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 (μHML) with respect to suppression enforcement. We develop an operational framework for enforcement which we then use to formalise when a monitor enforces a μHML 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 μHML [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* μHML 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 μHML 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 μHML 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 μHML 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, μHML, 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 μHML (a reformulation of the μ-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, μHML. 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

$$\begin{array}{llllll} \varphi,\psi \in \mu\text{HML} ::= \mathsf{tt} & \text{(truth)} & \mid \mathsf{ff} & \text{(falsehood)} & \mid \bigvee_{i\in I}\varphi_i & \text{(disjunction)} \\ \mid \bigwedge_{i\in I}\varphi_i & \text{(conjunction)} & \mid \langle\{p,c\}\rangle\varphi & \text{(possibility)} & \mid [\{p,c\}]\varphi & \text{(necessity)} \\ \mid \mathsf{min}\, X.\varphi & \text{(least fp.)} & \mid \mathsf{max}\, X.\varphi & \text{(greatest fp.)} & \mid X & \text{(fp. variable)} \end{array}$$

Semantics

$$\begin{array}{lll} [\![\mathsf{tt},\rho]\!] \stackrel{\text{def}}{=} \textsc{Sys} & [\![\mathsf{ff},\rho]\!] \stackrel{\text{def}}{=} \varnothing & [\![X,\rho]\!] \stackrel{\text{def}}{=} \rho(X) \\ [\![\bigwedge_{i\in I}\varphi_i,\rho]\!] \stackrel{\text{def}}{=} \bigcap_{i\in I}[\![\varphi_i,\rho]\!] & [\![\mathsf{max}\, X.\varphi,\rho]\!] \stackrel{\text{def}}{=} \bigcup \{S \mid S \subseteq [\![\varphi,\rho[X\mapsto S]]\!]\} & \\ [\![\bigvee_{i\in I}\varphi_i,\rho]\!] \stackrel{\text{def}}{=} \bigcup_{i\in I}[\![\varphi_i,\rho]\!] & [\![\mathsf{min}\, X.\varphi,\rho]\!] \stackrel{\text{def}}{=} \bigcap \{S \mid [\![\varphi,\rho[X\mapsto S]]\!] \subseteq S\} & \end{array}$$

$$[\![[\{p,c\}]\varphi,\rho]\!] \stackrel{\text{def}}{=} \{s \mid (\forall \alpha, r \cdot s \stackrel{\alpha}{\Rightarrow} r \text{ and } (\exists \sigma \cdot \mathsf{mtch}(p,\alpha){=}\sigma \text{ and } c\sigma \Downarrow \mathsf{true})) \text{ implies } q \in [\![\varphi\sigma,\rho]\!]\}$$

$$[\![\langle\{p,c\}\rangle\varphi,\rho]\!] \stackrel{\text{def}}{=} \{s \mid \exists \alpha, r, \sigma \cdot (s \stackrel{\alpha}{\Rightarrow} r \text{ and } \mathsf{mtch}(p,\alpha){=}\sigma \text{ and } c\sigma \Downarrow \mathsf{true} \text{ and } q \in [\![\varphi\sigma,\rho]\!])\}$$

Figure 1 μHML 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 $\langle \textsc{Sys}, \textsc{Act} \cup \{\tau\}, \rightarrow\rangle$ consisting of a set of *system states*, $s, r, q \in \textsc{Sys}$, a set of *observable actions*, $\alpha, \beta \in \textsc{Act}$, and a distinguished silent action $\tau \notin \textsc{Act}$ (where $\mu \in \textsc{Act} \cup \{\tau\}$), and a *transition* relation, $\longrightarrow \subseteq (\textsc{Sys} \times \textsc{Act} \cup \{\tau\} \times \textsc{Sys})$. We write $s \xrightarrow{\mu} r$ in lieu of $(s,\mu,r) \in \rightarrow$, and use $s \stackrel{\mu}{\Longrightarrow} s'$ to denote weak transitions representing $s(\xrightarrow{\tau})^* \cdot \xrightarrow{\mu} \cdot (\xrightarrow{\tau})^* s'$. We refer to s' as a μ-derivative of s. Traces, $t, u \in \textsc{Act}^*$ range over (finite) sequences of observable actions, and we write $s \stackrel{t}{\Rightarrow} r$ to denote a sequence of weak transitions $s \stackrel{\alpha_1}{\Longrightarrow} \ldots \stackrel{\alpha_n}{\Longrightarrow} r$ for $t = \alpha_1, \ldots, \alpha_n$. 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 μHML [32, 3] that uses *symbolic actions* of the form $\{p, c\}$. *Patterns*, p, abstract over actions using *data variables* $d, e, f \in \textsc{Var}$; 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 $\mathsf{mtch}(p,\alpha)$ that returns a substitution σ (when successful) mapping variables in p to the corresponding values in α, *i.e.*, if we instantiate every bound variable d in p with $\sigma(d)$ we obtain α. 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 $\mathbf{fv}(c) \subseteq \mathbf{bv}(p)$; it denotes the *set* of actions $[\![\{p,c\}]\!] \stackrel{\text{def}}{=} \{\, \alpha \mid \exists \sigma \cdot \mathsf{mtch}(p,\alpha){=}\sigma \text{ and } c\sigma \Downarrow \mathsf{true} \,\}$ 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 $X, Y \in \textsc{LVar}$. Apart from standard logical constructs such as conjunctions and disjunctions ($\bigwedge_{i\in I}\varphi_i$ describes a *compound* conjunction, $\varphi_1 \wedge \ldots \wedge \varphi_n$, where $I = \{1, .., n\}$ is a finite set of indices, and similarly for disjunctions), and the characteristic greatest and least fixpoints ($\mathsf{max}\, X.\varphi$ and $\mathsf{min}\, X.\varphi$ bind free occurrences of X in φ), the logic uses necessity

and possibility modal operators with symbolic actions, $[\{p,c\}]\varphi$ and $\langle\{p,c\}\rangle\varphi$, where $\mathbf{bv}(p)$ bind free data variables in c and φ. Formulas in μHML are interpreted over the system powerset domain where $S{\in}\mathcal{P}(\textsc{Sys})$. The semantic definition of Figure 1, $[\![\varphi,\rho]\!]$, is given for *both* open and closed formulas. It employs a valuation from logical variables to sets of states, $\rho \in (\textsc{LVar} \to \mathcal{P}(\textsc{Sys}))$, 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 ($p{=}\alpha$ for some α) and whose condition holds trivially ($c{=}\mathsf{true}$); 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 φ is independent of ρ. A system s *satisfies* formula φ whenever $s \in [\![\varphi]\!]$ whereas a formula φ is *satisfiable*, $\varphi \in \textsc{Sat}$, whenever there exists a system r such that $r \in [\![\varphi]\!]$.

▶ **Example 1.** Consider two systems (a good system, $s_{\mathbf{g}}$, and a bad one, $s_{\mathbf{b}}$) 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 $s_{\mathbf{g}}$ outputs an answer ($i!\mathsf{ans}$) for every request ($i?\mathsf{req}$), $s_{\mathbf{b}}$ occasionally refuses to answer a given request (see the underlined branch). Both systems terminate with $i?\mathsf{cls}$.

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

We can specify that two consecutive requests on port i indicate invalid behaviour via the μHML formula $\varphi_0 \stackrel{\text{def}}{=} \mathsf{max}\, X.[i?\mathsf{req}]\,([i!\mathsf{ans}]X \wedge [i?\mathsf{req}]\mathsf{ff})$; it defines an invariant property ($\mathsf{max}\, X.(\cdots)$) requiring that whenever a system interacting on i inputs a request, it cannot input a subsequent request, *i.e.,* $[i?\mathsf{req}]\mathsf{ff}$, unless it outputs an answer beforehand, in which case the formula recurses, *i.e.,* $[i!\mathsf{ans}]X$. Using symbolic actions, we can generalise φ_0 by requiring the property to hold for *any* interaction happening on *any* port number *except* j.

$$\varphi_1 \stackrel{\text{def}}{=} \mathsf{max}\, X.[\{(d)?\mathsf{req}, d{\neq}j\}]([\{d!\mathsf{ans}, \mathsf{true}\}]X \wedge [\{d?\mathsf{req}, \mathsf{true}\}]\mathsf{ff})$$

In φ_1, $(d)?\mathsf{req}$ binds the free occurrences of d found in $d{\neq}j$ and $[\{d!\mathsf{ans}, \mathsf{true}\}]X \wedge [\{d?\mathsf{req}, \mathsf{true}\}]\mathsf{ff}$. Using Figure 1, one can check that $s_{\mathbf{g}} {\in} [\![\varphi_1]\!]$, whereas $s_{\mathbf{b}} {\notin} [\![\varphi_1]\!]$ since $s_{\mathbf{b}} \xrightarrow{i?\mathsf{req}} \cdot \xrightarrow{i?\mathsf{req}} \ldots$ ◀

3 An Operational Model for Enforcement

Our operational mechanism for enforcing properties over systems uses the (symbolic) transducers $m, n \in \textsc{Trn}$ 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 $\big(\mathbf{fv}(c) \cup \mathbf{fv}(p') \cup \mathbf{fv}(m)\big) \subseteq \mathbf{bv}(p)$, and yield an LTS with labels of the form $\gamma \blacktriangleright \mu$, where $\gamma \in (\textsc{Act} \cup \{\bullet\})$. Our syntax assumes a well-formedness constraint where for every $\{p,c,p'\}.m$, $\mathbf{bv}(c) \cup \mathbf{bv}(p') = \varnothing$. Intuitively, a transition $m \xrightarrow{\alpha \blacktriangleright \mu} n$ denotes the fact that the transducer in state m *transforms* the visible action α (produced by the system) into the action μ (which can possibly become silent) and transitions into state n. In this sense, the transducer action $\alpha \blacktriangleright \tau$ represents the *suppression* of action α, action $\alpha \blacktriangleright \beta$ represents the *replacing* of α by β, and $\alpha \blacktriangleright \alpha$ denotes the *identity* transformation. The special case $\bullet \blacktriangleright \alpha$ encodes the *insertion* of α, where $\bullet$ represents that the transition is not induced by any system action.

The key transition rule in Figure 2 is $\textsc{eTrn}$. It states that the symbolic-prefix transducer $\{p,c,p'\}.m$ can transform an (extended) action γ into the concrete action μ, as long as

Syntax

$$m, n \in \textsc{Trn} ::= \quad \mathsf{id} \quad | \quad \{p, c, p'\}.m \quad | \quad \textstyle\sum_{i\in I} m_i \quad | \quad \mathsf{rec}\, x.m \quad | \quad x$$

Dynamics

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

$$\textsc{eTrn}\ \frac{\mathsf{mtch}(p, \gamma) = \sigma \qquad c\sigma \Downarrow \mathsf{true} \qquad \mu = p'\sigma}{\{p, c, p'\}.m \xrightarrow{\gamma \blacktriangleright \mu} m\sigma}$$

Instrumentation

$$\textsc{iTrn}\ \frac{s \xrightarrow{\alpha} s' \qquad m \xrightarrow{\alpha \blacktriangleright \mu} n}{m[s] \xrightarrow{\mu} n[s']} \qquad \textsc{iAsy}\ \frac{s \xrightarrow{\tau} s'}{m[s] \xrightarrow{\tau} m[s']} \qquad \textsc{iIns}\ \frac{m \xrightarrow{\bullet \blacktriangleright \mu} n}{m[s] \xrightarrow{\mu} n[s]} \qquad \textsc{iTer}\ \frac{s \xrightarrow{\alpha} s' \qquad m \overset{\alpha}{\not\longrightarrow} \qquad m \overset{\bullet}{\not\longrightarrow}}{m[s] \xrightarrow{\alpha} \mathsf{id}[s']}$$

Figure 2 A model for transducers (I is a finite index set and $m \overset{\gamma}{\not\longrightarrow}$ means $\nexists \mu, n \cdot m \xrightarrow{\gamma \blacktriangleright \mu} n$)

the action matches with pattern p with substitution σ, $\mathsf{mtch}(p, \gamma){=}\sigma$, and the condition is satisfied by σ, $c\sigma \Downarrow \mathsf{true}$ (the matching function is lifted to extended actions and patterns in the obvious way, where $\mathsf{mtch}(\bullet, \bullet){=}\varnothing$). In such a case, the transformed action is $\mu{=}p'\sigma$, *i.e.,* the action μ resulting from the instantiation of the free data variables in pattern p' with the corresponding values mapped by σ, and the transducer state reached is $m\sigma$. By contrast, in rule eId, the transducer 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 SuS s with the transformations of a transducer monitor m that *agrees* with the (observable) actions Act of s. The term $m[s]$ thus denotes the resulting *monitored system* whose behaviour is defined in terms of $\textsc{Act} \cup \{\tau\}$ from the system's LTS. Concretely, rule iTrn states that when a system s transitions with an observable action α to s' and the transducer m can *transform* this action into μ and transition to n, the instrumented system $m[s]$ transitions with action μ to $n[s']$. 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 μ 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 α to s', and the transducer m does not specify how to transform it ($m \overset{\alpha}{\not\longrightarrow}$), nor can it transition to a new transducer state by inserting an action ($m \overset{\bullet}{\not\longrightarrow}$), the system is still allowed to transition while the transducer's transformation activity is ceased, *i.e.,* it acts like the identity id from that point onwards.

▶ **Example 2.** Consider the insertion transducer $m_{\mathbf{i}}$ and the replacement transducer $m_{\mathbf{r}}$ below:

$$m_{\mathbf{i}} \overset{\text{def}}{=} \{\bullet, \mathsf{true}, i?\mathsf{req}\}.\{\bullet, \mathsf{true}, i!\mathsf{ans}\}.\mathsf{id}$$

$$m_{\mathbf{r}} \overset{\text{def}}{=} \mathsf{rec}\, x.\big(\{(d)?\mathsf{req}, \mathsf{true}, j?\mathsf{req}\}.x + \{(d)!\mathsf{ans}, \mathsf{true}, j!\mathsf{ans}\}.x + \{(d)?\mathsf{cls}, \mathsf{true}, j?\mathsf{cls}\}.x\big)$$

When instrumented with a system, $m_{\mathbf{i}}$ inserts the two successive actions $i?\mathsf{req}$ and $i!\mathsf{ans}$ before

behaving as the identity. Concretely in the case of $s_{\mathbf{b}}$ we can only start the computation as:

$$m_{\mathbf{i}}[s_{\mathbf{b}}] \xrightarrow{i?\mathsf{req}} \{\bullet, \mathsf{true}, i!\mathsf{ans}\}.\mathsf{id}[s_{\mathbf{b}}] \xrightarrow{i!\mathsf{ans}} \mathsf{id}[s_{\mathbf{b}}] \xrightarrow{\alpha} \ldots \qquad (\text{where } s_{\mathbf{b}} \xrightarrow{\alpha})$$

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

$$m_{\mathbf{r}}[s_{\mathbf{b}}] \xrightarrow{j?\mathsf{req}} m_{\mathbf{r}}[i!\mathsf{ans}.s_{\mathbf{b}}] \xrightarrow{j!\mathsf{ans}} m_{\mathbf{r}}[s_{\mathbf{b}}] \xrightarrow{j?\mathsf{cls}} m_{\mathbf{r}}[\mathsf{nil}]$$

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

$$m_{\mathbf{s}} \stackrel{\text{def}}{=} \mathsf{rec}\, x.\big(\{(d)?\mathsf{req}, d \neq j, \tau\}.x + \{(d)!\mathsf{ans}, \mathsf{true}, d!\mathsf{ans}\}.x\big)$$
$$m_{\mathbf{t}} \stackrel{\text{def}}{=} \mathsf{rec}\, x.\big(\{(d)?\mathsf{req}, d \neq j, d?\mathsf{req}\}.\mathsf{rec}\, y.\big(\{d!\mathsf{ans}, \mathsf{true}, d!\mathsf{ans}\}.x + \{d?\mathsf{req}, \mathsf{true}, \tau\}.y\big)\big)$$

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

$$m_{\mathbf{s}}[s_{\mathbf{b}}] \xrightarrow{\tau} m_{\mathbf{s}}[i!\mathsf{ans}.s_{\mathbf{b}}] \xrightarrow{i!\mathsf{ans}} m_{\mathbf{s}}[s_{\mathbf{b}}] \xrightarrow{\tau} m_{\mathbf{s}}[i!\mathsf{ans}.s_{\mathbf{b}}] \xrightarrow{i!\mathsf{ans}} m_{\mathbf{s}}[s_{\mathbf{b}}] \ldots$$

Note that $m_{\mathbf{s}}$ does not specify a transformation behaviour for when the monitored system produces inputs with payload other than req. The instrumentation handles this underspecification by ceasing suppression activity; in the case of $s_{\mathbf{b}}$ we get $m_{\mathbf{s}}[s_{\mathbf{b}}] \xrightarrow{i?\mathsf{cls}} \mathsf{id}[\mathsf{nil}]$. The transducer $m_{\mathbf{t}}$ 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 req followed by an output on ans) sequence. For $s_{\mathbf{b}}$ we can observe the following:

$$\begin{aligned} m_{\mathbf{t}}[s_{\mathbf{b}}] &\xrightarrow{i?\mathsf{req}} \mathsf{rec}\, y.\big(\{i!\mathsf{ans}, \mathsf{true}, i!\mathsf{ans}\}.m_{\mathbf{t}} + \{i?\mathsf{req}, \mathsf{true}, \tau\}.y\big)[s_{\mathbf{b}}] \\ &\xrightarrow{\tau} \mathsf{rec}\, y.\big(\{i!\mathsf{ans}, \mathsf{true}, i!\mathsf{ans}\}.m_{\mathbf{t}} + \{i?\mathsf{req}, \mathsf{true}, \tau\}.y\big)[i!\mathsf{ans}.s_{\mathbf{b}}] \xrightarrow{i!\mathsf{ans}} m_{\mathbf{t}}[s_{\mathbf{b}}] \end{aligned} \quad \blacktriangleleft$$

In the sequel, we find it convenient to refer to $\underline{p}$ 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 $p'{=}\underline{p}$ and simply write $\{p, c\}.m$; note that if $\mathbf{bv}(p) = \varnothing$, then $\underline{p}{=}p$. Similarly, we elide c whenever $c{=}\mathsf{true}$. This allows us to express $m_{\mathbf{t}}$ from Example 2 as $\mathsf{rec}\, x.\big(\{(d)?\mathsf{req}, d{\neq}j\}.\mathsf{rec}\, y.\big(\{d!\mathsf{ans}\}.x + \{d?\mathsf{req}, \tau\}.y\big)\big)$.

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 φ is *enforceable* iff there *exists* a transducer m such that m *enforces* φ. ◀

Definition 3 depends on what is considered to be an adequate definition for "m *enforces* φ". 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, φ, whenever this property *is satisfiable*.

▶ **Definition 4** (Sound Enforcement)**.** Enforcer m *soundly enforces* a formula φ, denoted as $\mathsf{senf}(m, \varphi)$, iff for *all* $s \in \textsc{Sys}$, $\varphi \in \mathit{Sat}$ implies $m[s] \in [\![\varphi]\!]$ holds. ◀

▶ **Example 5.** Recall φ_1, $s_{\mathbf{g}}$ and $s_{\mathbf{b}}$ from Example 1 where $s_{\mathbf{g}} \in [\![\varphi_1]\!]$ (hence $\varphi_1 \in \text{SAT}$) and $s_{\mathbf{b}} \notin [\![\varphi_1]\!]$. For the enforcers $m_{\mathbf{i}}$, $m_{\mathbf{r}}$, $m_{\mathbf{s}}$ and $m_{\mathbf{t}}$ presented in Example 2, we have:

- $m_{\mathbf{i}}[s_{\mathbf{b}}] \notin [\![\varphi_1]\!]$, since $m_{\mathbf{i}}[s_{\mathbf{b}}] \xrightarrow{i?\mathsf{req}} \cdot \xrightarrow{i!\mathsf{ans}} \mathsf{id}[s_{\mathbf{b}}] \xrightarrow{i?\mathsf{req}} \mathsf{id}[s_{\mathbf{b}}] \xrightarrow{i?\mathsf{req}} \mathsf{id}[s_{\mathbf{b}}]$. This counter example implies that $\neg\mathsf{senf}(m_{\mathbf{i}}, \varphi_1)$.
- $m_{\mathbf{r}}[s_{\mathbf{g}}] \in [\![\varphi_1]\!]$ and $m_{\mathbf{r}}[s_{\mathbf{b}}] \in [\![\varphi_1]\!]$. Intuitively, this is because the ensuing instrumented systems only generate (replaced) actions that are not of concern to φ_1. Since this behaviour applies to any system $m_{\mathbf{r}}$ is composed with, we can conclude that $\mathsf{senf}(m_{\mathbf{r}}, \varphi_1)$.
- $m_{\mathbf{s}}[s_{\mathbf{g}}] \in [\![\varphi_1]\!]$ and $m_{\mathbf{s}}[s_{\mathbf{b}}] \in [\![\varphi_1]\!]$ because the resulting instrumented systems never produce inputs with req on a port number other than j. We can thus conclude that $\mathsf{senf}(m_{\mathbf{s}}, \varphi_1)$.
- $m_{\mathbf{t}}[s_{\mathbf{g}}] \in [\![\varphi_1]\!]$ and $m_{\mathbf{t}}[s_{\mathbf{b}}] \in [\![\varphi_1]\!]$. 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 $\mathsf{senf}(m_{\mathbf{t}}, \varphi_1)$. ◀

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 $m_{\mathbf{s}}$ from Example 2. Although $m_{\mathbf{s}}$ manages to suppress the violating executions of system $s_{\mathbf{b}}$, thereby bringing it in line with property φ_1, it needlessly modifies the behaviour of $s_{\mathbf{g}}$ (namely it prohibits it from producing any inputs with req on port numbers that are not j), even though it satisfies φ_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 φ, 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 φ, denoted as $\mathsf{tenf}(m, \varphi)$, 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 $s_{\mathbf{g}}$—why $m_{\mathbf{s}}$ does *not* transparently enforce φ_1. We can also argue easily why $\neg\mathsf{tenf}(m_{\mathbf{r}}, \varphi_1)$ either: the simple system $i?\mathsf{req}.\mathsf{nil}$ trivially satisfies φ_1 but, clearly, we have the inequality $m_{\mathbf{r}}[i?\mathsf{req}.\mathsf{nil}] \not\sim i?\mathsf{req}.\mathsf{nil}$ since $m_{\mathbf{r}}[i?\mathsf{req}.\mathsf{nil}] \xrightarrow{j?\mathsf{req}} m_{\mathbf{r}}[\mathsf{nil}]$ and $i?\mathsf{req}.\mathsf{nil} \not\xrightarrow{j?\mathsf{req}}$.

It turns out that enforcer $\mathsf{tenf}(m_{\mathbf{t}}, \varphi_1)$, 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 $s_{\mathbf{g}}$: it satisfies φ_1 and $m_{\mathbf{t}}[s_{\mathbf{g}}] \sim s_{\mathbf{g}}$ holds. ◀

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

For any reasonably expressive logic (such as μHML), it is usually the case that *not* every formula can be enforced, as the following example informally illustrates.

▶ **Example 9.** Consider the μHML property φ_{ns}, together with the two systems s_{ra} and s_{r}:

$$\varphi_{\mathsf{ns}} \stackrel{\text{def}}{=} [i?\mathsf{req}]\mathsf{ff} \vee [i!\mathsf{ans}]\mathsf{ff} \qquad s_{\mathsf{ra}} \stackrel{\text{def}}{=} i?\mathsf{req}.\mathsf{nil} + i!\mathsf{ans}.\mathsf{nil} \qquad s_{\mathsf{r}} \stackrel{\text{def}}{=} i?\mathsf{req}.\mathsf{nil}$$

A system satisfies φ_{ns} if *either* it cannot produce action $i?\mathsf{req}$ *or* it cannot produce action $i!\mathsf{ans}$. 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 $m_{\mathsf{r}} \stackrel{\text{def}}{=} \mathsf{rec}\, y.\big(\{i?\mathsf{req}, \tau\}.y + \{i!\mathsf{ans}, \tau\}.y\big)$ would in

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

Figure 3 The syntax for the safety μHML fragment, sHML.

fact be able to suppress the offending actions produced by s_{ra}, thus obtaining $m_{\mathsf{r}}[s_{\mathsf{ra}}] \in [\![\varphi_{\mathsf{ns}}]\!]$. However, it would also suppress the sole action $i?\mathsf{req}$ produced by the system s_{r}, even though this system satisfies φ_{ns}. This would, in turn, violate the transparency criterion of Definition 6 since it needlessly suppresses s_{r}'s actions, *i.e.,* although $s_{\mathsf{r}} \in [\![\varphi_{\mathsf{ns}}]\!]$ we have $m_{\mathsf{r}}[s_{\mathsf{r}}] \not\sim s_{\mathsf{r}}$. 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 μHML 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 φ in this enforceable subset, we provide an *automated procedure* to *synthesise* a monitor m from it that correctly enforces φ 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 side-step 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 μHML that can be handled by violation-detecting (recogniser) monitors. We thus limit our enforceability study to this maximal safety fragment, called sHML, since a *transparent* suppression monitor cannot judiciously suppress actions without first detecting a (potential) violation. Figure 3 recalls the syntax for sHML. The logic is restricted to *truth* and *falsehood* (tt and ff), conjunctions ($\bigwedge_{i \in I} \varphi$), and necessity modalities ($[\{p, c\}]\varphi$), while recursion may only be expressed through greatest fixpoints ($\mathsf{max}\, X.\varphi$); the semantics follows that of Figure 1.

A standard way how to achieve our aims would be to (i) define a (total) synthesis function $(\!|-|\!) :: \text{sHML} \mapsto \textsc{Trn}$ from sHML formulas to suppression monitors and (ii) then show that for *any* $\varphi \in \text{sHML}$, the synthesised monitor $(\!|\varphi|\!)$ enforces φ. 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 naive approach to such a scheme is bound to fail, as discussed in the next example.

▶ **Example 10.** Consider a semantically equivalent reformulation of φ_1 from Example 1.

$$\varphi_2 \stackrel{\text{def}}{=} \mathsf{max}\, X.([\{(d)?\mathsf{req}, d{\neq}j\}][\{d!\mathsf{ans}, \mathsf{true}\}]X) \wedge ([\{(d)?\mathsf{req}, d{\neq}j\}][\{d?\mathsf{req}, \mathsf{true}\}]\mathsf{ff})$$

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

$$m_{\mathbf{b}} \stackrel{\text{def}}{=} \mathsf{rec}\, x.\big(\{(d)?\mathsf{req}, d \neq j\}.\{d!\mathsf{ans}\}.x\big) \;+\; \big(\{(d)?\mathsf{req}, d \neq j\}.\mathsf{rec}\, y.\{d?\mathsf{req}, \tau\}.y\big)$$

One can easily see that $m_{\mathbf{b}}$ does *not* behave deterministically, *nor* does it soundly enforce φ_2. For instance, for the violating system $i?\mathsf{req}.i?\mathsf{req}.\mathsf{nil} \notin [\![\varphi_2]\!] (= [\![\varphi_1]\!])$ we can observe the transition sequence $m_{\mathbf{b}}[i?\mathsf{req}.i?\mathsf{req}.\mathsf{nil}] \xrightarrow{i?\mathsf{req}} \{i!\mathsf{ans}\}.m_{\mathbf{b}}[i?\mathsf{req}.\mathsf{nil}] \xrightarrow{i?\mathsf{req}} \mathsf{id}[\mathsf{nil}]$. ◀

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:

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

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 $[\![\{p_j, c_j\}]\!] \cap [\![\{p_h, c_h\}]\!] = \varnothing$.
2. For every $\mathsf{max}\, X.\varphi$ we have $X \in \mathbf{fv}(\varphi)$. ◀

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 $\varphi \wedge \mathsf{ff}$ or $\mathsf{max}\, X_0. \ldots \mathsf{max}\, X_n.\mathsf{ff}$). 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 $(\!|-|\!) : \text{sHML}_{\mathbf{nf}} \mapsto \text{TRN}$ is defined inductively as:

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

$$(\!|\, \bigwedge_{i\in I} [\{p_i, c_i\}]\varphi_i \,|\!) \stackrel{\text{def}}{=} \mathsf{rec}\, y.\sum_{i\in I} \begin{cases} \{p_i, c_i, \tau\}.y & \text{if } \varphi_i{=}\mathsf{ff} \\ \{p_i, c_i, \underline{p_i}\}.(\!|\, \varphi_i \,|\!) & \text{otherwise} \end{cases}$$ ◀

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, $\mathsf{max}\, X.\varphi$, 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.*, $\mathsf{rec}\, y.m_i$, 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 $\varphi_i \neq \mathsf{ff}$, the synthesis produces an enforcer with the *identity transformation* prefix, $\{p_i, c_i, \underline{p_i}\}$, followed by the enforcer synthesised from the continuation φ_i, *i.e.*, $[\{p_i, c_i\}]\varphi_i$ is synthesised as $\{p_i, c_i, \underline{p_i}\}.(\!|\, \varphi_i \,|\!)$;

(*ii*) when m_i is derived from a branch of the form $[\{p_i, c_i\}]\mathsf{ff}$, the synthesis produces a *suppression transformation*, $\{p_i, c_i, \tau\}$, 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 $[\{p_i, c_i\}]\mathsf{ff}$ is translated into $\{p_i, c_i, \tau\}.y$.

▶ **Example 13.** Recall formula φ_1 from Example 1, recast in term of sHML$_{\mathbf{nf}}$'s grammar:

$$\varphi_1 \stackrel{\text{def}}{=} \mathsf{max}\, X.\bigwedge \big([\{(d)?\mathsf{req}, d{\neq}j\}] \big([\{d!\mathsf{ans}, \mathsf{true}\}]X \wedge [\{d?\mathsf{req}, \mathsf{true}\}]\mathsf{ff} \big) \big)$$

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

$$(\!|\, \varphi_1 \,|\!) = \mathsf{rec}\, x.\mathsf{rec}\, z.\sum \big(\{(d)?\mathsf{req}, d{\neq}j\}.\mathsf{rec}\, y.(\{d!\mathsf{ans}, \mathsf{true}\}.x \,+\, \{d?\mathsf{req}, \mathsf{true}, \tau\}.y) \big)$$

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

$$= \mathsf{rec}\, x.\{(d)?\mathsf{req}, d{\neq}j\}.\mathsf{rec}\, y.(\{d!\mathsf{ans}, \mathsf{true}\}.x \,+\, \{d?\mathsf{req}, \mathsf{true}, \tau\}.y) \;=\; m_{\mathbf{t}} \qquad \blacktriangleleft$$

We now present the first main result to the paper.

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

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

▶ **Proposition 15** (Enforcement Soundness)**.** *For every system* $s \in$ Sys *and* $\varphi \in$ sHML$_{\boldsymbol{nf}}$ *then* $\varphi \in$ Sat implies $(\!|\, \varphi \,|\!)[s] \in [\![\varphi]\!]$. ◀

▶ **Proposition 16** (Enforcement Transparency)**.** *For every system* $s \in$ Sys *and* $\varphi \in$ sHML$_{\boldsymbol{nf}}$ *then* $s \in [\![\varphi]\!]$ implies $(\!|\, \varphi \,|\!)[s] \sim s$. ◀

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

Our transformation $\langle\!\langle \varphi \rangle\!\rangle$ relies on a number of steps; here we provide an outline of these steps. First, we assume 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 $\{d!\mathsf{ans}, d \neq j\}$. 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 $\{(e)!(f), d{\neq}j \wedge e{=}d \wedge f{=}\mathsf{ans}\}$. ◀

Our algorithm for converting sHML formulas (with normalised patterns) to $\text{sHML}_{\mathbf{nf}}$ formulas, $\langle\!\langle - \rangle\!\rangle$, 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 $\mathsf{max}\, X.\big([\{p_1, c_1\}]X \wedge [\{p_2, c_2\}]\mathsf{ff}\big)$ is expanded to the formula $[\{p_1, c_1\}]\big(\mathsf{max}\, X.[\{p_1, c_1\}]X \wedge [\{p_2, c_2\}]\mathsf{ff}\big) \wedge [\{p_2, c_2\}]\mathsf{ff}$.
2. The formula is converted into a system of equations. *E.g.,* the expanded formula from the previous stage is converted into the set $\{X_0 = [\{p_1, c_1\}]X_0 \wedge [\{p_2, c_2\}]X_1, X_1 = \mathsf{ff}\}$.
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 $X_0 = [\{p_1, c_1\}]X_0 \wedge [\{p_2, c_2\}]X_1$ 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 $X_0 = [\{p_1, c_1\}]X_0 \wedge [\{p_1, c_3\}]X_1$ from the previous stage. We expand this to $X_0 = [\{p_1, c_1 \wedge c_3\}]X_0 \wedge [\{p_1, c_1 \wedge c_3\}]X_1 \wedge [\{p_1, c_1 \wedge \neg(c_3)\}]X_0 \wedge [\{p_1, \neg(c_1) \wedge c_3\}]X_1$.
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 $X_{\{0\}} = [\{p_1, c_1 \wedge c_3\}]X_{\{0,1\}} \wedge [\{p_1, c_1 \wedge \neg(c_3)\}]X_{\{0\}} \wedge [\{p_1, \neg(c_1) \wedge c_3\}]X_{\{1\}}$ 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.,* $\mathsf{max}\, X.\varphi$ where $X \notin \mathbf{fv}(\varphi)$, 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* $\psi{\in}\text{sHML}_{\boldsymbol{nf}}$ *s.t.* $[\![\varphi]\!]{=}[\![\psi]\!]$.

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

6 Alternative Transparency Enforcement

Transparency for a property φ, Definition 6, only restricts enforcers from modifying the behaviour of satisfying systems, *i.e.,* when $s{\in}[\![\varphi]\!]$, but fails to specify any enforcement behaviour for the cases when the SuS violates the property $s{\notin}[\![\varphi]\!]$. In this section, we consider an alternative transparency requirement for a property φ 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.

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

1. $(s,t) \in [\![\varphi]\!]_v$ implies $s \notin [\![\varphi]\!]$ and $s \stackrel{t}{\Longrightarrow}$,
2. $s \notin [\![\varphi]\!]$ implies $\exists t \cdot (s,t) \in [\![\varphi]\!]_v$. ◀

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

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

$$
\begin{array}{rl}
(s, \epsilon, \mathsf{ff}) \in \mathcal{R} & \text{always} \\
(s, t, \bigwedge_{i \in I} \varphi_i) \in \mathcal{R} & \text{if } \exists j \in I \text{ such that } (s, t, \varphi_j) \in \mathcal{R} \\
(s, \alpha t, [\{p, c\}]\varphi) \in \mathcal{R} & \text{if } \mathsf{mtch}(p, \alpha) = \sigma, c\sigma \Downarrow \mathsf{true} \text{ and } s \stackrel{\alpha}{\Longrightarrow} s' \text{ and } (s', t, \varphi\sigma) \in \mathcal{R} \\
(s, t, \mathsf{max}\, X.\varphi) \in \mathcal{R} & \text{if } (s, t, \varphi\{\mathsf{max}\, X.\varphi / X\}) \in \mathcal{R} \ .
\end{array}
$$

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

▶ **Example 21.** Recall $\varphi_1, s_{\mathbf{b}}$ from Example 1 where $\varphi_1 \in \text{sHML}$, and also $m_{\mathbf{t}}$ from Example 5 where we argued in Example 13 that $(\!|\, \varphi_1 \,|\!) = m_{\mathbf{t}}$ (modulo cosmetic optimisations). Even though $s_{\mathbf{b}} \notin [\![\varphi_1]\!]$, not all of its exhibited behaviours constitute violating traces: for instance, $s_{\mathbf{b}} \xRightarrow{i?\mathsf{req} \cdot i!\mathsf{ans}} s_{\mathbf{b}}$ is not a violating trace according to Definition 20. Correspondingly, we also have $m_{\mathbf{t}}[s_{\mathbf{b}}] \xRightarrow{i?\mathsf{req} \cdot i!\mathsf{ans}} m_{\mathbf{t}}[s_{\mathbf{b}}]$. ◀

▶ **Theorem 22** (Adapted and extended from [26]). *The alternative interpretation $[\![-]\!]_v$ of Definition 20 is a violating-trace semantics for* sHML *(with $[\![-]\!]$ 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 φ, denoted as $\mathsf{nvtenf}(m, \varphi)$, iff for *all* $s \in \textsc{Sys}$ and $t \in \textsc{Act}^*$, when $s, t \nvdash_v \varphi$ then

- $s \stackrel{t}{\Longrightarrow} s'$ *implies* $m[s] \stackrel{t}{\Longrightarrow} m'[s']$ for some m', and
- $m[s] \stackrel{t}{\Longrightarrow} m'[s']$ *implies* $s \stackrel{t}{\Longrightarrow} s'$. ◀

▶ **Proposition 24** (Non-Violating Trace Transparency). *For all $\varphi \in \text{sHML}$, $s \in \textsc{Sys}$ and $t \in \textsc{Act}^*$, when $s, t \nvdash_v \varphi$ then*

- $s \stackrel{t}{\Longrightarrow} s'$ implies $(\!|\, \varphi \,|\!)[s] \stackrel{t}{\Longrightarrow} m'[s']$, *and*
- $(\!|\, \varphi \,|\!)[s] \stackrel{t}{\Longrightarrow} m'[s']$ implies $s \stackrel{t}{\Longrightarrow} s'$. ◀

We can thus obtain a new definition for "*m enforces* φ" 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 $\text{sHML}_{\mathbf{nf}}$), 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 μHML. 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

1 Luca Aceto, Antonis Achilleos, Adrian Francalanza, and Anna Ingólfsdóttir. A framework for parameterized monitorability. In *Foundations of Software Science and Computation Structures*, pages 203–220, Cham, 2018. Springer International Publishing.

2 Luca Aceto, Antonis Achilleos, Adrian Francalanza, and Anna Ingólfsdóttir. Monitoring for silent actions. In Satya Lokam and R. Ramanujam, editors, *FSTTCS 2017: Foundations of Software Technology and Theoretical Computer Science*, volume 93 of *LIPIcs*, pages 7:1–7:14, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

3 Luca Aceto, Anna Ingólfsdóttir, Kim Guldstrand Larsen, and Jiri Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, New York, NY, USA, 2007.

4 Rajeev Alur and Pavol Černý. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 599–610. ACM, 2011.

5 Henrik Reif Andersen. Partial model checking. In *Proceedings of Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 398–407. IEEE, 1995.

6 Cyrille Artho, Howard Barringer, Allen Goldberg, Klaus Havelund, Sarfraz Khurshid, Michael R. Lowry, Corina S. Pasareanu, Grigore Rosu, Koushik Sen, Willem Visser, and Richard Washington. Combining test case generation and runtime verification. *Theoretical Computer Science*, 336(2-3):209–234, 2005.

7 Duncan Paul Attard, Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingolfsdottir. *A Runtime Monitoring Tool for Actor-Based Systems.*, chapter 3, pages 49–74. River Publishers, 2017.

8 Duncan Paul Attard and Adrian Francalanza. A monitoring tool for a branching-time logic. In *Runtime Verification*, pages 473–481, Cham, 2016. Springer International Publishing.

9 Jean Berstel and Luc Boasson. Transductions and context-free languages. *Ed. Teubner*, pages 1–278, 1979.

10 Nataliia Bielova. *A theory of constructive and predictable runtime enforcement mechanisms*. PhD thesis, University of Trento, 2011.

11 Nataliia Bielova and Fabio Massacci. Predictability of enforcement. In *International Symposium on Engineering Secure Software and Systems*, pages 73–86. Springer, 2011.

12 Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. Monitoring networks through multiparty session types. *Theoretical Computer Science*, 669:33 – 58, 2017.

13 Ian Cassar and Adrian Francalanza. On implementing a monitor-oriented programming framework for actor systems. In *International Conference on Integrated Formal Methods*, pages 176–192. Springer, 2016.

14 Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Jorge A. Pérez. Self-adaptation and secure information flow in multiparty communications. *Formal Aspects of Computing*, 28(4):669–696, July 2016.

15 Edward Chang, Zohar Manna, and Amir Pnueli. The safety-progress classification. In *Logic and Algebra of Specification*, pages 143–202. Springer, 1993.

16 Clare Cini and Adrian Francalanza. An LTL proof system for runtime verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 581–595. Springer, 2015.

17 Edmund M Clarke and E Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *25 Years of Model Checking*, pages 196–215. Springer, 2008.

18 Ankush Desai, Tommaso Dreossi, and Sanjit A. Seshia. Combining model checking and runtime verification for safe robotics. In *Runtime Verfication (RV)*, LNCS, pages 172–189, Cham, 2017. Springer International Publishing.

19 Yliès Falcone. You should better enforce than verify. In *Runtime Verification*, pages 89–105. Springer Berlin Heidelberg, 2010.

20 Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. What can you verify and enforce at runtime? *International Journal on Software Tools for Technology Transfer*, 14(3):349, June 2012.

21 Yliès Falcone, Laurent Mounier, Jean-Claude Fernandez, and Jean-Luc Richier. Runtime enforcement monitors: composition, synthesis, and enforcement abilities. *Formal Methods in System Design*, 38(3):223–262, June 2011.

22 Adrian Francalanza. A Theory of Monitors. In *International Conference on Foundations of Software Science and Computation Structures*, pages 145–161. Springer, 2016.

23 Adrian Francalanza. Consistently-Detecting Monitors. In *28th International Conference on Concurrency Theory (CONCUR 2017)*, volume 85 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:19, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

24 Adrian Francalanza, Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Ian Cassar, Dario Della Monica, and Anna Ingólfsdóttir. A foundation for runtime monitoring. In *Runtime Verification*, pages 8–29, Cham, 2017. Springer International Publishing.

25 Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. Monitorability for the Hennessy-Milner logic with recursion. *Formal Methods in System Design*, 51(1):87–116, 2017.

26 Adrian Francalanza and Aldrin Seychell. Synthesising correct concurrent runtime monitors. *Formal Methods in System Design*, 46(3):226–261, 2015.

27 Limin Jia, Hannah Gommerstadt, and Frank Pfenning. Monitors and blame assignment for higher-order session types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 582–594, New York, NY, USA, 2016. ACM.

28 Katarína Kejstová, Petr Ročkai, and Jiří Barnat. From Model Checking to Runtime Verification and Back. In *RV*. Springer, 2017.

29 Bettina Könighofer, Mohammed Alshiekh, Roderick Bloem, Laura Humphrey, Robert Könighofer, Ufuk Topcu, and Chao Wang. Shield synthesis. *Formal Methods in System Design*, 51(2):332–361, Nov 2017.

30 Dexter C. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

31 Frédéric Lang and Radu Mateescu. Partial model checking using networks of labelled transition systems and boolean equation systems. In Cormac Flanagan and Barbara König, editors, *TACAS*, pages 141–156, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

32 Kim G Larsen. Proof systems for satisfiability in Hennessy-Milner logic with recursion. *Theoretical Computer Science*, 72(2):265–288, 1990.

33 Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1):2–16, Feb 2005.

34 Jay Ligatti and Srikar Reddy. A theory of runtime enforcement, with results. In *CESORICS*, pages 87–100, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

35 Fabio Martinelli and Ilaria Matteucci. Partial model checking, process algebra operators and satisfiability procedures for (automatically) enforcing security properties. In *Foundations of Computer Security*, pages 133–144. Citeseer, 2005.

36 Fabio Martinelli and Ilaria Matteucci. Through modeling to synthesis of security automata. *Electronic Notes in Theoretical Computer Science*, 179:31–46, 2006.

37 Fabio Martinelli and Ilaria Matteucci. An approach for the specification, verification and synthesis of secure systems. *Electronic Notes in Theoretical Computer Science*, 168:29–43, 2007.

38 Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Information and computation*, 100(1):1–40, 1992.

39 A. Pnueli and A. Zaks. PSL model checking and run-time verification via testers. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *International Symposium on Formal Methods*, pages 573–586. Springer Berlin Heidelberg, 2006.

40 Z Manna A Pnueli. A hierarchy of temporal properties. *Proc. of the 2th symph. ACM of principle of distributed computer*, 1990.

41 Alexander Moshe Rabinovich. A complete axiomatisation for trace congruence of finite state behaviors. In *Proceedings of the 9th International Conference on Mathematical Foundations of Programming Semantics*, pages 530–543, London, UK, UK, 1994. Springer-Verlag.

42 Jacques Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, New York, NY, USA, 2009.

43 Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, New York, NY, USA, 2011.

44 Fred B Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):30–50, 2000.