On Runtime Enforcement via Suppressions

Luca Aceto
Gran Sasso Science Institute
& Reykjavik University
L’Aquila, Italy & Reykjavik Iceland
luca.aceto@gssi.it

Ian Cassar
Reykjavik University
& University of Malta
Reykjavik Iceland & Msida, Malta
ianc@ru.is

Adrian Francalanza
University of Malta
Msida, Malta
adrian.francalanza@um.edu.mt

Anna Ingólfsdóttir
Reykjavik University
Reykjavik, Iceland
annai@ru.is

Abstract
Runtime enforcement is a dynamic analysis technique that uses monitors to enforce the behaviour specified by some correctness property on an executing system. The enforceability of a logic captures the extent to which the properties expressible via the logic can be enforced at runtime. We study the enforceability of Hennessy-Milner Logic with Recursion ($\mu$HML) with respect to suppression enforcement. We develop an operational framework for enforcement which we then use to formalise when a monitor enforces a $\mu$HML property. We also show that the safety syntactic fragment of the logic, sHML, is enforceable by providing an automated synthesis function that generates correct suppression monitors from sHML formulas.

2012 ACM Subject Classification Theory of computation → Logic and verification Software and its engineering → Software verification Software and its engineering → Dynamic analysis

Keywords and phrases Enforceability, Suppression Enforcement, Monitor Synthesis, Logic

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2018.34

Related Version http://doi.org/10.13140/RG.2.2.11057.89447

Acknowledgements The research work disclosed in this publication is partially supported by the projects “Developing Theoretical Foundations for Runtime Enforcement” (184776-051) and “TheoFoMon: Theoretical Foundations for Monitorability” (163406-051) of the Icelandic Research Fund, and by the Endeavour Scholarship Scheme (Malta), part-financed by the European Social Fund (ESF) - Operational Programme II – Cohesion Policy 2014-2020.

1 Introduction
Runtime monitoring [22, 24] is a dynamic analysis technique that is becoming increasingly popular in the turbid world of software development. It uses code units called monitors to aggregate system information, compare system execution against correctness specifications, or steer the execution of the observed system. The technique has been used effectively to offload certain verification tasks to a post-deployment phase, thus complementing other (static) analysis techniques in multi-pronged verification strategies—see e.g., [6, 12, 27, 18, 28]. 
Runtime enforcement (RE) [33, 34, 21] is a specialized monitoring technique, used to ensure that the behaviour of a system-under-scrutiny (SuS) is always in agreement with some correctness specification. It employs a specific kind of monitor (referred to as a transducer [9, 42, 4] or an edit-automaton [33, 34]) to anticipate incorrect behaviour and counter it. Such a monitor thus acts as a proxy between the SuS and the surrounding environment interacting
with it, encapsulating the system to form a composite (monitored) system: at runtime, the monitor transforms any incorrect executions exhibited by the SuS into correct ones by either suppressing, inserting or replacing events on behalf of the system.

We extend a recent line of research [25, 24, 2, 1] and study RE approaches that adopt a separation of concerns between the correctness specification, describing what properties the SuS should satisfy, and the monitor, describing how to enforce these properties on the SuS. Our work considers system properties expressed in terms of the process logic $\mu$HML [30, 32], and explores what properties can be operationally enforced by monitors that can suppress system behaviour. A central element for the realisation of such an approach is the synthesis function: it automates the translation from the declarative $\mu$HML specifications to algorithmic descriptions formulated as executable monitors. Since analysis tools ought to form part of the trusted computing base, enforcement monitoring should be, in and of itself, correct. However, it is unclear what is to be expected of the synthesised monitor to adequately enforce a $\mu$HML formula. Nor is it clear for which type of specifications should this approach be expected to work effectively—it has been well established that a number of properties are not monitorable [15, 39, 16, 25, 2] and it is therefore reasonable to expect similar limits in the case of enforceability [19]. We therefore study the relationship between $\mu$HML specifications and suppression monitors for enforcement, which allows us to address the above-mentioned concerns and make the following contributions:

**Modelling:** We develop a general framework for enforcement instrumentation that is parametrisable by any system behaviour that is expressed via labelled transitions, and can express suppression, insertion and replacement enforcement, Figure 2.

**Correctness:** We give formal definitions for asserting when a monitor correctly enforces a formula defined over labelled transition systems, Definitions 3 and 8. These definitions are parametrisable with respect to an instrumentation relation, an instance of which is our enforcement framework of Figure 2.

**Expressiveness:** We provide enforceability results, Theorems 14 and 18 (but also Proposition 24), by identifying a subset of $\mu$HML formulas that can be (correctly) enforced by suppression monitors.

As a by-product of this study, we also develop a formally-proven correct synthesis function, Definition 12, that then can be used for tool construction, along the lines of [8, 7].

The setup selected for our study serves a number of purposes. For starters, the chosen logic, $\mu$HML, is a branching-time logic that allows us to investigate enforceability for properties describing computation graphs. Second, the use of a highly expressive logic allows us to achieve a good degree of generality for our results, and so, by working in relation to logics like $\mu$HML (a reformulation of the $\mu$-calculus), our work would also apply to other widely used logics (such as LTL and CTL [17]) that are embedded within this logic. Third, since the logic is verification-technique agnostic, it fits better with the realities of software verification in the present world, where a variety of techniques (e.g., model-checking and testing) straddling both pre- and post-deployment phases are used. In such cases, knowing which properties can be verified statically and which ones can be monitored for and enforced at runtime is crucial for devising effective multi-pronged verification strategies. Equipped with such knowledge, one could also employ standard techniques [35, 5, 31] to decompose a non-enforceable property into a collection of smaller properties, a subset of which can then be enforced at runtime.

**Structure of the paper:** Section 2 revisits labelled transition systems and our touchstone logic, $\mu$HML. The operational model for enforcement monitors and instrumentation is given in Section 3. In Section 4 we formalise the interdependent notions of correct enforcement
Syntax
\[
\begin{align*}
\varphi, \psi & \in \muHML \; ::= \; \text{tt} \quad \text{(truth)} \quad | \quad \text{ff} \quad \text{(falsehood)} \quad | \quad \bigvee_{i \in I} \varphi_i \quad \text{(disjunction)} \\
& \quad | \quad \bigwedge_{i \in I} \varphi_i \quad \text{(conjunction)} \quad | \quad \{p, c\} \varphi \quad \text{(possibility)} \quad | \quad \{p, c\}\varphi \quad \text{(necessity)} \\
& \quad | \quad \min X. \varphi \quad \text{(least fp.)} \quad | \quad \max X. \varphi \quad \text{(greatest fp.)} \quad | \quad X \quad \text{(fp. variable)}
\end{align*}
\]

Semantics
\[
\begin{align*}
\text{[tt, } \rho] & \equiv \text{Sys} \\
\text{[ff, } \rho] & \equiv \emptyset \\
\text{[}\bigwedge_{i \in I} \varphi_i, \rho]\ & \equiv \bigcap_{i \in I} [\varphi_i, \rho] \\
\text{[}\bigvee_{i \in I} \varphi_i, \rho]\ & \equiv \bigcup_{i \in I} [\varphi_i, \rho] \\
\text{[}\min X. \varphi, \rho]\ & \equiv \{S \mid S \subseteq [\varphi, \rho](X \rightarrow S)\} \\
\text{[}\max X. \varphi, \rho]\ & \equiv \{S \mid S \subseteq [\varphi, \rho](X \rightarrow S)\} \\
\text{[}\{p, c\}\varphi, \rho]\ & \equiv \{s \mid (\forall \alpha, r \cdot s \xrightarrow{\alpha} r \text{ and } (\exists \sigma \cdot \text{mtch}(p, \alpha)=\sigma \text{ and } \sigma \downarrow \text{true}) \text{ implies } q \in [\varphi, \rho]\} \\
\text{[}\{p, c\}\varphi, \rho]\ & \equiv \{s \mid \exists \alpha, r, \sigma \cdot (s \xrightarrow{\alpha} r \text{ and } \text{mtch}(p, \alpha)=\sigma \text{ and } \sigma \downarrow \text{true} \text{ and } q \in [\varphi, \rho]\}\}
\end{align*}
\]

Figure 1 \(\muHML\) Syntax and Semantics

and enforceability. These act as a foundation for the development of a synthesis function in Section 5, that produces correct-by-construction monitors. In Section 6 we consider alternative definitions for enforceability for logics with a specific additional interpretation, and show that our proposed synthesis function is still correct with respect to the new definition. Section 7 concludes and discusses related work.

2 Preliminaries

The Model: We assume systems described as labelled transition systems (LTSs), 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 observable actions, \(\alpha, \beta \in \text{ACT} \cup \{\tau\}\), and 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 use \(s \xrightarrow{*} s’\) to denote weak transitions representing \((s, \tau)r\times* \mu \xrightarrow{s} \cdot \cdot \cdot \cdot s’\). We refer to \(s’\) as a \(\mu\)-derivative of \(s\). Traces, \(t, u \in \text{ACT}^*\) range over (finite) sequences of observable actions, and we write \(s \xrightarrow{*} r\) to denote a sequence of weak transitions \(s \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_t} r\) for \(t = 1, \ldots, \alpha_t\). We also assume the classic notion of strong bisimilarity \([38, 43]\) for our model, \(s \sim r\), using it as our touchstone system equivalence. The syntax of the regular fragment of CCS [38] is occasionally used to concisely describe LTSs in our examples.

The Logic: We consider a slightly generalised version of \(\muHML\) [32, 3] that uses symbolic actions of the form \([p, c]\). Patterns, \(p\), abstract over actions using data variables \(d, e, f \in \text{VAR}\); in a pattern, they may either occur free, \(d\), or as binders, \((d)\) where a closed pattern is one without any free variables. We assume a (partial) matching function for closed patterns \(\text{mtch}(p, \alpha)\) that returns a substitution \(\sigma\) (when successful) mapping variables in \(p\) to the corresponding values in \(\alpha\), i.e., if we instantiate every bound variable \(d\) in \(p\) with \(\sigma(d)\) we obtain \(\alpha\). The filtering condition, \(c\), contains variables found in \(p\) and evaluates wrt. the substitutions returned by successful matches. Put differently, a closed symbolic action \([p, c]\) is one where \(p\) is closed and \(\text{fv}(c) \subseteq \text{bv}(p)\); it denotes the set of actions \([p, c]\) \(\equiv \{\alpha \mid \exists \sigma \cdot \text{mtch}(p, \alpha)=\sigma \text{ and } \sigma \downarrow \text{true}\}\) and allows more adequate reasoning about LTSs with infinite actions (e.g., actions carrying data from infinite domains).

The logic syntax is given in Figure 1 and assumes a countable set of logical variables \(X, Y \in \text{LVAR}\). Apart from standard logical constructs such as conjunctions and disjunctions (\(\bigwedge_{i \in I} \varphi_i\) describes a compound conjunction, \(\varphi_1 \land \ldots \land \varphi_n\), where \(I = \{1, \ldots, n\}\) is a finite set of indices, and similarly for disjunctions, and the characteristic greatest and least fixpoints (\(\max X. \varphi\) and \(\min X. \varphi\) bind free occurrences of \(X\) in \(\varphi\)), the logic uses necessity
and possibility modal operators with symbolic actions, \( [[p, c]] \varphi \) and \( (\{p, c\})\varphi \), where \( \text{bv}(p) \) bind free data variables in \( c \) and \( \varphi \). Formulas in \( \muHML \) are interpreted over the system powerset domain where \( S \in \mathcal{P}(\text{Sys}) \). 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, \( \rho \in (\text{LVar} \rightarrow \mathcal{P}(\text{Sys})) \), 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 cases are those for the modal formulas, due to the use of symbolic actions. Note that we recover the standard logic for symbolic actions \( \{p, c\} \) whose pattern \( p \) does not contain variables \( (p = \alpha \text{ for some } \alpha) \) and whose condition holds trivially \( (c = \text{true}) \); in such cases we write \( \alpha \varphi \) and \( \langle \alpha \rangle \varphi \) for short. We generally assume closed formulas, i.e., without free logical and data variables, and write \( [[\varphi]] \) in lieu of \( [[\varphi, \rho]] \) since the interpretation of a closed \( \varphi \) is independent of \( \rho \). A system \( s \) satisfies formula \( \varphi \) whenever \( s \in [[\varphi]] \) whereas a formula \( \varphi \) is satisfiable, \( \varphi \in \text{SAT} \), whenever there exists a system \( r \) such that \( r \in [[\varphi]] \).

**Example 1.** Consider two systems (a good system, \( s_g \), and a bad one, \( s_b \)) implementing a server that interacts on port \( i \), repeatedly accepting requests that are answered by outputting on the same port, and terminating the service once a close request is accepted (on the same port). Whereas \( s_g \) outputs an answer \( (\!\!\text{ans}) \) for every request \( (i?\text{req}) \), \( s_b \) occasionally refuses to answer a given request (see the underlined branch). Both systems terminate with \( i?\text{cls} \).

\[
s_g = \text{rec } x. (i?\text{req}, \!\!\text{ans}, x + i?\text{cls}, \text{nil}) \quad s_b = \text{rec } x. (i?\text{req}, \!\!\text{ans}, x + i?\text{req}, x + i?\text{cls}, \text{nil})
\]

We can specify that two consecutive requests on port \( i \) induce invalid behaviour via the \( \muHML \) formula \( \varphi_0 \equiv \max X. [[i?\text{req}]] (\!\!\!\text{ans}, X \land [i?\text{req}, \text{ff}]) \); it defines an invariant property \( \max X. (\cdots) \) requiring that whenever a system interacting on \( i \) inputs a request, it cannot input a subsequent request, i.e., \( [i?\text{req}, \text{ff}] \), unless it outputs an answer beforehand, in which case the formula recurses, i.e., \( [\!\!\!\text{ans}], X \). Using symbolic actions, we can generalise \( \varphi_0 \) by requiring the property to hold for any interaction happening on any port number except \( j \).

\[
\varphi_1 \equiv \max X. [[(d)\text{?req}, d\neq j]] ([[\!\!\!\text{ans}, \text{true}], X \land [d?\text{req}, \text{true}]], \text{ff})
\]

In \( \varphi_1 \), \( (d)?\!\!\text{req} \) binds the free occurrences of \( d \) found in \( d\neq j \) and \( [[\!\!\!\text{ans}, \text{true}], X \land [d?\text{req}, \text{true}]], \text{ff} \). Using Figure 1, one can check that \( s_g \in [\varphi_1] \), whereas \( s_b \not\in [\varphi_1] \) since \( s_b \overset{\ldots}{\overset{i?\text{req}}{\overset{i?\text{req}}{\overset{i?\text{req}}{\cdots}}}} \).

3 An Operational Model for Enforcement

Our operational mechanism for enforcing properties over systems uses the (symbolic) transducers \( m, n \in \text{Trn} \) defined in Figure 2. The transition rules in Figure 2 assume closed terms, i.e., for every symbolic-prefix transducer, \( \{p, c, p', m\}, p \) is closed and \( (\text{fv}(c) \cup \text{fv}(p') \cup \text{fv}(m)) \subseteq \text{bv}(p) \), and yield an LTS with labels of the form \( \gamma \cdot \mu \), where \( \gamma \in (\text{Act} \cup \{\bullet\}) \). Our syntax assumes a well-formedness constraint where for every \( \{p, c, p', m\}, \text{bv}(c) \cup \text{bv}(p') = \emptyset \). Intuitively, a transition \( m \overset{\alpha \bullet \mu}{\longrightarrow} n \) denotes the fact that the transducer in state \( m \) transforms the visible action \( \alpha \) (produced by the system) into the action \( \mu \) (which can possibly become silent) and transitions into state \( n \). In this sense, the transducer action \( \alpha \cdot \tau \) represents the suppression of action \( \alpha \), action \( \alpha \cdot \beta \) represents the replacing of \( \alpha \) by \( \beta \), and \( \bullet \cdot \alpha \) denotes the identity transformation. The special case \( \bullet \cdot \alpha \) encodes the insertion of \( \alpha \), where \( \bullet \) represents that the transition is not induced by any system action.

The key transition rule in Figure 2 is \( \text{eTrn} \). It states that the symbolic-prefix transducer \( \{p, c, p'\}, m \) can transform an (extended) action \( \gamma \) into the concrete action \( \mu \), as long as
the action matches with pattern \( p \) with substitution \( \sigma \), \( \text{mtch}(p, \gamma) = \sigma \), and the condition is satisfied by \( \sigma, c \sigma \Downarrow \text{true} \) (the matching function is lifted to extended actions and patterns in the obvious way, where \( \text{mtch}(\bullet, \bullet) = \emptyset \)). In such a case, the transformed action is \( \mu = p' \sigma \), i.e., the action \( \mu \) resulting from the instantiation of the free data variables in pattern \( p' \) with the corresponding values mapped by \( \sigma \), and the transducer state reached is \( m \sigma \). By contrast, in rule eID, the transducer \( \text{id} \) acts as the identity and leaves actions unchanged. The remaining rules are fairly standard and unremarkable.

Figure 2 also describes an instrumentation relation which relates the behaviour of the SUnS \( s \) with the transformations of a transducer monitor \( m \) that agrees with the (observable) actions \( \text{Act} \) of \( s \). The term \( m[s] \) thus denotes the resulting monitored system whose behaviour is defined in terms of \( \text{Act} \cup \{\tau\} \) from the system’s LTS. Concretely, rule iTRN states that when a system \( s \) transitions with an observable action \( \alpha \) to \( s' \) and the transducer \( m \) can transform this action into \( \mu \) and transition to \( n \), the instrumented system \( m[s] \) transitions with action \( \mu \) to \( n[s'] \). However, when \( s \) transitions with a silent action, rules iASY allows it to do so independently of the transducer. Dually, rule iINS allows the transducer to insert an action \( \mu \) independently of \( s \)’s behaviour. Rule iTER is analogous to standard monitor instrumentation rules for premature termination of the transducer [22, 25, 23, 1], and accounts for underspecification of transformations. Thus, if a system \( s \) transitions with an observable action \( \alpha \) to \( s' \), and the transducer \( m \) does not specify how to transform it \( (m \not\tau \alpha) \), nor can it transition to a new transducer state by inserting an action \( (m \not\tau \mu) \), the system is still allowed to transition while the transducer’s transformation activity is ceased, i.e., it acts like the identity \( \text{id} \) from that point onwards.

Example 2. Consider the insertion transducer \( m_i \) and the replacement transducer \( m_r \) below:

\[
m_i \equiv \{ \bullet, \text{true}, \text{i?req}, \{ \bullet, \text{true}, \text{i\text{\textbackslash\text{ans}}}, \text{id} \}
\]

\[
m_r \equiv \text{rec } x. \{(d)?\text{req}, \text{true}, j?\text{req}\}.x + \{(d)?\text{\text{\textbackslash\text{ans}}}, \text{true}, j?\text{\text{\textbackslash\text{ans}}}, x + \{(d)?\text{cls}, \text{true}, j?\text{cls}, x\}
\]

When instrumented with a system, \( m_i \) inserts the two successive actions \( \text{i?req} \) and \( \text{i\text{\textbackslash\text{ans}}} \) before
behaving as the identity. Concretely in the case of $s_b$ we can only start the computation as:

$$m_\text{t}[s_b] \xrightarrow{i\text{req}} [\bullet,\text{true},i\text{ans},i\text{id}[s_b] \xrightarrow{i\text{ans}} i\text{id}[s_b] \xrightarrow{\alpha} \ldots \xrightarrow{\alpha} \ldots$$ (where $s_b \xrightarrow{\alpha}$)

By contrast, $m_\text{r}$ transforms input actions with either payload $\text{req}$ or $\text{cls}$ and output actions with payload $\text{ans}$ on any port name, into the respective actions on port $j$. For instance:

$$m_\text{r}[s_b] \xrightarrow{j\text{req}} m_\text{r}[i\text{ans}.s_b] \xrightarrow{j\text{ans}} m_\text{r}[s_b] \xrightarrow{j\text{cls}} m_\text{r}[\text{nil}]$$

Consider now the two suppression transducers $m_\text{s}$ and $m_\text{t}$ for actions on ports other than $j$:

$$m_\text{s} \overset{\text{def}}{=} \text{rec}.x.([(d)?\text{req}, d \neq j, \tau],x + [(d)!\text{ans}, \text{true}, d!\text{ans}],x)$$

$$m_\text{t} \overset{\text{def}}{=} \text{rec}.x.([(d)?\text{req}, d \neq j, d?\text{req}], \text{rec}.y.([(d)!\text{ans}, \text{true}, d!\text{ans}],x + [d?\text{req}, \text{true}, \tau],y))$$

Monitor $m_\text{s}$ suppresses any requests on ports other than $j$, and continues to do so after any answers on such ports. When instrumented with $s_b$, we can observe the following behaviour:

$$m_\text{s}[s_b] \xrightarrow{\tau} m_\text{s}[i\text{ans}.s_b] \xrightarrow{i\text{ans}} m_\text{s}[s_b] \xrightarrow{\tau} m_\text{s}[i\text{ans}.s_b] \xrightarrow{i\text{ans}} m_\text{s}[s_b] \ldots$$

Note that $m_\text{s}$ does not specify a transformation behaviour for when the monitored system produces inputs with payload other than $\text{req}$. The instrumentation handles underspecification by ceasing suppression activity; in the case of $s_b$ we get $m_\text{s}[s_b] \xrightarrow{i\text{cls}} \text{id}[\text{nil}]$. The transducer $m_\text{t}$ performs slightly more elaborate transformations. For interactions on ports other than $j$, it suppresses consecutive input requests following any serviced request (i.e., an input on $\text{req}$ followed by an output on $\text{ans}$) sequence. For $s_b$, we can observe the following:

$$m_\text{t}[s_b] \xrightarrow{i\text{req}} \text{rec}.y.([(i)!\text{ans}, \text{true}, i\text{ans}],m_\text{t} + [i?\text{req}, \text{true}, \tau],y)|s_b]$$

$$\xrightarrow{\tau} \text{rec}.y.([(i)!\text{ans}, \text{true}, i\text{ans}],m_\text{t} + [i?\text{req}, \text{true}, \tau],y)]i\text{ans}.s_b] \xrightarrow{i\text{ans}} m_\text{t}[s_b]$$

In the sequel, we find it convenient to refer to $p$ as the transformed pattern $p$ where all the binding occurrences $(d)$ are converted to free occurrences $d$. As shorthand notation, we elide the second pattern $p'$ in a transducer $\{p,c,p',m\}$ whenever $p'=p$ and simply write $\{p,c,m\}$; note that if $\text{bv}(p) = \varnothing$, then $p=p$. Similarly, we elide $c$ whenever $c=\text{true}$. This allows us to express $m_\text{t}$ from Example 2 as $\text{rec}.x.([(d)?\text{req}, d \neq j], \text{rec}.y.([(d)!\text{ans}, x + [d?\text{req}, \tau],y])$.

## 4 Enforceability

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 (the transducers and instrumentation of Section 3 in our case) to enforce the specified behaviour on the other.

### Definition 3 (Enforceability). A logic $\mathcal{L}$ is enforceable iff every formula $\varphi \in \mathcal{L}$ is enforceable. A formula $\varphi$ is enforceable iff there exists a transducer $m$ such that $m$ enforces $\varphi$.

Definition 3 depends on what is considered to be an adequate definition for “$m$ enforces $\varphi$”. It is reasonable to expect that the latter definition should concern any system that the transducer $m$—hereafter referred to as the enforcer—is instrumented with. In particular, for any system $s$, the resulting composite system obtained from instrumenting the enforcer $m$ with it should satisfy the property of interest, $\varphi$, whenever this property is satisfiable.

### Definition 4 (Sound Enforcement). Enforcer $m$ soundly enforces a formula $\varphi$, denoted as $\text{senf}(m, \varphi)$, iff for all $s \in \text{Sys}$, $\varphi \in \text{Sat}$ implies $m[s] \in \llbracket \varphi \rrbracket$ holds.
**Example 5.** Recall $\varphi_1$, $s_g$ and $s_b$ from Example 1 where $s_g \notin [\varphi_1]$ (hence $\varphi_1 \in \text{SAT}$) and $s_b \notin [\varphi_1]$. For the enforcers $m_1$, $m_r$, $m_s$ and $m_t$ presented in Example 2, we have:

- $m_1[s_b] \notin [\varphi_1]$, since $m_1[s_b] \stackrel{\text{r?req}}{\rightarrow} \stackrel{\text{id}}{\rightarrow} \text{id}[s_b] - \stackrel{\text{r?req}}{\rightarrow} \text{id}[s_b]$. This counter example implies that $\neg \text{senf}(m_1, \varphi_1)$.
- $m_r[s_g] \in [\varphi_1]$ and $m_r[s_b] \in [\varphi_1]$. Intuitively, this is because the ensuing instrumented systems only generate (replaced) actions that are not of concern to $\varphi_1$. Since this behaviour applies to any system $m_r$ is composed with, we can conclude that $\text{senf}(m_r, \varphi_1)$.
- $m_s[s_g] \in [\varphi_1]$ and $m_s[s_b] \in [\varphi_1]$ because the resulting instrumented systems never produce inputs with $\text{req}$ on a port number other than $j$. We can thus conclude that $\text{senf}(m_s, \varphi_1)$.
- $m_t[s_g] \in [\varphi_1]$ and $m_t[s_b] \in [\varphi_1]$. Since the resulting instrumentation suppresses consecutive input requests (if any) after any number of serviced requests on any port other than $j$, we can conclude that $\text{senf}(m_t, \varphi_1)$.

By some measures, sound enforcement is a relatively weak requirement for adequate enforcement as it does not regulate the extent of the induced enforcement. More concretely, consider the case of enforcer $m_s$ from Example 2. Although $m_s$ manages to suppress the violating executions of system $s_b$, thereby bringing it in line with property $\varphi_1$, it needlessly modifies the behaviour of $s_g$ (namely it prohibits it from producing any inputs with $\text{req}$ on port numbers that are not $j$), even though it satisfies $\varphi_1$. Thus, in addition to sound enforcement we require a transparency condition for adequate enforcement. The requirement dictates that whenever a system $s$ already satisfies the property $\varphi$, the assigned enforcer $m$ should not alter the behaviour of $s$. Put differently, the behaviour of the enforced system should be behaviourally equivalent to the original system.

**Definition 6 (Transparent Enforcement).** An enforcer $m$ is transparent when enforcing a formula $\varphi$, denoted as $\text{tenf}(m, \varphi)$, iff for all $s \in \text{Sys}$, $s \in [\varphi]$ implies $m[s] \sim s$.

**Example 7.** We have already argued—via the counter example $s_g$—why $m_s$ does not transparently enforce $\varphi_1$. We can also argue easily why $\neg \text{tenf}(m_r, \varphi_1)$ either: the simple system $i?\text{req.nil}$ trivially satisfies $\varphi_1$ but, clearly, we have the inequality $m_r[i?\text{req.nil}] \not\sim i?\text{req.nil}$ since $m_r[i?\text{req.nil}] \stackrel{j?\text{req}}{\rightarrow} m_r[\text{nil}]$ and $i?\text{req.nil} \not\stackrel{j?\text{req}}{\rightarrow}$. It turns out that enforcer $\text{tenf}(m_t, \varphi_1)$, however. Although this property is not as easy to show—due to the universal quantification over all systems—we can get a fairly good intuition for why this is the case via the example $s_g$: it satisfies $\varphi_1$ and $m_t[s_g] \sim s_g$ holds.

**Definition 8 (Enforcement).** A monitor $m$ enforces property $\varphi$ whenever it does so (i) soundly, Definition 4 and (ii) transparently, Definition 6.

For any reasonably expressive logic (such as $\mu\text{HML}$), it is usually the case that not every formula can be enforced, as the following example informally illustrates.

**Example 9.** Consider the $\mu\text{HML}$ property $\varphi_{ns}$, together with the two systems $s_{ra}$ and $s_r$:

$$\varphi_{ns} \equiv [i?\text{req}.ff \lor i!\text{ans}.ff] \quad s_{ra} \equiv i?\text{req.nil} \lor i!\text{ans}.nil \quad s_r \equiv i?\text{req.nil}$$

A system satisfies $\varphi_{ns}$ if either it cannot produce action $i?\text{req}$ or it cannot produce action $i!\text{ans}$. Clearly, $s_{ra}$ violates this property as it can produce both. This system can only be enforced via action suppressions or replacements because insertions would immediately break transparency. Without loss of generality, assume that our monitors employ suppressions (the same argument applies for action replacement). The monitor $m_r \equiv \text{rec}.y.(\{i?\text{req},\tau\}.y \lor \{i!\text{ans},\tau\}.y)$ would in
fact be able to suppress the offending actions produced by \( s_{ra} \), thus obtaining \( m_r[s_{ra}] \in \llbracket \varphi_{ns} \rrbracket \). However, it would also suppress the sole action \( i?req \) produced by the system \( s_r \), even though this system satisfies \( \varphi_{ns} \). This would, in turn, violate the transparency criterion of Definition 6 since it needlessly suppresses \( s_r \)'s actions, i.e., although \( s_r \in \llbracket \varphi_{ns} \rrbracket \) we have \( m_r[s_r] \not\sim s_r \). The intuitive reason for this problem is that a monitor cannot, in principle, look into the computation graph of a system, but is limited to the behaviour the system exhibits at runtime.

5 Synthesising Suppression Enforcers

Despite their merits, Definitions 3 and 8 are not easy to work with. The universal quantifications over all systems in Definitions 4 and 6 make it hard to establish that a monitor correctly enforces a property. Moreover, according to Definition 3, in order to determine whether a particular property is enforceable or not, one would need to show the existence of a monitor that correctly enforces it; put differently, showing that a property is not enforceable entails another universal quantification, this time showing that no monitor can possibly enforce the property. Lifting the question of enforceability to the level of a (sub)logic entails a further universal quantification, this time on all the logical formulas of the logic; this is often an infinite set. We address these problems in two ways. First, we identify a non-trivial syntactic subset of \( \mu \text{HML} \) that is guaranteed to be enforceable; in a multi-pronged approach to system verification, this could act as a guide for whether the property should be considered at a pre-deployment or post-deployment phase. Second, for every formula \( \varphi \) in this enforceable subset, we provide an automated procedure to synthesise a monitor \( m \) from it that correctly enforces \( \varphi \) when instrumented over arbitrary systems, according to Definition 8. This procedure can then be used as a basis for constructing tools that automate property enforcement.

In this paper, we limit our enforceability study to suppression monitors, transducers that are only allowed to intervene by dropping (observable) actions. Despite being more constrained, suppression monitors sidestep problems associated with what data to use in a payload-carrying action generated by the enforcer, as in the case of insertion and replacement monitors: the notion of a default value for certain data domains is not always immediate. Moreover, suppression monitors are particularly useful for enforcing safety properties, as shown in [33, 10, 20]. Intuitively, a suppression monitor would suppress actions as soon as it becomes apparent that a violation is about to be committed by the SuS. Such an intervention intrinsically relies on the detection of a violation. To this effect, we use a prior result from [25], which identified a maximally-expressive logical fragment of \( \mu \text{HML} \) that can be handled by violation-detecting (recogniser) monitors. We thus limit our enforceability study to this maximal safety fragment, called \( s\text{HML} \), since a transparent suppression monitor cannot judiciously suppress actions without first detecting a (potential) violation. Figure 3 recalls the syntax for \( s\text{HML} \). The logic is restricted to truth and falsehood (\( \text{tt} \) and \( \text{ff} \)), conjunctions \( \bigwedge_{i\in I} \varphi_i \), and necessity modalities \( \llbracket [p,c]\rrbracket \varphi \), while recursion may only be expressed through greatest fixpoints \( \text{max} X.\varphi \); the semantics follows that of Figure 1.

A standard way how to achieve our aims would be to (i) define a (total) synthesis function \( L^{-M} : s\text{HML} \rightarrow \text{Trn} \) from \( s\text{HML} \) formulas to suppression monitors and (ii) then show that for any \( \varphi \in s\text{HML} \), the synthesised monitor \( \llbracket \varphi \rrbracket \) enforces \( \varphi \). Moreover, we would also
require the synthesis function to be compositional, whereby the definition of the enforcer for a composite formula is defined in terms of the enforcers obtained for the constituent subformulas. There are a number of reasons for this requirement. For one, it would simplify our analysis of the produced monitors and allow us to use standard inductive proof techniques to prove properties about the synthesis function, such as the aforementioned criteria (ii). However, a naïve approach to such a scheme is bound to fail, as discussed in the next example.

Example 10. Consider a semantically equivalent reformulation of \( \varphi_1 \) from Example 1.

\[
\varphi_2 \equiv \max X.((\{(d)?\text{req}, d \neq j\}[[d]\text{ans, true}]\},X) \land (\{(d)?\text{req}, d \neq j\}[[d]\text{req, true}][\text{ff}])
\]

At an intuitive level, the suppression monitor that one would expect to obtain for the subformula \( \varphi_2' \equiv \{(d)?\text{req, d \neq j}\}[[d]\text{ans, true}][\text{ff}] \) is \( \{(d)?\text{req, d \neq j}\}.\text{rec y, d}\text{req, r,y} \) (i.e., an enforcer that repeatedly drops any \text{req} input following a \text{req} input on the same port), whereas the monitor obtained for the subformula \( \varphi_2'' \equiv \{(d)?\text{req, d \neq j}\}[[d]\text{ans, true}][X] \) is \( \{(d)?\text{req, d \neq j}\}.\text{ans} \) (assuming some variable mapping from \( X \) to \( x \)). These monitors would then be combined in the synthesis for \( \max X.\varphi_2'' \land \varphi_2' \) as

\[
m_b \equiv \text{rec } x. \{(d)?\text{req, d \neq j}\}[[d]\text{ans}].x + \{(d)?\text{req, d \neq j}\}.\text{rec y, d}\text{req, r,y}
\]

One can easily see that \( m_b \) does not behave deterministically, nor does it soundly enforce \( \varphi_2 \). For instance, for the violating system \( i?'\text{req, i?'\text{req.nil} \notin \varphi_2'\eqqcolon \varphi_1} \), we can observe the transition sequence \( m_b[i?'\text{req, i?'\text{req.nil} \rightarrow id[\text{nil}].m_b[i?'\text{req.nil} \rightarrow id[\text{nil}]. \square
\]

Instead of complicating our synthesis function to cater for anomalies such as those presented in Example 10—also making it less compositional in the process—we opted for a two-stage synthesis procedure. First, we consider a normalised subset for sHML formulas which is amenable to a (straightforward) synthesis function definition that is compositional. This also facilitates the proofs for the conditions required by Definition 8 for any synthesised enforcer. Second, we show that every sHML formula can be reformulated in this normalised form without affecting its semantic meaning. We can then show that our two-stage approach is expressive enough to show the enforceability for all of sHML.

Definition 11 (sHML normal form). The set of normalised sHML formulas is defined as:

\[
\varphi, \psi \in \text{sHML}_{nf} \colon \equiv \tt \mid \ff \mid \bigwedge_{i \in I} [[p_i, c_i]] \varphi_i \mid X \mid \max X. \varphi.
\]

The above grammar combines necessity operators with conjunctions into one construct \( \bigwedge_{i \in I} [[p_i, c_i]] \varphi_i \). Normalised sHML formulas are required to satisfy two further conditions:
1. For every \( \bigwedge_{i \in I} [[p_i, c_i]] \varphi_i \), for all \( j, h \in I \) where \( j \neq h \) we have \( [[p_j, c_j]] \cap [[p_h, c_h]] = \emptyset \).
2. For every \( \max X. \varphi \) we have \( X \in \text{fv}(\varphi) \). \( \square \)

In a (closed) normalised sHML formula, the basic terms \( \tt \) and \( \ff \) can never appear unguarded unless they are at the top level (e.g., we can never have \( \varphi \lor \ff \) or \( \max X_0 \ldots \max X_n \ff \)). Moreover, in any conjunction of necessity subformulas, \( \bigwedge_{i \in I} [[p_i, c_i]] \varphi_i \), the necessity guards are disjoint and at most one necessity guard can satisfy any particular action.

Definition 12. The synthesis function \( \langle - \rangle : \text{sHML}_{nf} \rightarrow \text{TRN} \) is defined inductively as:

\[
\langle X \rangle \equiv x \quad \langle \tt \rangle \equiv \langle \ff \rangle \equiv \text{id} \quad \langle \max X. \varphi \rangle \equiv \text{rec } x. \langle \varphi \rangle
\]

\[
\langle \bigwedge_{i \in I} [[p_i, c_i]] \varphi_i \rangle \equiv \text{rec } y. \sum_{i \in I} \begin{cases} \text{if } \varphi_i = \ff \to y & \bigwedge_{i \in I} [[p_i, c_i]] \varphi_i \rangle \equiv \text{rec } y. \sum_{i \in I} \bigwedge_{i \in I} [[p_i, c_i]] \varphi_i \rangle \equiv \text{id} \end{cases}
\]

\( \square \)
The synthesis function is compositional. It assumes a bijective mapping between formula variables and monitor recursion variables and converts logical variables $X$ accordingly, whereas maximal fixpoints, $\text{max } X. \varphi$, are converted into the corresponding recursive enforcer. The synthesis also converts truth and falsehood formulas, $tt$ and $ff$, into the identity enforcer id. Normalized conjunctions, $\bigwedge_{i \in I} [p_i, c_i] \varphi_i$, are synthesised into a recursive summation of enforcers, i.e., $\text{rec } y.m_i$, where $y$ is fresh, and every branch $m_i$ can be either of the following:

(i) when $m_i$ is derived from a branch of the form $[p_i, c_i] \varphi_i$ where $\varphi_i \neq ff$, the synthesis produces an enforcer with the identity transformation prefix, $\{p_i, c_i, p_i^!\}$, followed by the enforcer synthesised from the continuation $\varphi_i$, i.e., $[p_i, c_i] \varphi_i$ is synthesised as $\{p_i, c_i, p_i^!\} [\varphi_i]$;

(ii) when $m_i$ is derived from a branch of the form $[p_i, c_i] ff$, the synthesis produces an enforcer with the identity transformation prefix, $\{p_i, c_i, \tau\}$, that drops every concrete action matching the symbolic action $\{p_i, c_i\}$, followed by the recursive variable of the branch $y$, i.e., a branch of the form $[p_i, c_i] ff$ is translated into $\{p_i, c_i, \tau\} y$.

**Example 13.** Recall formula $\varphi_1$ from Example 1, recast in term of $\text{sHML}_{nf}$'s grammar:

$$\varphi_1 \equiv \text{max } X. \bigwedge \left( [\{(d)?\req, d\neq j\} X \land [d?\req, \text{true}] ff \right)$$

Using the synthesis function defined in Definition 12, we can generate the enforcer

$$\langle \varphi_1 \rangle = \text{rec } x.\text{rec } z. \sum \left( [\{(d)?\req, d\neq j\} \text{rec } y.\{(d)!\ans, \text{true}\}, x + [d?\req, \text{true}, \tau], y \right)$$

which can be optimized by removing redundant recursive constructs (e.g., $\text{rec } z.$), obtaining:

$$= \text{rec } x.\{(d)?\req, d\neq j\} \text{rec } y.\{(d)!\ans, \text{true}\}, x + [d?\req, \text{true}, \tau], y = m_t$$

We now present the first main result to the paper.

**Theorem 14 (Enforcement).** The (sub)logic $\text{sHML}_{nf}$ is enforceable.

**Proof.** By Definition 3, the result follows if we show that for all $\varphi \in \text{sHML}_{nf}$, $\langle \varphi \rangle$ enforces $\varphi$. By Definition 8, this is a corollary following from Propositions 15 and 16 stated below. ◀

**Proposition 15 (Enforcement Soundness).** For every system $s \in \text{Sys}$ and $\varphi \in \text{sHML}_{nf}$ then $\varphi \in \text{SAT}$ implies $\langle \varphi \rangle[s] \in [\varphi]$. ◀

**Proposition 16 (Enforcement Transparency).** For every system $s \in \text{Sys}$ and $\varphi \in \text{sHML}_{nf}$ then $s \in [\varphi]$ implies $\langle \varphi \rangle[s] \sim s$. ◀

Following Theorem 14, to show that $\text{sHML}$ is an enforceable logic, we only need to show that for every $\varphi \in \text{sHML}$ there exists a corresponding $\psi \in \text{sHML}_{nf}$ with the same semantic meaning, i.e., $[\varphi] = [\psi]$. In fact, we go a step further and provide a constructive proof using a transformation $\langle - \rangle : \text{sHML} \rightarrow \text{sHML}_{nf}$ that derives a semantically equivalent $\text{sHML}_{nf}$ formula from a standard $\text{sHML}$ formula. As a result, from an arbitrary $\text{sHML}$ formula $\varphi$ we can then automatically synthesise a correct enforcer using $\langle \varphi \rangle$ which is useful for tool construction.

Our transformation $\langle \varphi \rangle$ relies on a number of steps; here we provide an outline of these steps. First, we assume $\text{sHML}$ formulas that only use symbolic actions with normalised patterns $p$, i.e., patterns that do not use any data or free data variables (but they may use bound data variables). In fact, any symbolic action $\{p, c\}$ can be easily converted into a corresponding one using normalised patterns as shown in the next example.
Example 17. Consider the symbolic action $\{d\text{ans}, d \neq j\}$. It may be converted to a corresponding normalised symbolic action by replacing every occurrence of a data or free data variable in the pattern by a fresh bound variable, and then add an equality constraint between the fresh variable and the data or data variable it replaces in the pattern condition. In our case, we would obtain $\{(e)((f), d \neq j \land e = d \land f = \text{ans}\}$.

Our algorithm for converting sHML formulas (with normalised patterns) to sHML$_{nf}$ formulas, $\llbracket - \rrbracket$, is based on Rabinovich’s work [41] for determinising systems of equations which, in turn relies on the standard powerset construction for converting NFAs into DFAs.

It consists in the following six stages that we outline below:

1. We unfold each recursive construct in the formula, to push recursive definitions inside the formula body. E.g., the formula $\max X.([p_1, c_1]X \land [p_2, c_2]ff)$ is expanded to the formula $\max \{p_1, c_1\}\{(\max X.[p_1, c_1]X \land [p_2, c_2]ff) \land [p_2, c_2]ff\}$.

2. The formula is converted into a system of equations. E.g., the expanded formula from the previous stage is converted into the set $\{X_0 = [p_1, c_1]X_0 \land [p_2, c_2]X_1, X_1 = ff\}$.

3. For every equation, the symbolic actions in the right hand side that are of the same kind are alpha-converted so that their bound variables match. E.g., Consider $X_0 = [p_1, c_1]X_0 \land [p_2, c_2]X_1$ from the previous stage where, for the sake of the example, $p_1 = (d_1)! (d_2)$ and $p_2 = (d_3)! (d_4)$. The patterns in the symbolic actions are made syntactically equivalent by renaming $d_3$ and $d_4$ in $\{p_2, c_2\}$ into $d_1$ and $d_2$ respectively.

4. For equations with matching patterns in the symbolic actions, we create a variant that symbolically covers all the (satisfiable) permutations on the symbolic action conditions. E.g., Consider $X_0 = [p_1, c_1]X_0 \land [p_1, c_3]X_1$ from the previous stage. We expand this to $X_0 = [p_1, c_1 \land c_3]X_0 \land [p_1, c_1 \land c_3]X_1 \land [p_1, c_1 \land c_3]X_0 \land [p_1, \neg (c_1) \land c_3]X_1$.

5. For equations with branches having syntactically equivalent symbolic actions, we carry out a unification procedure akin to standard powerset constructions. E.g., we convert the equation from the previous step to $X_{(0)} = [p_1, c_1 \land c_3]X_{(0, 1)} \land [p_1, c_1 \land \neg (c_3)]X_{(0)} \land [p_1, \neg (c_1) \land c_3]X_{(1)}$ using the (unified) fresh variables $X_{(0)}, X_{(1)}$ and $X_{(0, 1)}$.

6. From the unified set of equations we generate again the sHML formula starting from $X_{(0)}$. This procedure may generate redundant recursion binders, i.e., $\max X.\varphi$ where $X \notin \text{fv}(\varphi)$, and we filter these out in a subsequent pass.

We now state the second main result of the paper.

Theorem 18 (Normalisation). For any $\varphi \in \text{sHML}$ there exists $\psi \in \text{sHML}_{nf}$ s.t. $\llbracket \varphi \rrbracket = \llbracket \psi \rrbracket$.

Proof. The witness formula in normal form is $\llbracket \varphi \rrbracket$, where we show that each and every stage in the translation procedure preserves semantic equivalence.

6 Alternative Transparency Enforcement

Transparency for a property $\varphi$, Definition 6, only restricts enforcers from modifying the behaviour of satisfying systems, i.e., when $s \in \llbracket \varphi \rrbracket$, but fails to specify any enforcement behaviour for the cases when the SuS violates the property $s \notin \llbracket \varphi \rrbracket$. In this section, we consider an alternative transparency requirement for a property $\varphi$ that incorporates the expected enforcement behaviour for both satisfying and violating systems. More concretely, in the case of safety languages such as sHML, a system typically violates a property along a specific set of execution traces; in the case of a satisfying system this set of “violating traces” is empty. However, not every behaviour of a violating system would be part of this set of violating traces and, in such cases, the respective enforcer should be required to leave the generated behaviour unaffected.
On Runtime Enforcement via Suppressions

Definition 19 (Violating-Trace Semantics). A logic $\mathcal{L}$ with an interpretation over systems $[\cdot] : \mathcal{L} \mapsto \mathcal{P}(\text{Sys})$ has a violating-trace semantics whenever it has a secondary interpretation $[\cdot]_v : \mathcal{L} \mapsto \mathcal{P}(\text{Sys} \times \text{Act}^*)$ satisfying the following conditions for all $\varphi \in \mathcal{L}$:

1. $(s, t) \in \mathcal{L}[\varphi]$, implies $s \notin \mathcal{L}[\varphi]$ and $s \not\Rightarrow$.
2. $s \notin \mathcal{L}[\varphi]$ implies $\exists t : (s, t) \in \mathcal{L}[\varphi]_v$.

We adapt the work in [26] to give sHML a violating-trace semantics. Intuitively, the judgement $(s, t) \in \mathcal{L}[\varphi]_v$ according to Definition 20 below, denotes the fact that $s$ violates the sHML property $\varphi$ along trace $t$.

Definition 20 (Alternative Semantics for sHML [26]). The forcing relation $\vdash_v \subseteq (\text{Sys} \times \text{Act}^* \times \text{sHML})$ is the least relation satisfying the following rules:

- $(s, c, \text{ff}) \in \mathcal{R}$ always
- $(s, t, \bigwedge_{j \in I} \varphi_j) \in \mathcal{R}$ if $\exists j \in I$ such that $(s, t, \varphi_j) \in \mathcal{R}$
- $(s, ot, [p, c] \varphi) \in \mathcal{R}$ if $\text{match}(p, c) = \sigma, \sigma \downarrow \text{true}$ and $s \not\Rightarrow s'$ and $(s', t, \varphi\sigma) \in \mathcal{R}$
- $(s, t, \text{max} X. \varphi) \in \mathcal{R}$ if $(s, t, \varphi\{\text{max} X. \varphi/X\}) \in \mathcal{R}$.

We write $s, t \vdash_v \varphi$ (or $(s, t) \in \mathcal{L}[\varphi]_v$) in lieu of $(s, t, \varphi) \in \vdash_v$. We say that trace $t$ is a violating trace for $s$ with respect to $\varphi$ whenever $s, t \vdash_v \varphi$. Dually, $t$ is a non-violating trace for $\varphi$ whenever there does not exist a system $s$ such that $s, t \vdash_v \varphi$.

Example 21. Recall $\varphi_1, s_b$ from Example 1 where $\varphi_1 \in \text{sHML}$, and also $m_k$ from Example 5 where we argued in Example 13 that $\{\varphi_1\} = m_k$ (modulo cosmetic optimisations). Even though $s_b \notin \mathcal{L}[\varphi_1]$, not all of its exhibited behaviours constitute violating traces: for instance, $s_b \overset{\text{req ians}}{\rightarrow} s_b$ is not a violating trace according to Definition 20. Correspondingly, we also have $m_k[s_b] \overset{\text{req ians}}{\rightarrow} m_k[s_b]$.

Theorem 22 (Adapted and extended from [26]). The alternative interpretation $[\cdot]_v$ of Definition 20 is a violating-trace semantics for sHML (with $[\cdot]$ from Figure 1) in the sense of Definition 19.

Equipped with Definition 20 we can define an alternative definition for transparency that concerns itself with preserving exhibited traces that are non-violating. We can then show that the monitor synthesis for sHML of Definition 12 observes non-violating trace transparency.

Definition 23 (Non-Violating Trace Transparency). An enforcer $m$ is transparent with respect to the non-violating traces of a formula $\varphi$, denoted as $\text{nvtens}(m, \varphi)$, iff for all $s \in \text{Sys}$ and $t \in \text{Act}^*$, when $s, t \not\vdash_v \varphi$ then

- $s \not\Rightarrow s'$ implies $m[s] \not\Rightarrow m'[s']$ for some $m'$, and
- $m[s] \not\Rightarrow m'[s']$ implies $s \not\Rightarrow s'$.

Proposition 24 (Non-Violating Trace Transparency). For all $\varphi \in \text{sHML}$, $s \in \text{Sys}$ and $t \in \text{Act}^*$, when $s, t \not\vdash_v \varphi$ then

- $s \not\Rightarrow s'$ implies $\langle \varphi \rangle[s] \not\Rightarrow m'[s']$, and
- $\langle \varphi \rangle[s] \not\Rightarrow m'[s']$ implies $s \not\Rightarrow s'$.

We can thus obtain a new definition for “$m$ enforces $\varphi$" instead of Definition 8 by requiring sound enforcement, Definition 6, and non-violating trace transparency, Definition 23 (instead of the transparent enforcement of Definition 6). This in turn gives us a new definition for enforceability for a logic, akin to Definition 3. Using Propositions 15 and 24, one can show that sHML is also enforceable with respect to the new definition as well.
7 Conclusion

This paper presents a preliminary investigation of the enforceability of properties expressed in a process logic. We have focussed on a highly expressive and standard logic, µHML, and studied the ability to enforce µHML properties via a specific kind of monitor that performs suppression-based enforcement. We concluded that sHML, identified in earlier work as a maximally expressive safety fragment of µHML, is also an enforceable logic. To show this, we first defined enforceability for logics and system descriptions interpreted over labelled transition systems. Although enforceability builds upon soundness and transparency requirements that have been considered in other work, our branching-time framework allowed us to consider novel definitions for these requirements. We also contend that the definitions that we develop for the enforcement framework are fairly modular: e.g., the instrumentation relation is independent of the specific language constructs defining our transducer monitors and it functions as expected as long as the transition semantics of the transducer and the system are in agreement. Based on this notion of enforcement, we devise a two-phase procedure to synthesise correct enforcement monitors. We first identify a syntactic subset of our target logic sHML that affords certain structural properties and permits a compositional definition of the synthesis function. We then show that, by augmenting existing rewriting techniques to our setting, we can convert any sHML formula into this syntactic subset.

Related Work

In his seminal work [44], Schneider regards a property (in a linear-time setting) to be enforceable if its violation can be detected by a truncation automaton, and prevents its occurrence via system termination; by preventing misbehaviour, these enforcers can only enforce safety properties. Ligatti et al. in [33] extended this work via edit automata—an enforcement mechanism capable of suppressing and inserting system actions. A property is thus enforceable if it can be expressed as an edit automaton that transforms invalid executions into valid ones via suppressions and insertions. Edit automata are capable of enforcing instances of safety and liveness properties, along with other properties such as infinite renewal properties [33, 10]. As a means to assess the correctness of these automata, the authors introduced soundness and transparency. In both of these settings, there is no clear separation between the specification and the enforcement mechanism, and properties are encoded in terms of the languages accepted by the enforcement model itself, i.e., as edit/truncation automata. By contrast, we keep the specification and verification aspects of the logic separate.

Bielova et al. [10, 11] remark that soundness and transparency do not specify to what extent a transducer should modify an invalid execution. They thus introduce a predictability criterion to prevent transducers from transforming invalid executions arbitrarily. More concretely, a transducer is predictable if one can predict the number of transformations that it will apply in order to transform an invalid execution into a valid one, thereby preventing enforcers from applying unnecessary transformations over an invalid execution. Using this notion, Bielova et al. thus devise a more stringent notion of enforceability. Although we do not explore this avenue, Definition 23 may be viewed as an attempt to constrain transformations of violating systems in a branching-time setup, and should be complementary to these predictability requirements.

Könighofer et al. in [29] present a synthesis algorithm that produces action replacement transducers called shields from safety properties encoded as automata-based specifications. Shields analyse the inputs and outputs of a reactive systems and enforce properties by
modifying the least amount of output actions whenever the system deviates from the specified behaviour. By definition, shields should adhere to two desired properties, namely correctness and minimum deviation which are, in some sense, analogous to soundness and transparency respectively. Falcone et al. in [19, 21, 20], also propose synthesis procedures to translate properties – expressed as Streett automata – into the resp., enforcers. The authors show that most of the property classes defined within the Safety-Progress hierarchy [40] are enforceable, as they can be encoded as Streett automata and subsequently converted into enforcement automata. As opposed to Ligatti et al., both Könighofer et al. and Falcone et al. separate the specification of the property from the enforcement mechanism, but unlike our work they do not study the enforceability of a branching time logic.

To the best of our knowledge, the only other work that tackles enforceability for the modal µ-calculus [30] (a reformulation of µHML) is that of Martinelli et al. in [36, 37]. Their approach is, however, different from ours. In addition to the µ-calculus formula to enforce, their synthesis function also takes a “witness” system satisfying the formula as a parameter. This witness system is then used as the behaviour that is mimicked by the instrumentation via suppression, insertion or replacement mechanisms. Although the authors do not explore automated correctness criteria such as the ones we study in this work, it would be interesting to explore the applicability of our methods to their setting.

Bocchi et al. [12] adopt multi-party session types to project the global protocol specifications of distributed networks to local types defining a local protocol for every process in the network that are then either verified statically via typechecking or enforced dynamically via suppression monitors. To implement this enforcement strategy, the authors define a dynamic monitoring semantics for the local types that suppress process interactions so as to conform to the assigned local specification. They prove local soundness and transparency for monitored processes that, in turn, imply global soundness and transparency by construction. Their local enforcement is closely related to the suppression enforcement studied in our work with the following key differences: (i) well-formed branches in a session type are, by construction, explicitly disjoint via the use of distinct choice labels (i.e., similar to our normalised subset SHML), whereas we can synthesise enforcers for every SHML formula using a normalisation procedure; (ii) they give an LTS semantics to their local specifications (which are session types) which allows them to state that a process satisfies a specification when its behaviour is bisimilar to the operational semantics of the local specification—we do not change the semantics of our formulas, which is left in its original denotational form; (iii) they do not provide transparency guarantees for processes that violate a specification, along the lines of Definition 23; (iv) Our monitor descriptions sit at a lower level of abstraction than theirs using a dedicated language, whereas theirs have a session-type syntax with an LTS semantics (e.g., repeated suppressions have to be encoded in our case using the recursion construct while this is handled by their high-level instrumentation semantics).

In [14], Castellani et al. adopt session types to define reading and writing privileges amongst processes in a network as global types for information flow purposes. These global types are projected into local monitors capable of preventing read and write violations by adapting certain aspects of the network. Although their work is pitched towards adaptation [24, 13], rather than enforcement, in certain instances they adapt the network by suppressing messages or by replacing messages with messages carrying a default nonce value. It would be worthwhile investigating whether our monitor correctness criteria could be adapted or extended to this information-flow setting.
Future Work

We plan to extend this work along two different avenues. On the one hand, we will attempt to extend the enforceable fragment of $\mu$HML. For a start, we intend to investigate maximality results for suppression monitors, along the lines of [25, 2]. We also plan to consider more expressive enforcement mechanisms such as insertion and replacement actions. Finally, we will also investigate more elaborate instrumentation setups, such as the ones explored in [1], that can reveal refusals in addition to the actions performed by the system.

On the other hand, we also plan to study the implementability and feasibility of our framework. We will consider target languages for our monitor descriptions that are closer to an actual implementation (e.g., an actor-based language along the lines of [26]). We could then employ refinement analysis techniques and use our existing monitor descriptions as the abstract specifications that are refined by the concrete monitor descriptions. The more concrete synthesis can then be used for the construction of tools that are more amenable towards showing correctness guarantees.

References

On Runtime Enforcement via Suppressions


