in \llbracket \varphi\sigma, \rho \rrbracket\}$$
Fig. 2. The syntax and semantics for SHML, the safety fragment of μHML .

- (ii) A novel definition formalising what it means for a monitor to *adequately enforce* a property in a bidirectional setting (Definitions 2 and 6). These definitions are parametrisable with respect to an instrumentation relation, an instance of which is given by our enforcement framework of Fig. 4.
- (iii) A new result showing that the subclass of *disabling monitors* suffices to bidirectionally enforce any property expressed as an SHML formula (Theorem 1). A by-product of this result is a synthesis function (Definition 8) that generates a disabling monitor for any SHML formula.

Full proofs and additional details can to be found at [6, 13].

2 Preliminaries

The Model. We assume a countable set of communication ports $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \text{PORT}$, a set of values $v, w \in \text{VAL}$, and partition the set of actions ACT into *inputs* $\mathbf{a}?v \in \text{IACT}$, and *outputs* $\mathbf{a}!v \in \text{OACT}$ where $\text{IACT} \cup \text{OACT} = \text{ACT}$. Systems are described as *labelled transition systems* (LTSs); these are triples $\langle \text{SYS}, \text{ACT} \cup \{\tau\}, \rightarrow \rangle$ consisting of a set of *system states*, $s, r, q \in \text{SYS}$, a set of *visible actions*, $\alpha, \beta \in \text{ACT}$, along with a distinguished silent action $\tau \notin \text{ACT}$ (where $\mu \in \text{ACT} \cup \{\tau\}$), and a *transition relation*, $\rightarrow \subseteq (\text{SYS} \times (\text{ACT} \cup \{\tau\}) \times \text{SYS})$. We write $s \xrightarrow{\mu} r$ in lieu of $(s, \mu, r) \in \rightarrow$, and $s \xrightarrow{\alpha} r$ to denote weak transitions representing $s(\xrightarrow{\tau})^* \cdot \xrightarrow{\alpha} r$ where r is called the α -derivative of s . For convenience, we use the syntax of the regular fragment of value-passing CCS [23] to concisely describe LTSs. Traces $t, u \in \text{ACT}^*$ range over (finite) sequences of *visible actions*. We write $s \xrightarrow{t} r$ to denote a sequence of *weak transitions* $s \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_n} r$ where $t = \alpha_1 \dots \alpha_n$ for some $n \geq 0$; when $t = \varepsilon$, $s \xrightarrow{\varepsilon} r$ means $s \xrightarrow{\tau}^* r$. Additionally, we represent system runs as *explicit traces* that include τ -actions, $t_\tau, u_\tau \in (\text{ACT} \cup \{\tau\})^*$ and write $s \xrightarrow{\mu_1 \dots \mu_n} r$ to denote a sequence of strong transitions $s \xrightarrow{\mu_1} \dots \xrightarrow{\mu_n} r$. The function $\text{sys}(t_\tau)$ returns a canonical system that exclusively produces the sequence of actions defined by t_τ . *E.g.*, $\text{sys}(\mathbf{a}?3.\tau.\mathbf{a}!5)$ produces the process $\mathbf{a}?x.\tau.\mathbf{a}!5.\text{nil}$. We consider states in our system LTS modulo the classic notion of *strong bisimilarity* [23, 38] and write $s \sim r$ when states s and r are bisimilar.

The Logic. The behavioral properties we consider are described using sHML [7, 22], a subset of the value passing μ HML [24, 36] that uses *symbolic actions* of the form (p, c) consisting of an action pattern p and a condition c . Symbolic actions abstract over concrete actions using *data variables* $x, y, z \in \text{DVAR}$ that occur free in the constraint c or as binders in the pattern p . Patterns are subdivided into input $(x)?(y)$ and output $(x)!(y)$ patterns where (x) binds the information about the port on which the interaction has occurred, whereas (y) binds the payload; $\mathbf{bv}(p)$ denotes the set of binding variables in p whereas $\mathbf{fv}(c)$ represents the set of free variables in condition c . We assume a (partial) *matching function* $\text{match}(p, \alpha)$ that (when successful) returns a substitution σ mapping bound variables in p to the corresponding values in α ; by replacing every occurrence (x) in p with $\sigma(x)$ we get the matched action α . The *filtering condition*, c , is evaluated wrt. the substitution returned by successful matches, written as $c\sigma \Downarrow v$ where $v \in \{\text{true}, \text{false}\}$.

Whenever a symbolic action (p, c) is *closed*, i.e., $\mathbf{fv}(c) \subseteq \mathbf{bv}(p)$, it denotes the *set* of actions $\llbracket (p, c) \rrbracket \stackrel{\text{def}}{=} \{ \alpha \mid \exists \sigma \cdot \text{match}(p, \alpha) = \sigma \text{ and } c\sigma \Downarrow \text{true} \}$. Following standard value-passing LTS semantics [23, 34], our systems have *no control* over the data values supplied via inputs. Accordingly, we assume a well-formedness constraint where the condition c of an input symbolic action, $((x)?(y), c)$, *cannot* restrict the values of binder y , i.e., $y \notin \mathbf{fv}(c)$. As a shorthand, whenever a condition in a symbolic action equates a bound variable to a specific value we embed the equated value within the pattern, e.g., $((x)!(y), x = \mathbf{a} \wedge y = 3)$ and $((x)?(y), x = \mathbf{a})$ become $(\mathbf{a}!3, \text{true})$ and $(\mathbf{a}?(\mathbf{y}), \text{true})$; we also elide *true* conditions, and just write $(\mathbf{a}!3)$ and $(\mathbf{a}?(\mathbf{y}))$ in lieu of $(\mathbf{a}!3, \text{true})$ and $(\mathbf{a}?(\mathbf{y}), \text{true})$.

Figure 2 presents the sHML syntax for some countable set of logical variables $X, Y \in \text{LVAR}$. The construct $\bigwedge_{i \in I} \varphi_i$ describes a *compound* conjunction, $\varphi_1 \wedge \dots \wedge \varphi_n$, where $I = \{1, \dots, n\}$ is a finite set of indices. The syntax also permits recursive properties using greatest fixpoints, $\max X. \varphi$, which bind free occurrences of X in φ . The central construct is the (symbolic) universal modal operator, $[p, c]\varphi$, where the binders $\mathbf{bv}(p)$ bind the free data variables in c and φ . We occasionally use the notation $(-)$ to denote “don’t care” binders in the pattern p , whose bound values are not referenced in c and φ . We also assume that all fixpoint variables, X , are guarded by modal operators.

Formulas in sHML are interpreted over the system powerset domain where $S \in \mathcal{P}(\text{SYS})$. The semantic definition of Fig. 2, $\llbracket \varphi, \rho \rrbracket$, is given for *both* open and closed formulas. It employs a valuation from logical variables to sets of states, $\rho \in (\text{LVAR} \rightarrow \mathcal{P}(\text{SYS}))$, 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 $Y \neq X$. The only non-standard case is that for the universal modality formula, $[p, c]\varphi$, which is satisfied by any system that either *cannot* perform an action α that matches p while satisfying condition c , or for any such matching action α with substitution σ , its derivative state satisfies the continuation $\varphi\sigma$. We consider formulas modulo associativity and commutativity of \wedge , and unless stated explicitly, we assume *closed* formulas, i.e., without free logical and data variables. Since the interpretation of a closed φ is independent

of the valuation ρ we write $\llbracket \varphi \rrbracket$ in lieu of $\llbracket \varphi, \rho \rrbracket$. A system s *satisfies* formula φ whenever $s \in \llbracket \varphi \rrbracket$, and a formula φ is *satisfiable*, when $\llbracket \varphi \rrbracket \neq \emptyset$.

We find it convenient to define the function *after*, describing how an SHML formula *evolves* in reaction to an action μ . Note that, for the case $\varphi = [p, c]\psi$, the formula returns $\psi\sigma$ when μ matches successfully the symbolic action (p, c) with σ , and **tt** otherwise, to signify a trivial satisfaction. We lift the *after* function to (explicit) traces in the obvious way, i.e., $\text{after}(\varphi, t_\tau)$ is equal to $\text{after}(\text{after}(\varphi, \mu), u_\tau)$ when $t_\tau = \mu u_\tau$ and to φ when $t_\tau = \varepsilon$. Our definition of *after* is justified vis-a-vis the semantics of Fig. 2 via Proposition 1; it will play a role in defining our notion of enforcement in Sect. 4.

Definition 1. We define the function $\text{after} : (\text{SHML} \times \text{ACT} \cup \{\tau\}) \rightarrow \text{SHML}$ as:

$$\text{after}(\varphi, \alpha) \stackrel{\text{def}}{=} \begin{cases} \varphi & \text{if } \varphi \in \{\text{tt}, \text{ff}\} \\ \text{after}(\varphi' \{ \varphi / X \}, \alpha) & \text{if } \varphi = \max X. \varphi' \\ \bigwedge_{i \in I} \text{after}(\varphi_i, \alpha) & \text{if } \varphi = \bigwedge_{i \in I} \varphi_i \\ \psi\sigma & \text{if } \varphi = [p, c]\psi \text{ and } \exists \sigma. (\text{match}(p, \alpha) = \sigma \wedge c\sigma \Downarrow \text{true}) \\ \text{tt} & \text{if } \varphi = [p, c]\psi \text{ and } \nexists \sigma. (\text{match}(p, \alpha) = \sigma \wedge c\sigma \Downarrow \text{true}) \end{cases}$$

$$\text{after}(\varphi, \tau) \stackrel{\text{def}}{=} \varphi \quad \blacksquare$$

Proposition 1. For every system state s , formula φ and action α , if $s \in \llbracket \varphi \rrbracket$ and $s \xrightarrow{\alpha} s'$ then $s' \in \llbracket \text{after}(\varphi, \alpha) \rrbracket$. \square

Example 1. The *safety* property φ_1 *repeatedly* requires that *every* input request that is made on a port that is *not* **b**, cannot be followed by another input on the same port in succession. However, following this input it allows a *single* output answer on the same port in response, followed by the logging of the serviced request by outputting a notification on a dedicated port **b**. We note how the channel name bound to x is used to constrain sub-modalities. Similarly, values bound to y_1 and y_2 are later referenced in condition $y_3 = (\text{log}, y_1, y_2)$.

$$\varphi_1 \stackrel{\text{def}}{=} \max X. [((x)?(y_1), x \neq \mathbf{b})]([((x)?(-))] \text{ff} \wedge [(x)!(y_2)]) \varphi'_1$$

$$\varphi'_1 \stackrel{\text{def}}{=} ([((x)!(-))] \text{ff} \wedge [(\mathbf{b})!(y_3), y_3 = (\text{log}, y_1, y_2)]) X$$

Consider the systems $s_{\mathbf{a}}$, $s_{\mathbf{b}}$ and $s_{\mathbf{c}}$ (where $s_{\text{cls}} \stackrel{\text{def}}{=} (\mathbf{b}?z. \text{if } z = \text{cls} \text{ then nil else } X)$).

$$s_{\mathbf{a}} \stackrel{\text{def}}{=} \text{rec } X. ((\mathbf{a}?x. y := \text{ans}(x). \mathbf{a}!y. \mathbf{b}!(\text{log}, x, y). X) + s_{\text{cls}}) \quad s_{\mathbf{c}} \stackrel{\text{def}}{=} \mathbf{a}?y. s_{\mathbf{a}}$$

$$s_{\mathbf{b}} \stackrel{\text{def}}{=} \text{rec } X. ((\mathbf{a}?x. y := \text{ans}(x). \mathbf{a}!y. (\mathbf{a}!y. \mathbf{b}!(\text{log}, x, y). s_{\mathbf{a}} + \mathbf{b}!(\text{log}, x, y). X) + s_{\text{cls}})$$

$s_{\mathbf{a}}$ implements a request-response server that repeatedly inputs values (for some domain VAL) on port **a**, $\mathbf{a}?x$, for which it internally computes an answer and assigns it to the data variable y , $y := \text{ans}(x)$. It then outputs the answer on port **a** in response to each request, $\mathbf{a}!y$, and finally logs the serviced request by outputting the triple (log, x, y) on port **b**, $\mathbf{b}!(\text{log}, x, y)$. It terminates whenever it inputs a close request **cls** from port **b**, i.e., $\mathbf{b}?z$ when $z = \text{cls}$.

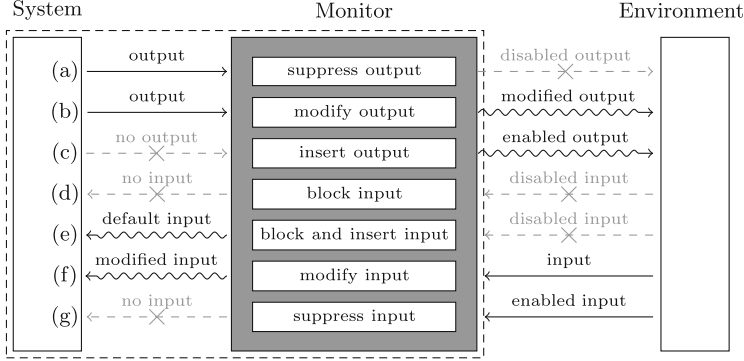


Fig. 3. Bidirectional enforcement via suppression, insertion and replacement.

Systems s_b and s_c are similar to s_a but define additional behaviour: s_c requires a startup input, $a?y$, before behaving as s_a , whereas s_b occasionally provides a redundant (underlined) answer prior to logging a serviced request. Using the semantics of Fig. 2, one can verify that $s_a \in \llbracket \varphi_1 \rrbracket$, $s_c \notin \llbracket \varphi_1 \rrbracket$ because of $s_c \xrightarrow{a?v_1.a?v_2}$, and $s_b \notin \llbracket \varphi_1 \rrbracket$ since we have $s_b \xrightarrow{a?v_1.a!ans(v_1).a!ans(v_1)}$ (for some values v_1 and v_2). ■

3 A Bidirectional Enforcement Model

Bidirectional enforcement seeks to transform the entire (visible) behaviour of the SuS in terms of input and output actions; this contrasts with unidirectional approaches that only modify output traces. In this richer setting, it helps to differentiate between the transformations performed by the monitor (*i.e.*, insertions, suppressions and replacements), and the way they can be used to affect the resulting behaviour of the composite system. In particular, we say that an action that can be performed by the SuS has been *disabled* when it is no longer visible in the resulting composite system (consisting of the SuS and the monitor). Dually, a visible action is *enabled* when the composite system can execute it while the SuS cannot. SuS actions are *adapted* when either their payload differs from that of the composite system, or when the action is rerouted through a different port.

We argue that implementing action enabling, disabling and adaptation differs according to whether the action is an input or an output; see Fig. 3. Enforcing actions instigated by the SuS—such as outputs—is more straightforward. Figure 3(a), (b) and (c) *resp.* state that disabling an output can be achieved by suppressing it, adapting an output amounts to replacing the payload or redirecting it to a different port, whereas output enabling can be attained via an insertion. However, enforcing actions instigated by the environment such as inputs

is harder. In Fig. 3(d), we propose to *disable* an input by concealing the input port. Since this may block the SuS from progressing, the instrumented monitor may additionally *insert* a default input to unblock the system, Fig. 3(e). Input *adaptation*, Fig. 3(f), is also attained via a *replacement* (applied in the opposite direction to the output case). Inputs can also be *enabled* (when the SuS is unable to carry them out), Fig. 3(g), by having the monitor accept the input in question and then *suppress* it: from the environment's perspective, the input would be effected.

Syntax

$$m, n \in \text{TRN} ::= (p, c, p').m \mid \sum_{i \in I} m_i \text{ (} I \text{ is a finite index set)} \mid \text{rec } X.m \mid X$$

Dynamics

$$\begin{array}{c} \text{ESEL} \frac{m_j \xrightarrow{\gamma \blacktriangleright \gamma'} n_j}{\sum_{i \in I} m_i \xrightarrow{\gamma \blacktriangleright \gamma'} n_j} \quad j \in I \qquad \text{EREC} \frac{m \{\text{rec } X.m / X\} \xrightarrow{\gamma \blacktriangleright \gamma'} n}{\text{rec } X.m \xrightarrow{\gamma \blacktriangleright \gamma'} n} \\ \text{ETRN} \frac{\text{match}(p, \gamma) = \sigma \quad c\sigma \Downarrow \text{true} \quad \gamma' = \pi\sigma}{(p, c, \pi).m \xrightarrow{\gamma \blacktriangleright \gamma'} m\sigma} \end{array}$$

Instrumentation

$$\begin{array}{c} \text{BITRNO} \frac{s \xrightarrow{\text{b!}w} s' \quad m \xrightarrow{(\text{b!}w) \blacktriangleright (\text{a!}v)} n}{m[s] \xrightarrow{\text{a!}v} n[s']} \qquad \text{BITRNI} \frac{m \xrightarrow{(\text{a?}v) \blacktriangleright (\text{b?}w)} n \quad s \xrightarrow{\text{b?}w} s'}{m[s] \xrightarrow{\text{a?}v} n[s']} \\ \text{BIDISO} \frac{s \xrightarrow{\text{a!}v} s' \quad m \xrightarrow{(\text{a!}v) \blacktriangleright \bullet} n}{m[s] \xrightarrow{\tau} n[s']} \qquad \text{BIDISI} \frac{m \xrightarrow{\bullet \blacktriangleright (\text{a?}v)} n \quad s \xrightarrow{\text{a?}v} s'}{m[s] \xrightarrow{\tau} n[s']} \\ \text{BIENO} \frac{m \xrightarrow{\bullet \blacktriangleright (\text{a!}v)} n}{m[s] \xrightarrow{\text{a!}v} n[s]} \qquad \text{BIENI} \frac{m \xrightarrow{(\text{a?}v) \blacktriangleright \bullet} n}{m[s] \xrightarrow{\text{a?}v} n[s]} \qquad \text{BIASY} \frac{s \xrightarrow{\tau} s'}{m[s] \xrightarrow{\tau} m[s']} \\ \text{BIDDEF} \frac{s \xrightarrow{\text{a!}v} s' \quad m \xrightarrow{\text{a!}v} \quad \forall \mathbf{b} \in \text{PORT}, w \in \text{VAL} \cdot m \xrightarrow{\bullet \blacktriangleright \text{b!}w}}{m[s] \xrightarrow{\text{a!}v} \text{id}[s']}$$

Fig. 4. A bidirectional instrumentation model for enforcement monitors.

Figure 4 presents an operational model for the bidirectional instrumentation proposal of Fig. 3 in terms of (symbolic) transducers¹. Transducers, $m, n \in \text{TRN}$, are monitors that define *symbolic transformation triples*, (p, c, π) , consisting of an action *pattern* p , *condition* c , and a *transformation action* π . Conceptually, the action pattern and condition determine the range of system (input or output)

¹ These transducers were originally introduced in [5] for unidirectional enforcement.

actions upon which the transformation should be applied, while the transformation action specifies the transformation that should be applied. The symbolic transformation pattern p is an extended version of those definable in symbolic actions, that may also include \bullet ; when $p = \bullet$, it means that the monitor can act independently from the system to insert the action specified by the transformation action. Transformation actions are possibly open actions (i.e., actions with possibly free variable such as $x?v$ or $a!x$) or the special action \bullet ; the latter represents the suppression of the action specified by p . We assume a well-formedness constraint where, for every $(p, c, \pi).m$, p and π cannot both be \bullet , and when neither is, they are of the *same* type i.e., an input (resp. output) pattern and action. Examples of well-formed symbolic transformations are $(\bullet, \text{true}, a?v)$, $((x)!(y), \text{true}, \bullet)$ and $((x)!(y), \text{true}, a!v)$.

The monitor transition rules in Fig. 4 assume closed terms, i.e., every *transformation-prefix transducer* of the form $(p, c, \pi).m$ must obey the constraint $(\mathbf{fv}(c) \cup \mathbf{fv}(\pi) \cup \mathbf{fv}(m)) \subseteq \mathbf{bv}(p)$ and similarly for recursion variables X and $\text{rec } X.m$. Each transformation-prefix transducer yields an LTS with labels of the form $\gamma \blacktriangleright \gamma'$, where $\gamma, \gamma' \in (\text{ACT} \cup \{\bullet\})$. Intuitively, transition $m \xrightarrow{\gamma \blacktriangleright \gamma'} n$ denotes the way that a transducer in state m *transforms* the action γ into γ' while transitioning to state n . The transducer action $\alpha \blacktriangleright \beta$ represents the *replacement* of α by β , $\alpha \blacktriangleright \alpha$ denotes the *identity* transformation, whereas $\alpha \blacktriangleright \bullet$ and $\bullet \blacktriangleright \alpha$ respectively denote the *suppression* and *insertion* transformations of action α . The key transition rule in Fig. 4 is ETRN. It states that the transformation-prefix transducer $(p, c, \pi).m$ transforms action γ into a (potentially) different action γ' and reduces to state $m\sigma$, whenever γ matches pattern p , i.e., $\text{match}(p, \gamma) = \sigma$, and satisfies condition c , i.e., $c\sigma \Downarrow \text{true}$. Action γ' results from instantiating the free variables in π as specified by σ , i.e., $\gamma' = \pi\sigma$. The remaining rules for selection (ESEL) and recursion (EREC) are standard. We employ the shorthand notation $m \xrightarrow{\gamma} n$ to mean $\nexists \gamma', n$ such that $m \xrightarrow{\gamma \blacktriangleright \gamma'} n$. Moreover, for the semantics of Fig. 4, we can encode the identity monitor, id , as $\text{rec } Y.((x)!(y), \text{true}, x!y).Y + ((x)?(y), \text{true}, x?y).Y$. As a shorthand notation, we write $(p, c).m$ instead of $(p, c, \pi).m$ when all the binding occurrences (x) in p correspond to free occurrences x in π , thus denoting an identity transformation. Similarly, we elide c whenever $c = \text{true}$.

The first contribution of this work lies in the new *instrumentation relation* of Fig. 4, linking the behaviour of the SuS s with that of a monitor m : the term $m[s]$ denotes their composition as a *monitored system*. Crucially, the instrumentation rules in Fig. 4 give us a semantics in terms of an LTS over the actions $\text{ACT} \cup \{\tau\}$, in line with the LTS semantics of the SuS. Following Fig. 3(b), rule BITRNO states that if the SuS transitions with an output $b!w$ to s' and the transducer can *replace* it with $a!v$ and transition to n , the *adapted* output can be externalised so that the composite system $m[s]$ transitions over $a!v$ to $n[s']$. Rule BIDISO states that if s performs an output $a!v$ that the monitor *can suppress*, the instrumentation withholds this output and the composite system silently transitions; this amounts to action *disabling* as outlined in Fig. 3(a). Rule BIENO is dual, and it *enables* the output $a!v$ on the SuS as outlined in Fig. 3(c): it aug-

ments the composite system $m[s]$ with an output $a!v$ whenever m can *insert* $a!v$, independently of the behaviour of s .

Rule **BIDEF** is analogous to standard rules for premature monitor termination [1, 20–22], and accounts for underspecification of transformations. We, however, restrict defaulting (termination) to output actions performed by the SuS exclusively, *i.e.*, a monitor only defaults to id when it cannot react to or enable a system output. By forbidding the monitor from defaulting upon unspecified inputs, the monitor is able to *block* them from becoming part of the composite system’s behaviour. Hence, any input that the monitor is unable to react to, *i.e.*, $m \xrightarrow{a?v\gamma}$, is considered as being *invalid and blocked* by default. This technique is thus used to implement Fig. 3(d). To avoid disabling valid inputs unnecessarily, the monitor must therefore explicitly define symbolic transformations that cover *all* the valid inputs of the SuS. Note, that rule **BIASY** still allows the SuS to silently transition independently of m . Following Fig. 3(f), rule **BITRNI** adapts inputs, provided the SuS can accept the adapted input. Similarly, rule **BIENI** *enables* an input on a port a as described in Fig. 3(g): the composite system accepts the input while suppressing it from the SuS. Rule **BIDISI** allows the monitor to generate a default input value v and forward it to the SuS on a port a , thereby unblocking it; externally, the composite system silently transitions to some state, following Fig. 3(e).

Example 2. Consider the following action disabling transducer m_d , that repeatedly disables every output performed by the system via the branch $((-)!(-), \bullet).Y$. In addition, it limits inputs to those on port b via the input branch $(b?(-)).Y$; inputs on other ports are disabled since none of the relevant instrumentation rules in Fig. 4 can be applied.

$$m_d \stackrel{\text{def}}{=} \text{rec } Y.(b?(-)).Y + ((-)!(-), \bullet).Y$$

When instrumented with s_c from Example 1, m_d blocks its initial input, *i.e.*, we have $m_d[s_c] \not\xrightarrow{\alpha}$ for any α . In the case of s_b , the composite system $m_d[s_b]$ can only input requests on port b , such as the termination request $m_d[s_b] \xrightarrow{b?\text{cls}} m_d[\text{nil}]$.

$$\begin{aligned} m_{dt} &\stackrel{\text{def}}{=} \text{rec } X.(((x)?(y_1), x \neq b).(((x_1)?(-), x_1 \neq x).\text{id} + (x!(y_2)).m'_{dt}) + (b?(-).\text{id})) \\ m'_{dt} &\stackrel{\text{def}}{=} (x!(-), \bullet).m_d + ((-)?(-).\text{id} + (b!(y_3), y_3 = (\log, y_1, y_2)).X) \end{aligned}$$

By defining branch $(b?(-).\text{id})$, the more elaborate monitor m_{dt} (above) allows the SuS to immediately input on port b (possibly carrying a termination request). At the same time, the branch prefixed by $((x)?(y_1), x \neq b)$ permits the SuS to input the first request via any port $x \neq b$, subsequently blocking inputs on the same port x (without deterring inputs on other ports) via the input branch $((x_1)?(-), x_1 \neq x).\text{id}$. In conjunction to this branch, m_{dt} defines another branch $(x!(y_2)).m'_{dt}$ to allow outputs on the port bound to variable x . The continuation monitor m'_{dt} then defines the suppression branch $(x!(-), \bullet).m_d$ by which it disables any *redundant* response that is output following the first one. Since it also

defines branches $(\mathbf{b}!(y_3), y_3=(\log, y_1, y_2)).X$ and $((-)?(-)).\text{id}$, it does not affect log events or further inputs that occur immediately after the first response.

When instrumented with system s_c from Example 1, m_{dt} allows the composite system to perform the first input but then blocks the second one, permitting only input requests on channel b , e.g., $m_{\text{dt}}[s_c] \xrightarrow{a?v} \cdot \xrightarrow{b?\text{cls}} \text{id}[\text{nil}]$. It also disables the first redundant response of system s_b while transitioning to m_{d} , which proceeds to suppress every subsequent output (including log actions) while blocking every other port except b , i.e., $m_{\text{dt}}[s_b] \xrightarrow{a?v} \cdot \xrightarrow{a!w} \cdot \xrightarrow{\tau} m_{\text{d}}[\mathbf{b}!(\log, v, w).s_a] \xrightarrow{\tau} m_{\text{d}}[s_a] \xrightarrow{a?v} \cdot$ (for every port a where $a \neq b$ and any value v). Rule iDEF allows it to default when handling unspecified outputs, e.g., for system $\mathbf{b}!(\log, v, w).s_a$ the composite system can still perform $m_{\text{dt}}[\mathbf{b}!(\log, v, w).s_a] \xrightarrow{\mathbf{b}!(\log, v, w)} \text{id}[s_a]$.

$$\begin{aligned} m_{\text{det}} &\stackrel{\text{def}}{=} \text{rec } X.(((x)?(y_1), x \neq \mathbf{b}).m'_{\text{det}} + (\mathbf{b}?(-)).\text{id}) \\ m'_{\text{det}} &\stackrel{\text{def}}{=} \text{rec } Y_1.(\underline{\bullet, x?v_{\text{def}}}.Y_1 + (x!(y_2)).m''_{\text{det}} + ((x_1)?(-), x_1 \neq x).\text{id}) \\ m''_{\text{det}} &\stackrel{\text{def}}{=} \text{rec } Y_2.(\underline{(x!(-), x \neq \mathbf{b}, \bullet).Y_2} + (\mathbf{b}!(y_3), y_3=(\log, y_1, y_2)).X + ((-)?(-)).\text{id}) \end{aligned}$$

Monitor m_{det} (above) is similar to m_{dt} but instead employs a loop of suppressions (underlined in m''_{det}) to disable further responses until a log or termination input is made. When composed with s_b , it permits the log action to go through:

$$m_{\text{det}}[s_b] \xrightarrow{a?v} \cdot \xrightarrow{a!w} \cdot \xrightarrow{\tau} m''_{\text{det}}[\mathbf{b}!(\log, v, w).s_b] \xrightarrow{\mathbf{b}!(\log, v, w)} m_{\text{det}}[s_b].$$

m_{det} also defines a branch prefixed by the insertion transformation $(\bullet, x?v_{\text{def}})$ (underlined in m'_{det}) where v_{def} is a default input domain value. This permits the instrumentation to silently unblock the SuS when this is waiting for a request following an unanswered one. In fact, when instrumented with s_c , m_{det} not only forbids invalid input requests, but it also (internally) unblocks s_c by supplying the required input via the added insertion branch. This allows the composite system to proceed, as shown below (where $s'_a \stackrel{\text{def}}{=} y := \text{ans}(v_{\text{def}}).a!y.\mathbf{b}!(\log, v_{\text{def}}, y).s_a$):

$$\begin{aligned} m_{\text{det}}[s_c] &\xrightarrow{a?v} \text{rec } Y.((\bullet, a?v_{\text{def}}).Y + (a!(y_2)).m''_{\text{det}} + (\mathbf{b}?(-)).\text{id})[s_a] \\ &\xrightarrow{\tau} \text{rec } Y.((\bullet, a?v_{\text{def}}).Y + (a!(y_2)).m''_{\text{det}} + (\mathbf{b}?(-)).\text{id})[s'_a] \\ &\xrightarrow{a!\text{ans}(v_{\text{def}}).\mathbf{b}!(\log, v_{\text{def}}, y)} m_{\text{det}}[s_a] \quad \blacksquare \end{aligned}$$

Although in this paper we mainly focus on action disabling monitors, using our model one can also define action enabling and adaptation monitors.

Example 3. Consider now transducers m_e and m_a below:

$$\begin{aligned} m_e &\stackrel{\text{def}}{=} ((x)?(y), x \neq \mathbf{b}, \bullet).(\bullet, x!\text{ans}(y)).(\bullet, \mathbf{b}!(\log, y, \text{ans}(y))).\text{id} \\ m_a &\stackrel{\text{def}}{=} \text{rec } X.(\mathbf{b}?(y), a?y).X + (a!(y), \mathbf{b}!y).X. \end{aligned}$$

Once instrumented, m_e first uses a suppression to enable an input on any port $x \neq \mathbf{b}$ (but then gets discarded). It then automates a response by inserting an

answer followed by a log action. Concretely, when composed with $r \in \{s_{\mathbf{b}}, s_{\mathbf{c}}\}$ from Example 1, the execution of the composite system can only start as follows, for some channel name $c \neq \mathbf{b}$, values v and $w = \text{ans}(v)$:

$$m_{\mathbf{e}}[r] \xrightarrow{c?v} (\bullet, c!w).(\bullet, \mathbf{b}!(\log, v, w)).\text{id}[r] \xRightarrow{c!w} (\bullet, \mathbf{b}!(\log, v, w)).\text{id}[r] \xrightarrow{\mathbf{b}!(\log, v, w)} \text{id}[r].$$

By contrast, $m_{\mathbf{a}}$ uses action adaptation to redirect the inputs and outputs from the SuS through port \mathbf{b} : it allows the composite system to exclusively input values on port \mathbf{b} forwarding them to the SuS on port \mathbf{a} , and dually allowing outputs from the SuS on port \mathbf{a} to rerout them to port \mathbf{b} . As a result, a composite system can *only* communicate on port \mathbf{b} . *E.g.*, $m_{\mathbf{a}}[s_{\mathbf{c}}] \xrightarrow{\mathbf{b}?v_1} m_{\mathbf{a}}[s_{\mathbf{a}}] \xrightarrow{\mathbf{b}?v_2.\mathbf{b}!w_2.\mathbf{b}!(\log, v_2, w_2)} m_{\mathbf{a}}[s_{\mathbf{a}}]$ and $m_{\mathbf{a}}[s_{\mathbf{a}}] \xrightarrow{\mathbf{b}?v_1.\mathbf{b}!w_1.\mathbf{b}!(\log, v_1, w_1)} m_{\mathbf{a}}[s_{\mathbf{b}}]$. ■

4 Enforcement

We are concerned with extending the enforceability result obtained in prior work [5] to the extended setting of bidirectional enforcement. The *enforceability* of a logic rests on the relationship between the semantic behaviour specified by the logic on the one hand, and the ability of the operational mechanism (that of Sect. 3 in this case) to enforce the specified behaviour on the other.

Definition 2 (Enforceability [5]). *A formula φ is enforceable iff there exists a transducer m such that m adequately enforces φ . A logic \mathcal{L} is enforceable iff every formula $\varphi \in \mathcal{L}$ is enforceable.* ■

Since we have limited control over the SuS that a monitor is composed with, “ m adequately enforces φ ” should hold for *any* (instrumentable) system. In [5] we stipulate that any notion of adequate enforcement should at least entail soundness.

Definition 3 (Sound Enforcement [5]). *Monitor m soundly enforces a satisfiable formula φ , denoted as $\text{senf}(m, \varphi)$, iff for every state $s \in \text{SYS}$, it is the case that $m[s] \in \llbracket \varphi \rrbracket$.* ■

Example 4. Although showing that a monitor soundly enforces a formula should consider *all* systems, we give an intuition based on $s_{\mathbf{a}}, s_{\mathbf{b}}, s_{\mathbf{c}}$ for formula φ_1 from Example 1 (restated below) where $s_{\mathbf{a}} \in \llbracket \varphi_1 \rrbracket$ (hence $\llbracket \varphi_1 \rrbracket \neq \emptyset$) and $s_{\mathbf{b}}, s_{\mathbf{c}} \notin \llbracket \varphi_1 \rrbracket$.

$$\begin{aligned} \varphi_1 &\stackrel{\text{def}}{=} \max X. [((x)?(y_1), x \neq \mathbf{b})]([((x)?(-))\text{ff} \wedge [(x! (y_2))]])\varphi'_1 \\ \varphi'_1 &\stackrel{\text{def}}{=} [((x!(-))\text{ff} \wedge [(\mathbf{b}!(y_3), y_3 = (\log, y_1, y_2))]])X \end{aligned}$$

Recall the transducers $m_{\mathbf{e}}, m_{\mathbf{a}}, m_{\mathbf{d}}, m_{\mathbf{dt}}$ and $m_{\mathbf{det}}$ from Example 2:

- $m_{\mathbf{e}}$ is *unsound* for φ_1 . When composed with $s_{\mathbf{b}}$, it produces two consecutive output replies (underlined), meaning that $m_{\mathbf{e}}[s_{\mathbf{b}}] \notin \llbracket \varphi_1 \rrbracket$: $m_{\mathbf{e}}[s_{\mathbf{b}}] \xrightarrow{t_{\mathbf{e}}^1} \text{id}[s_{\mathbf{b}}]$ where $t_{\mathbf{e}}^1 \stackrel{\text{def}}{=} c?v_1.c!\text{ans}(v_1).\mathbf{b}!(\log, v_1, \text{ans}(v_1)).\mathbf{a}?v_2.\mathbf{a}!\underline{w_2}.\mathbf{a}!\underline{w_2}$. Similarly,

- $m_e[s_c] \notin \llbracket \varphi_1 \rrbracket$ since the $m_e[s_c]$ executes the erroneous trace with two consecutive inputs on port **a** (underlined): $c?v_1.c!ans(v_1).b!(\log, v_1, ans(v_1)).\underline{a?v_2.a?v_3}$. This demonstrates that $m_e[s_c]$ can still input two consecutive requests on port **a** (underlined). Either one of these counter examples *disproves* $\text{senf}(m_e, \varphi_1)$.
- m_a turns out to be *sound* for φ_1 because once instrumented, the resulting composite system is adapted to only interact on port **b**. In fact we have $m_a[s_a], m_a[s_b], m_a[s_c] \in \llbracket \varphi_1 \rrbracket$. Monitors m_d, m_{dt} and m_{det} are also *sound* for φ_1 . Whereas, m_d prevents the violation of φ_1 by also blocking all input ports except **b**, m_{dt} and m_{det} achieve the same goal by disabling the invalid consecutive requests and answers that occur on a specific port (except **b**). ■

By itself, sound enforcement is a weak criterion because it does not regulate the *extent* to which enforcement is applied. More specifically, although m_d from Example 2 is sound, it needlessly modifies the behaviour of s_a even though s_a satisfies φ_1 : by blocking the initial input of s_a , m_d causes it to block indefinitely. The requirement that a monitor should not modify the behaviour of a system that satisfies the property being enforced can be formalised using a transparency criterion.

Definition 4 (Transparent Enforcement [5]). A monitor m transparently enforces a formula φ , $\text{tenf}(m, \varphi)$, iff for all $s \in \text{SYS}$, $s \in \llbracket \varphi \rrbracket$ implies $m[s] \sim s$. □

Example 5. As argued earlier, s_a suffices to disprove $\text{tenf}(m_d, \varphi_1)$. Monitor m_a from Example 3 also breaches Definition 4: although $s_a \in \llbracket \varphi_1 \rrbracket$, we have $m_a[s_a] \not\sim s_a$ since for any value v and w , $s_a \xrightarrow{b?v} \cdot \xrightarrow{b!w}$ but $m_a[s_a] \xrightarrow{b?v} \cdot \xrightarrow{b!w} \cdot$. By contrast, monitors m_{dt} and m_{det} turn out to satisfy Definition 4 as they only intervene when it becomes apparent that a violation will occur. For instance, they only disable inputs on a specific port, as a precaution, following an unanswered request on the same port, and they only disable the redundant responses that are produced after the first response to a request. ■

By some measures, Definition 4 is still a relatively weak requirement since it only limits transparency requirements to well-behaved systems, and disregards enforcement behaviour for systems that violate a property. For instance, consider monitor m_{dt} from Example 2 and system s_b from Example 1. At runtime s_b can exhibit the following invalid behaviour: $s_b \xrightarrow{t_1} b!(\log, v, w).s_a$ where $t_1 \stackrel{\text{def}}{=} a?v.a!w.a!w$. In order to rectify this violating behaviour wrt. formula φ_1 , it suffices to use a monitor that disables *one* of the responses in t_1 , i.e., $a!w$. Following this disabling, no further modifications are required since the SuS reaches a state that does not violate the remainder of the formula φ_1 , i.e., $b!(\log, v, w).s_a \in \llbracket \text{after}(\varphi_1, t_1) \rrbracket$. However, when instrumented with m_{dt} , this monitor does not only disable the invalid response, namely $m_{dt}[s_b] \xrightarrow{a?v.a!w} \cdot$ but subsequently disables every other action by reaching $m_d[b!(\log, v, w).s_a]$, but subsequently disables every other action by reaching $m_d, m_d[b!(\log, v, w).s_a] \xrightarrow{\tau} m_d[s_a]$. To this end, we introduce the novel requirement of *eventual transparency*.

Definition 5 (Eventually Transparent Enforcement). *Monitor m enforces property φ in an eventually transparent way, $\text{evtenf}(m, \varphi)$, iff for all systems s, s' , traces t and monitors m' , $m[s] \xrightarrow{t} m'[s']$ and $s' \in \llbracket \text{after}(\varphi, t) \rrbracket$ imply $m'[s'] \sim s'$. ■*

Example 6. We have already argued why m_{dt} does not adhere to eventual transparency via the counterexample s_{b} . This is not the case for m_{det} . Although the universal quantification over all systems and traces make it hard to prove this property, we get an intuition of why this is the case from s_{b} : when $m_{\text{det}}[s_{\text{b}}] \xrightarrow{a?v_1.a!w_1} \cdot \xrightarrow{\tau} m''_{\text{det}}[\mathbf{b}!(\log, v_1, w_1).s_{\text{a}}]$ we have $\mathbf{b}!(\log, v_1, w_1).s_{\text{a}} \in \llbracket \text{after}(\varphi_1, a?v_1.a!w_1) \rrbracket$ and that $m''_{\text{det}}[\mathbf{b}!(\log, v_1, w_1).s_{\text{a}}] \sim \mathbf{b}!(\log, v_1, w_1).s_{\text{a}}$. ■

Corollary 1. *For all monitors $m \in \text{TRN}$ and properties $\varphi \in \text{SHML}$, $\text{evtenf}(m, \varphi)$ implies $\text{tenf}(m, \varphi)$. □*

Along with Definition 3 (soundness), Definition 5 (eventual transparency) makes up our definition for “ m (adequately) enforces φ ”. From Corollary 1, it follows that this definition is stricter than the one given in [5].

Definition 6 (Adequate Enforcement). *A monitor m (adequately) enforces property φ iff it adheres to (i) soundness, Definition 3, and (ii) eventual transparency, Definition 5. ■*

5 Synthesising Action Disabling Monitors

Although Definition 2 enables us to rule out erroneous monitors that purport to enforce a property, the universal quantifications over all systems in Definitions 3 and 5 make it difficult to prove that a monitor does indeed enforce a property correctly in a bidirectional setting. Establishing that a formula is enforceable, Definition 6, involves a further existential quantification over a monitor that enforces it correctly. Moreover, establishing the enforceability of a logic entails yet another universal quantification, on all the formulas in the logic.

We address these problems through an *automated synthesis procedure* that produces an enforcement monitor for *every* SHML formula. We also show that the synthesised monitors are correct, according to Definition 6. For a unidirectional setting, it has been shown that monitors that only administer *omissions* are expressive enough to enforce *safety properties* [5, 19, 25, 32]. Analogously, for our bidirectional case, we restrict ourselves to action disabling monitors and show that they can enforce *any* property expressed in terms of SHML.

Our synthesis procedure is compositional, meaning that the monitor synthesis of a composite formula is defined in terms of the enforcement monitors generated from its constituent sub-formulas. Compositionality simplifies substantially our correctness analysis of the generated monitors (e.g., we can use standard inductive proof techniques). In order to ease a compositional definition, our synthesis procedure is defined in terms of a variant of SHML called SHML_{nf}: it is

a normalised syntactic subset of sHML that is still as expressive as sHML [2]. An automated procedure to translate an sHML formula into a corresponding sHML_{nf} one (with the same semantic meaning) is given in [2, 5].

Definition 7 (sHML Normal Form). *The set of normalised sHML formulas is generated by the following grammar:*

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

In addition, sHML_{nf} formulas are required to satisfy the following conditions:

1. Every branch in $\bigwedge_{i \in I} [p_i, c_i] \varphi_i$, must be disjoint, i.e., for every $i, j \in I$, $i \neq j$ implies $\llbracket (p_i, c_i) \rrbracket \cap \llbracket (p_j, c_j) \rrbracket = \emptyset$.
2. For every $\max X. \varphi$ we have $X \in \mathbf{fv}(\varphi)$. ■

In a (closed) sHML_{nf} formula, the basic terms tt and ff can never appear unguarded unless they are at the top level (e.g., we can never have $\varphi \wedge \text{ff}$ or $\max X_0. \dots \max X_n. \text{ff}$). Modal operators are combined with conjunctions into one construct $\bigwedge_{i \in I} [p_i, c_i] \varphi_i$ that is written as $[p_0, c_0] \varphi_0 \wedge \dots \wedge [p_n, c_n] \varphi_n$ when $I = \{0, \dots, n\}$ and simply as $[p_0, c_0] \varphi$ when $|I| = 1$. The conjunct modal guards must also be *disjoint* so that *at most one* necessity guard can satisfy any particular visible action. Along with these restrictions, we still assume that sHML_{nf} fixpoint variables are guarded, and that for every $((x)?(y), c)$, $y \notin \mathbf{fv}(c)$.

Example 7. The formula φ_3 defines a recursive property stating that, following an input on port **a** (carrying any value), prohibits that the system outputs a value of 4 (on any port), unless the output is made on port **a** with a value that is not equal to 3 (in which cases, it recurses).

$$\varphi_3 \stackrel{\text{def}}{=} \max X. [((x_1)?(y_1), x_1 = \mathbf{a})] \left(\begin{array}{l} [((x_2)!(y_2), x_2 = \mathbf{a} \wedge y_2 \neq 3)] X \\ \wedge [((x_3)!(y_3), y_3 = 4)] \text{ff} \end{array} \right)$$

φ_3 is not an sHML_{nf} formula since its conjunction is not disjoint (e.g., the action **a**!4 satisfies both branches). Still, we can reformulate φ_3 as $\varphi'_3 \in \text{sHML}_{\text{nf}}$:

$$\varphi'_3 \stackrel{\text{def}}{=} \max X. [((x_1)?(y_1), x_1 = \mathbf{a})] \left(\begin{array}{l} [((x_4)!(y_4), x_4 = \mathbf{a} \wedge y_4 \neq 4)] X \\ \wedge [((x_4)!(y_4), x_4 = \mathbf{a} \wedge y_4 = 4)] \text{ff} \end{array} \right)$$

where x_4 and y_4 are fresh variables. ■

Our monitor synthesis function in Definition 8 converts an sHML_{nf} formula φ into a transducer m . This conversion also requires information regarding the input ports of the SuS, as this is used to add the necessary insertion branches that silently unblock the SuS at runtime. The synthesis function must therefore be supplied with this information in the form of a *finite* set of input ports $\Pi \subset \text{PORT}$, which then relays this information to the resulting monitor.

Definition 8. The synthesis function $\llbracket - \rrbracket : \text{SHML}_{\text{nf}} \times \mathcal{P}_{\text{fin}}(\text{PORT}) \rightarrow \text{TRN}$ is defined inductively as:

$$\llbracket X, \Pi \rrbracket \stackrel{\text{def}}{=} X \quad \llbracket \text{tt}, \Pi \rrbracket \stackrel{\text{def}}{=} \llbracket \text{ff}, \Pi \rrbracket \stackrel{\text{def}}{=} \text{id} \quad \llbracket \max X.\varphi, \Pi \rrbracket \stackrel{\text{def}}{=} \text{rec } X.\llbracket \varphi, \Pi \rrbracket$$

$$\llbracket \varphi = \bigwedge_{i \in I} [(p_i, c_i)]\varphi_i, \Pi \rrbracket \stackrel{\text{def}}{=} \text{rec } Y. \left(\sum_{i \in I} \begin{cases} \text{dis}(p_i, c_i, Y, \Pi) & \text{if } \varphi_i = \text{ff} \\ (p_i, c_i).\llbracket \varphi_i, \Pi \rrbracket & \text{otherwise} \end{cases} \right) + \text{def}(\varphi)$$

$$\text{where } \text{dis}(p, c, m, \Pi) \stackrel{\text{def}}{=} \begin{cases} (p, c, \bullet).m & \text{if } p = (x)!(y) \\ \sum_{b \in \Pi} (\bullet, c\{b/x\}, b?v_{\text{def}}).m & \text{if } p = (x)?(y) \end{cases} \text{ and}$$

$$\text{def}(\bigwedge_{i \in I} [(x_i)?(y_i), c_i]\varphi_i \wedge \psi) \stackrel{\text{def}}{=} \begin{cases} ((-)?(-)).\text{id} & \text{when } I = \emptyset \\ ((x)?(y), \bigwedge_{i \in I} (\neg c_i\{x/x_i, y/y_i\})).\text{id} & \text{otherwise} \end{cases}$$

where ψ has no conjuncts starting with an input modality, variables x and y are fresh, and v_{def} is a default value. \blacksquare

The definition above assumes a bijective mapping between formula variables and monitor recursion variables. Normalised conjunctions, $\bigwedge_{i \in I} [p_i, c_i]\varphi_i$, are synthesised as a *recursive summation* of monitors, i.e., $\text{rec } Y. \sum_{i \in I} m_i$, where Y is fresh, and every branch m_i can be one of the following:

- (i) when m_i is derived from a branch of the form $[p_i, c_i]\varphi_i$ where $\varphi_i \neq \text{ff}$, the synthesis produces a monitor with the *identity transformation* prefix, (p_i, c_i) , followed by the monitor synthesised from the continuation φ_i , i.e., $\llbracket \varphi_i, \Pi \rrbracket$;
- (ii) when m_i is derived from a violating branch of the form $[p_i, c_i]\text{ff}$, the synthesis produces an *action disabling transformation* via $\text{dis}(p_i, c_i, Y, \Pi)$.

Specifically, in clause (ii) the dis function produces either a *suppression transformation*, (p_i, c_i, \bullet) , when p_i is an *output* pattern, $(x)!(y)$, or a *summation of insertions*, $\sum_{b \in \Pi} (\bullet, c_i\{b/x_i\}, b?v_{\text{def}}).m_i$, when p_i is an *input* pattern, $(x_i)?(y_i)$. The former signifies that the monitor must react to and suppress every matching (invalid) system output thus stopping it from reaching the environment. By not synthesising monitor branches that react to the erroneous input, the latter allows the monitor to hide the input synchronisations from the environment. At the same time, the synthesised insertion branches insert a default domain value v_{def} on every port $a \in \Pi$ whenever the branch condition $c_i\{b/x_i\}$ evaluates to true at runtime. This stops the monitor from blocking the resulting composite system unnecessarily.

This blocking mechanism can, however, block *unspecified* inputs, i.e., those that do not satisfy any modal necessity in the normalised conjunction. This is undesirable since the unspecified actions do not contribute towards a safety violation and, instead, lead to its trivial satisfaction. To prevent this, the *default monitor* $\text{def}(\varphi)$ is also added to the resulting summation. Concretely, the def function produces a *catch-all* identity monitor that forwards an input to the SuS whenever it satisfies the negation of *all* the conditions associated with modal necessities for input patterns in the normalised conjunction. This condition is

constructed for a normalised conjunction of the form $\bigwedge_{i \in I} [(x_i)?(y_i), c_i] \varphi_i \wedge \psi$ (assuming that ψ does not include further input modalities). Otherwise, if none of the conjunct modalities define an input pattern, every input is allowed, i.e., the default monitor becomes $((-)?(-)).\text{id}$, which transitions to id after forwarding the input to the SuS.

Example 8. Recall (the full version of) formula φ_1 from Example 1.

$$\begin{aligned} \varphi_1 &\stackrel{\text{def}}{=} \max X. [((x)?(y_1), x \neq \mathbf{b})] [(((x_1)?(-), x_1 = x)] \text{ff} \wedge [((x_2)!(y_2), x_2 = x)] \varphi'_1 \\ \varphi'_1 &\stackrel{\text{def}}{=} ([((x_3)!(y_3), x_3 = x)] \text{ff} \wedge [((x_4)!(y_3), x_4 = \mathbf{b} \wedge y_3 = (\log, y_1, y_2))] X) \end{aligned}$$

For any arbitrary set of ports Π , the synthesis of Definition 8 produces the following monitor.

$$\begin{aligned} m_{\varphi_1} &\stackrel{\text{def}}{=} \text{rec } X. \text{rec } Z. (((x)?(y_1), x \neq \mathbf{b}). \text{rec } Y_1. m'_{\varphi_1}) + ((x_{\text{def}})?(-), x_{\text{def}} = \mathbf{b}). \text{id} \\ m'_{\varphi_1} &\stackrel{\text{def}}{=} \sum_{\mathbf{a} \in \Pi} (\bullet, \mathbf{a} = x, \mathbf{a}?v_{\text{def}}). Y_1 + ((x_2)!(y_2), x_2 = x). \text{rec } Y_2. m''_{\varphi_1} + ((x_{\text{def}})?(-), x_{\text{def}} \neq x). \text{id} \\ m''_{\varphi_1} &\stackrel{\text{def}}{=} ((x_3)!(y_3), x_3 = x, \bullet). Y_2 + ((x_4)!(y_3), x_4 = \mathbf{b} \wedge y_3 = (\log, y_1, y_2)). X + ((-)?(-)). \text{id} \end{aligned}$$

Monitor m_{φ_1} can be optimised by removing redundant recursive constructs such as $\text{rec } Z.$ that are introduced mechanically by our synthesis. \blacksquare

Monitor m_{φ_1} from Example 8 (with $(\varphi_1, \Pi) = m_{\varphi_1}$) is very similar to m_{det} of Example 2, differing only in how it defines its insertion branches for unblocking the SuS. For instance, if we consider $\Pi = \{\mathbf{b}, \mathbf{c}\}$, (φ_1, Π) would synthesise two insertion branches, namely $(\bullet, \mathbf{b} = x, \mathbf{b}?v_{\text{def}})$ and $(\bullet, \mathbf{c} = x, \mathbf{c}?v_{\text{def}})$. By contrast, m_{det} attains the same result more succinctly via the single insertion branch $(\bullet, x?v_{\text{def}})$. Importantly, our synthesis provides the witness monitors needed to show enforceability.

Theorem 1 (Enforceability). *sHML is bidirectionally enforceable using the monitors and instrumentation of Fig. 4.* \square

6 Conclusions and Related Work

This work extends the framework presented in the precursor to this work [5] to the setting of bidirectional enforcement where observable actions such as inputs and outputs require different treatment. We achieve this by:

1. augmenting substantially our instrumentation relation (Fig. 4);
2. refining our definition of enforcement to incorporate transparency over violating systems (Definition 6); and
3. providing a more extensive synthesis function (Definition 8) that is proven correct (Theorem 1).

Future work. There are a number of possible avenues for extending our work. One immediate step would be the implementation of the monitor operational model presented in Sect. 3 together with the synthesis function described in Sect. 5. This effort should be integrated it within the detectEr tool suite [10, 14–16]. This would allow us to assess the overhead induced by our proposed bidirectional monitoring [4]. Another possible direction would be the development of behavioural theories for the transducer operational model presented in Sect. 3, along the lines of the refinement preorders studied in earlier work on sequence recognisers [3, 20, 21]. Finally, applications of the theory, along the lines of [30] are also worth exploring.

Related work. As we discussed already in the Introduction, most work on RE assumes a trace-based view of the SuS [32, 33, 39], where few distinguish between actions with different control profiles (e.g., inputs versus outputs). Although shields [28] can analyse both input and output actions, they still perform unidirectional enforcement and only modify the data associated with the output actions. The closest to our work is that by Pinisetty *et al.* [35], who consider bidirectional RE, modelling the system as a trace of input-output pairs. However, their enforcement is limited to replacements of payloads and their setting is too restrictive to model enforcements such as action rerouting and the closing of ports. Finally, Lanotte *et al.* [30] employ similar synthesis techniques and correctness criteria to ours (Definitions 3 and 4) to generate enforcement monitors for a timed setting.

References

1. Aceto, L., Achilleos, A., Francalanza, A., Ingólfssdóttir, 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
2. Aceto, L., Achilleos, A., Francalanza, A., Ingólfssdóttir, A., Kjartansson, S.Ö.: Determinizing monitors for HML with recursion. *J. Log. Algebraic Methods Program.* **111**, (2020). <https://doi.org/10.1016/j.jlamp.2019.100515>
3. Aceto, L., Achilleos, A., Francalanza, A., Ingólfssdóttir, A., Lehtinen, K.: The Best a Monitor Can Do. In: CSL. LIPIcs, vol. 183, pp. 7:1–7:23. Schloss Dagstuhl (2021). <https://doi.org/10.4230/LIPIcs.CSL.2021.7>
4. Aceto, L., Attard, D.P., Francalanza, A., Ingólfssdóttir, A.: On benchmarking for concurrent runtime verification. FASE 2021. LNCS, vol. 12649, pp. 3–23. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-71500-7_1
5. Aceto, L., Cassar, I., Francalanza, A., Ingólfssdóttir, A.: On Runtime Enforcement via Suppressions. In: CONCUR. vol. 118, pp. 34:1–34:17. Schloss Dagstuhl (2018). <https://doi.org/10.4230/LIPIcs.CONCUR.2018.34>
6. Aceto, L., Cassar, I., Francalanza, A., Ingólfssdóttir, A.: On bidirectional enforcement. Technical report Reykjavik University (2020). <http://icetcs.ru.is/theofomon/bidirectionalRE.pdf>
7. Aceto, L., Ingólfssdóttir, 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

8. Aceto, L., Ingólfssdóttir, A., Larsen, K.G., Srba, J.: *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, NY, USA (2007)
9. Alur, R., Černý, P.: Streaming Transducers for Algorithmic Verification of Single-pass List-processing Programs. In: *POPL*, pp. 599–610. ACM (2011). <https://doi.org/10.1145/1926385.1926454>
10. Attard, D.P., Francalanza, A.: A monitoring tool for a branching-time logic. In: Falcone, Y., Sánchez, C. (eds.) *RV 2016*. LNCS, vol. 10012, pp. 473–481. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46982-9_31
11. Bielova, N., Massacci, F.: Do you really mean what you actually enforced?-edited automata revisited. *J. Inf. Secur.* **10**(4), 239–254 (2011). <https://doi.org/10.1007/s10207-011-0137-2>
12. Bocchi, L., Chen, T.C., Demangeon, R., Honda, K., Yoshida, N.: Monitoring networks through multiparty session types. *TCS* **669**, 33–58 (2017)
13. Cassar, I.: *Developing Theoretical Foundations for Runtime Enforcement*. Ph.D. thesis, University of Malta and Reykjavik University (2021)
14. Cassar, I., Francalanza, A., Aceto, L., Ingólfssdóttir, A.: eAOP: an aspect oriented programming framework for Erlang. In: *Erlang*. ACM SIGPLAN (2017)
15. Cassar, I., Francalanza, A., Attard, D.P., Aceto, L., Ingólfssdóttir, A.: A Suite of Monitoring Tools for Erlang. In: *RV-CuBES*. Kalpa Publications in Computing, vol. 3, pp. 41–47. EasyChair (2017)
16. Cassar, I., Francalanza, A., Said, S.: Improving Runtime Overheads for detectEr. In: *FESCA*. EPTCS, vol. 178, pp. 1–8 (2015)
17. Chen, T.-C., Bocchi, L., Deniérou, P.-M., Honda, K., Yoshida, N.: Asynchronous distributed monitoring for multiparty session enforcement. In: Bruni, R., Sassone, V. (eds.) *TGC 2011*. LNCS, vol. 7173, pp. 25–45. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30065-3_2
18. Falcone, Y., Fernandez, J.-C., Mounier, L.: Synthesizing Enforcement Monitors w.r.t. the safety-progress classification of properties. In: Sekar, R., Pujari, A.K. (eds.) *ICISS 2008*. LNCS, vol. 5352, pp. 41–55. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89862-7_3
19. Falcone, Y., Fernandez, J.C., Mounier, L.: What can you verify and enforce at runtime? *J. Softw. Tools Technol. Transf.* **14**(3), 349 (2012)
20. Francalanza, A.: Consistently-Detecting Monitors. In: *CONCUR*. LIPIcs, vol. 85, pp. 8:1–8:19. Dagstuhl, Germany (2017). <https://doi.org/10.4230/LIPIcs.CONCUR.2017.8>
21. Francalanza, A.: A theory of monitors. *Inf. Comput* 104704 (2021). <https://doi.org/10.1016/j.ic.2021.104704>
22. Francalanza, A., Aceto, L., Ingólfssdóttir, A.: Monitorability for the Hennessy-Milner logic with recursion. *Formal Methods Syst. Des.* **51**(1), 87–116 (2017)
23. Hennessy, M., Lin, H.: Proof systems for message-passing process algebras. *Formal Aspects Comput.* **8**(4), 379–407 (1996). <https://doi.org/10.1007/BF01213531>
24. Hennessy, M., Liu, X.: A modal logic for message passing processes. *Acta Inf.* **32**(4), 375–393 (1995). <https://doi.org/10.1007/BF01178384>
25. van Hulst, A.C., Reniers, M.A., Fokkink, W.J.: Maximally permissive controlled system synthesis for non-determinism and modal logic. *Discr. Event Dyn. Syst.* **27**(1), 109–142 (2017)
26. Jia, L., Gommerstadt, H., Pfenning, F.: Monitors and blame assignment for higher-order session types. In: *POPL*, pp. 582–594. ACM, NY, USA (2016)
27. Khoury, R., Tawbi, N.: Which security policies are enforceable by runtime monitors? A survey. *Comput. Sci. Rev.* **6**(1), 27–45 (2012). <https://doi.org/10.1016/j.cosrev.2012.01.001>

28. Könighofer, B., et al.: Shield synthesis. *Formal Methods Syst. Des.* **51**(2), 332–361 (2017). <https://doi.org/10.1007/s10703-017-0276-9>
29. Kozen, D.C.: Results on the propositional μ -calculus. *Theor. Comput. Sci.* **27**, 333–354 (1983)
30. Lanotte, R., Merro, M., Munteanu, A.: Runtime enforcement for control system security. In: CSF, pp. 246–261. IEEE (2020). <https://doi.org/10.1109/CSF49147.2020.00025>
31. Larsen, K.G.: Proof systems for satisfiability in Hennessy-Milner logic with recursion. *Theor. Comput. Sci.* **72**(2), 265–288 (1990). [https://doi.org/10.1016/0304-3975\(90\)90038-J](https://doi.org/10.1016/0304-3975(90)90038-J)
32. Ligatti, J., Bauer, L., Walker, D.: Edit automata: enforcement mechanisms for runtime security policies. *J. Inf. Secur.* **4**(1), 2–16 (2005). <https://doi.org/10.1007/s10207-004-0046-8>
33. Ligatti, J., Bauer, L., Walker, D.: Run-time enforcement of nonsafety policies. *ACM Trans. Inf. Syst. Secur.* **12**(3), 19:1–19:41 (2009)
34. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes. I. *Inf. Comput.* **100**(1), 1–40 (1992). [https://doi.org/10.1016/0890-5401\(92\)90008-4](https://doi.org/10.1016/0890-5401(92)90008-4)
35. Pinisetty, S., Roop, P.S., Smyth, S., Allen, N., Tripakis, S., Hanxleden, R.V.: Runtime enforcement of cyber-physical systems. *ACM Trans. Embed. Comput. Syst.* **16**(5s), 1–25 (2017)
36. Rathke, J., Hennessy, M.: Local model checking for value-passing processes (extended abstract). In: Abadi, M., Ito, T. (eds.) TACS 1997. LNCS, vol. 1281, pp. 250–266. Springer, Heidelberg (1997). <https://doi.org/10.1007/BFb0014555>
37. Sakarovitch, J.: Elements of Automata Theory. Cambridge University Press, New York, NY, USA (2009)
38. Sangiorgi, D.: Introduction to Bisimulation and Coinduction. Cambridge University Press, New York, NY, USA (2011)
39. Schneider, F.B.: Enforceable security policies. *ACM Trans. Inf. Syst. Secur. (TISSEC)* **3**(1), 30–50 (2000)