Corrective Enforcement: A new Paradigm of Security Policy Enforcement by Monitors

RAPHAËL KHOURY, Université Laval
NADIA TAWBI, Université Laval

Runtime monitoring is an increasingly popular method to ensure the safe execution of untrusted codes. Monitors observe and transform the execution of these codes, responding when needed to correct or prevent a violation of a user-defined security policy. Prior research has shown that the set of properties monitors can enforce correlates with the latitude they are given to transform and alter the target execution. But for enforcement to be meaningful this capacity must be constrained, otherwise, the monitor can enforce any property, but not necessarily in a manner that is useful or desirable. However, such constraints are not significantly addressed in prior work. In this paper, we develop a new paradigm of security policy enforcement in which the behavior of the enforcement mechanism is restricted to ensure that valid aspects present in the execution are preserved notwithstanding any transformation it may perform. These restrictions capture the desired behavior of valid executions of the program, and are stated by way of a preorder over sequences. The resulting model is closer than previous ones to what would be expected of a real-life monitor, from which we demand a minimal footprint on both valid and invalid executions. We illustrate this framework with examples of real-life security properties. Since several different enforcement alternatives of the same property are made possible by the flexibility of this type of enforcement, our study also provides metrics that allow the user to compare monitors objectively and choose the best enforcement paradigm for a given situation.

Categories and Subject Descriptors: D.2.0 [**Software Engineering**]: General—*Protection Mechanisms*; D.2.4 [**Software Engineering**]: Network Protocols—*formal methods, Validation*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Monitors*; D.4.6 [**Operating Systems**]: Security and Protection—*Verification*; D.2.1 [**Software Engineering**]: Requirements/Specifications; F.1.1 [**Theory of Computation**]: Models of Computation—*Automata (eg, finite, push-down, resource-bounded)*

General Terms: Security

Additional Key Words and Phrases: Monitoring, Runtime Monitors, Dynamic Analysis, Security Policies Enforcement, Program Transformation

ACM Reference Format:
Khoury, R. and Tawbi, N., 2011. Corrective Enforcement of Security Policies Using Monitors. ACM Trans. Info. Syst. Sec. V, N, Article A (January YYYY), 39 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Security concerns become ever more present in our daily interactions with complex systems, and various tools have been suggested to address these concerns. A particularly efficient and flexible solution to the problem of ensuring the safety of untrusted code is runtime monitors. They allow untrusted code to run safely by injecting security guards or performing other code transformations on an untrusted source code or binary. In essence, the monitor replaces the execution it observes by an alternate execution that respects the desired security policy.

Prior research [Bauer et al. 2002; K. W. Hamlen and Schneider 2006] has shown that the range of policies enforceable in this manner is closely connected to the degree to which the monitor is al-

This work is supported by an NSERC Discovery grant.
Authors' addresses: R. Khoury and N. Tawbi, Department of Computer Science and Software Engineering, Laval University.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

lowed to transform its target's execution. Monitors given broad latitude to intervene in their target's execution naturally exhibit the widest range of enforceable properties. But for such an enforcement to be meaningful, the transformations performed on the target program must necessarily be constrained. Otherwise, the enforcement of the security policy might come at the cost of altering the target to the point where it is no longer useful. For instance Ligatti et al. [Ligatti et al. 2005] argue that any property is enforceable if the monitor considers all valid executions to be equivalent.

To address this issue, previous studies have introduced the notion of *transparency*. An enforcement mechanism is said to be transparent if the semantics of valid executions are preserved. As defined in [Ligatti et al. 2005], a monitor is only allowed to alter a valid execution if the resulting edited execution (the monitor's output) is equivalent to the original sequence with respect to some equivalence relation $\cong$.

In our opinion, two difficulties could arise with this approach. Firstly, it is often difficult to identify a suitable equivalence relation. Indeed, while much work has been done on the topic of effective$_{\cong}$ enforcement, the only equivalence relation $\cong$ for which the set of effectively$_{\cong}$ enforceable properties has been studied is syntactic equality. Secondly, in those cases where several possible alternative transformations could be used to enforce the property, this definition cannot guide the monitor to choose the most suitable transformation, nor does it give the user any tool to compare two alternative monitors enforcing the same property. In this regard, a third difficulty was raised by Bielova et al. in [2009]: this notion of transparency imposes no restriction on the behavior of the monitor when the observed execution does not respect the desired property. Once the monitor detects that the target's execution irremediably violates the desired security policy, it can, in principle, replace it with any valid sequence — even one completely unrelated to the original execution. Obviously, this paradigm does not adequately model the desired behavior of a real-life monitor, which is expected to perform only minimal alterations to bring an invalid execution in line with the security policy. In particular, valid behaviors present in an otherwise invalid sequence should be preserved, while invalid behaviors should be corrected or deleted. As Bielova et al. in [2009] put it concisely, "*What distinguishes an enforcement mechanism is not what happens when traces are good, because nothing should happen! The interesting part is how precisely bad traces are converted into good ones.*"

This study contributes to the literature in that it provides a new framework to model and assess the corrective capabilities of monitors. Our key insight is to organize executions into preorders, rather than equivalence classes. These preorders capture the desired behavior of a valid program execution, and impose that an execution be replaced only by a sequence that is equal or higher on the preorder. We found that preorders are more convenient than equivalence relations to model the restrictions we seek to impose on the monitor. To probe this issue further, we present four examples of real-life properties that highlight the operation of the new framework. Moreover, since several different enforcement alternatives of the same property are made possible by the flexibility of corrective enforcement, the study also provides metrics that allow the user to compare monitors objectively and choose the best enforcement paradigm for a given situation. Using preorders allows the user to impose natural constraints on the monitors' behavior, thus ensuring that the semantics of the original sequence are always preserved, in so far as doing so does not violate the security policy. By using preorders, we propose a model of monitors' enforcement that better approximates than previous work that which would be expected of a real-life monitor, from which we demand a minimal footprint on both valid and invalid executions.

This paper extends our earlier workshop paper "Corrective Enforcement of Security Policies" [Khoury and Tawbi 2010a]. The major extensions include:

— major edits throughout the paper to better motivate and explain our ideas,
— four metrics for objectively comparing several monitors enforcing the same security property. We prove several theorems related to the use of these metrics with our proposed automata,
— a much more thorough treatment of related works,
— proofs for all the theorems that had been omitted from the conference paper for lack of space,

— the addition of two automata $\mathcal{A}^t_{ap}$ and $\mathcal{A}^t_{cw}$ in sections 5.2 and 5.3 respectively. These automata better position our work in relation to that of previous authors.

The remainder of this paper is organized as follows: the next section provides an overview of related work, while Section 3 presents definitions of concepts and notations that are used throughout the paper. In Section 4, we show how preorders can form the basis of a corrective monitoring framework and motivate the use of such a framework by comparing it to other enforcement paradigms. Section 5 provides four examples of security properties that can be enforced in this manner and metrics that can be used to compare alternative enforcements of the same security policy. Concluding remarks and avenues for future work are presented in the final section.

2. RELATED WORK

This paper extends the body of research that studies the notion of security policy enforcement by monitors and seeks to identify the set of properties enforceable by monitors under various constraints. Prior research on this topic has focused on three questions, namely delineating the set of properties enforceable by monitors operating under various constraints, identifying how this set is constrained by computability and memory constraints, and exploring alternative definitions of enforcement.

2.1. Delineating the Set of Enforceable Properties

The question of which properties are enforceable by monitors is understandably central to any formal study of this enforcement mechanism. This in turn requires than we define what behavior is expected of a monitor that is said to enforce a property.

The first headways in answering these questions were given by Schneider in [Schneider 2000]. He argues that to enforce a property, a monitor should respect the following two principles:

(1) $Soundness$: The output must respect the desired property.
(2) $Transparency$: The semantics of valid executions must be preserved.

As a baseline to study the power of monitors, Schneider examines monitors that watch over the execution of a target program with no information about its possible future behavior and with no ability to affect its execution except to abort it. He also focuses on a very restrictive notion of transparency: every action of a valid sequence must be output by the monitor in lockstep with the output of the program. This is termed *precise enforcement* [Bauer et al. 2002]. In this context, he shows that the set of properties enforceable by monitors coincides with the set of *safety* properties.

Further research [Bauer et al. 2002; Ligatti et al. 2004] extend Schneider's definition of a monitor along three axes, namely (1) the different ways by which the monitor can respond to a possible violation of the security policy; (2) whether the monitor possesses information about the program's possible behavior; and (3) the degree to which the monitor is allowed to transform its target's execution. The authors find the set of enforceable properties cannot increase with the added power of the monitor to transform its target's execution (by inserting or suppressing program actions) if the monitor operates in the context of precise enforcement because of the rigidity of this enforcement paradigm.

However, if the monitor is allowed to alter or simply delay the execution of its target, a fairly wide set of policy becomes enforceable. The notion of *effective enforcement* [Bauer et al. 2002] characterizes this enforcement paradigm. A monitor effectively$_{\cong}$ enforces a property if any execution respecting the property is replaced by an equivalent execution with respect to some equivalence relation $\cong$. No restriction is imposed on the output of the monitor if its input is invalid. Even when operating under a fairly constraining equivalence relation, such as syntactic equality, a monitor can effectively enforce a wide range of security policies, termed (*renewal*) properties [Ligatti et al. 2005]. As defined by the authors, a property is a renewal property if every infinite valid sequence has infinitely many valid prefixes, while every invalid infinite sequence has only finitely many such prefixes. Every property over finite sequences is a renewal.

2.2. Imposing Memory and Computability Constraints

Schneider as well as Ligatti et al. argue that memory and computability constraints may impact the ability of a security mechanism to enforce a given property even though that property lies inside the set of properties this mechanism should be able to enforce. Several studies have thus examined exactly how such constraints affect the set of enforceable properties.

Fong was first to consider monitors operating with memory constraints [Fong 2004]. A key insight of his work is that the monitor can rely upon an abstraction of the input sequence rather than on the exact sequence itself. Subsequent work by Talhi et al. [Talhi et al. 2008] showed a close connection between the class of properties enforceable by monitors operating with a memory of bounded size and a class of languages termed locally testable languages (or local properties). The enforcement of security policies by such monitors is further studied in [Kupferman et al. 2006] while Beauquier et al. [Beauquier et al. 2009] consider the enforcement power of monitors with a finite but unbounded memory.

In [Kim et al. 2002], Kim et al. turn their attention to the limitations imposed on monitors by computational constraints. They observe that a monitor must be able to identify any invalid sequence upon the inspection of a finite prefix of this sequence. This restricts the set of enforceable properties to the class of co-recursively enumerable properties (coRE) [Viswanathan 2000]. However, in a subsequent paper [K. W. Hamlen and Schneider 2006], Hamlen and Schneider show that there are properties in coRE that are not monitorable. A better characterization of the set of properties enforceable by monitors is the intersection between the class coRE and the class of properties enforceable by program rewriters. This result agrees with an intuition that monitors can be inlined in the program they aim to protect, so that any property enforceable by monitors can also be enforced by program rewriting.

2.3. Alternative Notions of Enforcement

The foremost concern in the conception of any monitoring framework lies not only in the delineation of the set of enforceable properties, but also in the way the monitor reacts to a possible violation of the security policy.

As discussed above, the original definitions of precise and effective enforcement did not place any restrictions on the monitor's treatment of invalid sequence other than replacement with a valid output. The main advantage of the enforcement paradigm we propose is its capacity to state meaningful constraints on the permissible transformations that can be made on invalid sequences. We believe such constraints are an integral part of the enforcing security policies with real monitors and that their absence leads to a discrepancy between theoretical models and the practical monitors they represent.

In addition, since monitors operating in a precise or effective enforcement context are allowed to perform certain transformations on invalid sequences but not on valid sequences, the monitor's full capabilities cannot be used until the validity of the current input sequence is determined. By using the same restriction on the treatment of both valid and invalid sequences, our model frees the monitor to start altering the sequence from the onset of the execution. Furthermore, this approach simplifies mathematical reasoning about the monitor's behavior.

A particular subclass of effective$_{\cong}$ enforcement that has attracted much attention is effective$_{=}$ enforcement, in which the equivalence relation underlying effective enforcement is instantiated to syntactic equality. Once again, the monitor has no limitation in its choice of valid replacement for any invalid sequence. However, in most implementations, the monitor always returns the longest valid prefix of any invalid sequence. This restriction is the result of implementation choices, and not an integral part of the enforcement definition. Indeed, Ligatti's original paper [Ligatti et al. 2009] proposed monitors that effectively$_{=}$ enforced a property but output sequences that were not a prefix of the input.

In [Falcone et al. 2008], Falcone et al. stipulate that the monitor must always output the longest valid prefix of an invalid sequence into the definition of transparency. This can be seen as an in-

stantiation of effective$_{\cong}$ enforcement with an equivalence relation grouping together sequences that share the same longest prefix. While this solution makes it impossible for the monitor to replace an invalid sequence with some unrelated valid sequence, it also prevents the monitor from taking other corrective action. In particular, valid behaviors present in the input sequence after the occurrence of an irremediably invalid prefix can never be output.

More specific restrictions on the treatment of invalid sequences are stated explicitly in a definition proposed by Bielova and Massacci in [2011a]. They suggest *Late precise enforcement* to characterize the behavior of monitors limited to suppressing and reinserting some actions and consequently to always output a prefix of the input sequence. They also define and compare several subclasses to the edit-automaton (see definition 4.1), which model monitors that are correspondingly constrained. This includes monitors that always output a valid prefix of the input (modeled by the All-Or-Nothing automaton), and monitors that always output the longest valid prefix of the input sequence (modeled by the Late automaton for property $\hat{\mathcal{P}}$). While Bielova and Massacci impose explicit constraints on the monitor's treatment of invalid sequences, they do not explore the possibility of altering the invalid sequence beyond truncating it to a valid prefix, thus letting much of the monitor's capabilities go unused. The solution presented in this paper is more flexible in that it provides the monitor with multiple enforcement options beyond simply outputting a valid prefix of the input sequence. The authors then turn to the enforcement of properties, termed *iterative properties*, which describe the repeated execution of finite transactions. They propose a monitor capable of enforcing these properties while preserving all valid transactions, even if doing so requires that the output not be a prefix of the input. While this is a clear improvement over previous solutions, the model cannot be generalized beyond iterative properties. We revisit iterative properties and their enforcement in section 5.1 and show how the enforcement paradigm we propose can both emulate the enforcement proposed in [Bielova and Massacci 2011b] and be instantiated to other classes of properties.

Khoury and Tawbi [2010b] proposed a variant of effective enforcement: the monitor's output must always be equivalent to the input with respect to some equivalence relation $\cong$. This equivalence relation ensures that the monitor's output is related to the input via an approximation that preserves valid behaviors. The monitor is thus prevented from overly altering the execution to ensure compliance with the desired security policy. In section 5.1, we show that preorders are a more convenient way than equivalence relations to state the needed restrictions on the monitors' behavior. In particular, the use of preorders make it possible to suggest multiple valid replacements to the same invalid sequence and leave the final choice to the user.

An alternative solution presented by Bielova and Massacci [2011c] is to capture the constraints on the monitor's treatment of invalid sequences by a syntactic restriction termed *predictability*. In this regard, they limit the monitor's transformation of invalid sequences by limiting the number of permissible syntactic changes on the input sequence. By contrast our approach relates restrictions to the semantics of the property being enforced. The notion of predictability could then be expressed in our framework by a metric similar to those we will propose in Section 5.

In summary, the corrective enforcement model described above is a much more promising abstraction of real-life monitors than previous models because it allows us to model constraints in their allowed reactions to invalid inputs, which was not the case for previous models. By modeling such constraints on monitors' behavior, the corrective enforcement model we develop in this paper is a much more suitable foundation for any theoretical study of runtime enforcement and its capacities than previous models. Building monitors on the basis of a corrective rather than an effective enforcement framework will naturally restrict the set of enforceable properties available to them. However, this drawback is largely outweighed by the benefits of a more meaningful enforcement mechanism that is particularly suitable for practical applications.

2.4. Other studies in Monitoring

In our research, we have chosen to focus on the edit automaton-based representation of monitors (see definition in Section 4), and used this representation as a basis to illustrate our proposed enforcement

paradigm. While this representation is widely used in the literature, it is worth noting that alternative models have also been proposed. Jun et al. [2008] model monitors as Mealy Machines [Mealy 1955], Zhu et al. [2006] suggest modeling monitors as a *stream automata* while Ligatti et al. [2010] propose the *Mandatory results Automata* as an alternative. The last two models facilitate the interaction between the target program, the monitor and the system by virtue of their capacity to distinguish between the action set of the target and that of the system with which it interacts. While the focus of this research is the edit automaton, we believe that the ideas we propose could readily be applied to the alternative models of monitoring mentioned above.

Several algorithms have been proposed to construct a monitor from an automaton representing a security property. Ligatti et al. give a constructive proof that such a monitor can be constructed for all properties over finite sequences in [Bauer et al. 2002], and alternative algorithms are proposed in [Bielova et al. 2009; d'Amorim and Roşu 2005]. The monitor can then be inlined into an untrusted program using an algorithm given in [Ould-Slimane et al. 2009]. In [Falcone et al. 2008] Falcone et al. give algorithms to construct a monitor for four of the six classes of the safety-progress classification of properties.

3. PRELIMINARIES

Let us briefly start with some preliminary definitions.

Executions are modeled as sequences of atomic actions taken from a finite or countably infinite set of actions Σ. The empty sequence is noted ϵ, the set of all finite length sequences is noted Σ^*, that of all infinite length sequences is noted Σ^ω, and the set of all possible sequences is noted $\Sigma^\infty = \Sigma^\omega \cup \Sigma^*$. Likewise, for a set of sequences $\mathcal{S}$, $\mathcal{S}^*$ denotes the finite iterations of sequences of $\mathcal{S}$ and $\mathcal{S}^\omega$ that of infinite iterations, and $\mathcal{S}^\infty = \mathcal{S}^\omega \cup \mathcal{S}^*$. $\wp(S)$ denotes the powerset of S. Let $\tau \in \Sigma^*$ and $\sigma \in \Sigma^\infty$ be two sequences of actions. We write $\tau;\sigma$ for the concatenation of τ and σ. We say that τ is a prefix of σ noted $\tau \preceq \sigma$, or equivalently $\sigma \succeq \tau$ *iff* there exists a sequence σ' such that $\tau;\sigma' = \sigma$. We write $\tau \prec \sigma$ (resp. $\sigma \succ \tau$) for $\tau \preceq \sigma \wedge \tau \neq \sigma$ (resp. $\sigma \succeq \tau \wedge \tau \neq \sigma$). Finally, let $\tau, \sigma \in \Sigma^\infty$, τ is said to be a suffix of σ iff there exists a $\sigma' \in \Sigma^*$ such that $\sigma = \sigma';\tau$.

Following [Ligatti et al. 2009], if σ has already been quantified, we freely write $\forall \tau \preceq \sigma$ (resp. $\exists \tau \preceq \sigma$) as an abbreviation to $\forall \tau \in \Sigma^* : \tau \preceq \sigma$ (resp. $\exists \tau \in \Sigma^* : \tau \preceq \sigma$). Likewise, if τ has already been quantified, $\forall \sigma \in \Sigma^\infty : \sigma \succeq \tau$ (resp. $\exists \sigma \in \Sigma^\infty : \sigma \succeq \tau$) can be abbreviated as $\forall \sigma \succeq \tau$ (resp. $\exists \sigma \succeq \tau$).

We denote by $pref(\sigma)$ (resp. $suf(\sigma)$) the set of all prefixes (resp. suffixes) of σ. Let $A \subseteq \Sigma^\infty$ be a subset of sequences. Abusing the notation, we let $pref(A)$ (resp. $suf(A)$) stands for $\bigcup_{\sigma \in A} pref(\sigma)$ (resp. $\bigcup_{\sigma \in A} suf(\sigma)$). The i^{th} action in a sequence σ is given as σ_i, $\sigma[i,j]$ denotes the sequence occurring between the i^{th} and j^{th} actions of σ, $\sigma[i,..]$ denotes the remainder of the sequence starting from action σ_i while $\sigma[..,i]$ denotes the prefix of σ, up to action i. The length of a sequence $\tau \in \Sigma^*$ is given as $|\tau|$. Let τ be a sequence and a be an action, we write $\tau \backslash a$ for the left cancelation of a from τ, which is defined as the removal from τ of the first occurrence of a. Formally:

$$a;\tau\backslash a' = \begin{cases} \tau & \text{if } a = a'; \\ a;(\tau\backslash a') & \text{otherwise.} \end{cases}$$

Observe that $\epsilon \backslash a = \epsilon$. Abusing the notation, we write $\tau \backslash \tau'$ to denote the sequence obtained by left cancelation of each action of τ' from τ. Formally, $\tau\backslash a;\tau' = (\tau\backslash a)\backslash\tau'$. For example, $a;b;c;a;d;a\backslash d;a;a = b;c;a$.

A finite word $\tau \in \Sigma^*$ is said to be a subword of a word ω iff $\tau = a_0a_1a_2a_3...a_k$ and $\omega = \Sigma^*a_0\Sigma^*a_1\Sigma^*a_2\Sigma^*a_3...\Sigma^*a_k\Sigma^\omega$. Let τ, σ be sequences form Σ^*. We write $cs_\tau(\sigma)$ to denote the longest subword of τ which is also a subword of σ.

A word $\tau \in \Sigma^*$ is a factor of a word $\omega \in \Sigma^\infty$ if there exists two integers i, j such that $0 \leq i \leq j$ and $\tau = \omega[i..j]$. In other words, τ is a factor of ω if $\omega = \upsilon;\tau;\upsilon'$, with $\upsilon \in \Sigma^*$ and $\upsilon' \in \Sigma^\infty$. We say $fact(\sigma)$ for the set of factors present in σ.

A multiset, or bag [Syropoulos 2001] is a generalization of a set in which each element may occur multiple times. A multiset $\mathcal{M}$ can be formally defined as a pair $\langle A, f\rangle$ where A is a set and

$f : A \to \mathbb{N}$ is a function indicating the number of occurrences of each element of A in $\mathcal{M}$. Note that $a \notin A \Leftrightarrow f(a) = 0$. By using this insight to define basic operations on multisets one can consider a universal set A and different functions of type $A \to \mathbb{N}$ associated with it to form different multisets.

Given two multisets $\mathcal{M} = \langle A, f\rangle$ and $\mathcal{M}' = \langle A, g\rangle$, the multiset union $\mathcal{M} \uplus \mathcal{M}' = \langle A, h\rangle$ where $\forall a \in A : h(a) = f(a) + g(a)$. Furthermore, $\mathcal{M} \subseteq \mathcal{M}' \Leftrightarrow \forall a \in A : f(a) \leq g(a)$. The removal of an element $a \in A$ from multiset $\mathcal{M}$ is done by updating function f so that $f(a) = max(f(a)-1, 0)$. We let $acts(\tau)$ represent the multiset of actions occurring in sequence τ.

Finally, a security policy $P \subseteq \wp\{\Sigma^\infty\}$ is a set of sets of allowed executions. A policy P is a property iff it can be characterized as a set of sequences for which there exists a decidable predicate $\hat{\mathcal{P}}$ over the executions of Σ^∞ : $\hat{\mathcal{P}}(\sigma)$ iff σ is in the policy [Schneider 2000]. In other words, a property is a policy for which the membership of any sequence can be determined by examining only the sequence itself. Such a sequence is said to be valid or to respect the property. Since all policies enforceable by monitors are properties, P and $\hat{\mathcal{P}}$ are used interchangeably. Additionally, since the properties of interest represent subsets of Σ^∞, we follow the common usage in the literature and freely use $\hat{\mathcal{P}}$ to refer to these sets.

4. MONITORING WITH PREORDERS

In this section, we introduce the automata-based model of a monitor and the notions of effective$_{\cong}$ and corrective$_{\cong}$ enforcement used in the literature. We point out why these definitions are sometimes inadequate before presenting an alternative enforcement paradigm.

4.1. Enforcement Framework

The edit automaton [Bauer et al. 2002; Ligatti et al. 2005], is the most widely used model of a monitor. It captures the behavior of a monitor capable of inserting or suppressing any action in the execution in progress, as well as halting it.

Definition 4.1. An edit automaton is a tuple $\langle \Sigma, Q, q_0, \delta\rangle$ where[1]:

— Σ is a finite or countably infinite set of actions;
— Q is a finite or countably infinite set of states;
— $q_0 \in Q$ is the initial state;
— $\delta : \langle Q \times \Sigma\rangle \to \langle Q \times \Sigma^\infty\rangle$ is the transition function which, given the current state and input action, specifies the automaton's output and successor state. At any step, the automaton may accept the action and output it as it is, suppress it and move on to the next action (having not output anything), or output some other sequence in Σ^∞. If at a given state, the transition for a given action is undefined, the automaton aborts. This corresponds to halting the execution of the monitor's target. We write $q \xrightarrow[\tau]{a} q'$ to indicate that if the action a is input while the monitor is in state q, it will output τ and reach state q'.

The automaton is said to accept a sequence σ if it can follow a valid path upon being given σ as input. A valid path is a finite or infinite sequence of transitions of the form $q_0 \xrightarrow[\tau_1]{\sigma_1} q_1 \xrightarrow[\tau_2]{\sigma_2} q_2 ... \xrightarrow[\tau_n]{\sigma_n} q_n ...$ where $\forall i \in \mathbb{N} : 1 < i < |\sigma| \Rightarrow \delta\langle q_{i-1}, \sigma_i\rangle = \langle q_i, \tau_i\rangle$. The sequence $\tau_1; \tau_2; ...$ is thus the output of the automaton when the input is σ. Let $\mathcal{A}$ be an edit automaton and let $\mathcal{A}(\sigma)$ be the output of $\mathcal{A}$ when its input is σ.

Most related studies have focused on effective enforcement [Bauer et al. 2002]. A mechanism effectively enforces a security property iff all of its outputs respect this property and the semantics of executions which already respect the property must be preserved. The latter requirement is imposed by the use of an equivalence relation to determine when one sequence can be substituted for another.

[1]This definition, taken from [Talhi et al. 2008], is equivalent to the one given in [Bauer et al. 2002].

Definition 4.2. **From [Bauer et al. 2002]** Let $\mathcal{A}$ be an edit automaton. $\mathcal{A}$ effectively$_{\cong}$ enforces the property $\hat{\mathcal{P}}$ iff $\forall \sigma \in \Sigma^{\infty}$:

(1) $\hat{\mathcal{P}}(\mathcal{A}(\sigma))$ (i.e., $\mathcal{A}(\sigma)$ is valid)
(2) $\hat{\mathcal{P}}(\sigma) \Rightarrow \mathcal{A}(\sigma) \cong \sigma$

In the literature, the only equivalence relation $\cong$ for which the set of effectively$_{\cong}$ enforceable properties has been formally studied is syntactic equality [Ligatti et al. 2005]. Yet effective enforcement is not the only enforcement paradigm that has been suggested. Other enforcement paradigms include precise enforcement [Bauer et al. 2002], all-or-nothing delayed enforcement [Bielova et al. 2009], conservative enforcement [Bauer et al. 2002] and corrective$_{\cong}$ enforcement [Khoury and Tawbi 2010b].

Most previous work in monitoring has focused on effective$_{\cong}$ enforcement. This definition allows the monitor to replace an invalid execution with any valid sequence— even ϵ. A more intuitive model of a monitor's desired behavior would instead require that only minimal alterations be made to an invalid sequence, for instance by releasing a resource or adding an entry in a log. Valid parts of the input sequence should be preserved in the output, while invalid behaviors should be corrected or removed. A possible solution proposed by Khoury and Tawbi in [2010b], is termed corrective$_{\cong}$ enforcement. Informally, an enforcement mechanism correctively$_{\cong}$ enforces the desired property if every output sequence is both valid and equivalent to the input sequence. The equivalence relation used is built upon an abstraction function applied to sequences and is chosen in such a way that the valid behavior of the input sequence is preserved, while invalid behaviors are deleted or transformed.

Definition 4.3. **From [Khoury and Tawbi 2010b]** Let $\mathcal{A}$ be an edit automaton. $\mathcal{A}$ correctively$_{\cong}$ enforces the property $\hat{\mathcal{P}}$ iff $\forall \sigma \in \Sigma^{\infty}$:

(1) $\hat{\mathcal{P}}(\mathcal{A}(\sigma))$
(2) $\mathcal{A}(\sigma) \cong \sigma$

However, this definition also raises some difficulties. In particular, it implies that several distinct valid sequences, which are possible transformations of an invalid sequence, must be equivalent. Likewise, it requires that several invalid sequences be considered equivalent if a single valid sequence is a valid alternative to both.

In this paper, we examine an alternative notion of enforcement, termed corrective$_{\sqsubseteq}$ enforcement. Following previous work in monitoring by Fong [Fong 2004], we use an abstraction function $\mathcal{F} : \Sigma^{*} \rightarrow \mathcal{I}$ to capture the property of the input sequence that the monitor must preserve throughout its manipulation. While Fong made use of abstractions to reduce the memory overhead of the monitor, we use them as the basis for determining which transformations the monitor is or is not allowed to perform on both valid and invalid sequences. The set $\mathcal{I}$ can be any abstraction of the program's behavior. This may be, for example, the number of occurrences of certain subwords or factors or any other semantic property of interest. The main idea is to use this abstraction to capture the semantic property of the execution corresponding to the valid or desired behavior of the target program.

We wish to constrain the behavior of the monitor so that an invalid sequence is corrected in such a way that the monitor preserves all valid behaviors present in it. An intuitive solution to this problem would be to impose that the output sequence always have the same value of $\mathcal{F}$ as the input sequence. The abstraction function would thus form the basis of a partition of the sequences of Σ^{∞} into equivalence classes, and the monitor can then be forbidden to output a sequence that lies in a different equivalence class than the input sequence. But, as discussed above, this approach limits the monitor's ability to use the same valid sequences as a potential solution to several unrelated invalid sequences. Instead, we define a preorder $\leq$ over the abstract values of $\mathcal{I}$ and define a corresponding preorder $\sqsubseteq$ over sequences of Σ^{*} such that $\forall \sigma, \sigma' : \sigma \sqsubseteq \sigma' \Leftrightarrow \mathcal{F}(\sigma) \leq \mathcal{F}(\sigma')$. The monitor is allowed to transform an input σ into another sequence σ' iff $\sigma \sqsubseteq \sigma'$. By defining $\sqsubseteq$ appropriately, we can ensure that sequences that are greater on the preorder are always adequate replacements for

any inferior sequences, in the sense that they preserve the valid behaviors present in those sequences. We also find that preorders are a more natural way to state most security policies than equivalence relations.

The choice of the abstraction function, as well as that of how abstractions are organized into a preorder, are central to the transparency of the enforcement paradigm we propose. It is in the context of these choices that a solution is seen as a suitable replacement for another. In Section 4.2, we argue that no general method for choosing the most adequate abstraction and preorder for a given security property can be defined. However, as we illustrate with the examples presented in the next section, we believe that for any given property, a natural abstraction and preorder can be defined in a very intuitive manner. Formally:

Definition 4.4. Let $\mathcal{A}$ be an edit automaton, and let $\sqsubseteq$ be a preorder over the sequences of Σ^∞. $\mathcal{A}$ correctively$_\sqsubseteq$ enforces the property $\hat{\mathcal{P}}$ iff $\forall \sigma \in \Sigma^\infty$:

(1) $\hat{\mathcal{P}}(\mathcal{A}(\sigma))$
(2) $\sigma \sqsubseteq \mathcal{A}(\sigma)$

Let τ, τ' be two sequences such that $\tau \sqsubseteq \tau'$. In such a case, we write that τ is lower than τ' or conversely, that τ' is higher than τ.

A monitor often operates knowing that certain executions cannot occur. Prior research [Bauer et al. 2002; Chabot et al. 2009] has shown that a monitor operating in such a context (called a *nonuniform* context) can enforce a larger set of properties than one that considers every sequence in Σ^∞ to be a possible input (called a *uniform* context). To account for the possibility that the monitor might operate in a nonuniform context, we adapt the preceding definition as follows:

Definition 4.5. Let $\mathcal{S} \subseteq \Sigma^\infty$ be a subset of sequences, let $\sqsubseteq$ be a preorder over the sequences of Σ^∞ and let $\mathcal{A}$ be an edit automaton. $\mathcal{A}$ correctively$^{\mathcal{S}}_{\sqsubseteq}$ enforces the property $\hat{\mathcal{P}}$ iff $\forall \sigma \in \mathcal{S}$:

(1) $\hat{\mathcal{P}}(\mathcal{A}(\sigma))$
(2) $\sigma \sqsubseteq \mathcal{A}(\sigma)$

We write corrective$_\sqsubseteq$ enforcement when $\mathcal{S} = \Sigma^\infty$ or $\mathcal{S}$ is obvious from the context.

The preorder is initially defined over the sequences of Σ^* and extended to infinite sequences in the following manner. Let σ, σ' be infinite sequences, we have $\sigma \sqsubseteq \sigma'$ iff σ and σ' have infinitely many common prefixes τ, τ' such that $\tau \sqsubseteq \tau'$ with $\tau \prec \sigma$ and $\tau' \prec \sigma'$.

$$\forall \sigma, \sigma' \in \Sigma^\omega : \sigma \sqsubseteq \sigma' \Leftrightarrow \forall \tau \prec \sigma : \exists \upsilon \succeq \tau : \exists \tau' \prec \sigma' : \upsilon \sqsubseteq \tau' \tag{1}$$

Finally, since the monitor operates by transforming sequences, we must impose that every preorder respects the following closure restriction:

$$\forall \tau, \tau' \in \Sigma^* : \forall \sigma \in \Sigma^\infty : \tau \sqsubseteq \tau' \Rightarrow \tau;\sigma \sqsubseteq \tau';\sigma \tag{2}$$

To understand the need for this restriction, consider the possible behavior of a monitor that is presented with an invalid prefix τ of a longer input sequence. It may opt to transform τ into a valid higher sequence τ'. However, if the closure restriction given in equation 2 is not respected by the preorder, it is possible that the full input sequence $\tau;\sigma$ is actually valid, but there is no valid extension of τ' greater than $\tau;\sigma$. The monitor would have inadvertently ruined a valid sequence.

This study also makes headway on the complex issue of choosing the most effective monitor by devising metrics computed on automata. This paves the way for the next step: developing guidelines as to which automata to adopt, a topic that lies outside the scope of this paper.

4.2. Preorders vs Security Properties

Before moving on, let us briefly examine the interconnectedness between preorders and properties in the context of our framework. It is clear from the previous discussion that preorders formalize a particular aspect of an enforcement mechanism's specification, namely its desired behavior with an

invalid input. This stands in stark contrast with the security property, which captures information of a completely different nature: it divides the space of executions into valid and invalid executions. Since the information expressed in the preorder is completely absent in the security property, the former cannot be extracted from the latter through an algorithmic process.

In this regard, it is also important to stress that, for a given property, there may exist an unlimited number of different and even conflicting preorders, each one associated to specific constraints on the monitor's treatment of invalid sequences. To illustrate this, consider the example of a monitor in charge of scheduling the allocation of a scarce resource between processes and of enforcing the starvation freedom property. In some situations, it may be desirable that the monitor enforce this property while simultaneously satisfying one of any number of possible constraints such as minimizing the number of times the ownership of the resource is transferred from one process to another, minimizing the average delay before each process acquires the resource, allocating the resource as evenly as possible between the processes, maximizing access to the resource for processes that have been assigned a higher priority, or any other of the many possible scheduling constraints. The great variety of situation-specific conditions under which the monitor may operate makes it impossible to devise a general algorithmic process to instantiate a preorder for a given property.

Other meaningful constraints could be considered. From a practical perspective, there are as many preorders as there are users, and the requirements of each user may evolve over time. Thus, for any property there exists a multitude of different preorders, each tailored to a specific set of user-defined constraints on the monitor's behavior. Any general algorithmic process to derive preorders would therefore unduly constrain the user to a level incompatible with the fundamental premise of the model we develop. In light of this, preorders are considered exogenous in our model, a position that mirrors that of other researchers who studied the analogous issue of the selecting the equivalence relation in an effective enforcement context.

5. EXAMPLES

In this section, we illustrate the use of corrective$_{\sqsubseteq}$ enforcement using four real-life security properties: transactional properties, the assured pipeline property, the Chinese Wall property and general availability. When several possible monitors can enforce the same property, we show how these different enforcement paradigms can be compared.

5.1. Transactional Properties

The first class of properties we wish to correctively$_{\sqsubseteq}$ enforce is that of transactional properties, suggested in [Ligatti et al. 2005]. Let Σ be an action set and let $\mathcal{T} \subseteq \Sigma^*$ be a set of finite transactions. $\hat{\mathcal{P}}_{\mathcal{T}}$ is a *transactional* property over set Σ^∞ iff

$$\forall \sigma \in \Sigma^\infty : \hat{\mathcal{P}}_{\mathcal{T}}(\sigma) \Leftrightarrow \sigma \in \mathcal{T}^\infty \qquad \text{(transactional)}$$

The set of transactional properties thus forms a subset to that of iterative properties defined in [Bielova et al. 2009]. Transactional properties also form a subset to the set of *renewal* properties, and include some but not all *safety* properties, *liveness* [Alpern and Schneider 1985] properties, and properties which are neither. Transaction properties are also a subclass of *response* property in the safety-progress classification established by A. Chang et al. [Chang et al. 1991].

Previous studies have suggested enforcing this property either in a context of effective$_{=}$ enforcement [Beauquier et al. 2009], or in that of iterative enforcement [Bielova and Massacci 2011b]. In the first case, valid transactions present in the sequence after the occurrence of an invalid transaction are deleted. In the second case, all valid transactions present in the input sequence are output, but the monitor is prevented from taking any corrective action other than deleting invalid transactions.

We rely upon the notion of factors to reason about transactions in a formalized manner and construct a preorder that captures the constraints described above.

We only consider transactional properties built from a set of sequences $\mathcal{T}$ which meets the following two restrictions. Taken together, the conjunction of these restrictions implies a desirable

property, which we term *unambiguity*:

$$\forall\sigma,\sigma' \in \mathcal{T} : \forall\tau \in \mathit{pref}(\sigma) : \forall\tau' \in \mathit{suf}(\sigma') : (\tau \neq \epsilon \wedge \tau' \neq \epsilon) \Rightarrow (\tau';\tau \notin \mathcal{T}) \quad \text{(unambiguity 1)}$$

$$\forall\sigma,\sigma' \in \mathcal{T} : \sigma \in \mathit{fact}(\sigma') \Rightarrow \sigma = \sigma' \quad \text{(unambiguity 2)}$$

Informally, these restrictions forbid the occurrence of transactions with overlapping prefixes and suffixes or overlapping factors. If these restrictions are not met, there exists sequences which cannot be parsed as the concatenation of valid transactions—and thus not in the property—but for which every action is a part of at least one valid transaction. On the other hand, if these restrictions are met, each occurrence of an action σ_i in a given sequence can be part of at most one transaction.

To enforce this property, we use an abstraction function that returns the multiset of factors occurring in a given sequence. We use a multiset rather than simply comparing the set of factors from $\mathcal{T}$ occurring in each sequence so as to be able to distinguish between sequences containing a different number of occurrences of the same subset of factors. For any two sequences σ and σ', we write $\sigma \sqsubseteq \sigma'$ iff σ' has more valid factors (w.r.t. $\mathcal{T}$) than σ, and σ' is thus an acceptable replacement sequence for σ, provided that σ' is valid and σ is not. This captures the intuition that if certain valid transactions are present in the input sequence, they must still be present in the output sequence regardless of any other transformation made to ensure compliance with the security property. The monitor may add valid transactions, for instance by inserting a few missing actions to correct an invalid transaction, and must remove invalid ones outright, but it may not remove any valid transactions present in the original execution.

Let $\mathit{valid}_{\mathcal{T}}(\sigma)$, which stands for the multiset of factors from the sequence σ which are present in $\mathcal{T}$, be the abstraction function $\mathcal{F}$. The preorder $\sqsubseteq$ used to correctively enforce this property in the manner described above is thus given as $\forall\sigma,\sigma' \in \Sigma^\infty : \sigma \sqsubseteq \sigma' \Leftrightarrow \mathit{valid}_{\mathcal{T}}(\sigma) \subseteq \mathit{valid}_{\mathcal{T}}(\sigma')$.

The automaton $\mathcal{A}_t$ correctively enforces transactional properties. This automaton operates by suppressing the input sequence until it finds a transaction in $\mathcal{T}$, at which point the transaction is output while actions that are not part of any valid transaction are discarded.

Definition 5.1. Let $\mathcal{A}_t = \langle \Sigma, Q, q_0, \delta \rangle$ where

— Σ is a finite or countably infinite action set.
— $Q = \Sigma^* \times \Sigma^*$ is the set of automaton states. Each state consists of a pair $\langle \sigma_o, \sigma_s \rangle$ where σ_o is the sequence that has been output so far, and σ_s is a sequence that the monitor has suppressed and may eventually output or discard.
— $q_0 = \langle \epsilon, \epsilon \rangle$ is the initial state.
— The transition function $\delta : \langle \langle \Sigma^*, \Sigma^* \rangle \times \Sigma \rangle \to \langle \langle \Sigma^*, \Sigma^* \rangle \times \Sigma^\infty \rangle$ is given as

$$\delta\langle\langle \sigma_o, \sigma_s \rangle, a\rangle = \begin{cases} \langle\langle \sigma_o;\tau, \epsilon\rangle, \tau\rangle & \text{if } \exists\tau \in \mathit{suf}(\sigma_s;a) : \tau \in \mathcal{T} \wedge \tau \neq \epsilon \\ \langle\langle \sigma_o, \sigma_s;a\rangle, \epsilon\rangle & \text{otherwise} \end{cases}$$

PROPOSITION 5.2. *Let $\mathcal{T} \subseteq \Sigma^*$ be a subset of sequences, let $\hat{\mathcal{P}}_{\mathcal{T}}$ be the corresponding transactional property and let $\sqsubseteq$ be a preorder over the sequences of Σ^∞ defined such that $\forall\sigma,\sigma' \in \Sigma^\infty : \sigma \sqsubseteq \sigma' \Leftrightarrow \mathit{valid}_{\mathcal{T}}(\sigma) \subseteq \mathit{valid}_{\mathcal{T}}(\sigma')$. The automaton $\mathcal{A}_t$ correctively$_{\sqsubseteq}$ enforces $\hat{\mathcal{P}}_{\mathcal{T}}$.*

PROOF. Please consult appendix A for the proof. □

The method above shows how a transactional property can be enforced such that the output is always valid and always contains as many or more valid transactions than the input. In this particular example, we may be able to prove an even stronger enforcement paradigm, namely that the output will always contain exactly the same valid transactions as the input. This is unsurprising because equivalence relations are a special case of preorders, and imposing such a constraint would be tantamount to using the corrective$_{\cong}$ enforcement proposed in section 4. The method presented here can thus be seen as a generalization of corrective$_{\cong}$ enforcement.

This example also highlights why corrective$_{\cong}$ enforcement is too rigid to be useful in many practical cases. Consider what would happen if we proposed a more elaborate enforcement paradigm, in which the monitor corrects invalid transactions by, for instance, inserting actions to correct an incomplete transaction. In this case, the output sequence would contain strictly *more* valid transactions then the corresponding input, and thus would not be equivalent to it. Additionally, there may be several alternative valid replacements to any given invalid transaction. An equivalence relation grouping together the invalid sequence with all its possible replacements will necessarily include several sequences containing different transactions. Furthermore, if the monitor also maintains the option of deleting the offending transaction, the equivalence relation will necessarily include sequences which differ in the number of valid transactions they contain and imply that a valid transaction can permissibly be deleted from the input sequence.

The preorder we propose expresses no limits on the monitor's ability to correct invalid transactions. It follows that the monitor could, in principle, insert additional valid transactions. A tighter preorder could be used to restrict the monitor further, and prevent this from occurring. Such a preorder would indicate in which case a factor can permissibly be corrected in this manner and in which case it must simply be deleted. However, such decisions are highly user-dependant, as well as dependant on the specific transactional property being enforced. For this reason, we do not discuss here which specific modifications can or cannot be made to an invalid transaction and give instead the most general preorder that can be used to enforce transactional property.

5.2. Assured Pipelines

In the previous section, we showed how transactional properties can be enforced by an edit automaton that simply suppresses some actions from the input stream. However, part of the power of edit automata resides in their ability to insert actions not present in the input to correct an invalid sequence. Naturally, this ability must be constrained for the enforcement to remain meaningful, otherwise the monitor may simply replace any invalid sequence in its entirety by some unrelated valid sequence. In this section, we suggest three possible enforcement paradigms for the Assured Pipeline policy based on truncation, suppression and insertion. We also propose metrics that allow us to compare paradigms objectively.

The assured pipeline property was suggested in [Boebert and Kain 1985; Young et al. 1986] to ensure that data transformations are performed in a specific order. Let O be a set of data objects, and S be a set of transformations. We assume that S contains a distinguished member **create**. Finally, let $E = \langle S \times O \rangle$ be a set of access events. $\langle s, o \rangle \in E$ denotes the application of transformation s to the data object o. An assured pipeline policy restricts the application of transformations from S to data objects using an enabling relation $R = \langle S \times S \rangle$ with the following two restrictions: the relation R must define an acyclic graph, the **create** process can only occur at the root of this graph and the unique root of R must be the **create** process. The presence of a pair $\langle s, s' \rangle$ in R is represented in the graph by the occurrence of an edge between the vertice s and the vertice s' and indicates that any action of the form $\langle s', o \rangle$ is only permissible if s is the last process that accessed o. Because of the restriction that R must be an acyclic graph, each action $\langle s', o \rangle$ can occur at most once during an execution.

For the purpose of this example, we add another restriction, namely that the enabling relation must be linear. Formally: $\forall \langle s, s' \rangle \in R : \forall \langle s, s'' \rangle \in R \Rightarrow s' = s''$. Defined in this manner, each vertice has at most one predecessor and one successor. This condition makes easier for insertion monitors to add actions to the output without compromising transparency.

The assured pipeline policy can formally be defined as follows: Let R be an enabling relation over a set of objects O and transformations S. $\hat{\mathcal{P}}_R$ is an assured pipeline property over set $E = S \times O$ iff

Definition 5.3. Let R be an enabling relation over a set of objects O and transformations S. $\hat{\mathcal{P}}_R$ is an assured pipeline property over set $E = S \times O$ iff $\forall \sigma \in E^{\infty} : \sigma \in \hat{\mathcal{P}}_R \Leftrightarrow \forall i \in \mathbb{N} : \sigma_i = \langle s, o \rangle \Rightarrow ((s = create \vee \exists j \in \mathbb{N} : j < i \wedge \sigma_j = \langle s', o \rangle \wedge \langle s', s \rangle \in R) \wedge (\nexists k \in \mathbb{N} : k < i : \sigma_i = \sigma_k))$.

A truncation-based monitor was suggested by Fong [Fong 2004] and Talhi et al. [2008] to enforce this property. Their monitor effectively$_=$ enforces the assured pipeline property by aborting the execution if an unauthorized data transaction is encountered. Following their ideas, we propose the automaton $\mathcal{A}^t_{ap}$ below which enforces the assured pipelines in this manner.

Definition 5.4. $\mathcal{A}^t_{ap} = \langle E, Q, q_0, \delta_t \rangle$ where

— E is a set of access events over objects from O and a set of transformations from S as defined above.
— $Q : \wp(E)$ is the state space. Each state is an unordered set of access events from E.
— $q_0 = \emptyset$ is the initial state.
— $\delta_t : \wp(E) \times E \to \wp(E) \times E^\infty$ is given as

$$\delta_t(q, \langle s, o \rangle) = \begin{cases} (q \cup \{\langle s, o \rangle\}, \langle s, o \rangle) & \text{if } (s = create \wedge \langle create, o \rangle \notin q) \vee \\ & (\exists s' \in S : \langle s', o \rangle \in q \wedge \langle s', s \rangle \in R \wedge \langle s, o \rangle \notin q) \\ \text{undefined} & \text{otherwise} \end{cases}$$

PROPOSITION 5.5. *Let R be an enabling relation, that defines an assured pipeline policy $\hat{\mathcal{P}}_R$ over a set of actions $E = S \times O$. The automaton $\mathcal{A}^t_{ap}$ effectively$_=$ enforces $\hat{\mathcal{P}}_R$.*

PROOF. See appendix A. □

At each step of the execution, the monitor either outputs the current action if it is valid or else aborts by attempting an undefined transition otherwise. It follows that if the input sequence is invalid, the monitor outputs its longest valid prefix. Can this enforcement paradigm be improved upon? A corrective enforcement framework could allow the monitor to continue the execution after an illicit transformation has been attempted.

As was the case with transactional properties, the preorder defining the desired behavior of the program is stated in terms of the presence of valid actions, i.e., those occurring in a manner allowed by the enabling relation. Since each action can only occur once, a function returning the set of valid atomic actions is an adequate abstraction function. We write $valid_R(\sigma)$ for the set of valid transformations (w.r.t. the enabling relation R) occurring in σ. We write $\sigma \sqsubseteq \sigma' \Leftrightarrow valid_R(\sigma) \subseteq valid_R(\sigma')$.

Instead of simply aborting the execution, a corrective monitor may suppress an invalid action and allow the execution to proceed. We need to make only minor adjustments to $\mathcal{A}^t_{ap}$ to create the automaton $\mathcal{A}^s_{ap}$, that enforces the property in this manner.

Definition 5.6. Let $\mathcal{A}^s_{ap} = \langle E, Q, q_0, \delta_s \rangle$ where E, Q and q_0 are defined as in $\mathcal{A}^t_{ap}$ and $\delta_s : \wp(E) \times E \to \wp(E) \times E^\infty$ is given as follows.

$$\delta_s(q, \langle s, o \rangle) = \begin{cases} (q \cup \{\langle s, o \rangle\}, \langle s, o \rangle) & \text{if } (s = create \wedge \langle create, o \rangle \notin q) \vee \\ & (\exists s' \in S : \langle s', o \rangle \in q \wedge \langle s', s \rangle \in R \wedge \langle s, o \rangle \notin q) \\ (q, \epsilon) & \text{otherwise} \end{cases}$$

PROPOSITION 5.7. *Let R be an enabling relation, that defines an assured pipeline policy $\hat{\mathcal{P}}_R$ over a set of actions $E = S \times O$ and let $\sqsubseteq$ be a preorder over the sequences of E^∞ defined such that $\forall \sigma, \sigma' \in E^\infty : \sigma \sqsubseteq \sigma' \Leftrightarrow valid_R(\sigma) \subseteq valid_R(\sigma')$. The automaton $\mathcal{A}^s_{ap}$ correctively$_\sqsubseteq$ enforces $\hat{\mathcal{P}}_R$.*

PROOF. See appendix A. □

The automaton thus either outputs an action if it is valid, or it suppresses it before allowing the execution to continue. Yet, the edit automaton is capable of not only suppressing or outputting the actions present in the input, but also of adding actions not present in the execution to the input. It may be reasonable, for instance, for a monitor to suppress only those transformations that have already occurred. Otherwise, if an action occurs in the input sequence before the actions preceding

it in R have occurred, the monitor can enforce the property by adding the required actions. The automaton $\mathcal{A}^R_{ap}$, given in definition 5.8, enforces the property in the manner described above.

Definition 5.8. Let $\mathcal{A}^e_{ap} = \langle E, Q, q_0, \delta_e \rangle$ where Σ, Q and q_0 are defined as in $\mathcal{A}^s_{ap}$ and $\delta_e : \wp(E) \times E \to \wp(E) \times E^\infty$ is given as follows.

$$\delta_e(q, \langle s, o\rangle) = \begin{cases} (q \cup \{\langle s, o\rangle\}, \langle s, o\rangle) & \text{if } (s = create \wedge \langle create, o\rangle \notin q) \vee \\ & (\exists s' \in S : \langle s', o\rangle \in q \wedge \langle s', s\rangle \in R \wedge \langle s, o\rangle \notin q) \\ (q \cup acts(\tau; \langle s, o\rangle), \tau; \langle s, o\rangle) & \text{if } (\langle s, o\rangle \notin q \wedge (s \neq create) \wedge \\ & (\forall s' \in S : \langle s', s\rangle \in R \Rightarrow \langle s', o\rangle \notin q)) \\ & \text{where } \tau = path_R(\langle s, o\rangle, q) \\ (q, \epsilon) & \text{otherwise} \end{cases}$$

Where $path_R : (E \times \wp(E)) \to E^*$ is recursively defined as follows:

$$path_R(\langle s, o\rangle, q) = \begin{cases} \epsilon & \text{if } \langle s, o\rangle \in q \\ \langle s, o\rangle & \text{if } \langle s, o\rangle \notin q \wedge s = create \\ path_R(\langle s', o\rangle, q); \langle s, o\rangle & \text{otherwise where} (s', s) \in R \end{cases}$$

In fact, this function returns the sequence of actions, ending with the input sequence $\langle s, o\rangle$, that must be output so that the occurrence of $\langle s, o\rangle$ in the output does not violate the security property.

PROPOSITION 5.9. *Let R be an enabling relation, that defines an assured pipeline policy $\hat{\mathcal{P}}_R$ over a set of actions $E = S \times O$ and let $\sqsubseteq$ be a preorder over the sequences of E^∞ defined such that $\forall \sigma, \sigma' \in E^\infty : \sigma \sqsubseteq \sigma' \Leftrightarrow valid_R(\sigma) \subseteq valid_R(\sigma')$. The automaton $\mathcal{A}^e_{ap}$ correctively$_\sqsubseteq$ enforces $\hat{\mathcal{P}}_R$.*

PROOF. See appendix A. □

Note that had we used an equivalence relation rather than a preorder to constrain the monitor's behavior, and imposed that the monitor's output always be equivalent to its input rather than higher or equal on a preorder, it would not have been possible to correct the sequence in the manner described above. Indeed, an invalid input sequence cannot be thought of as equivalent to a corrected sequence to which valid transactions have been added, since it implies that those same transactions can permissibly be removed by the monitor in other cases. On the other hand, preorder allows us to impose on the monitor exactly those constraints we find necessary without further limiting its capacities.

We now have three possible execution monitors for the assured pipeline policy; two of them are based upon corrective enforcement and the other on effective$_=$ enforcement. In what follows, we propose metrics that allow us to compare these enforcement paradigms and enable us to select the most adequate monitor for a given situation.

The first metric we consider is based on the preorder used by the monitor to correctively$_\sqsubseteq$ enforce the property. Since this preorder is designed to capture the desired behavior of the target program, it is natural to also rely upon it to compare enforcement paradigms among themselves. If the output of a monitor $\mathcal{A}$ is consistently higher on the preorder $\sqsubseteq$ than that of another monitor $\mathcal{A}'$, we say that $\mathcal{A}$ is more corrective than $\mathcal{A}'$, noted $\mathcal{A}' \sqsubseteq \mathcal{A}$.

Definition 5.10. Let $\sqsubseteq$ be a preorder over a set of sequences from Σ^∞, let $\mathcal{S} \subseteq \Sigma^\infty$ be a set of input sequences and let $\mathcal{A}, \mathcal{A}'$ be edit automata enforcing the same property. $\mathcal{A}' \sqsubseteq^{\mathcal{S}} \mathcal{A} \Leftrightarrow \forall \sigma \in \mathcal{S} : \mathcal{A}'(\sigma) \sqsubseteq \mathcal{A}(\sigma)$. We write $\mathcal{A}' \equiv^{\mathcal{S}} \mathcal{A} \Leftrightarrow \mathcal{A} \sqsubseteq^{\mathcal{S}} \mathcal{A}' \wedge \mathcal{A}' \sqsubseteq^{\mathcal{S}} \mathcal{A}$. We omit the superscripted set of monitorable sequences when it is clear from the context.

In the case of the assured pipeline property, $\mathcal{A}^e_{ap}$ provides the most corrective enforcement followed by $\mathcal{A}^s_{ap}$ and finally by $\mathcal{A}^t_{ap}$. This follows from the fact that for the assured pipeline property, $\sqsubseteq$ is defined with respect to the presence of valid actions in the output. $\mathcal{A}^s_{ap}$ is more corrective than

$\mathcal{A}^t_{ap}$ since it allows more of the actions present in the input sequence to be output, while sometimes altering the sequence so that invalid actions become valid. $\mathcal{A}^e_{ap}$ provides an even more corrective enforcement by adding some valid actions.

PROPOSITION 5.11. *Let Σ be a set of atomic actions, $\mathcal{A}^t_{ap} \sqsubseteq^{\Sigma^\infty} \mathcal{A}^s_{ap} \sqsubseteq^{\Sigma^\infty} \mathcal{A}^e_{ap}$.*

PROOF. See appendix A. □

In some cases, one may favor a monitor that allows as much as possible of the input sequence to be output while also preserving the ordering between these input actions. This is particularly desirable when the user is not assumed to be malicious, as is the case, for example, with a monitor allocating resources between trusted users. A monitor $\mathcal{A}$ is more permissive than another monitor $\mathcal{A}'$ (noted $\mathcal{A}' \trianglelefteq^{\mathcal{S}}_p \mathcal{A}$) iff for any input sequence in a set $\mathcal{S}$, the output of $\mathcal{A}$ contains more of the actions present in the input than the output of $\mathcal{A}'$, regardless of any other actions that are inserted.

Definition 5.12. Let $\mathcal{S} \subseteq \Sigma^\infty$ be a set of input sequences and let $\mathcal{A}, \mathcal{A}'$ be edit automata enforcing the same property. $\mathcal{A}' \trianglelefteq^{\mathcal{S}}_p \mathcal{A} \Leftrightarrow$

— $\forall \sigma \in \mathcal{S} \cap \Sigma^* : |cs_\sigma(\mathcal{A}'(\sigma))| \leq |cs_\sigma(\mathcal{A}(\sigma))|$
— $\forall \sigma \in \mathcal{S} \cap \Sigma^\omega : \forall \tau \prec \sigma : \exists \tau' \prec \sigma \wedge \tau' \succeq \tau \wedge |cs'_\tau(\mathcal{A}'(\tau'))| \leq |cs'_\tau(\mathcal{A}(\tau'))|$.

We omit the superscripted set of monitorable sequences when it is clear from the context.

Of the three monitors proposed in this section for the enforcement of the assured pipeline property, $\mathcal{A}^e_{ap}$ provides the most permissive enforcement followed by $\mathcal{A}^s_{ap}$ and finally by $\mathcal{A}^t_{ap}$. This result follows immediately from the semantics of the three automata: $\mathcal{A}^t_{ap}$ operates by truncation, which prevents all actions from being output after a violation of the security policy occurs; $\mathcal{A}^s_{ap}$ operates by suppression, allowing at least some subsequent actions to be output; and $\mathcal{A}^e_{ap}$ operates both by insertion and suppression, deleting actions only if it is absolutely necessary and outputting it otherwise.

PROPOSITION 5.13. *Let Σ be a set of atomic actions, $\mathcal{A}^t_{ap} \trianglelefteq^{\Sigma^\infty}_p \mathcal{A}^s_{ap} \trianglelefteq^{\Sigma^\infty}_p \mathcal{A}^e_{ap}$.*

PROOF. See appendix A. □

5.3. Chinese Wall

As a third example, we consider the Chinese Wall policy, suggested in [Brewer and Nash 1989] to avoid conflicts of interest. In this model, a user who access a data object o is forbidden to simultaneously accessing certain other data objects identified as being in conflict with o. Several implementations of this model have been suggested in the literature. In this paper, we consider the framework of Sobel and Alves-Foss [1999], which includes a useful notion of data releasing.

Let S be a set of subjects and O a set of objects. The set of conflicts of interest is given as a set of pairs $C = \langle O \times O \rangle$. The presence of $\langle o_i, o_j \rangle \in C$ indicates that objects o_i and o_j conflict. Naturally, C is a symmetric set ($\forall o_i, o_j \in O : \langle o_i, o_j \rangle \in C \Leftrightarrow \langle o_j, o_i \rangle \in C$). For all objects $o_i \in O$, we write C_{o_i} for the set of objects which are in conflict with o_i. Let $O' \subseteq O$ be a subset of objects; overloading the notation we write $C_{O'}$ for $\bigcup_{o \in O'} C_o$. An action of the form $\langle \textbf{access}, s, o \rangle$ indicates that subject s holds the right to access object o. After this action is performed the subject can freely access the resource but can no longer access an object which conflicts with o.

It can often be too restrictive to impose that subjects never again access any data that conflicts with any other data objects they have previously accessed. In practice, the involvement of a subject with a given entity will eventually come to an end. Once this occurs, the subject should no longer be prevented from collaborating with the competitors of his former client. In [Sobel and Alves-Foss 1999], the model is enriched along this line by giving subjects the capacity to release previously acquired data. This can be modeled by the action $\langle \textbf{rel}, s, o \rangle$, indicating that subject s releases a previously accessed object o. After this action occurs in a sequence, s is once again allowed to access objects that conflict with o as long as they do not conflict with any other objects previously

accessed by s and have not yet been released. To simplify the notation, we define function $live$: $S \times \Sigma^* \rightarrow \wp(O)$ as follows: let s be a subject and τ be a finite sequence, $live(s, \tau)$ returns the set of objects which s has accessed in τ and has not yet released.

In the presence of a release action, the security property predicate is stated in the following manner:

Definition 5.14. Let C be a set of conflicts of interests over a set of objects O and let S be a set of subjects. $\hat{\mathcal{P}}_C$ is a Chinese Wall property of the set of actions $\Sigma = \{\textbf{access, rel}\} \times S \times O$ iff $\forall \sigma \in \Sigma^\infty : \hat{\mathcal{P}}_C(\sigma) \Leftrightarrow \forall s \in S : \forall o \in O : \forall i \in \mathbb{N} : \sigma_i = \langle \textbf{access}, s, o\rangle \Rightarrow (\nexists o' \in live(s, \sigma[..i-1]) : o \in C_{o'})$.

Where $live(\sigma) = \{o \in O \mid \exists i \in \mathbb{N} : \tau_i = \langle \textbf{access}, s, o\rangle \wedge \nexists j \in \mathbb{N} : j > i \wedge \tau_j = \langle \textbf{rel}, s, o\rangle\}$.

Both Fong [Fong 2004] and Talhi et al. [2008] suggest that this property be enforced by truncation using an automaton that aborts the execution as soon as a conflicting data access is encountered. The following automaton enforces the property in this manner.

Definition 5.15. $\mathcal{A}^t_{cw} = \langle \Sigma, Q, q_0, \delta_t\rangle$ where

— Σ : $\{\textbf{access, rel}\} \times S \times O$ is the set of all possible access and release events over objects from O and subjects from S,
— $Q = \Sigma^*$ is the state space. Each state is the finite sequence which has been output so far.
— $q_0 = \epsilon$ is the initial state.
— $\delta_t : \Sigma^* \times \Sigma \rightarrow \Sigma^*$ is given as:

$$\delta_t(q, a) = \begin{cases} \langle q; a, a\rangle & \text{if } (a = \langle \textbf{access}, s, o\rangle \wedge o \notin C_{live(s,q)}) \vee a = \langle \textbf{rel}, s, o\rangle \\ \text{undefined} & \text{otherwise} \end{cases}$$

PROPOSITION 5.16. *Let C be a set of conflicts of interest, and let $\hat{\mathcal{P}}_C$ be the corresponding Chinese Wall property. The automaton $\mathcal{A}^t_{cw}$ effectively$_=$ enforces $\hat{\mathcal{P}}_C$.*

PROOF. See appendix A. □

Since the monitor enforces the property by truncation, any authorized data access present in the input sequence after a conflicting data access has occurred is absent from the output sequence. Corrective enforcement can provide a more flexible enforcement paradigm. First, we define the preorder $\sqsubseteq$. Analogous to the preorder proposed in section 5.1 (to weed out invalid transactions while preserving valid ones) we impose that authorized data accesses be preserved while conflicting ones be deleted. Let σ be a sequence and let C be the corresponding conflict of interest class, we write $valid_C(\sigma)$ for the multiset of actions from σ occurring in that sequence in a manner consistent with C. The preorder is defined as $\forall \sigma, \sigma' \in \Sigma^\infty : \sigma \sqsubseteq \sigma' \Leftrightarrow valid_C(\sigma) \sqsubseteq valid_C(\sigma')$. The most intuitive manner to correctively$_\sqsubseteq$ enforce this property is to suppress conflicting data access but allow the execution to proceed afterward. Only a slight adjustment needs to be made to $\mathcal{A}^t_{cw}$ to create a monitor enforcing the Chinese Wall property in this manner.

Definition 5.17. Let $\mathcal{A}^s_{cw} = \langle \Sigma, Q, q_0, \delta_s\rangle$ where Σ, Q and q_0 are defined as in $\mathcal{A}^t_{cw}$ and δ_s : $\Sigma^* \times \Sigma \rightarrow \Sigma^*$ is given as follows:

$$\delta_s(q, a) = \begin{cases} \langle q; a, a\rangle & \text{if } (a = \langle \textbf{access}, s, o\rangle \wedge o \notin C_{live(s,q)}) \vee a = \langle \textbf{rel}, s, o\rangle \\ \langle q, \epsilon\rangle & \text{otherwise .} \end{cases}$$

PROPOSITION 5.18. *Let C be a set of conflicts of interests, and let $\hat{\mathcal{P}}_C$ be the corresponding Chinese Wall property. The automaton $\mathcal{A}^s_{ap}$ correctively$_\sqsubseteq$ enforces $\hat{\mathcal{P}}_C$.*

PROOF. The proof follows exactly as that of proposition 5.7. □

As discussed above, the involvement of any subject with any given entity eventually ends. When this occurs, objects in conflict with that entity become available to this subject for access if they do not conflict with any other object currently held by this subject.

Thus it is natural to suggest an alternative enforcement paradigm in which the monitor keeps track of data access that have been denied and inserts them after a release action makes them available to the subject that has requested them. In fact, access to conflicting objects is delayed rather than suppressed.

The automaton $\mathcal{A}^e_{cw}$ enforces the property in this manner:

Definition 5.19. $\mathcal{A}^e_{cw} = \langle \Sigma, Q, q_0, \delta_e \rangle$ where

— $\Sigma = \{access, rel\} \times S \times O$ is the set of all possible access and release events over objects from O and subjects from S,
— $Q = \Sigma^* \times \Sigma^*$ is the state space. Each state is a pair $\langle \sigma_o, \sigma_s \rangle$ of finite sequences where σ_o is the sequence that has been output so far, and σ_s is the sequence that has been seen and suppressed.
— $q_0 = \langle \epsilon, \epsilon \rangle$ is the initial state.
— $\delta_e : (\langle \Sigma^*, \Sigma^* \rangle \times \Sigma) \to (\langle \Sigma^*, \Sigma^* \rangle \times \Sigma^\infty)$ is given as:

$$\delta_e(\langle \sigma_o, \sigma_s \rangle, a) = \begin{cases} \langle \langle \sigma_o; a, \sigma_s \rangle, a \rangle & \text{if } a = \langle \textbf{access}, s, o \rangle \wedge o \notin C_{live(s,\sigma_o)} \\ \langle \langle \sigma_o; a; \tau, \sigma_s \backslash \tau \rangle, a; \tau \rangle & \text{if } a = \langle \textbf{rel}, s, o \rangle \text{ where } \tau = f(\sigma_o; a, \sigma_s, s, \epsilon) \\ \langle \langle \sigma_o, \sigma_s; a \rangle, \epsilon \rangle & \text{otherwise} \end{cases}$$

where $f : \Sigma^* \times \Sigma^* \times S \times \Sigma^* \to \Sigma^*$ is recursively defined as follows:

$$f(\sigma_o, \sigma, s, \tau) = \begin{cases} \tau & \text{if } \sigma = \epsilon \\ f(\sigma_o; \sigma_1, \sigma[2..], s, \tau; \sigma_1) & \text{if } \sigma_1 = \langle \textbf{access}, s, o \rangle \wedge o \notin C_{live(s,\sigma_o)} \\ f(\sigma_o, \sigma[2..], s, \tau) & \text{otherwise} \end{cases}$$

This function examines the sequences that have been suppressed and output up to that point. It then returns a sequence composed of all actions that were previously suppressed but can now be output without causing a conflict.

PROPOSITION 5.20. *Let C be a set of conflicts of interest, and let $\hat{\mathcal{P}}_C$ be the corresponding Chinese Wall property. The automaton $\mathcal{A}^e_{cw}$ correctively$_\sqsubseteq$ enforces $\hat{\mathcal{P}}_C$.*

PROOF. See appendix A. □

Once again, we can compare the three proposed enforcement paradigms using several metrics. Let us begin with the question of which is more corrective and which is more permissive. As was the case with the assured pipeline property, we find that the monitor based upon truncation is both the most corrective and the least permissive; the one based on edition is both the most corrective and the most permissive; and the one based on suppression occupies an intermediary position with respect to both metrics. Furthermore, as was also the case with the three automata proposed to enforce the assured pipeline property, this gradation follows from the different manner in which each automaton reacts to an action violating the security policy: namely by truncation, suppression or a mix of insertions and suppressions.

PROPOSITION 5.21. $\mathcal{A}^t_{cw} \sqsubseteq^{\Sigma^\infty} \mathcal{A}^s_{cw} \sqsubseteq^{\Sigma^\infty} \mathcal{A}^e_{cw}$.

PROOF. See appendix A. □

PROPOSITION 5.22. $\mathcal{A}^t_{cw} \trianglelefteq_p^{\Sigma^\infty} \mathcal{A}^s_{cw} \trianglelefteq_p^{\Sigma^\infty} \mathcal{A}^e_{cw}$.

PROOF. See appendix A. □

Another aspect that must be taken into consideration while selecting an enforcement paradigm is the amount of memory the monitor may require. The monitor may need to store the execution that has occurred so far or an abstraction of it, as well as any sequence that has been suppressed, but not yet output.

Since different enforcement paradigms for the same security property can vary with respect to the amount of memory they require, it is natural that this factor also be used to compare enforcement mechanisms.

Definition 5.23. Let $mem(\mathcal{A})$ stand for the space efficiency of the monitoring algorithm $\mathcal{A}$ (measured in O notation). Let $\mathcal{A}, \mathcal{A}'$ be edit automata enforcing the same property. We write $\mathcal{A} \trianglelefteq_m^{\mathcal{S}} \mathcal{A}' \Leftrightarrow mem(\mathcal{A}) \subseteq mem(\mathcal{A}')$. We write $\mathcal{A} \equiv_m^{\Sigma^\infty} \mathcal{A}' \Leftrightarrow \mathcal{A} \trianglelefteq_m^{\Sigma^\infty} \mathcal{A}' \wedge \mathcal{A}' \trianglelefteq_m^{\Sigma^\infty} \mathcal{A}$. We omit the superscripted set of monitorable sequences when it is clear from the context.

Comparing the three monitors proposed for the Chinese Wall property with respect to this metric, we find that $\mathcal{A}_{cw}^e$ necessitates the most memory while $\mathcal{A}_{cw}^s$ and $\mathcal{A}_{cw}^t$ both have the same memory requirements. This is because while all three monitors must maintain a list of the resources that are live (accessed and not yet released) for each subject at any given time, $\mathcal{A}_{cw}^e$ must additionally keep track of resources that have been suppressed, with the expectation that they will be output when no longer cause conflicts.

PROPOSITION 5.24. $\mathcal{A}_{cw}^t \trianglelefteq_m^{\Sigma^\infty} \mathcal{A}_{cw}^s \trianglelefteq_m^{\Sigma^\infty} \mathcal{A}_{cw}^e$.

PROOF. See appendix A. □

5.4. General Availability

The final security property that we examine is general availability: a policy requiring that any resource that is acquired is eventually released. Despite its apparent simplicity, the property is remarkably difficult to monitor in a system with more than one resource and is given by Ligatti et al. as an example of a property that is not effectively$_=$ enforceable.

Modifying Ligatti et al.'s formulation slightly, we define the property as follows: let $\langle ac, r\rangle, \langle use, r\rangle$ and $\langle rel, r\rangle$ stand for accessing, using and releasing a resource r from a set of resources $\mathcal{R}$. The set of possible actions is given as $\Sigma = \{ac, use, rel\} \times \mathcal{R}$. The property states that only acquired resources are used, and that any acquired resource is eventually released. Note that this seemingly straightforward property combines a liveness component that cannot be monitored [Ligatti et al. 2005] with a safety component.

From the onset of the study of formal monitors, there has been an interest in examining how restricting the set of possible executions can increase that of enforceable properties. The intuition is that if the monitor knows that certain execution paths cannot occur in its target, it should be able to use this information to enforce properties that would otherwise not be enforceable. This question was first raised in [Schneider 2000] and was later addressed in [Bauer et al. 2002; Chabot et al. 2009].

It this section, we use the availability property proposed above to show that an a priori knowledge of the program's possible execution paths not only increases the set of properties enforceable by the monitor, but also provides a different, and possibly preferable, enforcement of a given property. First, consider the difficulties facing a monitor trying to enforce the general availability property in a context in which $\mathcal{S} = \Sigma^\infty$. The monitor may not simply abort an invalid execution since the liveness component implies that some invalid sequences can be corrected. Nor can it suppress a potentially invalid sequence only to output it when a valid prefix is reached. As observed in [Ligatti et al. 2005], an infinite sequence of the form $\langle ac, r_1\rangle; \langle ac, r_2\rangle; \langle rel, r_1\rangle; \langle ac, r_3\rangle; \langle rel, r_2\rangle; \langle ac, r_4\rangle; \langle rel, r_3\rangle ...$ is valid but has no finite valid prefix. The property can only be enforced by a monitor transforming the input more aggressively.

Definition 5.25. Let $\mathcal{R}$ be a set of resources and let $\Sigma = \{ac, use, rel\} \times \mathcal{R}$ be a set of atomic actions. $\hat{\mathcal{P}}$ is a general availability property iff $\forall \sigma \in \Sigma^\infty : \sigma \in \hat{\mathcal{P}} \Leftrightarrow \forall j \in \mathbb{N} : \sigma_j = \langle ac, r\rangle \Rightarrow (\exists k \in \mathbb{N} : k > j : \sigma_k = \langle rel, r\rangle) \wedge \sigma_j = \langle use, r\rangle \Rightarrow i \in live(\sigma[..j])$.

Where $live(\sigma) = \{r \in \mathcal{R} \mid \exists j \in \mathbb{N} : \sigma_j = \langle ac, r\rangle \wedge \nexists k \in \mathbb{N} : k > i \wedge \sigma_k = \langle rel, r\rangle\}$.

Since the desired behavior of the program is given in terms of the presence of $\langle use, r\rangle$ actions bracketed by $\langle ac, r\rangle$ and $\langle rel, r\rangle$ actions, a preorder based on the number of occurrences of such actions is a natural way to compare sequences. We can designate such use actions as valid. Let the function *valid*(σ) that returns the multiset of actions $\langle use, r\rangle$ which occur in sequence σ and are both preceded by a $\langle ac, r\rangle$ action and followed by a $\langle rel, r\rangle$ action be the abstraction function $\mathcal{F}$. We thus have that $\forall \sigma, \sigma' \in \Sigma^* : \sigma \sqsubseteq \sigma' \Leftrightarrow$ *valid*$(\sigma) \subseteq$ *valid*(σ').

A trivial way to enforce this property is for the monitor to insert an $\langle ac, r\rangle$ action prior to each $\langle use, r\rangle$ action, and follow it with a $\langle rel, r\rangle$ action. The $\langle ac, r\rangle$ and $\langle rel, r\rangle$ actions present in the original sequence can then simply be suppressed, as every $\langle use, r\rangle$ action is made available by a pair of actions $\langle ac, r\rangle$ and $\langle rel, r\rangle$ inserted by the monitor. While theoretically feasible, it is obvious that a monitor which opens and closes a resource after every program step would be of limited use in practice. Furthermore, for many applications, a sequence of the form $\langle ac, r\rangle; \langle use, r\rangle; \langle use, r\rangle; \langle rel, r\rangle$ cannot be considered equivalent to the sequence $\langle ac, r\rangle; \langle use, r\rangle; \langle rel, r\rangle; \langle ac, r\rangle; \langle use, r\rangle; \langle rel, r\rangle$. We thus limit this study to monitors which insert $\langle ac, r\rangle$ actions a finite number of times.

We first propose a monitoring framework capable of enforcing the general availability property in a uniform context, where every sequence in Σ^∞ can occur in the target program. A monitor can enforce the property in this context by suppressing every acquire action, as well as subsequent use actions for the same resource, and output them only when a release action is reached. Any use action not preceded by a corresponding acquire action is simply suppressed and never inserted again. A monitor enforcing the property in this manner needs to keep track only of the actions that have been suppressed and not output so far. The automaton $\mathcal{A}_{ga}^{\Sigma^\infty}$ enforces the property as described above.

Definition 5.26. Let $\mathcal{A}_{ga}^{\Sigma^\infty} = \langle \Sigma, Q, q_0, \delta_e\rangle$ where

— $\Sigma = \{ac, rel, use\} \times \mathcal{R}$ is the set of all possible acquire, release and use actions for all objects;
— $Q = \langle \Sigma^*, \Sigma^*\rangle$ is a pair composed of the sequence that has been output so far and the sequence that has been suppressed so far;
— $q_0 = \langle \epsilon, \epsilon\rangle$ is the initial state;
— $\delta_{\Sigma^\infty} : \langle \Sigma^*, \Sigma^*\rangle \times \Sigma \to \langle \Sigma^*, \Sigma^*\rangle \times \Sigma^\infty$ is given as:

$$\delta_{\Sigma^\infty}(\langle \sigma_o, \sigma_s\rangle, a) = \begin{cases} \langle \langle \sigma_o, \sigma_s\rangle, \epsilon\rangle & \text{if } a = \langle use, r\rangle \wedge \langle ac, r\rangle \notin acts(\tau) \\ \langle \langle \sigma_o, \sigma_s; a\rangle, \epsilon\rangle & \text{if } a = \langle ac, r\rangle \vee (a = \langle use, r\rangle \wedge \\ & \langle ac, r\rangle \in acts(\sigma_s)) \\ \langle \langle \sigma_o; \tau; \langle rel, r\rangle, \sigma_s \backslash \tau\rangle, \tau; \langle rel, r\rangle\rangle & \text{if } a = \langle rel, r\rangle \text{where } \tau = f_r(\sigma_s, \epsilon) \end{cases}$$

Where $f_r : (\Sigma^* \times \Sigma^*) \to \Sigma^*$ is recursively defined as follows:

$$f_r(\sigma, \tau) = \begin{cases} f_r(\sigma[2..], \tau; \sigma_1) & \text{if } \sigma_1 = \langle use, r\rangle \vee \langle ac, r\rangle \vee \langle rel, r\rangle \\ \tau & \text{if } \sigma_1 = \epsilon \\ f_r(\sigma[2..], \tau) & \text{otherwise} \end{cases}$$

The purpose of f_r is simply to examine the suppressed sequence and retrieve the actions over resource r. Observe that σ_s does not contain any rel action.

PROPOSITION 5.27. *The automaton $\mathcal{A}_{ga}^{\Sigma^\infty}$ correctively$_\sqsubseteq$ enforces the general availability property.*

PROOF. See appendix A. □

Next we consider a monitor operating in a context in which every computation of the target program is *fair* which means certain actions must occur infinitely often in infinite paths. For the purpose of the general availability property it would mean that any action $\langle ac, r\rangle$ is eventually followed by a $\langle rel, r\rangle$ action, or by the end of the execution. The property can thus be violated only by the presence of use actions that are not preceded by a corresponding ac action. Formally, let $\mathcal{R}$ be a set of resources and let $\Sigma = \{ac, use, rel\} \times \mathcal{R}$. We say that a set $\mathcal{S} \subseteq \Sigma^\infty$ is fair iff

$$\forall \sigma \in \mathcal{S} \cap \Sigma^\omega : \forall r \in \mathcal{R} : \forall j \in \mathbb{N} : \sigma_j = \langle ac, r\rangle \Rightarrow \exists k \in \mathbb{N} : k > j \wedge \sigma_k = \langle rel, r\rangle. \quad \text{(fairness)}$$

If the monitor is operating in a fair context, a new, more permissive enforcement method becomes possible, as is illustrated by automaton $\mathcal{A}_{ga}^{fair}$.

Definition 5.28. Let $\mathcal{A}_{ga}^{fair} = \langle \Sigma, Q, q_0, \delta_e \rangle$ where Σ is defined as above and

— $Q : \wp(\mathcal{R})$ is the set of resources that have been acquired but not yet released;
— $q_0 = \emptyset$ is the initial state;
— $\delta_{fair} : (\wp(\mathcal{R}) \times \Sigma) \to \wp((\mathcal{R}) \times \Sigma^\infty)$ is given as:

$$\delta_{fair}(q,a) = \begin{cases} \langle q, \epsilon \rangle & \text{if } a = \langle use, r \rangle \wedge r \notin q \\ \langle q \cup \{r\}, a \rangle & \text{if } a = \langle ac, r \rangle \\ \langle q \backslash \{r\}, a \rangle & \text{if } a = \langle rel, r \rangle \\ \langle q, a \rangle & \text{if } a = \langle use, r) \wedge r \in q \\ \langle \emptyset, f(q) \rangle & \text{if } a = a_{end} \end{cases}$$

Where $f : \wp(\mathcal{R}) \to \Sigma^*$ is a function returning a sequence of the form $\langle rel, r_i \rangle, \langle rel, r_j \rangle, \langle rel, r_k \rangle ...$ for all resources present in its input and a_{end} is a token action marking the end of the input sequence.

PROPOSITION 5.29. *Let $\mathcal{S} \subseteq \Sigma^\infty$ be a subset of sequences such that $\forall \sigma \in \mathcal{S} \cap \Sigma^\omega : \forall r \in \mathcal{R} : \forall j \in \mathbb{N} : \sigma_j = \langle ac, r \rangle \Rightarrow \exists k \in \mathbb{N} : k > j \wedge \sigma_k = \langle rel, r \rangle$.*
The automaton $\mathcal{A}_{ga}^{fair}$ correctively$_{\sqsubseteq}^{\mathcal{S}}$ enforces the general availability property.

PROOF. See appendix A. □

Finally, we consider the enforcement power of a monitor operating under the assumption that every execution eventually terminates. If this is the case, an even more corrective enforcement paradigm is available to the monitor, since it can acquire resources as needed and close them at the end of the execution. Since every execution is of finite length, the number of use actions it contains is also necessarily finite. It follows that a monitor that systematically inserts release actions into the input stream whenever an unacquainted use action is encountered will necessarily insert only a finite number of actions.

Definition 5.30. Let $\mathcal{A}_{ga}^{\Sigma^*} = \langle \Sigma, Q, q_0, \delta_e \rangle$ where Σ is defined as above and

— $Q : \wp(\mathcal{R})$ is the set of resources that have been acquired but not yet released;
— $q_0 = \emptyset$ is the initial state;
— $\delta_{\Sigma^*} : (\wp(\mathcal{R}) \times \Sigma) \to (\wp(\mathcal{R}) \times \Sigma^\infty)$ is given as: $\delta_{\Sigma^*}(q,a) =$

$$\begin{cases} \langle q \cup \{r\}, \langle ac, r \rangle; a \rangle & \text{if } a = \langle use, r \rangle \wedge r \notin q \\ \langle q \cup \{r\}, a \rangle & \text{if } a = \langle ac, r \rangle \\ \langle q \backslash \{r\}, a \rangle & \text{if } a = \langle rel, r \rangle \\ \langle q, a \rangle & \text{if } a = \langle use, r \rangle \wedge r \in q \\ \langle \emptyset, f(q) \rangle & \text{if } a = a_{end} \end{cases}$$

Where $f : \wp(\mathcal{R}) \to \Sigma^*$ is a function returning a sequence of the form $\langle rel, r_i \rangle, \langle rel, r_j \rangle, \langle rel, r_k \rangle ...$ for all resources present in its input.

PROPOSITION 5.31. *The automaton $\mathcal{A}_{ga}^{\Sigma^*}$ correctively$_{\sqsubseteq}^{\Sigma^*}$ enforces the general availability property.*

PROOF. See appendix A. □

The three monitors presented above thus differ with respect to the set of possible input sequences for which they are able to correctively$_{\sqsubseteq}$ enforce the general availability property. The monitor $\mathcal{A}_{ga}^{\Sigma^\infty}$ can enforce the property in all cases while $\mathcal{A}_{ga}^{\Sigma^*}$ and $\mathcal{A}_{ga}^{fair}$ can only do so if the presence of some execution paths has been rules out. It is natural to use this restriction on the monitor's generality as a basis for comparing monitors. We say that a monitor is more versatile than another iff it can enforce the same property when strictly more sequences are present.

Definition 5.32. Let $\mathcal{A}, \mathcal{A}'$ be edit automata enforcing the same property $\hat{\mathcal{P}}$. $\mathcal{A} \trianglelefteq_v \mathcal{A}' \Leftrightarrow \forall S \in \wp(\Sigma^\infty) : \forall \sigma \in S : (\hat{\mathcal{P}}(\mathcal{A}(\sigma)) \wedge \sigma \sqsubseteq \mathcal{A}(\sigma)) \Rightarrow (\hat{\mathcal{P}}(\mathcal{A}'(\sigma)) \wedge \sigma \sqsubseteq \mathcal{A}'(\sigma))$.

We write $\mathcal{A} \lhd_v \mathcal{A}' \Leftrightarrow \mathcal{A} \trianglelefteq_v \mathcal{A}' \wedge \neg(\mathcal{A}' \trianglelefteq_v \mathcal{A})$ and $\mathcal{A} \equiv_v \mathcal{A}' \Leftrightarrow \mathcal{A} \trianglelefteq_v \mathcal{A}' \wedge \mathcal{A}' \trianglelefteq_v \mathcal{A}$.

Clearly $\mathcal{A}_{ga}^{\Sigma^\infty}$ is the most versatile monitor for the general availability property, $\mathcal{A}_{ga}^{fair}$ is strictly less versatile than $\mathcal{A}_{ga}^{\Sigma^\infty}$ and $\mathcal{A}_{ga}^{\Sigma^*}$ is strictly less versatile than $\mathcal{A}_{ga}^{fair}$.

PROPOSITION 5.33. $\mathcal{A}_{ga}^{\Sigma^*} \lhd_v \mathcal{A}_{ga}^{fair} \lhd_v \mathcal{A}_{ga}^{\Sigma^\infty}$.

PROOF. See appendix A. □

The automaton enforcing the general availability property can also be compared using the metrics suggested in the previous sections. Such comparisons can only be performed over a set of input sequences for which both automata are capable of enforcing the property. In this case, we compare the monitors over the set Σ^*. We observe that over this set, $\mathcal{A}_{ga}^{\Sigma^\infty}$ is less corrective than $\mathcal{A}_{ga}^{fair}$, and both automata as less corrective than $\mathcal{A}_{ga}^{\Sigma^*}$.

PROPOSITION 5.34. $\mathcal{A}_{ga}^{fair} \sqsubseteq^{\Sigma^*} \mathcal{A}_{ga}^{\Sigma^\infty} \sqsubseteq^{\Sigma^*} \mathcal{A}_{ga}^{\Sigma^*}$.

PROOF. See appendix A. □

Conversely, using the memory usage metric introduced in section 5.3, we find that $\mathcal{A}_{ga}^{\Sigma^*}$ necessitates a higher memory overhead than and $\mathcal{A}_{ga}^{fair}$. Indeed, while both maintain the same information throughout the execution of their target, namely a list of resources that have been acquired and not yet released, there are cases where $\mathcal{A}_{ga}^{\Sigma^*}$ will output an acquire action while $\mathcal{A}_{ga}^{fair}$ simply deletes this action. In those cases, $\mathcal{A}_{ga}^{\Sigma^*}$ must keep track of the fact that this specific resource has been acquired. $\mathcal{A}_{ga}^{\Sigma^\infty}$ is incomparable to the two other automata, because of its differing semantics.

PROPOSITION 5.35. $\mathcal{A}_{ga}^{\Sigma^*} \trianglelefteq_m^{\Sigma^*} \mathcal{A}_{ga}^{fair}$.

PROOF. See appendix A. □

$\mathcal{A}_{ga}^{\Sigma^\infty}$ is incomparable to the two other automata with respect to the permissiveness metric because of its ability to reorder the actions present in the input sequence. However, $\mathcal{A}_{ga}^{fair}$ is less permissive than $\mathcal{A}_{ga}^{\Sigma^*}$, since the former sometimes suppresses use actions that $\mathcal{A}_{ga}^{\Sigma^*}$ outputs.

PROPOSITION 5.36. $\mathcal{A}_{ga}^{fair} \trianglelefteq_p^{\Sigma^*} \mathcal{A}_{ga}^{\Sigma^*}$.

PROOF. See appendix A. □

5.5. Discussion

The driving idea behind this paper is that the transformations performed by a monitor faced with an invalid execution constitute an integral part of the notion of *enforcement*. Our main insight is that constraints on the monitor's permissible corrective actions can be stated in the form of preorders, and that the behavior of edit-automata, a widely used model of monitors, can then be checked against these preorders to ensure compliance with the original constraints.

In section 4.2, we argued that the preorder should not be extracted directly from the property being enforced. Instead, the preorder should be provided alongside the property, as both reflect a different aspect of the security specification. This realization opens a new avenue of research in which the range of possible corrective actions available to a security policy enforcement mechanism is explored and formalized. This new avenue would mirror mainstream research in security property specification and classification and would allow users to select the constraints that most adequately reflect their security needs. We have illustrated these new concepts with four examples of real life

security policies. In most cases, we proposed several monitors enforcing the same property but performing different transformations on invalid executions. Four metrics, measuring corrective degree, permissiveness, memory use and versatility, are proposed to contrast different monitors enforcing a given property. We summarize our results in Table I. Observe that in this table, all comparisons of the automata enforcing Assured pipelines and Chinese wall property are done over the set Σ^ω while those for the General availability property (except for versatility) are over the set Σ^*.

	Assured Pipelines	Chinese Wall	General Availability
More Corrective (definition 5.10)	$\mathcal{A}^t_{ap} \sqsubseteq \mathcal{A}^s_{ap} \sqsubseteq \mathcal{A}^e_{ap}$	$\mathcal{A}^t_{cw} \sqsubseteq \mathcal{A}^s_{cw} \sqsubseteq \mathcal{A}^e_{cw}$	$\mathcal{A}^{fair}_{ga} \sqsubseteq \mathcal{A}^{\Sigma^\infty}_{ga} \sqsubseteq \mathcal{A}^{\Sigma^*}_{ga}$
Permissiveness (definition 5.12)	$\mathcal{A}^t_{ap} \trianglelefteq_p \mathcal{A}^s_{ap} \trianglelefteq_p \mathcal{A}^e_{ap}$	$\mathcal{A}^t_{cw} \trianglelefteq_p \mathcal{A}^s_{cw} \trianglelefteq_p \mathcal{A}^e_{cw}$	$\mathcal{A}^{fair}_{ga} \trianglelefteq_p \mathcal{A}^{\Sigma^*}_{ga}$
Memory Use (definition 5.23)	—	$\mathcal{A}^t_{cw} \trianglelefteq^{\Sigma^\infty}_m \mathcal{A}^s_{cw} \trianglelefteq^{\Sigma^\infty}_m \mathcal{A}^e_{cw}$	$\mathcal{A}^{\Sigma^*}_{ga} \trianglelefteq_m \mathcal{A}^{fair}_{ga}$
Versatility (definition 5.32)	—	—	$\mathcal{A}^{\Sigma^*}_{ga} \lhd_v \mathcal{A}^{fair}_{ga} \lhd_v \mathcal{A}^{\Sigma^\infty}_{ga}$

Table I: Comparison of multiple automata enforcing 3 security properties using 4 metrics.

Multiple automata constrained by the same preorder can still differ substantially in how they enforce a given property. The preorder could be defined in a more inflexible manner, to further tighten the monitor's behavior. However, as we argued in section 4.1, imposing too many constraints may limit the monitor's ability to take needed corrective actions. Instead, we propose to codify in the preorder only those aspects that are essential to the enforcement of the property by relying on the metrics to capture other desirable properties such as low memory footprint. Furthermore, the aspects captured by the metrics may conceivably conflict, for example, with a more corrective enforcement of certain properties that may necessarily be less permissive or less general. It would thus be difficult to capture both concerns simultaneously using a preorder.

One of the most interesting applications of the enforcement paradigm we propose is in the area of monitor inlining [Erlingsson and Schneider 2000]: the process by which untrusted code is instrumented to ensure its compliance with a user-supplied security policy. Corrective enforcement allows us to set limits to the degree to which these transformations alter the original code. Furthermore, the inlining process can be associated with a certification, to guarantee that the instrumented code respects the desired security policy. However, previous work has focused only on the certification of soundness [Hamlen 2011]. We believe that using corrective enforcement will make the certification of transparency easier.

6. CONCLUSION AND FUTURE WORK

We proposed a framework to analyze the security properties enforceable by monitors capable of transforming their input. By imposing constraints on the enforcement mechanism to the effect that some behaviors existing in the input sequence must still be present in the output, we are able to model the desired behavior of real-life monitors in a more realistic and effective way. We also show that real life properties are enforceable in this paradigm and give four examples of relevant real-life properties. Finally, we propose metrics that can be used to compare alternative enforcements of the same security property.

The framework presented in this paper allows us to transform a program execution to ensure its compliance with a security policy while also preserving the semantics of the execution. We believe

this framework is sufficiently flexible to be useful in other program rewriting contexts and even in situations where security is not the main concern such as controller synthesis or specification refinement. Corrective enforcement could also be adapted to other enforcement paradigms apart from monitoring such as program rewriting. In future work, we hope to adapt our corrective framework to such contexts.

APPENDIX A: Proof of Propositions

PROPOSITION 5.2 Let $\mathcal{T} \subseteq \Sigma^*$ be a subset of sequences, let $\hat{\mathcal{P}}_\mathcal{T}$ be the corresponding transactional property and let $\sqsubseteq$ be a preorder over the sequences of Σ^∞ defined such that $\forall \sigma, \sigma' \in \Sigma^\infty$: $\sigma \sqsubseteq \sigma' \Leftrightarrow \mathit{valid}_\mathcal{T}(\sigma) \subseteq \mathit{valid}_\mathcal{T}(\sigma')$. The automaton $\mathcal{A}_t$ correctively$_\sqsubseteq$ enforces $\hat{\mathcal{P}}_\mathcal{T}$.

PROOF.

By definition 4.4, we have that the automaton $\mathcal{A}_t$ correctively$_\sqsubseteq$ enforces the property iff $\forall \sigma \in \Sigma^\infty : \mathcal{A}_t(\sigma) \in \hat{\mathcal{P}}_\mathcal{T} \wedge \sigma \sqsubseteq \mathcal{A}_t(\sigma)$, in other words, if the output of $\mathcal{A}_t$ is in $\hat{\mathcal{P}}_\mathcal{T}$ and is higher or equal to the input on the preorder.

Let $\sigma \in \Sigma^\infty$ be an input sequence, let $\sigma[..i]$ be the prefix of σ processed so far and let $q = \langle \sigma_o^i, \sigma_s^i \rangle$ stand for the state of $\mathcal{A}_t$ reached after processing the input $\sigma[..i]$. We show that the automaton maintains the invariant INV(i)= $\sigma_o^i \in \hat{\mathcal{P}}_\mathcal{T} \wedge \sigma[..i] \sqsubseteq \sigma_o^i \wedge \sigma_s^i \in \mathit{suf}(\sigma[..i])$, at each step i. The invariant holds initially since $\sigma[..0] = \epsilon$ and the automaton is in state $\langle \epsilon, \epsilon \rangle$. By the definition of transactional properties we have $\epsilon \in \hat{\mathcal{P}}_\mathcal{T}$ and $\epsilon \sqsubseteq \epsilon$. Let us assume that INV(i) holds and let a be the next input action. We show that INV($i+1$) holds, where $\sigma[..i+1] = \sigma[..i]; a$.

There are two cases to consider.

(1) $\exists \tau \in \mathit{suf}(\sigma_s^i; a) : \tau \in \mathcal{T} \wedge \tau \neq \epsilon$. In this case, the automaton outputs a sequence $\tau \in \hat{\mathcal{P}}_\mathcal{T}$. Since by assumption $\sigma_o^i \in \hat{\mathcal{P}}_\mathcal{T}$, and by the definition of transactional properties, $\forall \upsilon, \upsilon' \in \hat{\mathcal{P}}$: $\upsilon; \upsilon' \in \hat{\mathcal{P}}$, it follows that $\sigma_o^i; \tau \in \hat{\mathcal{P}}_\mathcal{T}$. The fact that σ_s^{i+1} is a suffix of $\sigma[..i+1]$ holds trivially since after processing a, $\sigma_s^{i+1} = \epsilon$. Finally, to show that $\sigma[..i]; a \sqsubseteq \mathcal{A}_t(\sigma[..i]; a)$, first observe that since by the induction hypothesis we have $\sigma[..i] \sqsubseteq \mathcal{A}_t(\sigma[..i])$, any transaction present in $\sigma[..i]; a$ but not present in $\sigma[..i]$ necessarily includes action a. By the assumption that the set $\mathcal{T}$ is unambiguous, a given occurrence of an action can be comprised in at most one transaction. Since by assumption transaction τ is present in $\mathcal{A}_t(\sigma[..i+1])$ we have $\sigma[..i+1] \sqsubseteq \mathcal{A}_t(\sigma[..i+1])$.

(2) In all other cases, the automaton enters state $\langle \sigma_o^i, \sigma_s^i; a \rangle$. Since after this transition $\sigma_o^{i+1} = \sigma_o^i$, we have $\hat{\mathcal{P}}_\mathcal{T}(\sigma_o^{i+1})$ from the induction hypothesis. Likewise, since a is appended to both σ_s^i and $\sigma[..i]$, it flows immediately from the induction hypothesis that σ_s^{i+1} is a suffix of $\sigma[..i+1]$. Finally, to show that $\sigma[..i+1] \sqsubseteq \sigma_o^{i+1}$, we must show that every transaction present in $\sigma[..i+1]$ is also present in σ_o^{i+1}. However, since by the induction hypothesis $\sigma[..i] \sqsubseteq \sigma_o[..i]$, any transaction present in $\sigma[..i]; a$ but absent from σ_o^i necessarily comprises a suffix of $\sigma[..i]; a$ and includes a. Yet, we assumed in this case that no such suffix exists. Thus we have $\sigma[..i+1] \sqsubseteq \sigma_o^{i+1}$.

Since the invariant INV holds at each step, we are sure that the output of a *valid* sequence exhibits infinitely many valid prefixes. Since transactional properties are a subset of renewal properties, it follows that the output is itself valid. Likewise, the invariant ensures that any infinite input sequence exhibits infinitely many prefixes for which the corresponding output is higher or equal on the preorder as the input. By definition 1 this ensures that $\sigma \sqsubseteq \mathcal{A}_t(\sigma)$. □

PROPOSITION 5.5 Let R be an enabling relation, defining an assured pipeline policy $\hat{\mathcal{P}}$ over a set of objects O and a set of transformations S. The automaton $\mathcal{A}_{ap}^t$ effectively$_=$ enforces $\hat{\mathcal{P}}_R$.

PROOF. By definition 4.2, we have that the automaton $\mathcal{A}_{ap}^t$ effectively$_=$ enforces the property iff $\forall \sigma \in \Sigma^\infty : \mathcal{A}_{ap}^t(\sigma) \in \hat{\mathcal{P}}_R \wedge \hat{\mathcal{P}}_R(\sigma) \Rightarrow \mathcal{A}_{ap}^t(\sigma) = \sigma$.

Recall that the output of $\mathcal{A}^t_{ap}$ is the concatenation of actions that are output after each transition and that $\hat{\mathcal{P}}_R(\sigma) = \forall i \in \mathbb{N} : \sigma_i = \langle s, o\rangle \Rightarrow ((s = create \vee \exists j \in \mathbb{N} : j < i \wedge \sigma_j = \langle s', o\rangle \wedge \langle s', s\rangle \in R) \wedge (\nexists k \in \mathbb{N} : k < i : \sigma_i = \sigma_k))$. Informally, this property states that each action σ_i in σ does not introduce redundancy with preceding actions in $\sigma[..i-1]$, and that its immediate predecessor in R is present in the prefix $\sigma[..i-1]$, unless the action is a $create$ action.

We begin by showing that $\forall \sigma \in \Sigma^\infty : \hat{\mathcal{P}}_R(\mathcal{A}^t_{ap}(\sigma))$.

Let $\sigma \in \Sigma^\infty$ be any input sequence, let $\sigma[..i]$ be the prefix of σ processed so far, let σ^i_o stand for $\mathcal{A}^t_{ap}(\sigma[..i])$ and finally let q_i stand for the state reached after step i. We show that the automaton maintains the following invariant at each step i. INV(i)= $\hat{\mathcal{P}}_R(\sigma^i_o) \wedge ((q_i = acts(\sigma^i_o)) \vee q_i$ is undefined).

The invariant holds initially since $q_0 = \emptyset \wedge \sigma^0_o = \epsilon$, and by definition we have $\hat{\mathcal{P}}_R(\epsilon)$. Let us assume that INV(i) holds, and let $a = \sigma_{i+1} = \langle s, o\rangle$ be the next input action. We show that INV($i+1$) = $\hat{\mathcal{P}}_R(\sigma^{i+1}_o) \wedge ((q_{i+1} = acts(\sigma^{i+1}_o)) \vee q_{i+1}$ is undefined) holds.

There are two cases to consider.

(1) The condition $(s = create \wedge \langle create, o\rangle \notin q_i) \vee (\exists s' \in S \langle s', o\rangle \in q_i \wedge \langle s', s\rangle \in R \wedge \langle s, o\rangle \notin q_i)$ holds. In this case, the automaton outputs the action a, thus $\sigma^{i+1}_o = \sigma^i_o; a$. Since by assumption $\hat{\mathcal{P}}_R(\sigma^i_o)$ the output sequence $\sigma^i_o; a$ satisfies $\hat{\mathcal{P}}_R$ as by the condition imposed in this case, the action $\langle s, o\rangle$ does not introduce redundancy and either $s = create$ or it's predecessor is in $q_i = acts(\sigma^i_o)$. Thus $\hat{\mathcal{P}}_R(\sigma^{i+1}_o)$. Furthermore, since we have $q_i = acts(\sigma^i_o)$ and after step $i+1$ the state is $q_{i+1} = q_i \cup \{a\}$ and $\sigma^{i+1}_o = \sigma^i_o; a$, we have $q_{i+1} = acts(\sigma^{i+1}_o)$.

(2) The condition $(s = create \wedge \langle create, o\rangle \notin q_i) \vee (\exists s' \in S \langle s', o\rangle \in q_i \wedge \langle s', s\rangle \in R \wedge \langle s, o\rangle \notin q_i)$ does not hold. In this case the execution is aborted leaving the output unchanged and q_{i+1} undefined. We thus have $\hat{\mathcal{P}}_R(\sigma^{i+1}_o)$, since the automaton aborts we have $\sigma^i_o = \sigma^{i+1}_o = \mathcal{A}^t_{ap}(\sigma)$ thus $\hat{\mathcal{P}}_R(\mathcal{A}^t_{ap}(\sigma))$.

For a finite sequence σ, $\hat{\mathcal{P}}_R(\mathcal{A}^t_{ap}(\sigma))$ follows immediately from the invariant above. For infinite sequences, the invariant ensures that the output of the automaton exhibits infinitely many valid prefixes. Since $\hat{\mathcal{P}}_R$ is a renewal property, such an output is valid.

Next, we prove that if the input sequence σ is valid, the automaton's output is syntactically equal to σ by an induction on the length of σ.

At step 0, $\sigma[..0] = \epsilon$, and $\mathcal{A}^t_{ap}(\sigma[..0]) = \epsilon$. Assume that at step i, $\mathcal{A}^t_{ap}(\sigma[..i]) = \sigma[..i]$ and that $\sigma_{i+1} = a$. Since $\hat{\mathcal{P}}_R(\sigma)$, we have $\forall i \in \mathbb{N} : \sigma_i = \langle s_i, o_i\rangle \Rightarrow ((s_i = create \vee \exists j \in \mathbb{N} : j < i \wedge \sigma_j = \langle s', o_i\rangle \wedge \langle s', s_i\rangle \in R) \wedge (\nexists k \in \mathbb{N} : k < i : \sigma_i = \sigma_k))$.

This implies that σ_{i+1} does not introduce redundancy and that the action either has its predecessor present in $\sigma[..i]$ or it is a $create$ action. Thus according to the definition of the transition function, the automaton concatenates a to the output σ^i_o. Since $\sigma[..i] = \sigma^i_o$, after step $i+1$, we have $\sigma[..i+1] = \sigma^i_o; \sigma_{i+1} = \sigma^{i+1}_o$. We conclude that the actions of the sequence σ are output as they are read without any change. Thus $\mathcal{A}^t_{ap}(\sigma) = \sigma$

We can conclude that $\mathcal{A}^t_{ap}$ effectively$_=$ enforces $\hat{\mathcal{P}}_R$. □

PROPOSITION 5.7 Let R be an enabling relation, that defines an assured pipeline policy $\hat{\mathcal{P}}_R$ over a set of actions $E = S \times O$ and let $\sqsubseteq$ be a preorder over the sequences of E^∞ defined such that $\forall \sigma, \sigma' \in E^\infty : \sigma \sqsubseteq \sigma' \Leftrightarrow valid_R(\sigma) \subseteq valid_R(\sigma')$. The automaton $\mathcal{A}^s_{ap}$ correctively$_\sqsubseteq$ enforces $\hat{\mathcal{P}}_R$.

PROOF. By definition 4.4, we have that the automaton $\mathcal{A}^s_{ap}$ correctively$_\sqsubseteq$ enforces $\hat{\mathcal{P}}_R$ iff $\forall \sigma \in \Sigma^\infty : \hat{\mathcal{P}}_R(\mathcal{A}^s_{ap}(\sigma)) \wedge \sigma \sqsubseteq \mathcal{A}^s_{ap}(\sigma)$.

Let $\sigma \in \Sigma^\infty$ be any input sequence, let $\sigma[..i]$ be the prefix of σ processed so far, let σ_o^i stand for $\mathcal{A}_{ap}^s(\sigma[..i])$ and finally let q_i stand for the state reached after step i. We show that the automaton maintains the invariant INV(i)= $\hat{\mathcal{P}}_R(\sigma_o^i) \wedge q_i = acts(\sigma_o^i) \wedge \sigma[..i] \sqsubseteq \sigma_o^i$. The invariant holds initially since $\sigma[..0] = \sigma_o^0 = \epsilon$, $\hat{\mathcal{P}}_R(\epsilon)$, $q_i = \emptyset$ and $\epsilon \sqsubseteq \epsilon$. Let us assume that INV(i) holds and let $a = \langle s, o\rangle$ be the next input action, meaning $\sigma[..i+1] = \sigma[..i]; a$. We show that INV($i+1$) holds.

There are two cases to consider.

(1) $(s = create \wedge \langle create, o\rangle \notin q_i) \vee (\exists s' \in S : \langle s', o\rangle \in q_i \wedge \langle s', s\rangle \in R \wedge \langle s, o\rangle \notin q_i)$ holds. This is the first case of the transition function δ_s. In this case, the automaton outputs the action a. Thus the entire output sequence at that point is $\sigma_o^i; a$. By assumption $\hat{\mathcal{P}}_R(\sigma_o^i)$ and by the definition of $\hat{\mathcal{P}}_R$ we conclude that the output sequence $\sigma_o^i; a$ satisfies $\hat{\mathcal{P}}_R$. By the condition imposed in this case, the action $\langle s, o\rangle$ does not introduce redundancy and it is either $s = create$ or it has an immediate predecessor in $q_i = acts(\sigma_o^i)$. Thus $\hat{\mathcal{P}}_R(\sigma_o^{i+1})$. We have $q_i = acts(\sigma_o^i)$ since after step $i+1$, the reached state is $q_{i+1} = q_i \cup \{a\}$ and $\sigma_o^{i+1} = \sigma_o^i; a$, thus $q_{i+1} = acts(\sigma_o^{i+1})$. Finally we have $\sigma[..i] \sqsubseteq \sigma_o^i$. Since $\sqsubseteq$, $\sigma[..i]; a \sqsubseteq \sigma_o^i; a$ is equivalent to $\mathit{valid}_R(\sigma_o^i) \subseteq \mathit{valid}_R(\sigma_o^i; a)$ and we know that action a is valid with respect to the prefix σ_o^i, we can conclude That $\sigma[..i+1] \sqsubseteq \sigma_o^{i+1}$.

(2) $(s = create \wedge \langle create, o\rangle \notin q_i) \vee (\exists s' \in S : \langle s', o\rangle \in q_i \wedge \langle s', s\rangle \in R \wedge \langle s, o\rangle \notin q_i)$ does not hold. This is the second case of the transition function δ_s. In this case, the automaton outputs ϵ, thus leaving the output sequence unchanged and thus $\sigma_o^{i+1} = \sigma_o^i$. Likewise, the state q is left unchanged and $q_{i+1} = q_i = acts(\sigma_o^{i+1})$. Finally, by assumption $\sigma[..i] \sqsubseteq \sigma_i^o$ and we have by the condition of this case $\sigma[..i]; a \notin \hat{\mathcal{P}}_R$. Thus $\mathit{valid}_R(\sigma[..i]) = \mathit{valid}_R(\sigma[..i+1])$ leading to $\mathit{valid}_R(\sigma[..i+1]) \subseteq \mathit{valid}_R(\sigma_o^{i+1})$.

Thus INV($i+1$) holds in both cases. For any finite sequence σ the invariant INV is sufficient to ensure the respect of proposition 5.7. For infinite sequences, the invariant ensures that the output exhibits infinitely many valid prefixes. Since the assured pipeline property is a renewal property the output is valid by definition. Likewise, the invariant ensures that any infinite input sequence exhibits infinitely many prefixes for which the corresponding output is higher or equal to the input on the preorder . By definition 1 this ensures that $\sigma \sqsubseteq \mathcal{A}_s(\sigma)$. $\square$

PROPOSITION 5.9 Let R be an enabling relation, defining an assured pipeline policy $\hat{\mathcal{P}}_R$ over a set of actions $E = S \times O$, and let $\sqsubseteq$ be a preorder over the sequences of E^∞ defined such that $\forall \sigma, \sigma' \in E^\infty : \sigma \sqsubseteq \sigma' \Leftrightarrow \mathit{valid}_R(\sigma) \subseteq \mathit{valid}_R(\sigma')$. The automaton $\mathcal{A}_{ap}^e$ correctively$_\sqsubseteq$ enforces $\hat{\mathcal{P}}_R$.

PROOF. By definition 4.4, we have the automaton $\mathcal{A}_{ap}^e$ correctively$_\sqsubseteq$ enforces the property iff $\forall \sigma \in \Sigma^\infty : \mathcal{A}_{ap}^e(\sigma) \in \hat{\mathcal{P}} \wedge \sigma \sqsubseteq \mathcal{A}_{ap}^e(\sigma)$. The first conjunct requires that the automaton only output valid sequences, while the second imposes that the output is higher or equal to the input on the preorder.

Let $\sigma \in \Sigma^\infty$ be any input sequence, let $\sigma[..i]$ be the prefix of σ processed so far, let σ_o^i stand for $\mathcal{A}_{ap}^e \sigma[..i])$ and finally let q_i stand for the state reached after step i.

We show that the automaton maintains the invariant INV(i)= $\hat{\mathcal{P}}_R(\sigma_o^i) \wedge q_i = acts(\sigma_o^i) \wedge \sigma[..i] \sqsubseteq \sigma_o^i$. The invariant holds initially since $\sigma[..0] = \sigma_o^0 = \epsilon$, $\hat{\mathcal{P}}_R(\epsilon)$, $q_i = \emptyset$ and $\epsilon \sqsubseteq \epsilon$. Let us assume that INV(i) holds and let $a = \langle s, o\rangle$ be the next input action, meaning $\sigma[..i+1] = \sigma[..i]; a$. We show that INV($i+1$) holds.

There are three cases to consider, which correspond to the three cases of the automaton's transition function.

(1) $s = create \wedge \langle create, o\rangle \notin q_i) \vee (\exists s' \in S : \langle s', o\rangle \in q_i \wedge \langle s', s\rangle \in R \wedge \langle s, o\rangle \notin q_i)$ holds. This is the first case of the transition function δ_e, in which $\mathcal{A}_{ap}^e$ outputs a in lockstep with the output

of the program. In this case, the automaton $\mathcal{A}^e_{ap}$ behaves identically as $\mathcal{A}^s_{ap}$. The proof of the invariant thus proceeds identically as it did in the case for proposition 5.7 above.

(2) $(\langle s, o\rangle \notin q_i \wedge (s \neq create) \wedge (\forall s' \in S : \langle s', s\rangle \in R \Rightarrow \langle s', o\rangle \notin q_i))$ holds. In this case, $\sigma[..i]; a$ is invalid because the immediate predecessor s in R is not in $\sigma[..i]$. Note that, by the definition of R and because $s \neq create$ in this case, there must exists exactly one subject $s' \in S$ such that $\langle s', s\rangle \in R$. Let $pred : S \to S$ be the partial function defined as $pred(s) = s'$ iff $(s', s) \in R$.
Instead of rejecting the action $a = \langle s, o\rangle$, the automaton computes the path $\tau = path(a, q_i)$ of actions to be output after σ^i_o. It is easy to see, by the definition of $path$ that $\tau = \langle s_1, o\rangle; \langle s_2, o\rangle; \ldots \langle s_n, o\rangle$ with $s_n = s$ and $\forall j \in \mathbb{N} : 1 \leq j < n \Rightarrow \langle s_j, s_{j+1}\rangle \in R \wedge \langle s_j, o\rangle \notin q_i \wedge (s_1 = create \vee \langle pred(s_1), o\rangle \in q_i)$. We also have, by the condition of this case, $\langle s, o\rangle \notin q_i$. Observe that $\forall i, j \in \mathbb{N} : (i \neq j \wedge \tau_i = \langle s_i, o\rangle \wedge \tau_j = \langle s_j, o\rangle) \Rightarrow s_i \neq s_j$, since the graph of the relation R is acyclic.
The output of the automaton in this case is $\sigma^{i+1}_o = \sigma^i_o; \tau$ and the successor state is $q_{i+1} = q_i \cup acts(\tau)$. $\hat{\mathcal{P}}_R(\sigma^{i+1}_o)$ holds, since by construction of τ and by the invariant assumption, each action occurs in σ^{i+1}_o only if it is a $create$ action or if its predecessor in R precedes it in σ^{i+1}_o. Moreover, there are no actions that occur more than once in σ^{i+1}_o. It is clear that $q_{i+1} = acts(\sigma^{i+1}_o)$ since q_i is updated to reflect the fact that τ has been output. To see that $\sigma^i_o; a \sqsubseteq \sigma^{i+1}_o$, it is sufficient to observe that if this case of the transition function is taken, then a is necessarily an invalid action at that point of the execution. Recall that function $valid_R$ upon which the preorder $\sqsubseteq$ is built returns the set of valid actions in a given sequence. Thus we have $valid_R(\sigma[..i]; a) = valid_R(\sigma[..i])$. Since by the induction assumption $\sigma[..i] \sqsubseteq \sigma^i_o$, we have $valid_R(\sigma[..i]) \subseteq valid_R(\sigma^i_o)$ and thus $valid_R(\sigma[..i+1]) \subseteq valid_R(\sigma^i_o)$. Moreover we have $valid_R(\sigma^i_o) \subseteq valid_R(\sigma^{i+1}_o)$, since $valid_R(\sigma^{i+1}_o)$ includes all valid actions of σ^i_o by construction. It follows that $\sigma[..i+1] \sqsubseteq \sigma^{i+1}_o$.

(3) In all other cases, the automaton suppresses a, leaving σ^i_o and q_i unchanged.

Thus INV$(i+1)$ holds in the three cases. For any finite sequence σ the invariant INV is sufficient to ensure the respect of proposition 5.9. For infinite sequences, the invariant ensures that the output exhibits infinitely many valid prefixes. Since the assured pipeline property is a renewal property, it follows that the output will itself be valid. Likewise, the invariant ensures that any infinite input sequence exhibits infinitely many prefixes for which the corresponding output is higher or equal on the preorder as the input. By definition 1 this ensures that $\sigma \sqsubseteq \mathcal{A}^e_{ap}(\sigma)$. $\square$

PROPOSITION 5.11 Let Σ be a set of atomic actions, $\mathcal{A}^t_{ap} \sqsubseteq^{\Sigma^\infty} \mathcal{A}^s_{ap} \sqsubseteq^{\Sigma^\infty} \mathcal{A}^e_{ap}$.

PROOF.

Let us first compare $\mathcal{A}^t_{ap}$ and $\mathcal{A}^s_{ap}$. Let $\sigma \in \Sigma^\infty$ be any input sequence and let $\sigma[..i]$ be the prefix of σ processed so far. We show that the automata maintain the invariant INV(i)= $\mathcal{A}^t_{ap}(\sigma[..i]) \sqsubseteq \mathcal{A}^s_{ap}(\sigma[..i])$ and prove proposition 5.11 using this invariant.

The invariant holds initially because when $\sigma[..0] = \epsilon$, $\mathcal{A}^t_{ap}(\epsilon) = \mathcal{A}^s_{ap}(\epsilon) = \epsilon$. Assume that INV$(i)$ holds and let $a = \langle s, o\rangle$ be the next input action. We show that INV$(i+1)$ holds. There are three cases to consider.

(1) $\sigma[..i]; a \in \hat{\mathcal{P}}_R$. By definition 5.3, this case occurs when $(s = create \wedge \langle create, o\rangle \notin q_i) \vee (\exists s' \in S \langle s', o\rangle \in q_i \wedge \langle s', s\rangle \in R \wedge \langle s, o\rangle \notin q_i)$. When this is the case, $\mathcal{A}^t_{ap}$ and $\mathcal{A}^s_{ap}$ behave identically and the automata output the action a in lockstep with the output of the program. Recall that the preorder $\sqsubseteq$ is defined by function $valid_R$ which returns the set of valid actions in a given sequence. It follows that $valid_R$ grows monotonically as actions are appended to its argument. Thus, $\mathcal{A}^t_{ap}(\sigma[..i]; a) \sqsubseteq \mathcal{A}^s_{ap}(\sigma[..i]; a)$ follows immediately from the assumption that $\mathcal{A}^t_{ap}(\sigma[..i]) \sqsubseteq \mathcal{A}^s_{ap}(\sigma[..i])$.

(2) $\sigma[..i]; a \notin \hat{\mathcal{P}}_R \wedge \sigma[..i] \in \hat{\mathcal{P}}_R$. In this case, $\mathcal{A}^t_{ap}(\sigma[..i])$ is undefined, while $\mathcal{A}^s_{ap}$ suppresses action a. The output of both automata on $\sigma[..i]; a$ is thus the same as it was on $\sigma[..i]$ and that the invariant INV($i+1$) holds thus follows immediately from the assumption that INV(i) holds.

(3) $\sigma[..i] \in \hat{\mathcal{P}}_R$. In all other cases, since $\hat{\mathcal{P}}_R$ is a safety property, then by definition there must exist an action σ_j such that $\sigma[..j-1] \in \hat{\mathcal{P}}_R$ and $\sigma[..j] \notin \hat{\mathcal{P}}_R$. Upon encountering σ_j $\mathcal{A}^t_{ap}$ aborted its execution while $\mathcal{A}^s_{ap}$ suppressed it and continued processing the input. Thus the output of $\mathcal{A}^t_{ap}$ remains unchanged from this point on while that of $\mathcal{A}^s_{ap}$ continues to evolve as new valid actions are encountered. Since the value $valid_R$ grows monotonically as actions are appended to its argument, it follows that INV($i+1$) holds in this case.

Let us now compare $\mathcal{A}^s_{ap}$ and $\mathcal{A}^e_{ap}$. Let $\sigma \in \Sigma^\infty$ be any input sequence and let $\sigma[..i]$ be the prefix of σ processed so far. We show that the automata maintain the invariant INV'(i)= $\mathcal{A}^s_{ap}(\sigma[..i]) \sqsubseteq \mathcal{A}^e_{ap}(\sigma[..i])$. The invariant holds initially since $\mathcal{A}^s_{ap}(\epsilon) = \mathcal{A}^e_{ap}(\epsilon) = \epsilon$. Assume that INV'($i$) holds and let $a = \langle s, o\rangle$ be the next input action. We show that INV'($i+1$) holds. There are three cases to consider:

(1) $(s = create \wedge \langle create, o\rangle \notin q_i) \vee (\exists s' \in S \langle s', o\rangle \in q_i \wedge \langle s', s\rangle \in R \wedge \langle s, o\rangle \notin q_i)$. In this case, both $\mathcal{A}^s_{ap}$ and $\mathcal{A}^e_{ap}$ behave identically and output a immediately. The respect of invariant INV'($i+1$) thus follows immediately from the assumption that INV'(i) holds and the fact that the preorder $\sqsubseteq$ respects the closure property 2.

(2) $(\langle s, o\rangle \notin q \wedge (s \neq create) \wedge (\forall s' \in S : \langle s', s\rangle \in R \Rightarrow \langle s', o\rangle \notin q))$. In this case, $\mathcal{A}^s_{ap}$ suppresses a (second case of the transition function δ_s) while $\mathcal{A}^e_{ap}$ outputs it alongside with any other action needed to make a valid (second case of the transition function δ_e). Once again, since $\sqsubseteq$ is based on a function that computes the set of valid actions in a given sequence, $\mathcal{A}^s_{ap}(\sigma[..i]; a) \sqsubseteq \mathcal{A}^e_{ap}(\sigma[..i]; a)$ follows immediately from the assumption that $\mathcal{A}^s_{ap}(\sigma[..i]) \sqsubseteq \mathcal{A}^e_{ap}(\sigma[..i])$.

(3) In all other cases, neither automata outputs any actions and $\mathcal{A}^s_{ap}(\sigma[..i]; a) \sqsubseteq \mathcal{A}^e_{ap}(\sigma[..i]; a)$ follows trivially from the induction hypothesis.

Proposition 5.11 follows immediately from those two invariants and from the fact that the preorder $\sqsubseteq$ respects the constraint 1. □

PROPOSITION 5.13 Let Σ be a set of atomic actions, $\mathcal{A}^t_{ap} \trianglelefteq^{\Sigma^\infty}_p \mathcal{A}^s_{ap} \trianglelefteq^{\Sigma^\infty}_p \mathcal{A}^e_{ap}$.

PROOF.

Let us first compare $\mathcal{A}^t_{ap}$ and $\mathcal{A}^s_{ap}$. Let $\sigma \in \Sigma^\infty$ be any input sequence and let $\sigma[..i]$ be the prefix of σ processed so far. We show that the automata maintain the invariant INV(i)=$|cs_{\sigma[..i]}(\mathcal{A}^t_{ap}(\sigma[..i]))| \leq |cs_{\sigma[..i]}(\mathcal{A}^s_{ap}(\sigma[..i]))|$, and prove proposition 5.13 using this invariant. The invariant holds initially because when $\sigma[..0] = \epsilon$, $\mathcal{A}^t_{ap}(\epsilon) = \mathcal{A}^s_{ap}(\epsilon) = \epsilon$. Let us assume that INV(i) holds and let $a = \langle s, o\rangle$ be the next input action, meaning $\sigma[..i+1] = \sigma[..i]; a$. We show that INV($i+1$) holds. There are three cases to consider.

(1) $\sigma[..i]; a \in \hat{\mathcal{P}}_R$. By definition 5.3, this case occurs when $(s = create \wedge \langle create, o\rangle \notin q_i) \vee (\exists s' \in S \langle s', o\rangle \in q_i \wedge \langle s', s\rangle \in R \wedge \langle s, o\rangle \notin q_i)$. In this case, the transition functions and thus the behavior of both automata coincide: they both output the same sequence that has been input action by action, in lockstep with the output of the program. Thus $\mathcal{A}^t_{ap}(\sigma[..i]; a) = \mathcal{A}^t_{ap}(\sigma[..i]); a$, $\mathcal{A}^s_{ap}(\sigma[..i]; a) = \mathcal{A}^s_{ap}(\sigma[..i]); a$ and $|cs_\sigma(\mathcal{A}^t_{ap}(\sigma[..i]; a))| \leq |cs_{\sigma[..i]}(\mathcal{A}^s_{ap}(\sigma[..i]; a))|$ follows immediately from the induction hypothesis.

(2) $\sigma[..i]; a \notin \hat{\mathcal{P}}_R \wedge \sigma[..i] \in \hat{\mathcal{P}}_R$. In this case, a causes a violation of the security property. $\mathcal{A}^s_{ap}$'s transition function will suppress a while $\mathcal{A}^t_{ap}$'s transition function is undefined. In both cases, the automaton outputs nothing and thus $\mathcal{A}^s_{ap}(\sigma[..i]; a) = \mathcal{A}^s_{ap}(\sigma[..i])$ and $\mathcal{A}^t_{ap}(\sigma; a) = \mathcal{A}^t_{ap}(\sigma[..i])$. It follows that INV($i+1$) holds from the assumption that INV(i) holds.

(3) $\sigma[..i] \in \hat{\mathcal{P}}_R$. In all other cases, since $\hat{\mathcal{P}}_R$ is a safety property by definition there must exist an action σ_j such that $\sigma[..j-1] \in \hat{\mathcal{P}}_R$ and $\sigma[..j] \notin \hat{\mathcal{P}}_R$. Upon encountering σ_j $\mathcal{A}^t_{ap}$ aborted its execution while $\mathcal{A}^s_{ap}$ suppressed it and continued processing the input. Thus the output of $\mathcal{A}^t_{ap}$ remains unchanged from this point on while that of $\mathcal{A}^s_{ap}$ continues to evolve as new valid actions are encountered. Thus $|cs_{\sigma[..i]}(\mathcal{A}^t_{ap}(\sigma[..i];a))| \leq |cs_{\sigma[..i]}(\mathcal{A}^s_{ap}(\sigma[..i];a))|$.

Let us now compare $\mathcal{A}^s_{ap}$ and $\mathcal{A}^e_{ap}$. Let $\sigma \in \Sigma^\infty$ be any input sequence and let $\sigma[..i]$ be the prefix of σ processed so far. Let INV'(i) stand for the invariant that after having processed $\sigma[..i]$, $|cs_\sigma(\mathcal{A}^s_{ap}(\sigma[..i]))| \leq |cs_{\sigma[..i]}(\mathcal{A}^e_{ap}(\sigma[..i]))|$. The invariant holds initially because when $\sigma = \epsilon$, $\mathcal{A}^s_{ap}(\sigma[..i]) = \mathcal{A}^e_{ap}(\sigma[..i]) = \epsilon$. Assume that INV'($i$) holds and let $a = \langle s, o\rangle$ be the next input action. We show that INV'($i+1$) holds.

There are three cases to consider.

(1) $(s = create \wedge \langle create, o\rangle \notin q_i) \vee (\exists s' \in S \langle s', o\rangle \in q_i \wedge \langle s', s\rangle \in R \wedge \langle s, o\rangle \notin q_i)$. In this case, both automata behave in the same manner and output a. Thus, INV'($i+1$) follows immediately form INV'(i).
(2) $(\langle s, o\rangle \notin q \wedge (s \neq create) \wedge (\forall s' \in S : \langle s', s\rangle \in R \Rightarrow \langle s', o\rangle \notin q))$. In this case, $\mathcal{A}^e_{ap}(\sigma[..i])$ will allow a to be output (second case of the transition function of δ_e) while $\mathcal{A}^s_{ap}(\sigma[..i])$ will suppress a and output nothing. Again, INV'($i+1$) follows immediately form INV'(i).
(3) Finally, in all other cases, both automata will suppress a and output nothing. Since the output of both automata is the same if the input sequence is $\sigma[..i]$ as if it is $\sigma; a$, INV'($i+1$) follows immediately from the assumption that INV'(i) holds.

Proposition 5.13 follows immediately from those two invariants and definition 5.12. □

PROPOSITION 5.16 Let C be a set of conflicts of interests, and let $\hat{\mathcal{P}}_C$ be the corresponding Chinese Wall property. The automaton $\mathcal{A}^t_{cw}$ effectively$_=$ enforces $\hat{\mathcal{P}}_C$.

PROOF. By definition 4.2, the automaton $\mathcal{A}^t_{cw}$ effectively$_=$ enforces the property $\hat{\mathcal{P}}_C$ iff $\forall \sigma \in \Sigma^\infty : \hat{\mathcal{P}}_C(\mathcal{A}^t_{cw}(\sigma)) \wedge \hat{\mathcal{P}}_C(\sigma) \Rightarrow \mathcal{A}^t_{cw}(\sigma) = \sigma$.

We begin by showing that $\forall \sigma \in \Sigma^\infty : \hat{\mathcal{P}}_C(\mathcal{A}^t_{cw}(\sigma))$. Let $\sigma \in \Sigma^\infty$ be any input sequence, let $\sigma[..i]$ be the prefix of σ processed so far and let $q = \sigma^i_o$ stand for the state of $\mathcal{A}^t_{cw}$ reached after processing $\sigma[..i]$. By the definition of $\mathcal{A}^t_{cw}$, σ^i_o also equals the output of $\mathcal{A}^t_{cw}$ after it has processed $\sigma[..i]$. We show that the automaton maintains the invariant INV(i)= $\hat{\mathcal{P}}_C(\sigma^i_o)$. The invariant holds initially since $\sigma^0_o = \epsilon$, the automaton is in state ϵ and $\hat{\mathcal{P}}_C(\epsilon)$. Let us assume that INV(i) holds and let $a = \sigma_{i+1}$ be the next input action. We show that INV($i+1$) holds.

There are two cases to consider.

(1) The condition $(a = \langle \textbf{access}, s, o\rangle \wedge o \notin C_{live(s,q)}) \vee a = \langle \textbf{rel}, s, o\rangle$ holds. In this case, the automaton outputs the sequence $\sigma^i_o; a$. Since by assumption $\hat{\mathcal{P}}_C(\sigma^i_o)$ and the definition of $\hat{\mathcal{P}}_C$ implies that $\forall s \in S : \forall o \in O : \forall j \in \mathbb{N} : \sigma^i_{o_j} = \langle \textbf{access}, s, o\rangle \Rightarrow (\nexists o' \in live(s, \sigma^i_o[..j-1]) : o \in C_{o'})$, it follows that the output sequence satisfies $\hat{\mathcal{P}}_C(\sigma^i_o; a)$. Indeed, by the condition imposed in this case, a is either a release action (which can never cause a violation of the security property) or an access action $\langle \textbf{access}, s, o\rangle$ that does not conflict with any other live action. Thus, the invariant holds after this step.
(2) The condition $(a = \langle \textbf{access}, s, o\rangle \wedge o \notin C_{live(s,q)}) \vee a = \langle \textbf{rel}, s, o\rangle$ does not hold. In this case the execution is aborted leaving the output unchanged, thus satisfying the invariant INV.

For any finite sequence σ the invariant INV is sufficient to ensure the respect of $\hat{\mathcal{P}}_C(\mathcal{A}^t_{cw}(\sigma))$. For infinite sequences, the invariant ensures that the output exhibits infinitely many valid prefixes. Since the Chinese Wall property is a renewal property, it follows that the output of the automaton satisfies the property for any sequence finite or infinite.

Next, we prove that if the input sequence σ is valid, the automaton's output is syntactically equal to σ by an induction on the length of σ.

At step 0, $\sigma[..0] = \epsilon$, and $\mathcal{A}^t_{cw}(\sigma[..0]) = \epsilon$. Assume that at step i, $\mathcal{A}^t_{cw}(\sigma[..i]) = \sigma[..i]$ and that $\sigma_{i+1} = a$. Since $\hat{\mathcal{P}}_C(\sigma)$, we have $\forall s \in S : \forall o \in O : \forall j \in \mathbb{N} : \sigma_j = \langle \textbf{access}, s, o\rangle \Rightarrow (\nexists o' \in live(s, \sigma[..j]) : o \in C_{o'})$ implying that $(\sigma_{i+1} = \langle \textbf{access}, s, o\rangle \wedge o \notin C_{live(s,q=\sigma[1..i])}) \vee \sigma_{i+1} = \langle \textbf{rel}, s, o\rangle$. Thus at step $i+1$, and according to the definition of the transition function, the automaton outputs $\sigma^i_o; a = \sigma[..i]; \sigma_{i+1} = \sigma[..i+1]$. We conclude that the actions of the sequence σ are output as they are read without any change. Thus $\mathcal{A}^t_{cw}(\sigma) = \sigma$

Thus $\mathcal{A}^t_{cw}$ effectively$_=$ enforces $\hat{\mathcal{P}}_C$. $\square$

PROPOSITION 5.18 Let C be a set of conflicts of interests, and let $\hat{\mathcal{P}}_C$ be the corresponding Chinese Wall property. The automaton $\mathcal{A}^s_{cw}$ correctively$_\sqsubseteq$ enforces $\hat{\mathcal{P}}_C$.

PROOF.

By definition 4.4, we have that the automaton $\mathcal{A}^s_{cw}$ correctively$_\sqsubseteq$ enforces the property $\hat{\mathcal{P}}_C$ iff $\forall \sigma \in \Sigma^\infty : \mathcal{A}^s_{cw}(\sigma) \in \hat{\mathcal{P}}_C \wedge \sigma \sqsubseteq \mathcal{A}^s_{cw}(\sigma)$.

Let $\sigma \in \Sigma^\infty$ be any input sequence, let $\sigma[..i]$ be the prefix of σ processed so far and let $q_i = \sigma^i_o$ stand for the state of $\mathcal{A}^s_{cw}$ reached after processing $\sigma[..i]$. By the definition of $\mathcal{A}^s_{cw}$, σ^i_o also equals the output of $\mathcal{A}^s_{cw}$ after it has processed $\sigma[..i]$. We show that the automaton maintains the invariant INV(i)= $\hat{\mathcal{P}}_C(\sigma^i_o) \wedge \sigma[..i] \sqsubseteq \sigma^i_o$. The invariant holds initially since $\sigma^0_o = \epsilon$, the automaton is in state $q_0 = \epsilon$, and $valid_C(\epsilon) = \emptyset$ and by definition we have $\hat{\mathcal{P}}_C(\epsilon) \wedge \epsilon \sqsubseteq \epsilon$. Let us assume that INV(i) holds and let $a = \sigma_{i+1}$ be the next input action. We show that INV($i+1$) holds. There are two cases to consider.

(1) The condition $(a = \langle \textbf{access}, s, o\rangle \wedge o \notin C_{live(s,q_i)}) \vee a = \langle \textbf{rel}, s, o\rangle$ holds. In this case, the automaton outputs the sequence $\sigma^i_o; a$. Since by assumption $\hat{\mathcal{P}}_C(\sigma^i_o)$ and since the definition of $\hat{\mathcal{P}}_C$ implies that $\forall s \in S : \forall o \in O : \forall j \in \mathbb{N} : \sigma^i_{o_j} = \langle \textbf{access}, s, o\rangle \Rightarrow (\nexists o' \in live(s, \sigma^i_o[..j-1]) : o \in C_{o'})$, it is obvious that the output sequence $\sigma^i_o; a = \sigma^{i+1}_o$ satisfies $\hat{\mathcal{P}}_C$. Indeed, by the condition imposed in this case, a is either a release action (which can never cause a violation of the security property) or an access action $\langle \textbf{access}, s, o\rangle$ that does not conflict with any other live action. Thus $\hat{\mathcal{P}}_C(\mathcal{A}^s_{cw}(\sigma[..i+1))$. Furthermore, the preorder used in enforcing this property is based on function $valid_C$ which computes the multiset of valid actions occurring in a given sequence. Thus $\sigma[..i] \sqsubseteq \sigma^i_o \Leftrightarrow valid_C(\sigma[..i]) \subseteq valid_C(\sigma^i_o)$. Note that if A, B and C are multisets then $A \subseteq B \Rightarrow (A \uplus C) \subseteq (B \uplus C)$. Since a is a valid action in this context, we have $valid_C(\sigma[..i]; a) = valid_C(\sigma[..i]) \uplus \{a\}$ and $valid_C(\sigma^i_o; a) = valid_C(\sigma^i_o) \uplus \{a\}$. Thus, $valid_C(\sigma[..i]; a) \subseteq valid_C(\sigma^i_o; a)$ and $\sigma[..i]; a \sqsubseteq \sigma^i_o; a$ follow immediately from the induction assumption that $\sigma[..i] \sqsubseteq \sigma^i_o$. From this we conclude $valid_C(\sigma[..i+1]) \subseteq valid_C(\sigma^{i+1}_o)$ and thus, $\sigma[..i+1] \subseteq \mathcal{A}^s_{cw}(\sigma[..i+1)$.

(2) The condition $(a = \langle \textbf{access}, s, o\rangle \wedge o \notin C_{live(s,q_i)}) \vee a = \langle \textbf{rel}, s, o\rangle$ does not hold. This indicates that a causes a conflict in σ^{i+1}_o. In this case, $\sigma^i_o = \sigma^{i+1}_o = \mathcal{A}^s_{cw}(\sigma[..i+1])$, thus, by the induction assumption, $\hat{\mathcal{P}}_C(\mathcal{A}^s_{cw}(\sigma[..i+1]))$ holds together with $valid_C(\sigma^i_o) = valid_C(\sigma^{i+1}_o)$. Note that $\neg\hat{\mathcal{P}}_C(\sigma^i_o; a)$, implies that $\neg\hat{\mathcal{P}}_C(\sigma[..i]; a)$, since any **access** action present in σ^i_o is also present in $\sigma[..i]$ while $\sigma[..i]$ does not contain any **rel** action not also present in σ^i_o Thus $valid_C(\sigma[..i]; a) = valid_C(\sigma[..i])$ and by the induction assumption, $valid_C(\sigma[..i]) \subseteq valid_C(\sigma^i_o)$ which leads to $valid_C(\sigma[..i+1]) \subseteq valid_C(\sigma^{i+1}_o)$ and $\sigma[..i+1] \sqsubseteq \sigma^{i+1}_o$.

Thus INV(i+1) holds in both cases.

For finite sequences the invariant is sufficient to ensure the respect of proposition 5.18. For infinite sequences, the invariant guarantees that the output exhibits infinitely many valid prefixes. Since the Chinese Wall property is a renewal property, it follows that the output is itself valid. Likewise, the invariant ensures that any infinite input sequence exhibits infinitely many prefixes for which the

corresponding output is higher or equal on the preorder as the input. By definition 1 this ensures that $\sigma \sqsubseteq \mathcal{A}^s_{cw}(\sigma)$. $\square$

PROPOSITION 5.20 Let C be a set of conflicts of interests, and let $\hat{\mathcal{P}}_C$ be the corresponding Chinese Wall property. The automaton $\mathcal{A}^e_{cw}$ correctively$_{\sqsubseteq}$ enforces $\hat{\mathcal{P}}_C$.

PROOF.

By definition 4.4, we have that the automaton $\mathcal{A}^e_{cw}$ correctively$_{\sqsubseteq}$ enforces the property $\hat{\mathcal{P}}_C$ iff $\forall \sigma \in \Sigma^\infty : \mathcal{A}^e_{cw}(\sigma) \in \hat{\mathcal{P}}_C \wedge \sigma \sqsubseteq \mathcal{A}^e_{cw}(\sigma)$.

Let $\sigma \in \Sigma^\infty$ be any input sequence, let $\sigma[..i]$ be the prefix of σ processed so far and let $q_i = \langle \sigma^i_o, \sigma^i_s \rangle$ stand for the state of $\mathcal{A}^e_{cw}$ reached after processing $\sigma[..i]$. By the definition of $\mathcal{A}^e_{cw}$, σ^i_o equals the output of $\mathcal{A}^e_{cw}$ after it has processed $\sigma[..i]$, and σ^i_s stand for the suppressed sequence under observation. We show that the automaton maintains the invariant INV(i)= $\hat{\mathcal{P}}_C(\sigma^i_o) \wedge \sigma[..i] \sqsubseteq \sigma^i_o$. The invariant holds initially since $\sigma^0_o = \epsilon$, the automaton is in state $q_0 = \epsilon$, $\mathit{valid}_C(\epsilon) = \emptyset$ and by definition we have $\hat{\mathcal{P}}_C(\epsilon) \wedge \epsilon \sqsubseteq \epsilon$. Let us assume that INV(i) holds and let $a = \sigma_{i+1}$ be the next input action. We show that INV($i+1$) holds. There are three cases to consider.

(1) The condition $(a = \langle \textbf{access}, s, o \rangle \wedge o \notin C_{live(s,q_i)}) \vee a = \langle \textbf{rel}, s, o \rangle$ holds. This is the first case of the transition function δ_e, in which $\mathcal{A}^e_{cw}$ outputs a in lockstep with the output of the program. In this case, the automaton $\mathcal{A}^e_{cw}$ behaves identically as $\mathcal{A}^s_{cw}$. The proof of the invariant thus proceeds identically as it did in the case of proposition 5.18 above.

(2) The condition $a = \langle \textbf{rel}, s, o \rangle$ holds. In this case, $\sigma^{i+1}_o = \sigma^i_o; a; \tau$, where $\tau = f(\sigma^i_o; a, \sigma^i_s, s, \epsilon)$ is a sequence of **access** actions that become available by the occurrence of action a. These actions are selected by the recursive function f, whose semantics ensure that only actions that do not cause conflicts of interest occur. We thus have $\hat{\mathcal{P}}_C(\sigma^{i+1}_o)$.
On the other hand, we have by assumption $\sigma[..i] \sqsubseteq \sigma^i_o$. We want to prove that $\sigma[..i+1] \sqsubseteq \sigma^{i+1}_o$. By definition, $\sigma[..i] \sqsubseteq \sigma^i_o$ is equivalent to $\mathit{valid}_C(\sigma[..i]) \subseteq \mathit{valid}_C(\sigma^i_o)$ Since $a = \langle \textbf{rel}, o \rangle$, we have $\mathit{valid}_C(\sigma[..i]; a) = \mathit{valid}_C(\sigma[..i+1]) = \mathit{valid}_C(\sigma[..i]) \uplus \{a\}$. On the other hand, by the definition of the function f, $\mathit{valid}_C(\sigma^i_o; a; \tau) = \mathit{valid}_C(\sigma^i_o) \uplus \{a\} \uplus \mathit{acts}(\tau)$. Since $\mathit{valid}_C(\sigma[..i]) \uplus \{a\} \subseteq \mathit{valid}_C(\sigma^i_o) \uplus \{a\} \uplus \mathit{acts}(\tau)$ we can immediately conclude that $\sigma[..i+1] \sqsubseteq \sigma^{i+1}_o$.

(3) In all other cases, the automaton $\mathcal{A}^e_{cw}$ behaves identically as did automaton $\mathcal{A}^s_{cw}$ in the second case of its transition function. The proof of the invariant thus proceeds identically as in the corresponding case of the proof of proposition 5.18.

Thus INV($i+1$) holds for all three cases.

For finite sequences the invariant is sufficient to ensure the respect of proposition 5.18. For infinite sequences, the invariant guarantees that the output exhibits infinitely many valid prefixes. Since the Chinese Wall property is a renewal property, it follows that the output is itself valid. Likewise, the invariant ensures that any infinite input sequence exhibits infinitely many prefixes for which the corresponding output is higher or equal on the preorder as the input. By definition 1 this ensures that $\sigma \sqsubseteq \mathcal{A}^e_{cw}(\sigma)$. $\square$

PROPOSITION 5.21 $\mathcal{A}^t_{cw} \sqsubseteq^{\Sigma^\infty} \mathcal{A}^s_{cw} \sqsubseteq^{\Sigma^\infty} \mathcal{A}^e_{cw}$.

PROOF. Let us first compare $\mathcal{A}^t_{cw}$ and $\mathcal{A}^s_{cw}$. Let $\sigma \in \Sigma^\infty$ be any input sequence and let $\sigma[..i]$ be the prefix of σ processed so far. We show that the automata maintain the invariant INV(i)= $\mathcal{A}^t_{cw}(\sigma[..i]) \sqsubseteq^{\Sigma^\infty} \mathcal{A}^s_{cw}(\sigma[..i])$. The invariant holds initially because when $\sigma[..0] = \epsilon$, $\mathcal{A}^t_{cw}(\epsilon) = \mathcal{A}^s_{cw}(\epsilon) = \epsilon$. Let us assume that INV(i) holds and let $a = \langle s, o \rangle$ be the next input action. We show that INV($i+1$) holds. There are three cases to consider.

(1) $\sigma; a \in \hat{\mathcal{P}}_C$. By definition 5.14, this occurs if the condition $(a = \langle \textbf{access}, s, o \rangle \wedge o \notin C_{live(s,q_i)}) \vee a = \langle \textbf{rel}, s, o \rangle$ holds. In this case, the transition functions and thus the behavior of both automata coincide: they both output the same sequence that has been input, action

by action, in lockstep with the output of the program. Thus $\mathcal{A}^t_{cw}(\sigma[..i];a) = \mathcal{A}^t_{cw}(\sigma[..i]);a$, $\mathcal{A}^s_{cw}(\sigma[..i];a) = \mathcal{A}^s_{cw}(\sigma[..i]);a$ and $\mathcal{A}^t_{cw}(\sigma[..i];a) \sqsubseteq \mathcal{A}^s_{cw}(\sigma[..i];a)$ follow immediately from the induction hypothesis.

(2) $\sigma[..i];a \notin \hat{\mathcal{P}}_C \wedge \sigma[..i] \in \hat{\mathcal{P}}_C$. In this case, $\mathcal{A}^s_{cw}$'s transition function will suppress a while $\mathcal{A}^t_{cw}$'s transition function is undefined. Neither automata outputs an action and thus $\mathcal{A}^s_{cw}(\sigma[..i];a) = \mathcal{A}^s_{cw}(\sigma[..i])$ and $\mathcal{A}^t_{cw}(\sigma;a) = \mathcal{A}^t_{cw}(\sigma[..i])$. It follows that INV($i+1$) holds from the assumption that INV(i) holds.

(3) $\sigma[..i] \notin \hat{\mathcal{P}}_C$. In this case, since $\hat{\mathcal{P}}_R$ is a safety property and $\sigma[..i];a \notin \hat{\mathcal{P}}_R$, then by definition there must exist an action σ_j such that $\sigma[..j-1] \in \hat{\mathcal{P}}_R$ and $\sigma[..j] \notin \hat{\mathcal{P}}_C$. Upon encountering σ_j $\mathcal{A}^t_{ap}$ aborted its execution while $\mathcal{A}^s_{ap}$ suppressed it and continued processing the input. Thus the output of $\mathcal{A}^t_{ap}$ remains unchanged from this point on while that of $\mathcal{A}^s_{ap}$ continues to evolve as new valid actions are encountered. Since the value $valid_R$ grows monotonically as actions are appended to its argument, it follows that INV($i+1$) holds in this case.

Now let us compare $\mathcal{A}^s_{cw}$ and $\mathcal{A}^e_{cw}$. Let $\sigma \in \Sigma^\infty$ be any input sequence and let $\sigma[..i]$ be the prefix of σ processed so far. We show that the automata maintain the invariant INV'(i)= $\mathcal{A}^s_{cw}(\sigma[..i]) \sqsubseteq \mathcal{A}^e_{cw}(\sigma[..i])$. The invariant holds initially because when $\sigma[..0] = \epsilon$, $\mathcal{A}^s_{cw}(\sigma[..i]) = \mathcal{A}^e_{cw}(\sigma[..i]) = \epsilon$. Assume that INV'($i$) holds and let a be the next input action. We show that INV'($i+1$) holds. There are three cases to consider.

(1) $a = \langle \textbf{access}, s, o\rangle \wedge o \notin C_{live(s,q)}$. In this case, a is a valid access request and both automata behave in the same manner and output a. Thus, INV'($i+1$) follows immediately form INV'(i).

(2) a is a release action. In this case,$\mathcal{A}^s_{ap}(\sigma[..i])$ will output a (first case of the transition function of δ_e) while $\mathcal{A}^e_{ap}(\sigma[..i])$ will output a as well as a number of other valid actions. From the proofs of theorems 5.18 and 5.20 above we have that the output of both automata is valid and from the fact the the Chinese Wall property is a safety property, we have that every action present in a valid sequence is itself valid. Since the preorder $\sqsubseteq$ is based upon the multiset of valid actions present in the output, INV'($i+1$) follows immediately from INV'(i).

(3) Finally, in all other cases, both automata will suppress a and output nothing. Since the output of both automata is the same as when the input sequence is σ as it is when the input sequence is $\sigma;a$, INV'($i+1$) follows immediately from the assumption that INV'(i) holds.

Proposition 5.21 follows immediately from those two invariants and the fact that the preorder $\sqsubseteq$ respects the constraint 1. □

PROPOSITION 5.22 $\mathcal{A}^t_{cw} \trianglelefteq^{\Sigma^\infty}_p \mathcal{A}^s_{cw} \trianglelefteq^{\Sigma^\infty}_p \mathcal{A}^e_{cw}$.

PROOF.

This proof proceeds analogously to that of proposition 5.21 above. Let us first compare $\mathcal{A}^t_{ap}$ with $\mathcal{A}^s_{ap}$. Let $\sigma \in \Sigma^\infty$ be any input sequence and let $\sigma[..i]$ be the prefix of σ processed so far. We show that the automata maintain the invariant INV(i)=$|cs_{\sigma[..i]}(\mathcal{A}^t_{cw}(\sigma[..i]))| \leq |cs_{\sigma[..i]}(\mathcal{A}^s_{cw}(\sigma[..i]))|$, and prove proposition 5.22 using this invariant. The invariant holds initially because when $\sigma[..0] = \epsilon$, $\mathcal{A}^t_{cw}(\epsilon) = \mathcal{A}^s_{cw}(\epsilon) = \epsilon$. Let us assume that INV(i) holds and let $a = \langle s, o\rangle$ be the next input action, which means that $\sigma[..i+1] = \sigma[..i];a$. We show that INV($i+1$) holds. There are three cases to consider.

(1) $\sigma;a \in \hat{\mathcal{P}}$. As discussed in the proof of proposition 5.21 above, in this case the behavior of both automata coincide: they both output the same sequence that has been input action by action, in lockstep with the output of the program. Thus $\mathcal{A}^t_{cw}(\sigma;a) = \mathcal{A}^t_{cw}(\sigma);a$, $\mathcal{A}^s_{cw}(\sigma;a) = \mathcal{A}^s_{cw}(\sigma);a$ and $\mathcal{A}^t_{cw}(\sigma;a) \trianglelefteq^{\Sigma^\infty}_p \mathcal{A}^s_{cw}(\sigma;a)$ follow immediately from the induction hypothesis.

(2) $\sigma[..i];a \notin \hat{\mathcal{P}}_C \wedge \sigma[..i] \in \hat{\mathcal{P}}_C$. In this case, $\mathcal{A}^s_{cw}$'s transition function will suppress a while $\mathcal{A}^t_{cw}$'s transition function is undefined. Since both automata's output is unchanged, it follows immediately from the induction hypothesis that INV($i+1$) holds.

(3) $\neg\hat{\mathcal{P}}(\sigma; a)$. As discussed above, this case occurs only if the execution of $\mathcal{A}^t_{cw}$ was aborted at some previous step $j < i$. Since by the induction hypothesis INV(j) holds and since $\mathcal{A}^t_{cw}$'s output is unchanged from the previous step, while that of $\mathcal{A}^s_{cw}$ continues to evolve, INV($i+1$) holds.

Let us now compare $\mathcal{A}^s_{cw}$ and $\mathcal{A}^e_{cw}$. Let $\sigma \in \Sigma^\infty$ be any input sequence and let $\sigma[..i]$ be the prefix of σ processed so far. We show that the automata $\mathcal{A}^s_{cw}$ and $\mathcal{A}^e_{cw}$ maintain the invariant INV'(i)=$\mathcal{A}^s_{cw} \trianglelefteq^{\Sigma^\infty}_p \mathcal{A}^e_{cw}$, and prove proposition 5.22 by induction using this invariant. The invariant holds initially because when $\sigma = \epsilon$, $\mathcal{A}^s_{cw}(\sigma[..i]) = \mathcal{A}^e_{cw}(\sigma[..i]) = \epsilon$. Assume that INV'($i$) holds and let a be the next input action. We show that INV'($i+1$) holds. There are three cases to consider.

(1) $a = \langle$**access**$, s, o\rangle \wedge o \notin C_{live(s,q)}$. In this case, a is a valid access request and both automata behave in the same manner and output a. Thus, INV'($i+1$) follows immediately from INV'(i).
(2) $a = \langle$**rel**$, s, o\rangle$. In this case, $\mathcal{A}^s_{ap}$ outputs a (first case of the transition function of δ_s) while $\mathcal{A}^e_{ap}$ outputs a as well as a number of other valid actions (second case of the transition function of δ_e). Since any action from $\sigma[..i+1]$ output by $\mathcal{A}^s_{ap}$ is also output by $\mathcal{A}^e_{ap}$, that $|cs_{\sigma[..i+1]}(\mathcal{A}^t_{cw}(\sigma[..i+1]))| \leq |cs_{\sigma[..i+1]}(\mathcal{A}^s_{cw}(\sigma[..i+1]))|$.
(3) Finally, in all other cases, both automata will suppress a and output nothing. Since the output of both automata is the same if the input sequence is σ as if it is $\sigma; a$, INV'($i+1$) follows immediately from the assumption that INVi holds.

Proposition 5.22 follows immediately from those two invariants. □

PROPOSITION 5.24 $\mathcal{A}^t_{cw} \trianglelefteq^{\Sigma^\infty}_m \mathcal{A}^s_{cw} \trianglelefteq^{\Sigma^\infty}_m \mathcal{A}^e_{cw}$.

PROOF. We first compare $\mathcal{A}^t_{cw}$ and $\mathcal{A}^s_{cw}$. Both automata keep track of the set of resources that are live (accessed and not yet released) for each subject at any given time. For valid sequences, $\mathcal{A}^t_{cw} \equiv^{\hat{\mathcal{P}}_C}_m \mathcal{A}^s_{cw}$ holds trivially since until an invalid action is encountered, both automata's transition function are identical. Upon encountering an invalid action, $\mathcal{A}^t_{cw}$ aborts immediately, while $\mathcal{A}^s_{cw}$ continues its execution, possibly accessing more resources and recording them while they are live. It follows that $\mathcal{A}^t_{cw} \trianglelefteq^{\Sigma^\infty}_m \mathcal{A}^s_{cw}$.

We now compare $\mathcal{A}^s_{cw}$ and $\mathcal{A}^e_{cw}$. The information recorded by each automata is captured in the current state $q = \langle \sigma_o, \sigma_s \rangle$. Both automata maintain two sequences, one recording the sequence that has been output so far and the other the sequence that has been suppressed. Let $\sigma \in \Sigma^\infty$ be any input sequence, let $\sigma[..i]$ be the prefix of σ processed so far and let $\mathcal{A}.\sigma^i_o$ (resp. $\mathcal{A}.\sigma^i_s$) stand for the value of σ_o (resp. σ_s) for automaton $\mathcal{A}$ after at execution step i. We show that the automata maintain the invariant INV(i)=$|\mathcal{A}^s_{cw}.\sigma^i_o| + |\mathcal{A}^s_{cw}.\sigma^i_s| \leq |\mathcal{A}^e_{cw}.\sigma^i_o| + |\mathcal{A}^e_{cw}.\sigma^i_s|$. The invariant holds initially since when $\sigma[..0] = \epsilon$, $\mathcal{A}^s_{cw}.\sigma^0_o = \mathcal{A}^s_{cw}.\sigma^0_s = \mathcal{A}^e_{cw}.\sigma^0_o = \mathcal{A}^e_{cw}.\sigma^0_s = \epsilon$. Let us assume that INV(i) holds and let a be the next input action. We show that INV($i+1$) holds. There are 3 cases to consider.

(1) $a = \langle$**access**$, s, o\rangle \wedge o \notin C_{live(s,q)}$. In this case, both monitors record the same new information, namely that action a has been accessed. Thus, that INV($i+1$) holds follows immediately from the assumption that INV(i) holds.
(2) $a = \langle$**rel**$, s, o\rangle$. In this case, $\mathcal{A}^s_{cw}$ behaves identically as it did in the previous case, and appends a $\mathcal{A}^s_{cw}.\sigma^i_s$. Upon encountering a **rel** action, $\mathcal{A}^e_{cw}$ may move several actions from $\mathcal{A}^e_{cw}.\sigma^i_s$ to $\mathcal{A}^s_{cw}.\sigma^i_o$ but the only new information that is added to $\mathcal{A}^e_{cw}$'s memory is the occurrence of a. We thus have $|\mathcal{A}^s_{cw}.\sigma^{i+1}_o| + |\mathcal{A}^s_{cw}.\sigma^{i+1}_s| = |\mathcal{A}^s_{cw}.\sigma^i_o| + |\mathcal{A}^s_{cw}.\sigma^i_s| + 1$ and $|\mathcal{A}^e_{cw}.\sigma^{i+1}_o| + |\mathcal{A}^e_{cw}.\sigma^{i+1}_s| = |\mathcal{A}^e_{cw}.\sigma^i_o| + |\mathcal{A}^e_{cw}.\sigma^i_s| + 1$. Thus that $|\mathcal{A}^s_{cw}.\sigma^{i+1}_o| + |\mathcal{A}^s_{cw}.\sigma^{i+1}_s| \leq |\mathcal{A}^e_{cw}.\sigma^{i+1}_o| + |\mathcal{A}^e_{cw}.\sigma^{i+1}_s|$ holds follows immediately from the induction hypothesis.
(3) In all other cases, $\mathcal{A}^s_{cw}$ simply suppresses the input action, leaving its memory footprint unchanged, while $\mathcal{A}^e_{cw}$ records the occurrence of a in $\mathcal{A}^e_{cw}.\sigma_s$. Thus we have $\mathcal{A}^s_{cw} \trianglelefteq^{\Sigma^\infty}_m \mathcal{A}^e_{cw}$.

That $\mathcal{A}^s_{cw} \trianglelefteq^{\Sigma^\infty}_m \mathcal{A}^e_{cw}$ follows immediately from this invariant. □

PROPOSITION 5.27 The automaton $\mathcal{A}_{ga}^{\Sigma^\infty}$ correctively$_\sqsubseteq$ enforces the general availability property.

PROOF. By definition 4.4, the automaton correctively$_\sqsubseteq$ enforces the property $\hat{\mathcal{P}}$ iff $\forall \sigma \in \Sigma^\infty : \mathcal{A}_{ga}^{\Sigma^\infty}(\sigma) \in \hat{\mathcal{P}} \wedge \sigma \sqsubseteq \mathcal{A}(\sigma)$.

Let $\mathcal{R}$ be the set of resources, and $open_r(\tau)$ stand for the sequence of actions over resource r that has been opened but not yet released in a sequence τ. Recall that function $f_r(\tau, \epsilon)$ returns the sequence of actions over a resource r present in τ and that the preorder $\sqsubseteq$ is defined with respect to function $valid(\tau)$, which returns the multiset of valid use actions. A use action present in a sequence τ is valid if it has been acquired before it occurs in τ and if it is subsequently released.

Let $\sigma \in \Sigma^\infty$ be any input sequence, let $\sigma[..i]$ be the prefix of σ processed so far and let $q_i = (\sigma_o^i, \sigma_s^i)$ stand for the state of $\mathcal{A}_{ga}^{\Sigma^\infty}$ reached after processing $\sigma[..i]$. By the definition of $\mathcal{A}_{ga}^{\Sigma^\infty}$, $\sigma_o^i = \mathcal{A}_{ga}^{\Sigma^\infty}(\sigma[..i])$ and σ_s^i is the sequence which the automaton has suppressed. We show that the automaton maintains the invariant INV(i)= $\hat{\mathcal{P}}(\sigma_o^i) \wedge \sigma[..i] \sqsubseteq \sigma_o^i \wedge (\forall r_j \in \mathcal{R} : f_{r_j}(\sigma_s^i, \epsilon) = open_{r_j}(\sigma[..i]))$. The invariant holds initially since when $\sigma_o^0 = \epsilon$ and the automaton is in state $q_0 = \epsilon$, *valid*$(\epsilon) = \emptyset$, $\forall r_j \in \mathcal{R} : open_{r_j}(\epsilon) = f_{r_j}(\epsilon, \epsilon)$, and by definition we have $\hat{\mathcal{P}}(\epsilon) \wedge \epsilon \sqsubseteq \epsilon$. Let us assume that INV(i) holds and let $a = \sigma_{i+1}$ be the next input action and let us also assume that a is an action over some resource r. We show that INV($i+1$) holds. There are three cases to consider.

(1) a is a use action for a resource r that has not yet been acquired. In this case, a is simply suppressed, with $\mathcal{A}_{ga}^{\Sigma^\infty}(\sigma[..i]), \sigma_o^i, \sigma_s^i$ and $open_i(\sigma[..i])$ unchanged. Since the output is unchanged $\sigma_o^{i+1} = \sigma_o^i$ and by assumption we have $\hat{\mathcal{P}}(\sigma_o^{i+1})$ which is the same as $\hat{\mathcal{P}}(\mathcal{A}_{ga}^{\Sigma^\infty}(\sigma[..i+1]))$. Since a is an invalid action in the input sequence at that point, we have *valid*$(\sigma[..i]; a) =$ *valid*$(\sigma[..i])$, thus $\sigma[..i]; a \sqsubseteq \mathcal{A}_{ga}^{\Sigma^\infty}(\sigma[..i]; a)$ holds since, by the induction hypothesis, $\sigma[..i] \sqsubseteq \mathcal{A}_{ga}^{\Sigma^\infty}(\sigma[..i])$. Likewise, $open_r(\sigma[..i]; a) = f_r(\sigma_s^{i+1}, \epsilon)$ follows from the induction hypothesis $open_r(\sigma[..i]) = f_r(\sigma_s^i, \epsilon)$ and the fact that $\sigma_s^i = \sigma_s^{i+1}$.

(2) a is a use action for a resource r that has previously been acquired or a is an ac action. The automaton suppresses action a, $\sigma_s^i = \sigma_s^i; a$ and *valid*$(\sigma[..i])$ is unchanged. Since the output is unchanged we have $\hat{\mathcal{P}}(\mathcal{A}_{ga}^{\Sigma^\infty}(\sigma[..i+1]))$ immediately from the induction hypothesis. Furthermore, $open_r(\sigma[..i]; a) = open_r(\sigma[..i]); a$, and $f_r(\sigma_s^{i+1}, \epsilon) = f_r(\sigma_s^i, \epsilon); a$. By assumption we have $open_r(\sigma[..i]) = f_r(\sigma_s^i, \epsilon)$. Thus $open_r(\sigma[..i+1]) = f_r(\sigma_s^{i+1}, \epsilon)$. Finally, note that since appending an ac action or a use action to the end of a sequence tau cannot affect the value of *valid*(τ) we have *valid*$(\sigma[..i+1]) =$ *valid*$(\sigma[..i])$, and since $\sigma_o^i = \sigma_o^{i+1}$ we have $\sigma[..i+1] \sqsubseteq \sigma_o^{i+1}$.

(3) $a = \langle rel, r\rangle$. In this case $\sigma_o^{i+1} = \sigma_o^i; f_r(\sigma_s^i, \epsilon); a = \mathcal{A}_{ga}^{\Sigma^\infty}(\sigma[..i+1])$. By construction, the function f_r introduces a sequence of actions over r that starts with an $\langle ac, r\rangle$ action followed by $\langle use, r\rangle$ or $\langle ac, r\rangle$ actions over the same r. By the definition of general availability we have $\hat{\mathcal{P}}(f_r(\sigma_s^i, \epsilon); a)$. Furthermore, since $\forall \tau, \tau' \in \Sigma^* : \hat{\mathcal{P}}(\tau) \wedge \hat{\mathcal{P}}(\tau') \Rightarrow \hat{\mathcal{P}}(\tau; \tau')$ and $\hat{\mathcal{P}}(\sigma_o^i)$ we have $\hat{\mathcal{P}}(\sigma_o^{i+1})$. Moreover, note that by definition, the use actions present in $\sigma[..i]$ that are preceded by a corresponding ac action do not affect the value of *valid*$(\sigma[..i])$ until a corresponding rel action is appended to the sequence. Recall that $open_r(\sigma[..i])$ returns a multiset containing these actions and by the induction hypothesis, $open_r(\sigma[..i]) = f_r(\sigma_s^i, \epsilon)$. Thus, since $\sigma_{i+1} = \langle rel, r\rangle$, *valid*$(\sigma[..i+1]) =$ *valid*$(\sigma[..i]) \uplus$ *valid*$(open_r(\sigma[..i]; a))$. Likewise, *valid*$(\sigma_o^{i+1}) =$ *valid*$(\sigma_o^i) \uplus$ *valid*$(f_r(\sigma_s^i, \epsilon); a)$, and since $open_r(\sigma[..i]) = f_r(\sigma_s^i, \epsilon)$, by assumption, *valid*$(\sigma[..i]) \subseteq$ *valid*$(\mathcal{A}_{ga}^{\Sigma^\infty}(\sigma[..i]))$ we have *valid*$(\sigma[..i+1]) \subseteq$ *valid*$(\mathcal{A}_{ga}^{\Sigma^\infty}(\sigma[..i+1]))$ and thus $\sigma[..i+1] \sqsubseteq \mathcal{A}_{ga}^{\Sigma^\infty}(\sigma[..i+1])$. Finally, since $\sigma_s^i; a$ and $open_r(\sigma[..i]; a)$ are simultaneously purged of the same actions over resource r, $open_r(\sigma[..i+1]) = f_r(\sigma_s^{i+1}, \epsilon)$ is also maintained.

For finite sequences the invariant is sufficient to ensure the respect of proposition 5.27. For infinite sequences, the invariant guarantees that the output exhibits infinitely many valid prefixes. This

means that every acquired resource is eventually released. Furthermore, the invariant also ensures that every use action is preceded by an ac action. It follows that the output is itself valid. Likewise, the invariant ensures that any infinite input sequence exhibits infinitely many prefixes for which the corresponding output is higher or equal on the preorder as the input. By definition 1 this ensures that $\sigma \sqsubseteq \mathcal{A}_{ga}^{\Sigma^\infty}(\sigma)$. □

Proposition 5.27 follows immediately from the invariant. □

PROPOSITION 5.29 Let $\mathcal{S} \subseteq \Sigma^\infty$ be a subset of sequences such that $\forall \sigma \in \mathcal{S} \cap \Sigma^\omega : \forall r \in \mathcal{R} : \forall j \in \mathbb{N} : \sigma_j = (ac, r) \Rightarrow \exists k \in \mathbb{N} : k > j \wedge \sigma_k = (rel, r)$. The automaton $\mathcal{A}_{ga}^{fair}$ correctively$_{\sqsubseteq}^{\mathcal{S}}$ enforces the general availability property.

PROOF. By definition 4.4, the automaton $\mathcal{A}_{ga}^{fair}$ correctively$_{\sqsubseteq}^{\mathcal{S}}$ enforces the property $\hat{\mathcal{P}}$ iff $\forall \sigma \in \Sigma^{\mathcal{S}} : \mathcal{A}_{ga}^{fair}(\sigma) \in \hat{\mathcal{P}} \wedge \sigma \sqsubseteq \mathcal{A}_{ga}^{fair}(\sigma)$. We first show that $\forall \sigma \in \Sigma^\infty : \sigma \sqsubseteq \mathcal{A}_{ga}^{fair}(\sigma)$.

Let $live(\tau)$ stand for the set of resources that are live (i.e., acquired and not yet released) at the end of a finite sequence τ and let *UseSet*(τ) be the multiset of use actions present in a sequence τ that involve resources that are live. Let q_i be the automaton state reached after the prefix $\sigma[..i]$ has been processed and let σ_o^i be the sequence output so far. We show that the automaton maintains the invariant INV(i) = $(\sigma[..i] \sqsubseteq \mathcal{A}_{ga}^{fair}(\sigma[..i]) \wedge q_i = live(\sigma[..i]) \wedge$ *UseSet*$(\mathcal{A}_{ga}^{fair}(\sigma[..i])) =$ *UseSet*$(\sigma[..i]) \vee (a = a_{end} \wedge \sigma[..i] \sqsubseteq \mathcal{A}_{ga}^{fair}(\sigma[..i]) \wedge q_i = live(\sigma[..i])))$. We then use this invariant to prove proposition 5.29. The invariant holds initially since when $\sigma[..0] = \sigma_o^0 = \epsilon$, $q_0 = \emptyset$, no resources have yet been acquired in $\sigma[..0]$, and *UseSet*$(\epsilon) =$ *UseSet*(ϵ). Assume that INV(i) holds and let $a = \sigma_{i+1}$ be the next input action and let us also assume that a is an action over some resource r. We show that INV($i+1$) holds. There are 5 cases to consider.

(1) $a = \langle use, r\rangle \wedge r \notin q_i$. In this case, the automaton does not output anything, and the set q_i is unchanged (first case of the transition function). Since by the condition imposed in this case a is not live in $\sigma[..i]$, the value of function *valid*$(\sigma[..i]; a)$ is the same as that of *valid*$(\sigma[..i])$. Thus, $\sigma[..i]; a \sqsubseteq \mathcal{A}_{ga}^{fair}(\sigma[..i]; a)$ follows immediately from the assumption that $\sigma[..i] \sqsubseteq \mathcal{A}_{ga}^{fair}(\sigma[..i])$. Furthermore, the set of live resources is unchanged and $q_i = live(\sigma[..i]; a)$ holds trivially from the induction assumption. Likewise, since a is not a use action for a live resource, *UseSet*$(\sigma[..i]) =$ *UseSet*$(\sigma[..i]; a)$ and since $\mathcal{A}_{ga}^{fair}(\sigma[..i]; a) = \mathcal{A}_{ga}^{fair}(\sigma[..i])$, *UseSet*$(\mathcal{A}_{ga}^{fair}(\sigma[..i]; a)) =$ *UseSet*$(\sigma[..i]; a)$ follows immediately from the induction hypothesis.

(2) $a = \langle ac, r\rangle$. In this case, the automaton $\mathcal{A}_{ga}^{fair}$ outputs action a and adds r to q_i (second case of the transition function). The occurrence of this action implies that r is now live in $\sigma[..i]; a$ and since $q_i = live(\sigma[..i])$ we have $q_{i+1} = live(\sigma[..i]; a)$. Function *valid* upon which the preorder $\sqsubseteq$ is based is not affected by the affixation of an ac action to the sequence, thus $\sigma[..i]; a \sqsubseteq \mathcal{A}_{ga}^{fair}(\sigma[..i]; a)$ follows immediately from the assumption that $\sigma[..i] \sqsubseteq \mathcal{A}_{ga}^{fair}(\sigma[..i])$ holds. Finally, since a is not a use action for a live resource, *UseSet*$(\sigma[..i]) =$ *UseSet*$(\sigma[..i]; a)$ and *UseSet*$(\mathcal{A}_{ga}^{fair}(\sigma[..i])) =$ *UseSet*$(\mathcal{A}_{ga}^{fair}(\sigma[..i]; a))$. Thus *UseSet*$(\mathcal{A}_{ga}^{fair}(\sigma[..i]; a)) =$ *UseSet*$(\sigma[..i]; a)$.

(3) $a = \langle rel, r\rangle$. In this case, the automaton $\mathcal{A}_{ga}^{fair}$ outputs a and removes r from q_i. Since r is the only resource live in $\sigma[..i]$ that is no longer live in $\sigma[..i]; a$, we can see that $q_{i+1} = q_i \backslash \{r\} = live(\sigma[..i+1])$ holds. Furthermore, by the induction hypothesis, we have *valid*$(\sigma[..i]) \sqsubseteq$ *valid*$(\mathcal{A}_{ga}^{fair}(\sigma[..i]))$. After action a is output, *valid*$(\sigma[..i]; a)$ contains every action present in *valid*$(\sigma[..i])$ as well as every use action over resource r that was live in $\sigma[..i]$. Formally, *valid*$(\sigma[..i]; a) =$ *valid*$(\sigma[..i]) \uplus \{\langle use, r\rangle \mid \langle use, r\rangle \in$ *UseSet*$(\sigma[..i])\}$. Likewise, *valid*$(\mathcal{A}(\sigma[..i]; a)) =$ *valid*$(\mathcal{A}_{ga}^{fair}(\sigma[..i])) \uplus \{\langle use, r\rangle \mid \langle use, r\rangle$*UseSet*$(\mathcal{A}_{ga}^{fair}(\sigma[..i]))$. Since by the induction hypothesis *UseSet*$(\mathcal{A}_{ga}^{fair}(\sigma[..i])) =$ *UseSet*$(\sigma[..i])$, we have *valid*$(\sigma[..i]; a) \subseteq$ *valid*$(\mathcal{A}_{ga}^{fair}(\sigma[..i]; a))$ and thus $\sigma[..i]; a \sqsubseteq \mathcal{A}_{ga}^{fair}(\sigma[..i]; a)$. Finally, *UseSet*$(\sigma[..i])$ and

UseSet$(\mathcal{A}_{ga}^{fair}(\sigma[..i]))$ are both updated by the removal of the same actions, hence they remain identical.

(4) $a = \langle use, r\rangle \wedge r \in q_i$. In this case the automaton $\mathcal{A}_{ga}^{fair}$ outputs a, leaving q_i unchanged. Since the presence of this action in the input sequence does not affect the set of resources that are live, the fact that $q_{i+1} = live(\sigma[..i]; a)$ follows immediately from the induction hypothesis. Likewise, since the preorder is based on the set of use actions that have both been precedingly acquired and subsequently released and r has not yet been released, the value of *valid*$(\sigma[..i]; a)$ is the same as that of *valid*$(\sigma[..i])$ and the invariant holds by assumption. Finally, in this case, *UseSet*$(\sigma[..i]; a) =$ *UseSet*$(\sigma[..i]); a$. Since r has previously been acquired in both the input and the output sequence, we have that *UseSet*$(\mathcal{A}_{ga}^{fair}(\sigma[..i]; a)) =$ *UseSet*$(\mathcal{A}_{ga}^{fair}(\sigma[..i]); a)$ and *UseSet*$(\sigma[..i]; a) =$ *UseSet*$(\sigma[..i]); a$. Thus *UseSet*$(\mathcal{A}_{ga}^{fair}(\sigma[..i]; a)) =$ *UseSet*$(\sigma[..i]; a)$ follows from the induction hypothesis.

(5) $a = a_{end}$. In this case the automaton outputs the sequence $\tau =$ of all release actions of live resources, as computed by $f(q_i)$. After the automaton outputs $f(q_i)$, $q_{i+1} = \emptyset$ and $live(\sigma[...i{+}1]) = \emptyset$, thus $q_{i+1} = live(\sigma[..i{+}1])$. We have by assumption, $\sigma[..i] \sqsubseteq \mathcal{A}_{ga}^{fair}(\sigma[..i])$ which is equivalent to *valid*$(\sigma[..i]) \subseteq$ *valid*$(\mathcal{A}_{ga}^{fair}(\sigma[..i]))$. After τ has been output, every action present in the output sequence that is also listed in *UseSet*$(\sigma[..i])$ becomes valid but the corresponding actions in the input sequence remain invalid, thus *valid*$(\sigma[..i+1]) =$ *valid*$(\sigma[..i])$ and *valid*$(\mathcal{A}_{ga}^{fair}(\sigma[..i+1])) =$ *valid*$(\mathcal{A}_{ga}^{fair}(\sigma[..i])) \uplus$ *UseSet*$(\sigma[..i])$. From these observations, we can conclude $\sigma[..i+1] \sqsubseteq \mathcal{A}_{ga}^{fair}(\sigma[..i+1])$.

Thus INV$(i+1)$ holds in all cases. For any finite sequence σ the invariant is sufficient to ensure the respect of $\sigma \sqsubseteq \mathcal{A}_{ga}^{fair}(\sigma)$. For an infinite sequence σ, the preorder guarantees that $\forall \sigma' \preceq \sigma : \sigma' \sqsubseteq \mathcal{A}_{ga}^{fair}(\sigma')$. This together with the fact that the preorder $\sqsubseteq$ respects the closure property 2 is sufficient to show that $\sigma \sqsubseteq \mathcal{A}_{ga}^{fair}(\sigma)$ holds for all infinite sequences σ.

We prove that the output of $\mathcal{A}_{ga}^{fair}$ is always valid by contradiction. Let τ be an input sequence. If $\mathcal{A}_{ga}^{fair}(\tau)$ is invalid, it is either because a resource is acquired and never released, or because a resource is used without having previously been acquired. The former case is impossible by the assumption that the input set is fair. Furthermore, the induction argument above shows that $\mathcal{A}_{ga}^{fair}$ only outputs use actions for resources that have previously been acquired (fourth case of the transition function). The output of $\mathcal{A}_{ga}^{fair}$ is thus necessarily valid. □

PROPOSITION 5.31 The automaton $\mathcal{A}_{ga}^{\Sigma^*}$ correctively$_{\sqsubseteq}^{\Sigma^*}$ enforces the general availability property.

PROOF. By definition 4.4, the automaton $\mathcal{A}_{ga}^{\Sigma^*}$ correctively$_{\sqsubseteq}^{\Sigma^*}$ enforces the property $\hat{\mathcal{P}}$ iff $\forall \sigma \in \Sigma^* : \mathcal{A}_{ga}^{\Sigma^*}(\sigma) \in \hat{\mathcal{P}} \wedge \sigma \sqsubseteq \mathcal{A}_{ga}^{\Sigma^*}(\sigma)$.

We first show that the output of $\mathcal{A}_{ga}^{\Sigma^\infty}$ is valid. By assumption, every sequence is finite and an automaton processing an input sequence will eventually encounter the final action token a_{end}. Let $\sigma; a_{end} \in \Sigma^*$ be an input sequence and q_i be the automaton state reached after the prefix $\sigma[..i]$ has been processed. We begin by showing that after having processed any prefix $\sigma[..1]$, the automaton maintains the following invariant INV(i)= the current state q_i records every resource that has been acquired and not yet released, and use actions are present in $\mathcal{A}_{ga}^{\Sigma^*}(\sigma[..1])$ if they have previously been acquired and not yet released.

The invariant holds initially since when $\sigma = \epsilon$ and $q = \emptyset$ and $\mathcal{A}_{ga}^{\Sigma^*}(\epsilon) = \epsilon$. Assume that INV($i$) holds and let a be the next input action. We show that INV$(i+1)$ holds. There are four cases to consider:

(1) $a = \langle use, r\rangle \wedge \langle ac, r\rangle \notin q$. In this case, a is a use action which is immediately acquired and output. Since state q_i is updated to reflect the fact that r has been acquired and since the action

a present in the input sequence $\sigma[..1]$ is immediately acquired and output, the fact that INV(i) holds implies that INV($i+1$) also holds.

(2) $a = \langle ac, r\rangle$ In this case a is an acquire action which is immediately output. Since the state q_i is updated to reflect the newly acquired resource and since a is not a use action, the fact that INV(i) holds implies that INV($i+1$) also holds.

(3) $a = \langle rel, r\rangle$. In this case, a is a release action, which is deleted from q_i. As with the previous case, since q_i is updated and no new use actions are present in the output sequence, the fact that INV($i+1$) holds follows immediately from the assumption that INV(i) also holds.

(4) $a = \langle use, r\rangle \wedge r \in q_i$. In this case, a is a use action for a resource already present in q_i. Since by assumption, the state q_i, after the automaton has processed σ, records every resource that has been acquired and not yet released, we find that a is output while preserving the invariant. Further, as no new resources are acquired or released and q_i is unchanged, we have that INV($i+1$) holds.

Thus the invariant holds in all cases.

The induction argument above proves that when the automaton encounters the final action of the input sequence a_{end}, every use action present in the output sequence is preceded by a corresponding acquire resource and every resource that has been acquired and not yet released is recorded in the current state q_i. Upon encountering a_{end}, the transition function δ_{Σ^*} will release every resource listed in q_i. This, together with the invariant, ensure that the output of $\mathcal{A}_{ga}^{\Sigma^*}$ is valid.

That $\sigma \sqsubseteq \mathcal{A}_{ga}^{\Sigma^*}(\sigma)$ follows immediately from the fact that the automaton's transition function does not allow it to suppress any ac action present in the input sequence and from the fact that the output of $\mathcal{A}_{ga}^{\Sigma^*}$ is valid. Indeed, the preorder $\sqsubseteq$ used to enforce this property is based on the multiset of valid use actions occurring in a given sequence, and since $\mathcal{A}_{ga}^{\Sigma^*}$ never deletes a use action present in the input sequence and always outputs a valid sequence, then for every use action present in σ, there is a corresponding use action in $\mathcal{A}_{ga}^{\Sigma^*}(\sigma)$ that is necessarily valid. □

PROPOSITION 5.33 $\mathcal{A}_{ga}^{\Sigma^*} \lhd_v \mathcal{A}_{ga}^{fair} \lhd_v \mathcal{A}_{ga}^{\Sigma^\infty}$.

PROOF.

From propositions 5.27, 5.29 and 5.31, we have $\mathcal{A}_{ga}^{\Sigma^*} \trianglelefteq_v \mathcal{A}_{ga}^{fair} \trianglelefteq_v \mathcal{A}_{ga}^{\Sigma^\infty}$. Further, observe that $\mathcal{A}_{ga}^{fair}$ cannot enforce the property if the input set contains a sequence from $\Sigma^\infty \backslash \Sigma^{fair}$. In this case, the input sequence contains ac actions unmatched by any rel action. However, $\mathcal{A}_{ga}^{fair}$ systematically outputs any ac action it encounters, and only outputs a rel action if it is present in the input sequence or if the execution terminates. Yet sequences in $\Sigma^\infty \backslash \Sigma^{fair}$ are infinite sequences in which some resources are acquired and never released. It follows that if the input is a sequence from this set, the output of $\mathcal{A}_{ga}^{fair}$ would not be sound.

Likewise, observe that $\mathcal{A}_{ga}^{\Sigma^*}$ cannot enforce the property if the input contains sequences from Σ^∞. By $\mathcal{A}_{ga}^{\Sigma^*}$'s transition function, acquired resources are only released as the execution terminates which does not occur for sequences in $\mathcal{A}_{ga}^{\Sigma^\infty}$. It follows that $\mathcal{A}_{ga}^{\Sigma^*} \lhd_v \mathcal{A}_{ga}^{fair} \lhd_v \mathcal{A}_{ga}^{\Sigma^\infty}$. □

PROPOSITION 5.34 $\mathcal{A}_{ga}^{fair} \sqsubseteq^{\Sigma^*} \mathcal{A}_{ga}^{\Sigma^\infty} \sqsubseteq^{\Sigma^*} \mathcal{A}_{ga}^{\Sigma^*}$.

PROOF.

First observe that from propositions 5.27, 5.29 and 5.31 we know that the output of all three automata is always valid, which implies that every use action is bracketed by ac and rel actions. It follows that we can compare how corrective each monitor is by contrasting the number of use actions output by each. Note that for all three automata, any use action present in automata's output can be matched to a corresponding use action present in the input. We thus prove the validity of proposition 5.34 by contrasting the behavior of each automaton when faced with a use action.

We first compare $\mathcal{A}_{ga}^{fair}$ and $\mathcal{A}_{ga}^{\Sigma^\infty}$. Let a be a use action present in the input sequence. We show that if a is present in the output of $\mathcal{A}_{ga}^{\Sigma^\infty}$, it is also necessarily present in the output of $\mathcal{A}_{ga}^{fair}$. However, there are cases where the output of $\mathcal{A}_{ga}^{fair}$ can contain one or more use actions not present in the input sequence. $\mathcal{A}_{ga}^{\Sigma^\infty}$. There are three cases to consider.

(1) a is a use action for a resource that has not been acquired. In this instance, the first case of each automaton's transition function permanently suppresses a.
(2) a is a use action for a resource that has both precedingly been acquired but is never released. The automaton $\mathcal{A}_{ga}^{fair}$ will output a as soon as it is encountered, and will output a corresponding release action when the token a_{end} is encountered. Since $\mathcal{A}_{ga}^{\Sigma^\infty}$ only outputs use actions upon encountering the corresponding release action (third case of its transition function), this use action will never be output.
(3) a is a use action that is both precedingly acquired and subsequently released. $\mathcal{A}_{ga}^{fair}$ outputs each such action immediately (fourth case of the transition function), while $\mathcal{A}_{ga}^{\Sigma^\infty}$ will suppress them (second case of the transition function) until the corresponding release action is encountered. Since by assumption a is eventually followed by a release action, $\mathcal{A}_{ga}^{\Sigma^\infty}$ will also eventually output a.

Let us now compare $\mathcal{A}_{ga}^{fair}$ and $\mathcal{A}_{ga}^{\Sigma^*}$. Once again, we will show that every use action present in the output of $\mathcal{A}_{ga}^{fair}$ will also be present in the output of $\mathcal{A}_{ga}^{\Sigma^*}$ while the output of $\mathcal{A}_{ga}^{\Sigma^*}$ may additionally contain other use actions.

(1) $a = \langle use, r\rangle \wedge r \notin q$. In this case, $\mathcal{A}_{ga}^{\Sigma^*}$ outputs a while $\mathcal{A}_{ga}^{fair}$ suppresses it.
(2) In all other cases, the transition functions, and thus the behaviors of $\mathcal{A}_{ga}^{\Sigma^*}$ and $\mathcal{A}_{ga}^{fair}$ are identical.

□

PROPOSITION 5.35 $\mathcal{A}_{ga}^{\Sigma^*} \trianglelefteq_m^{\Sigma^*} \mathcal{A}_{ga}^{fair}$.

PROOF.
Both automata must maintain a list of those resources that have been acquired and not yet released, captured in state q. Let $\sigma \in \Sigma^\infty$ be any input sequence, let $\sigma[..i]$ be the prefix of σ processed so far, let $\mathcal{A}.q_i$ stand for the value of q for automaton $\mathcal{A}$ after execution i and let $|q|$ stand for the cardinality of this set. We show that the automata $\mathcal{A}_{cw}^s$ and $\mathcal{A}_{cw}^e$ maintain the invariant INV(i)= $|\mathcal{A}_{ga}^{\Sigma^*}.q_i| \leq |\mathcal{A}_{ga}^{fair}.q_i|$. The invariant holds initially since when $\sigma = \epsilon$, $q = \emptyset$. Assume that INV(i) holds and let a be the next input action. We show that INV($i+1$) holds. There are two cases to consider.

(1) $a = \langle use, r\rangle \wedge r \notin q$. In this case, $\mathcal{A}_{ga}^{fair}$ simply suppresses the action leaving q unchanged, while $\mathcal{A}_{ga}^{\Sigma^*}$ outputs a and adds the value i to q. It follows that INV($i+_1$) holds.
(2) It all other cases, the two automata's transition functions, and thus their behaviors, are identical, and INV($i+1$) holds immediately from the induction assumption.

Proposition 5.35 follows immediately from this invariant. □

PROPOSITION 5.36 $\mathcal{A}_{ga}^{fair} \trianglelefteq_p^{\Sigma^*} \mathcal{A}_{ga}^{\Sigma^*}$.

PROOF. Let $\sigma \in \Sigma^\infty$ be any input sequence and let $\sigma[..i]$ be the prefix of σ processed so far Let INV(i) stand for the invariant that after having processed the input sequence $\sigma[..i]$, $|cs_{\sigma[..i]}(\mathcal{A}_{ga}^{fair}(\sigma[..i]))| \leq |cs_{\sigma[..i]}(\mathcal{A}_{ga}^{\Sigma^*}(\sigma[..i]))|$. The invariant holds initially since when $\sigma[..0] = \epsilon$, $\mathcal{A}_{ga}^{fair}(\sigma[..0]) = \mathcal{A}_{ga}^{\Sigma^*}(\sigma[..i]) = \epsilon$. Assume that INV($i$) holds and let a be the next input action. We show that INV($i+1$) holds. There are two cases to consider.

(1) $a = \langle use, r\rangle \wedge r \notin q$. In this case, $\mathcal{A}_{ga}^{fair}$ simply suppresses a while $\mathcal{A}_{ga}^{\Sigma^*}$ outputs a. It follows that if INV(i) holds, then INV($i+1$) also holds.
(2) It all other cases, the two automata's transition functions, and thus their behaviors, are identical, and INV($i+1$) holds immediately from the induction assumption.

Proposition 5.36 follows immediately from this invariant. □

ACKNOWLEDGMENTS

We wish to thank Richard Khoury and Nataliia Bielova for their helpful remarks and the three anonymous reviewers for their insightful comments.

REFERENCES

ALPERN, B. AND SCHNEIDER, F. 1985. Defining liveness. *Information Processing Letters 21,* 4, 181–185.

BAUER, L., LIGATTI, J., AND WALKER, D. 2002. More enforceable security policies. In *Foundations of Computer Security*. Copenhagen, Denmark.

BEAUQUIER, D., COHEN, J., AND LANOTTE, R. 2009. Security policies enforcement using finite edit automata. *Electronic Notes in Theoretical Computer Science 229,* 3, 19–35.

BIELOVA, N. AND MASSACCI, F. 2011a. Do you really mean what you actually enforced? *International Journal of Information Security 10,* 4, 1–16. 10.1007/s10207-011-0137-2.

BIELOVA, N. AND MASSACCI, F. 2011b. Iterative enforcement by suppression: Towards practical enforcement theories. *Journal of Computer Security*. To appear.

BIELOVA, N. AND MASSACCI, F. 2011c. Predictability of enforcement. In *Proceedings of the International Symposium on Engineering Secure Software and Systems 2011*. Vol. 6542. Springer, 73–86.

BIELOVA, N., MASSACCI, F., AND MICHELETTI, A. 2009. Towards practical enforcement theories. In *Proceedings of the 14th Nordic Conference on Secure IT Systems, NordSec 2009*. LNCS Series, vol. 5838. Springer, 239–254.

BOEBERT, W. E. AND KAIN, R. Y. 1985. A practical alternative to hierarchical integrity policies. In *Proceedings of the 8th National Computer Security Conference*.

BREWER, D. F. C. AND NASH, M. J. 1989. The chinese wall security policy. In *IEEE Symposium on Security and Privacy*. 206–214.

CHABOT, H., KHOURY, R., AND TAWBI, N. 2009. Generating in-line monitors for Rabin automata. In *Proceedings of the 14th Nordic Conference on Secure IT Systems, NordSec 2009*. LNCS Series, vol. 5838. Springer, 287–301.

CHANG, E., MANNA, Z., AND PNUELI, A. 1991. The safety-progress classification. In *Logic and Algebra of Specifications*, F. Bauer, W. Brauer, and H. Schwichtenberg, Eds. Springer-Verlag, 143–202.

D'AMORIM, M. AND ROŞU, G. 2005. Efficient monitoring of omega-languages. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*. Lecture Notes in Computer Science Series, vol. 3576. Springer, 364–378.

ERLINGSSON, U. AND SCHNEIDER, F. 2000. IRM enforcement of java stack inspection. In *IEEE Symposium on Security and Privacy*. 246–255.

FALCONE, Y., FERNANDEZ, J.-C., AND MOUNIER, L. 2008. Synthesizing enforcement monitors wrt. the safety-progress classification of properties. In *Information Systems Security, 4th International Conference, ICISS 2008, Hyderabad, India, December 16-20, 2008. Proceedings*. Lecture Notes in Computer Science Series, vol. 5352. 41–55.

FONG, P. 2004. Access control by tracking shallow execution history. In *In Proceedings of the 2004 IEEE Symposium on Security and Privacy, Oakland, California, USA, May 2004.*

HAMLEN, M. S. K. W. 2011. Flexible in-lined reference monitor certification: challenges and future directions. In *Proceedings of the 5th ACM workshop on Programming languages meets program verification*. PLPV '11. ACM, New York, NY, USA, 55–60.

JUN, P., XINGYUAN, C., BEI, W., XIANGDONG, D., AND YONGLIANG, W. 2008. Policy monitoring and a finite state automata model. In *CSSE '08: Proceedings of the 2008 International Conference on Computer Science and Software Engineering*. IEEE Computer Society, Washington, DC, USA, 646–649.

K. W. HAMLEN, G. M. AND SCHNEIDER, F. 2006. Computability classes for enforcement mechanisms. *ACM Transactions on Programming Languages and Systems (TOPLAS) 28,* 1, 175–205.

KHOURY, R. AND TAWBI, N. 2010a. Corrective enforcement of security policies. In *the 7th International Workshop on Formal Aspects of Security & Trust (FAST2010)*.

KHOURY, R. AND TAWBI, N. 2010b. Using equivalence relations for corrective enforcement of security policies. In *the Fifth International Conference Mathematical Methods, Models, and Architectures for Computer Networks Security*.

KIM, M., KANNAN, S., LEE, I., SOKOLSKY, O., AND VISWANATHAN, M. 2002. Computational analysis of run-time monitoring - fundamentals of java-mac. *Electr. Notes Theor. Comput. Sci. 70,* 4.

KUPFERMAN, O., LUSTIG, Y., AND VARDI, M. 2006. On locally checkable properties. In *Proccedings of the 13th International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Lecture Notes in Computer Science. Springer-Verlag.

LIGATTI, J., BAUER, L., AND WALKER, D. 2004. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security 4,* 1-2, 2–16.

LIGATTI, J., BAUER, L., AND WALKER, D. 2005. Enforcing non-safety security policies with program monitors. In *10th European Symposium on Research in Computer Security (ESORICS)*. Milan.

LIGATTI, J., BAUER, L., AND WALKER, D. 2009. Run-time enforcement of nonsafety policies. *ACM Transactions on Information and System Security 12,* 3, 1–41.

LIGATTI, J. AND REDDY, S. 2010. A theory of runtime enforcement, with results. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*.

MEALY, G. H. 1955. A method for synthesizing sequential circuits. *Bell System Technical Journal 34,* 5, 1045–1079.

OULD-SLIMANE, H., MEJRI, M., AND ADI, K. 2009. Using edit automata for rewriting-based security enforcement. In *Data and Applications Security XXIII, 23rd Annual IFIP WG 11.3 Working Conference, Montreal, Canada, July 12-15, 2009. Proceedings*. 175–190.

SCHNEIDER, F. 2000. Enforceable security policies. *Information and System Security 3,* 1, 30–50.

SOBEL, A. E. K. AND ALVES-FOSS, J. 1999. A trace-based model of the chinese wall security policy. In *In Proceedings of the 22nd National Information Systems Security Conference*.

SYROPOULOS, A. 2001. Mathematics of multisets. In *Proceedings of the Workshop on Multiset Processing*. Springer-Verlag, 347–358.

TALHI, C., TAWBI, N., AND DEBBABI, M. 2008. Execution monitoring enforcement under memory-limitations constraints. *Information and Computation 206,* 1, 158–184.

VISWANATHAN, M. 2000. Foundations for the run-time analysis of software systems. Ph.D. thesis, University of Pennsylvania.

YOUNG, W., TELEGA, P., AND BOEBERT, W. 1986. A verified labeler for the secure ada target. In *Proceedings of the 9th National Computer Security Conference*.

ZHU, G., TYAGI, A., AND ROOP, P. 2006. Stream automata as run-time monitors for open system security policies. Department of Electrical and Computer Engineering, Iowa State University, Ames, Iowa, USA Tech Report 06-101.