Model checking usage policies

Massimo Bartoletti[1], Pierpaolo Degano[1], Gian Luigi Ferrari[1], and Roberto Zunino[2]

[1] Dipartimento di Informatica, Università di Pisa, Italy
[2] Dipartimento di Ingegneria e Scienza dell'Informazione, Università di Trento, Italy

Abstract. We propose a model for specifying, analysing and enforcing safe usage of resources. Our usage policies allow for parametricity over resources, and they can be enforced through finite state automata. The patterns of resource access and creation are described through a basic calculus of usages. In spite of the augmented flexibility given by resource creation and by policy parametrization, we devise an efficient (polynomial-time) model-checking technique for deciding when a usage is resource-safe, i.e. when it complies with all the relevant usage policies.

1 Introduction

A fundamental concern of security is to ensure that resources are used correctly. Devising expressive, flexible and efficient mechanisms to control resource usages is therefore a major issue in the design and implementation of security-aware programming languages. The problem is made even more crucial by the current programming trends, which provide for reusing code, and exploiting services and components, offered by (possibly untrusted) third parties. It is indeed common practice to pick from the Web some scripts, or plugins, or packages, and assemble them into a bigger program, with little or no control about the security of the whole. To cope with this situation, we proposed in [4] *local policies*, that formalise and enhance the concept of *sandbox*, while being more flexible than global policies and local checks spread over program code. Local policies smoothly allow for safe composition of programs with their own security requirements, also in mobile code scenarios, and they can drive call-by-contract composition of services [5, 8].

Our contribution is twofold. First, we propose a model for local usage policies. Our policies are quite general: in the spirit of history-based security [1], they can inspect the whole trace of security-relevant events generated by a running program. Unlike e.g. [22], our policies are not hard-wired to resources, yet they are *parametric* over resources. For instance, a policy $\varphi(x, y)$ means that for all x and y the obligation expressed by φ must be obeyed. This is particularly relevant in mobile code scenarios, where you need to impose constraints on how external programs access the resources created in your local environment, without being able to alter the code (e.g. to insert local security checks). We prove that run-time enforcement of our policies is possible through finite state automata.

The second contribution is a model-checking technique to statically detect when a program violates the relevant local policies. The patterns of resource access and creation are described by a calculus of *usages*, which can be automatically inferred by a static analysis of programs [6]. We then devise a technique

to decide whether a given usage, locally annotated with policies, respects them in every possible execution. Since programs may create an arbitrary number of fresh resources, this may give rise to an infinite number of formulae to be inspected while checking a policy. We solve this problem by suitably abstracting resources so that only a finite number of cases needs to be considered. This allows us to extract from a usage a Basic Process Algebra and a regular formula, to be used in model-checking [19]. The proposed technique correctly and completely handles the case in which the number of fresh resources generated at run-time has no bound known at static time. Our algorithm runs in polynomial time on the size of the checked usage and policies.

Examples. To illustrate our model, we present some examples of real-world usage policies, which are particularly relevant to the field of security.

Information flow. Consider a Web application that allows for editing documents, storing them on a remote site, and sharing them with other users. The editor is implemented as an applet run by a local browser. The user can tag any of her documents as *private*. To avoid direct information flows, the policy requires that private files cannot be sent to the server in plain text, yet they can be sent encrypted. This policy is modelled by $\varphi_{\mathsf{IF}}(x)$ in Fig. 1, left. After having tagged the file x as private (edge from q_0 to q_1), if x were to be sent to the server (edge from q_1 to q_2), then the policy would be violated: the double circle around q_2 marks it as an offending state. Instead, if x is encrypted (edge from q_1 to q_3), then x can be freely transmitted: indeed, the absence of paths from q_3 to an offending state indicates that once in q_3 the policy will not be violated on file x. A further policy is applied to our editor, to avoid information flow due to covert channels. It requires that, after reading a private file, any other file must be encrypted before it can be transmitted. This is modelled by $\varphi_{\mathsf{CC}}(x, y)$ in Fig. 1, right. A violation occurs if after some private file x is read (path from q'_0 to q'_2), then some other file y is sent (edge from q'_2 to the offending state q'_4).

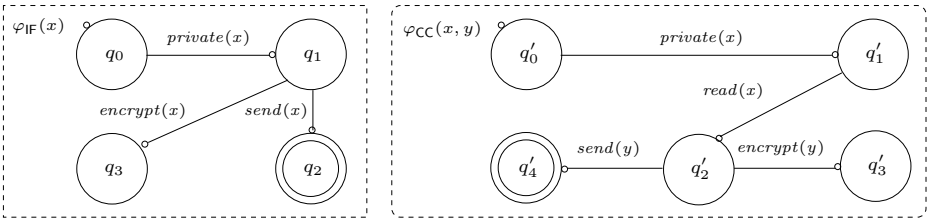

Fig. 1. The information flow policy $\varphi_{\mathsf{IF}}(x)$ and the covert channels policy $\varphi_{\mathsf{CC}}(x, y)$.

Chinese Wall. A classical security policy used in commercial and corporate business services is the Chinese Wall policy [13]. In such scenarios, it is usual to assume that all the objects which concern the same corporation are grouped together into a *company dataset*, e.g. $bank_A, bank_B, oil_A, oil_B$, etc. A *conflict of interest class* groups together all company datasets whose corporations are in competition, e.g. *Bank* containing $bank_A, bank_B$ and *Oil* containing oil_A, oil_B.

The Chinese Wall policy then requires that accessing an object is only permitted in two cases. Either the object is in the same company dataset as an object already accessed, or the object belongs to a different conflict of interest class. E.g., the trace $read(oil_A, Oil)\, read(bank_A, Bank)\, read(oil_B, Oil)$ violates the policy, because reading an object in the dataset oil_B is not permitted after having accessed oil_A, which is in the same conflict of interests class Oil. The Chinese Wall policy is specified by $\varphi_{\mathsf{CW}}(x, y)$ in Fig. 2, left. The edge from q_0 to q_1 represents accessing the company dataset x in the conflict of interests class y. The edge leading from q_1 to the offending state q_2 means that a dataset different from x (written as $\bar{x}$) has been accessed in the same conflict of interests class y.

Applet confinement. As a further example, consider the case of a Web browser which can run applets. Assume that an applet needs to create files on the local disk, e.g. to save and retrieve status information. Direct information flows are avoided by denying applets the right to access local files. Also, to avoid interference, applets are not allowed to access files created by other applets. This behaviour is modelled by the policy $\varphi_{\mathsf{AC}}(f, a)$ in Fig. 2, right. The edge from q'_0 to q'_1 represents the applet a creating a file f. Accessing f is denied to any applet other than a (modelled by $\bar{a}$) through the edge from q'_1 to the offending state q'_2. The edge from q_0 to q'_2 prohibits accessing local files.

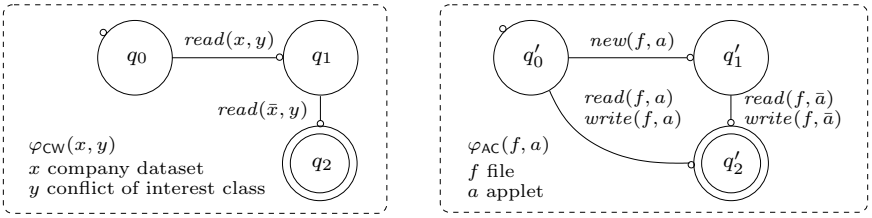

Fig. 2. The Chinese Wall policy $\varphi_{\mathsf{CW}}(x, y)$ and the Applet Isolation policy $\varphi_{\mathsf{AI}}(f, a)$.

The paper is organized as follows. We first introduce our usage policies, and we show them enforceable through finite state automata. We then define a calculus of usages, and we characterize when a usage is valid. Our model checking technique follows, together with the main results, i.e. its correctness, completeness and polynomial time complexity. We conclude by presenting some possible extensions, and we discuss some related work. For simplicity we shall only present the formal treatment of monadic usage policies (i.e. those with one formal parameter). The extension to the polyadic case $\varphi(x_1, \ldots, x_n)$ is quite direct; at each step of our formal development, we shall give the intuition for adapting our theory to the polyadic case. We have designed and implemented a tool for model checking usage policies [9]. The tool is written in Haskell, and it is quite general: it supports all of the features presented in this paper, including polyadic policies and events. We have used our tool to experiment with the case studies presented in this paper, and also in some more complex ones. The results and the performance are consistent with those anticipated by the theoretical results presented in this paper.

2 Usage policies

We start by introducing the needed syntactic categories. *Resources* $r, r', \ldots \in \mathsf{Res} = \mathsf{Res}_s \cup \mathsf{Res}_d$ are objects that can either be already available in the environment (*static*, included in the finite set Res_s), or be freshly created at run-time (*dynamic*, Res_d). We assume a distinguished resource $? \notin \mathsf{Res}$ to play the role of an "unknown" one (typically, ? will result from static approximations). Resources can be accessed through a given finite set of *actions* $\alpha, \alpha', new, \ldots \in \mathsf{Act}$. An *event* $\alpha(r) \in \mathsf{Ev}$ abstracts from accessing the resource r through the action α. The special action *new* represents the creation of a fresh resource; this means that for each dynamically created resource r, the event $new(r)$ must precede any other $\alpha(r)$. In our examples, we sometimes use polyadic events such as $\alpha(r_1, r_2)$. These events do not increase the expressivity of our model, since they can be simulated by e.g. a sequence $\alpha_1(r_1)\alpha_2(r_2)$. A *trace* is a finite sequence of events, typically denoted by $\eta, \eta', \ldots \in \mathsf{Ev}^*$. For notational convenience, when the target resource of an action α is immaterial, we stipulate that α acts on some special (static) resource, and we write just α for the event.

Usage policies (Def. 1) constrain the usage of resources to obey a regular property. For instance, a file usage policy $\varphi(x)$ might require that "before reading or writing a file x, that file must have been opened, and not closed by the while". A usage policy gives rise to an automaton with finite states, named *policy automaton*, when the formal parameter x is instantiated to an actual resource r. Policy automata will be exploited to recognize those traces obeying φ.

Definition 1. Usage policies

A usage policy $\varphi(x)$ *is a 5-tuple* $\langle S, Q, q_0, F, E \rangle$*, where:*

- $S \subset \mathsf{Act} \times (\mathsf{Res}_s \cup \{x, \bar{x}\})$ *is the input alphabet,*
- Q *is a* finite *set of states,*
- $q_0 \in Q \setminus F$ *is the start state,*
- $F \subset Q$ *is the set of final "offending" states,*
- $E \subseteq Q \times S \times Q$ *is a finite set of edges, written* $q \overset{\vartheta}{\multimap} q'$

Edges in a usage policy can be of three kinds: either $\vartheta = \alpha(r)$ for a static resource r, or $\vartheta = \alpha(x)$, or $\vartheta = \alpha(\bar{x})$, where $\bar{x}$ means "different from x". The extension to the polyadic case is as follows: the edges of a policy $\varphi(x, y)$ can mention static resources, the parameters x and y, and a special symbol $\bar{*}$ denoting "different from x and y". Quite notably, polyadic policies (unlike polyadic events) increase the expressivity of our model e.g. the policies in Fig. 2 cannot be expressed with a single parameter.

Given $r \in \mathsf{Res}$ and a set of resources $\mathcal{R}$, a usage policy $\varphi(x)$ is instantiated into a policy automaton $A_{\varphi(r,\mathcal{R})}$ by binding x to the resource r and, accordingly, making $\bar{x}$ range over each resource in $\mathcal{R} \setminus \{r\}$ (see Def. 2). The auxiliary relation $\ddot{\delta}$ (i) instantiates x to the given resource r, (ii) instantiates $\bar{x}$ with all $r' \neq r$, (iii) maintains the transitions $\alpha(r')$ for r' static. Then, the relation $\dot{\delta}$ adds self-loops

Definition 2. Policy automata and policy compliance

Let $\varphi(x) = \langle S, Q, q_0, F, E\rangle$ *be a usage policy, let* $r \in \mathsf{Res}$*, and let* $R \subseteq \mathsf{Res}$*. The* usage automaton $A_{\varphi(r,R)} = \langle \Sigma, Q, q_0, F, \delta\rangle$ *is defined as follows:*

$$
\begin{array}{lll}
\Sigma = \{\, \alpha(r') \mid \alpha \in \mathsf{Act} \text{ and } r' \in R \,\} & & \\
\delta = \dot\delta \cup \{\, q \xrightarrow{\alpha(?)} q' \mid \exists \zeta \in R : q \xrightarrow{\alpha(\zeta)} q' \in \dot\delta \,\} & & (\textit{unknown resources}) \\
\dot\delta = \ddot\delta \cup \{\, q \xrightarrow{\alpha(r')} q \mid r' \in R, \nexists q' : q \xrightarrow{\alpha(r')} q' \in \ddot\delta \,\} & & (\textit{self-loops}) \\
\ddot\delta = \{\, q \xrightarrow{\alpha(r)} q' \mid q \xrightarrow{\alpha(x)\circ} q' \in E \,\} & & (\textit{instantiation of } x) \\
\quad \cup \bigcup_{r' \in (R \cup \{?\}) \setminus \{r\}} \{\, q \xrightarrow{\alpha(r')} q' \mid q \xrightarrow{\alpha(\bar x)\circ} q' \in E \,\} & & (\textit{instantiation of } \bar x) \\
\quad \cup \{\, q \xrightarrow{\alpha(r')} q' \mid q \xrightarrow{\alpha(r')\circ} q' \in E \,\} & & (\textit{static resources})
\end{array}
$$

We write $\eta \lhd A_{\varphi(r,\mathcal{R})}$ *when* η *is not in the language of the automaton* $A_{\varphi(r,\mathcal{R})}$*. We say that* η respects φ*, written* $\eta \models \varphi$*, when* $\eta \lhd A_{\varphi(r,\mathsf{Res})}$*, for all* $r \in \mathsf{Res}$*. Otherwise, we say that* η violates φ*, written* $\eta \not\models \varphi$*.*

for all the events not explicitly mentioned in the policy. Finally, we derive δ by adding transitions to deal with ?, that represents resources not known at static time. Intuitively, any transition $q \xrightarrow{\alpha(r)} q'$ can also be played by $\alpha(?)$. Note that policy automata are non-deterministic, e.g. because of the transitions labelled $\alpha(?)$. Given a trace η and a policy φ, we want that *all* the paths of the instances of $\varphi(x)$ comply with φ. This is a form of diabolic (or internal) non-determinism. To account for that, we make the "offending" states as final — thus going into a final state represents a violation of the policy, while the other states mean compliance to the policy. The construction of Def. 2 can be easily adapted to polyadic policies such as $\varphi(x, y)$, yielding the automata $A_{\varphi(r,r',\mathcal{R})}$, where x is instantiated to r, y to r', and $\bar *$ to $\mathcal{R} \setminus \{r, r'\}$.

According to Def. 2, to check $\eta \models \varphi$ we should instantiate *infinitely* many policy automata (one for each $r \in \mathsf{Res}$). These automata have a finite number of states, but they may have *infinitely* many transitions, e.g. instantiating the abstract edge $q \xrightarrow{\alpha(\bar x)\circ} q'$ originates $q \xrightarrow{\alpha(r)} q'$ for all but one $r \in \mathsf{Res}$. Similarly for self-loops. Interestingly enough, the compliance of a trace with a policy is a decidable property: the next lemma shows that checking $\eta \models \varphi$ can be decided through a *finite* set of *finite* state automata.

Lemma 1. *Let* η *be a trace, let* φ *be a policy, and let* $\mathsf{R}(\eta)$ *be the set of resources occurring in* η*. Then,* $\eta \models \varphi$ *if and only if* $\eta \lhd A_{\varphi(r,\mathsf{R}(\eta))}$ *for each* $r \in \mathsf{R}(\eta) \cup \{r'\}$*, where* r' *is an arbitrary resource in* $\mathsf{Res} \setminus \mathsf{R}(\eta)$*.*

When polyadic policies are used, e.g. $\varphi(x, y)$, then Lemma 1 should be changed to require $\eta \lhd A_{\varphi(r,r',\mathsf{R}(\eta))}$ for each $r, r' \in \mathsf{R}(\eta) \cup \{r''\}$, where r'' is an arbitrary resource in $\mathsf{Res} \setminus \mathsf{R}(\eta)$.

Example 1. Consider the policy requiring that, in a history $\alpha(r_1)\alpha(r_2)\alpha(r_3)$, the three resources r_0, r_1, r_2 are actually distinct. This is modelled by the usage policy $\varphi_3(x, y)$ in Fig. 3. A straightforward generalisation of the above allows for keeping distinct k resources, using a policy with arity $k-1$. □

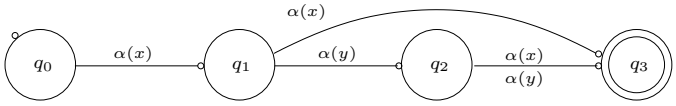

Fig. 3. The usage policy $\varphi_3(x, y)$ allows for keeping distinct three resources.

Example 2. To justify the extra resource of Lemma 1, let $\varphi_{\neg\alpha}(x)$ be the policy $\langle\{\alpha(\bar{x})\}, \{q_0, q_1\}, q_0, \{q_1\}, \{q_0 \xrightarrow{\alpha(\bar{x})}\circ\ q_1\}\rangle$. When a trace η contains some event $\alpha(r)$, the instantiation $A_{\varphi_{\neg\alpha}(r', \mathsf{Res})}$ recognizes η as offending, for all $r' \neq r$ – so $\varphi_{\neg\alpha}(x)$ actually forbids any α actions. Consider e.g. $\eta = \alpha(r_0)\,\beta(r_0)$. Although $\eta \lhd A_{\varphi(r, \mathsf{R}(\eta))}$ for all $r \in \mathsf{R}(\eta) = \{r_0\}$, as noted above η violates $\varphi_{\neg\alpha}$, which motivates Lemma 1 checking $A_{\varphi(r', \mathsf{R}(\eta))}$ also for some $r' \in \mathsf{Res} \setminus \mathsf{R}(\eta)$. Finally, note that replacing $\alpha(\bar{x})$ with $\alpha(x)$ in the policy would not affect trace validity, because x is implicitly universally quantified in the definition of validity. □

Example 3. Consider a trojan-horse applet that maliciously attempts to send spam e-mail through the mail server of the browser site. To do that, the applet first connects to the site it was downloaded from, and it implements some apparently useful and harmless activity. Then, the applet inquires the local host to check if relaying of e-mails is allowed: if so, it connects to the local SMTP server to send unsolicited bulk e-mails. To protect from such malicious behaviour, the browser prevents the applet from connecting to two different URLs. This can be enforced by sandboxing the applet with the policy $\varphi_{\mathsf{Spam}}(x)$ defined in Fig. 4, left. E.g., consider the trace $\eta = \mathit{start}\,\mathit{connect}(u_0)\,\mathit{stop}\,\mathit{start}\,\mathit{connect}(u_1)\,\mathit{connect}(u_2)$. The policy automaton $A_{\varphi_{\mathsf{Spam}}(u_1, \mathcal{R})}$, with $\mathcal{R} = \{u_0, u_1, u_2\}$, is in Fig. 4, right (the self-loops for events other than *start*, *stop* and *connect* are omitted). Since η drives an offending run in the policy automaton $A_{\varphi_{\mathsf{Spam}}(u_1, \mathcal{R})}$, then $\eta \not\models \varphi_{\mathsf{Spam}}$. This correctly models the fact that the policy φ_{Spam} prevents applets from connecting to two different sites. Observe that removing the last $\mathit{connect}(u_2)$ event from η would make the trace obey φ_{Spam}, since the *stop* event resets the policy φ_{Spam} to the starting state. Actually, $\mathit{start}\ \mathit{connect}(u_0)\ \mathit{stop}\ \mathit{start}\ \mathit{connect}(u_1)$ is *not* in the language of any $A_{\varphi_{\mathsf{Spam}}(r, \mathcal{R}')}$, for all $r \in \mathsf{Res}$ and $\mathcal{R}' \subseteq \mathsf{Res}$. □

3 A calculus for usage control

We now introduce a basic calculus for usage control (the abstract syntax of our calculus is in Def. 3). Usages are *history expressions* [6], that inherit from Basic Process Algebras (BPAs, [12]) sequential composition, non-deterministic choice, and recursion (though with a slightly different syntax). Quite differently from

BPAs, our atomic actions have a parameter which indicates the resource upon which the action is performed; more precisely, our atomic actions are the events from Sec. 2. We also have events the target of which is a *name* $n, n', \ldots \in \mathsf{Nam}$. In $\nu n.\, U$, the ν acts as a binder of the free occurrences of the name n in U. The intended meaning is to keep track of the binding between n and a freshly created resource. In a recursion $\mu h.U$, the free occurrences of h in U are bound by μ. A usage is *closed* when it has no free names and variables, and it is *initial* when closed and with no dynamic resources. The sandboxing construct $\varphi[U]$ asserts that each step of the execution of U must obey the policy φ. Aiming at minimality, it is convenient to treat $\varphi[U]$ as syntactic sugar for $[_\varphi \cdot U \cdot]_\varphi$, where $[_\varphi$ and $]_\varphi$ are special *framing* events that represent opening/closing of the scope of φ. We assume that such framing events are disjoint from those in Ev.

The behaviour of a usage (Def. 4) is defined as the set of sequential traces of its events. As usual, ε denotes the empty trace, and $\varepsilon\eta = \eta = \eta\varepsilon$. The trace semantics is defined through a labelled transition relation $U, \mathcal{R} \xrightarrow{a} U', \mathcal{R}'$, where $a \in \mathsf{Ev} \cup \{\varepsilon\}$. The set $\mathcal{R}$ in configurations accumulates the resources created at run-time, so that no resource can be created twice. We assume that $\mathsf{Res}_s \subseteq \mathcal{R}$, to prevent dynamically created resources from colliding with static ones.

Example 4. Let $U = \mu h.\, \nu n.\, (\varepsilon + \mathit{new}(n) \cdot \alpha(n) \cdot h)$. Given a starting $\mathcal{R}$, the traces of U are the prefixes of the strings with form $\mathit{new}(r_1)\alpha(r_1) \cdots \mathit{new}(r_k)\alpha(r_k)$ for all $k \geq 0$ and pairwise distinct resources r_i such that $r_i \notin \mathcal{R}$. □

A trace is *well-formed* when (i) no static resource is the target of a *new* event, (ii) no *new* is fired twice on the same resource, and (iii) no event $\alpha(r)$, with r dynamic and $\alpha \neq \mathit{new}$, is fired without a prior $\mathit{new}(r)$. Hereafter, we shall only consider usages U with well-formed traces. We conjecture this is a decidable property of usages, e.g. suitably adapting the techniques of [21] should enable us to identify and discard those U that produce non well-formed traces.

We now define when a trace respects all the relevant usage policies, i.e. when the trace is *valid* (Def. 5). For example, let $\eta = \mathit{private}(f)\, \mathit{read}(f)\, [_{\varphi_{\mathsf{Ed}}}\, \mathit{send}(f)\,]_{\varphi_{\mathsf{Ed}}}$ where φ_{Ed} is the Editor policy of Sect. 1. Then, η is *not* valid, because the *send* event occurs within a framing enforcing φ_{Ed}, and $\mathit{private}(f)\, \mathit{read}(f)\, \mathit{send}(f)$ does not obey φ_{Ed}. Note that we check the *whole* past, and not just the $\mathit{send}(f)$ event within the framing, as we follow the history–based security approach. This makes our policies more expressive than those that can only look at the part of the history enclosed within the framing events.

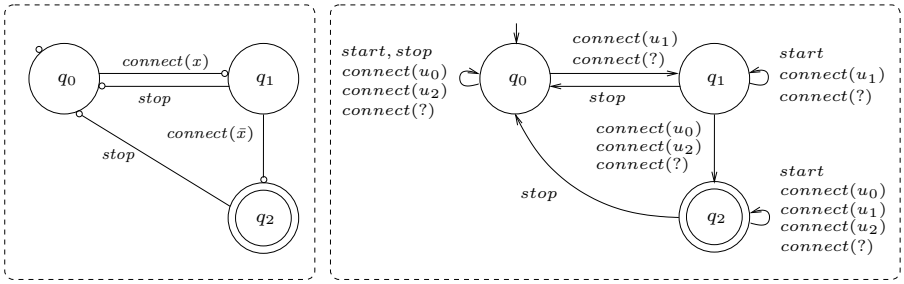

Fig. 4. The policy $\varphi_{\mathsf{Spam}}(x)$ (left) and the policy automaton $A_{\varphi_{\mathsf{Spam}}(u_1,\{u_0,u_1,u_2\})}$ (right).

Definition 3. Usages

$U, U' ::= \varepsilon$		*empty*	
h		*variable*	
$\alpha(\rho)$		*event*	$(\rho \in \mathsf{Res} \cup \mathit{Nam} \cup \{?\})$
$U \cdot V$		*sequence*	
$U + V$		*choice*	
$\nu n.U$		*resource creation*	
$\mu h.\, U$		*recursion*	
$\varphi[U]$		*sandboxing*	$(\varphi[U] = [\_{\varphi} \cdot U \cdot]\_{\varphi})$

Definition 4. Trace semantics of usages

$$\alpha(r),\, \mathcal{R} \xrightarrow{\alpha(r)} \varepsilon,\, \mathcal{R} \qquad \nu n.\, U,\, \mathcal{R} \xrightarrow{\varepsilon} U\{r/n\},\, \mathcal{R} \cup \{r\} \quad \text{if } r \in \mathsf{Res}_d \setminus \mathcal{R}$$

$$\varepsilon \cdot U,\, \mathcal{R} \xrightarrow{\varepsilon} U,\, \mathcal{R} \qquad U \cdot V,\, \mathcal{R} \xrightarrow{a} U' \cdot V,\, \mathcal{R}' \quad \text{if } U,\, \mathcal{R} \xrightarrow{a} U',\, \mathcal{R}'$$

$$U + V,\, \mathcal{R} \xrightarrow{\varepsilon} U,\, \mathcal{R} \qquad U + V,\, \mathcal{R} \xrightarrow{\varepsilon} V,\, \mathcal{R} \qquad \mu h.\, U,\, \mathcal{R} \xrightarrow{\varepsilon} U\{\mu h.\, U/h\},\, \mathcal{R}$$

Definition 5. Active policies and validity

The multiset $act(\eta)$ of the active policies of a trace η is defined as follows:

$$act(\varepsilon) = \{\!|\,|\!\} \qquad act(\eta\,[_{\varphi}) = act(\eta) \cup \{\!|\varphi|\!\}$$
$$act(\eta\,\alpha(\rho)) = act(\eta) \qquad act(\eta\,]_{\varphi}) = act(\eta) \setminus \{\!|\varphi|\!\}$$

A trace η is valid *when $\models \eta$, defined inductively as follows:*

$$\models \varepsilon \qquad \models \eta'\beta \quad \text{if} \quad \models \eta' \text{ and } (\eta'\beta)^{-[]} \models \varphi \text{ for all } \varphi \in act(\eta'\beta)$$

where $\eta^{-[]}$ is the trace η depurated from all the framing events.
A usage U is valid *when, for all $U', \mathcal{R}, \mathcal{R}'$ and η: $U, \mathcal{R} \xrightarrow{\eta} U', \mathcal{R}' \implies \models \eta$.*

Validity of traces is a prefix-closed (i.e., safety) property (Lemma 2), yet it is not compositional: in general, $\models \eta$ and $\models \eta'$ do not imply $\models \eta\eta'$. This is a consequence of the assumption that no past events can be hidden (see Ex. 7).

Lemma 2. *For all traces η and η', if $\eta\eta'$ is valid, then η is valid.*

Example 5. Let $\eta = [_{\varphi}\alpha_1[_{\varphi'}\alpha_2]_{\varphi'}\alpha_3$. The active policies of η are as follows:

$$\begin{aligned} act(\eta) &= act([_{\varphi}\alpha_1[_{\varphi'}\alpha_2]_{\varphi'}) = act([_{\varphi}\alpha_1[_{\varphi'}\alpha_2) \setminus \{\!|\varphi'|\!\} = act([_{\varphi}\alpha_1[_{\varphi'}) \setminus \{\!|\varphi'|\!\} \\ &= (act([_{\varphi}\alpha_1) \cup \{\!|\varphi'|\!\}) \setminus \{\!|\varphi'|\!\} = act([_{\varphi}\alpha_1) = act([_{\varphi}) = \{\!|\varphi|\!\} \end{aligned}$$

Therefore, η is valid if and only if: $\varepsilon \models \varphi$, $\alpha_1 \models \varphi$, $\alpha_1\alpha_2 \models \varphi$, $\alpha_1\alpha_2\alpha_3 \models \varphi$, $\alpha_1 \models \varphi'$, and $\alpha_1\alpha_2 \models \varphi'$. □

Example 6. Consider φ_{Loan} that prevents anyone from taking out a loan if their account is in the red: $\langle\{red, black\}, \{q_0, q_1\}, q_0, \{q_1\}, \{q_0 \xrightarrow{red}\!\!\circ\ q_1, q_1 \xrightarrow{black}\!\!\circ\ q_0\}\rangle$. Let $\eta = red\ black\,[_{\varphi_{\mathsf{Loan}}}$, which is valid. Indeed, we have that $red\ black \models \varphi_{\mathsf{Loan}}$ – while the prefix *red* is not required to respect the policy φ_{Loan} (so matching the intuition that one can recover from a red balance and obtain a loan). Note that, differently from validity, the relation $\eta \models \varphi$ is *not* prefix-closed. So, we can define policies which, as in this example, permit to recover from bad states. □

Example 7. Consider the trace $\eta = \alpha[_\varphi\, \alpha\,]_\varphi\alpha$, where the policy φ requires that α is not executed three times. Since $\alpha \models \varphi$ and $\alpha\alpha \models \varphi$, then η is valid. Note that the first α is checked, even though it is outside of the framing: since it happens in the past, our policies can inspect it. Instead, the third α occurs after the framing has been closed, therefore it is not checked. Now, consider the trace $\eta' = \alpha\eta$. In spite of both α and η being valid, their composition η' is not. To see why, consider the trace $\bar{\eta} = \alpha\alpha[_\varphi\, \alpha$, which is a prefix of η'. Then, $act(\bar{\eta}) = \{\!|\varphi|\!\}$, but $\bar{\eta}^{-[\,]} = \alpha\alpha\alpha \not\models \varphi$. This shows that validity is not compositional. □

We specify in Def. 6 a transformation of policy automata that makes them able to recognize valid traces. This is done by instrumenting a policy automaton $A_{\varphi(r,\mathcal{R})}$ with framing events, so obtaining a *framed* usage automaton $A_{\varphi_{[\,]}(r,\mathcal{R})}$ that will recognize those traces that are valid with respect to the policy φ.

Definition 6. Instrumenting policy automata with framing events

Let $A_{\varphi(r,\mathcal{R})} = \langle \Sigma, Q, q_0, F, \delta\rangle$ *be a policy automaton. Then, we define* $A_{\varphi_{[\,]}(r,\mathcal{R})} = \langle \Sigma', Q', q_0, F', \delta'\rangle$ *as follows:* $\Sigma' = \Sigma \cup \{[_\varphi,]_\varphi, [_{\varphi'},]_{\varphi'}, \ldots\}$, $Q' = Q \cup \{\, \dot{q} \mid q \in Q \,\}$, $F' = \{\, \dot{q} \mid q \in F \,\}$, *and:*

$$\delta' = \delta \ \cup\ \{\, q \xrightarrow{[_\varphi} \dot{q} \mid q \in Q \,\} \ \cup\ \{\, \dot{q} \xrightarrow{]_\varphi} q \mid q \in Q \setminus F \,\} \ \cup\ \{\, \dot{q} \xrightarrow{\vartheta} \dot{q} \mid \dot{q} \in F' \,\}$$
$$\cup\, \{\, \dot{q} \xrightarrow{\vartheta} \dot{q}' \mid q \xrightarrow{\vartheta} q' \in \delta \text{ and } q \in Q \setminus F \,\} \ \cup\ \{\, q \xrightarrow{[_\psi} q, q \xrightarrow{]_\psi} q \mid \psi \neq \varphi \,\}$$

Intuitively, the automaton $A_{\varphi_{[\,]}(r,\mathcal{R})}$ is partitioned into two layers. Both are copies of $A_{\varphi(r,\mathcal{R})}$, but in the first layer of $A_{\varphi_{[\,]}(r,\mathcal{R})}$ all the states are made non-final. This represents being compliant with φ. The second layer is reachable from the first one when opening a framing for φ, while closing gets back – unless we are in a final (i.e. offending) state. The transitions in the second layer are a copy of those in $A_{\varphi(r,\mathcal{R})}$, the only difference being that the final states are sinks. The final states in the second layer are exactly those final in $A_{\varphi(r,\mathcal{R})}$. Note that we shall only consider traces without "redundant" framings, i.e. of the form $\eta\,[_\varphi\,\eta'\,[_\varphi$ with $]_\varphi \notin \eta'$. In [4] we defined a static transformation of usages that removes these redundant framings. For instance, $\varphi[U \cdot \varphi[U']]$ is rewritten as $\varphi[U \cdot U']$ since the inner $\varphi[\ldots]$ is redundant. Hereafter, we assume that usages have been undergone to this transformation (minor modifications of [4] suffice).

Example 8. Consider the file usage policy $\varphi_{\mathsf{File}}(x)$ in Fig. 5 (left), requiring that only open files can be read or written. The initial state q_0 represents the file being closed, while q_1 is for an open file. Reading and writing x in q_0 leads to

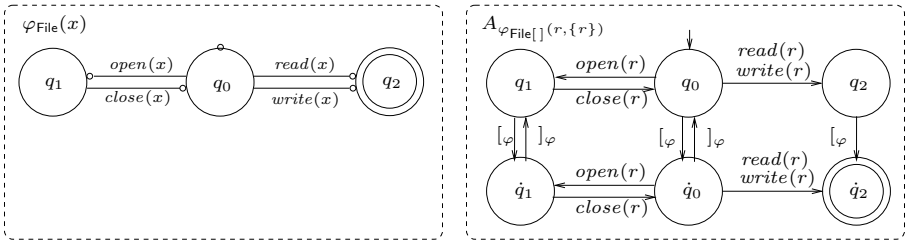

Fig. 5. The file usage policy $\varphi_{\mathsf{File}}(x)$ and the framed usage automaton $A_{\varphi_{\mathsf{File}}(r,\{r\})}$.

the offending state q_2, while in q_1 you can read and write x. The instrumentation $A_{\varphi_{\mathsf{File}[\,]}(r,\{r\})}$ of $A_{\varphi_{\mathsf{File}}(r,\{r\})}$ is in Fig. 5 (right) – the self-loops are omitted. □

We now relate framed usage automata with validity. A trace η (which has no redundant framings, after the assumed transformation) is valid if and only if it complies with the framed automata $A_{\varphi_{[\,]}(r,\mathsf{R}(\eta))}$ for all the policies φ spanning over η. The adaptation of Lemma 3 to polyadic policies proceeds as for Lemma 1.

Lemma 3. *A trace η is valid if and only if $\eta \lhd A_{\varphi_{[\,]}(r,\mathsf{R}(\eta))}$, for all φ occurring in η, and for all $r \in \mathsf{R}(\eta) \cup \{r'\}$, where r' is an arbitrary resource in $\mathsf{Res} \setminus \mathsf{R}(\eta)$.*

4 Model checking validity of usages

We statically verify the validity of usages by model-checking Basic Process Algebras with policy automata. Note that the arbitrary nesting of framings and the infinite alphabet of resources make validity *non-regular*, e.g. the usage $\mu h.\, \nu n.\, new(n) \cdot \alpha(n) + h \cdot h + \varphi[h]$ has traces with unbounded pairs of balanced $[_\varphi$ and $]_\varphi$ and unbounded number of symbols - so it is *not* a regular language. This prevents us from directly applying the standard decision technique for verifying that a BPA P satisfies a regular property φ, i.e. checking the emptiness of the pushdown automaton resulting from the conjunction of P and the finite state automaton recognizing $\neg\varphi$.

To cope with the first source of non-regularity – due to the arbitrary nesting of framings – we use the static transformation of usages that removes the redundant framings [4]. For the second source of non-regularity, due to the ν-binders, the major challenge for verification is that usages may create fresh resources, while BPAs cannot. A naïve solution could lead to the generation of an unbounded set of automata $A_{\varphi(r)}$ that must be checked to verify validity. For example, the traces denoted by $U = \varphi[\mu h.\, (\varepsilon + \nu n.\, new(n) \cdot \alpha(n) \cdot h)]$ must satisfy all the policies $\varphi(r_0), \varphi(r_1), \ldots$ for each fresh resource. Thus, we would have to intersect an infinite number of finite state automata to verify U valid, which is unfeasible.

To this purpose, we shall define a mapping from usages to BPAs that reflects and preserves validity. Our mapping groups freshly created resources in just two categories. The intuition is that any policy $\varphi(x)$ can only distinguish between x and all the other resources, represented by $\bar{x}$. There is no way for $\varphi(x)$ to further discriminate among the dynamic resources. Thus, it is sound to consider

Definition 7. Mapping usages to BPAs

The BPA associated with a usage U is defined as $\mathsf{B}(U) = \mathsf{B}_{\_}(U)_{\emptyset}$, where $\mathsf{B}_d(U)_\Theta$, inductively defined below, takes as input a usage U and a function Θ from variables h to BPA variables X. The parameter d can either be $\_$ or $\#$.

$$\mathsf{B}_d(\varepsilon)_\Theta = \langle 0, \emptyset\rangle \qquad \mathsf{B}_d(h)_\Theta = \langle \Theta(h), \emptyset\rangle \qquad \mathsf{B}_d(\alpha(\rho))_\Theta = \langle \alpha(\rho), \emptyset\rangle$$

$$\mathsf{B}_d(U \cdot V)_\Theta = \mathsf{B}_d(U)_\Theta \cdot \mathsf{B}_d(V)_\Theta \qquad \mathsf{B}_d(U + V)_\Theta = \mathsf{B}_d(U)_\Theta + \mathsf{B}_d(V)_\Theta$$

$$\mathsf{B}_{\_}(\nu n.U)_\Theta = \mathsf{B}_{\_}(U\{\_/n\})_\Theta + \mathsf{B}_{\#}(U\{\#/n\})_\Theta \qquad \mathsf{B}_{\#}(\nu n.U)_\Theta = \mathsf{B}_{\#}(U\{\_/n\})_\Theta$$

$$\mathsf{B}_d(\mu h.U)_\Theta = \langle X, \Delta \cup \{X \triangleq p\}\rangle \text{ where } \langle p, \Delta\rangle = \mathsf{B}_d(U)_{\Theta\{X/h\}},\ X \text{ fresh}$$

only two representatives of dynamic resources: the "witness" resource $\#$ that represents x, and the "don't care" resource $\_$ for $\bar{x}$.

The transformation from usages U into BPAs $\mathsf{B}(U)$ is given in Def. 7. The syntax of a BPA P and its trace semantics $[\![P]\!]$ are standard; for reference, we include them in [7]. Events, variables, concatenation and choice are mapped by $\mathsf{B}(U)$ into the corresponding BPA counterparts. A usage $\mu h.\, U$ is mapped to a fresh BPA variable X, bound to the translation of U in the set of definitions Δ. The crucial case is that of new name generation $\nu n.\, U$, which is dealt with two rules. If $d = \_$, then we generate a choice between two BPA processes: in the first, the name n is replaced by the "don't care" resource $\_$, while in the second, n is replaced by the "witness" resource $\#$. If $d = \#$, this means that we have already witnessed $\#$, so we proceed by generating $\_$.

Theorem 1 below states the correspondence between usages and BPAs. The traces of a usage are all and only the strings that label the computations of the extracted BPA, where resources are suitably renamed to permit model-checking.

Definition 8. *Let σ be a substitution from $\mathsf{Res} \cup \{?\}$ to $\mathsf{Res} \cup \{?\}$. We say that σ is* name-collapsing *when, for some set of dynamic resources $\mathcal{R} \subset \mathsf{Res}_d$:*

$$\begin{array}{llll} \sigma(r) = \# & \text{if } r \in \mathcal{R} & \sigma(\#) = \# & \sigma(?) = ? \\ \sigma(r) = \_ & \text{if } r \in \mathsf{Res}_d \setminus \mathcal{R} & \sigma(\_) = \_ & \sigma(r) = r \quad \text{otherwise} \end{array}$$

We call such σ uniquely-collapsing *if $\sigma(r) = \#$ for exactly one $r \neq \#$.*

Theorem 1. *For each initial usage U, trace η, and name-collapsing σ:*

$$U, \mathcal{R} \xrightarrow{\eta} U', \mathcal{R}' \implies \exists P \,:\, \mathsf{B}(U) \xrightarrow{\eta\sigma} P$$

Example 9. Let $U = \mu h.\, (\nu n.\, \varepsilon + new(n) \cdot \alpha(n) \cdot h)$. Then, $\mathsf{B}(U) = \langle X, \Delta\rangle$, where $\Delta = \{X \triangleq 0 + new(\_) \cdot \alpha(\_) \cdot X + 0 + new(\#) \cdot \alpha(\#) \cdot X\}$. Consider the trace $\eta = new(r_0)\alpha(r_0)new(r_1)\alpha(r_1)new(r_2)\alpha(r_2)$ of U. Let $\sigma = \{\_/r_0, \#/r_1, \_/r_2\}$. Then, $\eta\sigma = new(\_)\alpha(\_)new(\#)\alpha(\#)new(\_)\alpha(\_)$ is a trace in $[\![\langle X, \Delta\rangle]\!]$. □

Example 10. Let $U = \psi[(\nu n.\, new(n) \cdot \alpha(n)) \cdot (\nu n'.\, new(n') \cdot \alpha(n')) \cdot \alpha(?)]$ where the policy ψ asks that the action α cannot be performed twice on the same resource (left-hand side of Fig. 6). Then:

$$\mathsf{B}(U) = [_\psi \cdot \big(new(\_) \cdot \alpha(\_) + new(\#) \cdot \alpha(\#)\big) \cdot \big(new(\_) \cdot \alpha(\_) + new(\#) \cdot \alpha(\#)\big) \cdot \alpha(?)\ \cdot]_\psi$$

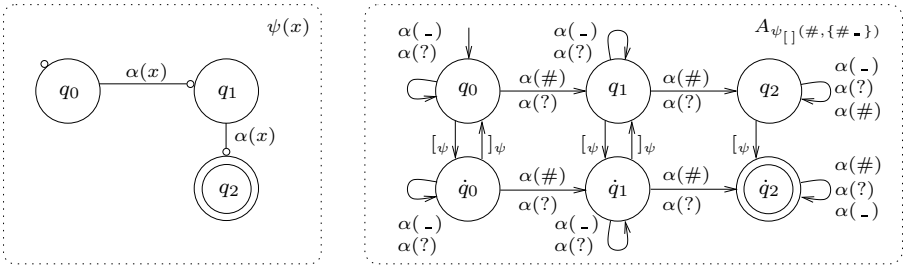

Fig. 6. The policy $\psi(x)$ and its instantiation $A_{\psi_{[]}(\#,\{\#,\_\})}$ (some self-loops omitted).

The BPA $\mathsf{B}(U)$ violates $A_{\psi_{[]}(\#,\{\_,\#\})}$, displayed in the right-hand side of Fig. 6 (where we have omitted all the self-loops for the *new* action). The violation is consistent with the fact that the wildcard ? represents any resource, e.g. $new(r')\alpha(r')new(r)\alpha(r)\alpha(r)$ is a trace of U that violates ψ. Although U is valid whenever $\mathsf{B}(U)$ is valid, such verification technique would not be complete. Consider e.g. ψ' requiring that α is not executed *three* times on the same resource, and let $U' = \psi'[(\nu n.\, new(n)\cdot\alpha(n))\cdot(\nu n'.\, new(n')\cdot\alpha(n'))\cdot\alpha(?)]$. Note that $\mathsf{B}(U')$ violates $\psi'_{[]}(\#)$, while all the traces denoted by U' respect ψ'. Theorem 2 below provides us with a sound and complete verification technique. □

As shown in Ex. 10, the validity of U does not imply the validity of $\mathsf{B}(U)$, so leading to a sound but incomplete decision procedure. The problem is that $\mathsf{B}(U)$ uses the same "witness" resource # for all the resources created in U. This leads to violations of policies, e.g. those that prevent some action from being performed twice (or more) on the same resource, because $\mathsf{B}(U)$ identifies (as #) resources that are distinct in U. To recover a (sound and) complete decision procedure for validity, it suffices to check any trace of $\mathsf{B}(U)$ only *until* the second "witness" resource is generated (i.e. before the second occurrence of $new(\#)$). This is accomplished by composing $\mathsf{B}(U)$ with the "unique witness" automaton through a "weak until" operator. The weak until W is standard; the unique witness $A_\#$ is a finite state automaton that reaches an offending state on those traces containing more than one $new(\#)$ event (see [7] for details).

Example 11. Consider again the usage U' in Ex. 10. The maximal traces generated by $\mathsf{B}(U')$ are shown below.

$$[_{\psi'}\ new(\_)\alpha(\_)\ new(\#)\alpha(\#)\ \alpha(?)\ ]_{\psi'} \quad [_{\psi'}\ new(\#)\alpha(\#)\ new(\_)\alpha(\_)\ \alpha(?)\ ]_{\psi'}$$
$$[_{\psi'}\ new(\_)\alpha(\_)\ new(\_)\alpha(\_)\ \alpha(?)\ ]_{\psi'} \quad [_{\psi'}\ new(\#)\alpha(\#)\ new(\#)\alpha(\#)\ \alpha(?)\ ]_{\psi'}$$

The first three traces above comply with $A_{\psi'_{[]}(\#,\{\#,\_\})}$, which instead is violated by the last trace. Indeed, in $\eta = [_{\psi'}\ new(\#)\alpha(\#)\ new(\#)\alpha(\#)\ \alpha(?)\ ]_{\psi'}$ the two fresh resources are identical, so contrasting with the semantics of usages. To avoid incompleteness, η is not considered in model-checking, since $\eta \lhd (A_{\psi'_{[]}(\#,\{\#,\_\})}\ \mathsf{W}\ A_\#)$, i.e. η is accepted by $A_\#$ and filtered out by the weak until. Note also that there is no need to consider the automaton $A_{\psi'_{[]}(\_,\{\#,\_\})}$, because if a violation has to occur, it will occur on $A_{\psi'_{[]}(\#,\{\#,\_\})}$. □

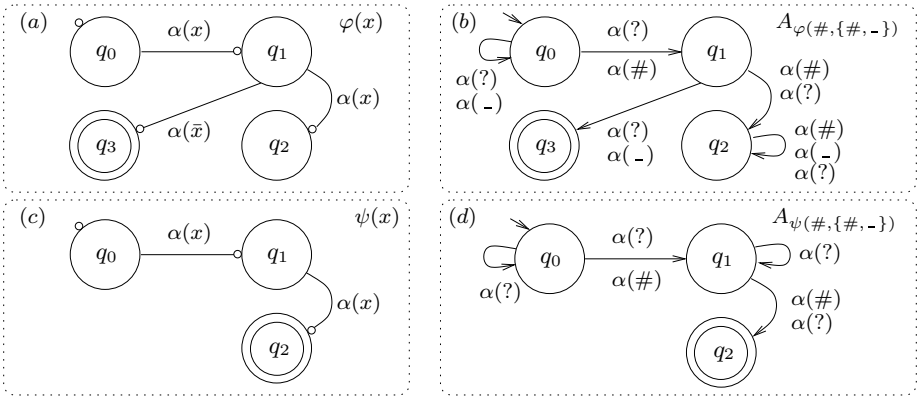

Fig. 7. The usage policies $\varphi(x)$ and $\psi(x)$ and the policy automata $A_{\varphi(\#,\{\#,\_\})}$ and $A_{\psi(\#,\{\#,\_\})}$. The self-loops for the *new* events are omitted.

The following theorem enables us to efficiently verify the validity of a usage U by (i) extracting the BPA $\mathsf{B}(U)$ from U, and (ii) model-checking $\mathsf{B}(U)$ against a finite set of finite state automata.

Theorem 2. *Let* $\Phi(U) = \{\, A_{\varphi_{[\,]}(r_0,\mathsf{R}(U))} \mid r_0, \varphi \in U \,\} \cup \{\, A_{\varphi_{[\,]}(\#,\mathsf{R}(U))} \mid \varphi \in U \,\}$, *where* $\mathsf{R}(U)$ *comprises* $\#$, $\_$, *and all the static resources occurring in* U.

(a) *An initial usage* U *is valid if and only if, for all* $A_{\varphi_{[\,]}(r,\mathcal{R})} \in \Phi(U)$:

$$[\![\mathsf{B}(U)]\!] \lhd A_{\varphi_{[\,]}(r,\mathcal{R})} \,\mathsf{W}\, A_{\#}$$

(b) *The computational complexity of this method is PTIME in the size* U.

Valid usages are recognized by checking all $A_{\varphi_{[\,]}(r,\mathcal{R})}$ in $\Phi(U)$. The set $\Phi(U)$ contains, for each policy φ and static resource r_0, the framed usage automaton $A_{\varphi_{[\,]}(r_0,\mathsf{R}(U))}$. Also, $\Phi(U)$ includes the instantiations $A_{\varphi_{[\,]}(\#,\mathsf{R}(U))}$, to be ready on controlling φ on dynamic resources, represented by the witness $\#$.

Example 12. Let $U = \mu h.\,(\varepsilon + \nu n.\, new(n) \cdot \alpha(n) \cdot h)$. Consider then $U_\varphi = \varphi[U]$, where $\varphi(x)$ asks that, for each resource x, the first occurrence of the event $\alpha(x)$ is necessarily followed by another $\alpha(x)$ (Fig. 7a). Then, U_φ is *not* valid, because e.g. $\eta = [_\varphi new(r)\alpha(r)new(r')\alpha(r')$ is a trace of U_φ, and $\eta \not\models A_{\varphi_{[\,]}(r,\{r,r'\})}$ (the non-framed policy automaton is in Fig. 7b). So, $\Phi(U_\varphi) = \{A_{\varphi_{[\,]}(\#,\{\#,\_\})}\}$, and:

$$\mathsf{B}(U_\varphi) = \langle [_\varphi \cdot X \cdot]_\varphi, X \triangleq \varepsilon + (new(\#) \cdot \alpha(\#) \cdot X) + (new(\_) \cdot \alpha(\_) \cdot X) \rangle$$

and the trace $[_\varphi new(\#)\alpha(\#)new(\_)\alpha(\_) \in [\![\mathsf{B}(U_\varphi)]\!]$ drives the framed automaton $A_{\varphi_{[\,]}(\#,\{\#,\_\})}$ to an offending state. Consider now $U_\psi = \psi[U]$, where the policy $\psi(x)$ says that the action α cannot be performed twice on the same resource x (Fig. 7c). We have that $\Phi(U_\psi) = \{A_{\psi_{[\,]}(\#,\{\#,\_\})}\}$, and:

$$\mathsf{B}(U_\psi) = \langle [_\psi \cdot X \cdot]_\psi, X \triangleq \varepsilon + (new(\#) \cdot \alpha(\#) \cdot X) + (new(\_) \cdot \alpha(\_) \cdot X) \rangle$$

Although U_ψ obeys ψ, the BPA does not, since $[_\psi new(\#)\alpha(\#)new(\#)\alpha(\#)]_\psi$ violates $A_{\psi_{[\,]}(\#,\{\#,\_\})}$ (the non-framed instantiation $A_{\psi(\#,\{\#,\_\})}$ is in Fig. 7d). Completeness is recovered through the weak until and unique witness automata, that filter the traces, like the one above, where $new(\#)$ is fired twice. □

Example 13. Recall from Ex. 8 the policy φ_{File} (only open files can be read/written) and consider the policy φ_{DoS} that forbids the creation of more than k files. Let:

$$U = \varphi_{\mathsf{File}}[\varphi_{\mathsf{DoS}}[\mu h.\, \varepsilon + \nu n.\, new(n) \cdot open(n) \cdot read(n) \cdot close(n) \cdot h]]$$

Then, $\Phi(U) = \{\varphi_{\mathsf{File}}(\#), \varphi_{\mathsf{DoS}}(\#)\}$, and $\mathsf{B}(U) = \langle [_{\varphi} \cdot [_{\varphi_{\mathsf{DoS}}} \cdot X \cdot]_{\varphi_{\mathsf{DoS}}} \cdot]_{\varphi},\, \Delta \rangle$, where Δ comprises the following definition:

$$\begin{aligned} X \;\triangleq\; & \varepsilon + (new(\_) \cdot open(\_) \cdot read(\_) \cdot close(\_) \cdot X) \\ & + (new(\#) \cdot open(\#) \cdot read(\#) \cdot close(\#) \cdot X) \end{aligned}$$

Note that each computation of $\mathsf{B}(U)$ obeys $A_{\varphi_{\mathsf{File}[\,]}(\#,\{\#,\_\})}$, while there exist computations that violate $A_{\varphi_{\mathsf{DoS}[\,]}(\#,\{\#,\_\})}$. □

To handle polyadic policies (say with arity k), our technique can be adjusted as follows. We use k witnesses $\#_1, \ldots, \#_k$. The set $\Phi(U)$ is computed as $\Phi(U) = \{\, A_{\varphi_{[\,]}(r_1,\ldots,r_k,\mathsf{R}(U))} \mid \varphi \in U,\ \forall i.\ r_i \in U \vee r_i \in \{\#_1, \ldots, \#_k\} \,\}$. Moreover, now $A_{\#}$ must check that each $\#_i$ has at most an associated *new* event. The transformation of usages into BPAs should then be modified as follows:

$$\begin{aligned} \mathsf{B}_{\_}(\nu n.U)_{\Theta} &= \mathsf{B}_{\_}(U\{\_/n\})_{\Theta} + \mathsf{B}_{\#_1}(U\{\#_1/n\})_{\Theta} \\ \mathsf{B}_{\#_i}(\nu n.U)_{\Theta} &= \mathsf{B}_{\#_i}(U\{\_/n\})_{\Theta} + \mathsf{B}_{\#_{i+1}}(U\{\#_{i+1}/n\})_{\Theta} \qquad \text{if } i < k \\ \mathsf{B}_{\#_k}(\nu n.U)_{\Theta} &= \mathsf{B}_{\#_k}(U\{\_/n\})_{\Theta} \end{aligned}$$

Theorem 2 still holds. The complexity of model-checking is still PTIME in the size of U, and EXPTIME in k. Note that being EXPTIME in k has no big impact in practice, as one expects k to be very small even for complex policies.

5 Conclusions

We proposed a model for policies that control the usage of resources. Usage policies can be enforced through finite state automata. A basic calculus of usages was presented to describe the patterns of resource access and creation, and obligation to respect usage policies. We call a usage *valid* when *all* its possible traces comply with *all* the relevant usage policies. In spite of the augmented flexibility given by resource creation and by policy parametrization, we devised an efficient (polynomial-time) and complete model-checking technique for deciding the validity of usages. Our technique manages to represent the generation of an unbounded number of resources in a finitary manner. Yet, we do not lose the possibility of verifying interesting security properties of programs.

Local usage policies were originally introduced in [6]. The current work thoroughly improves and simplifies them. Usage policies can now have an arbitrary number of parameters, which augments their expressive power and allows for modelling significant real-world policies (see Sec. 1). The model checking technique in [6] has exponential time complexity, while the current one is polynomial.

Extensions. A first simple extension to our model is to assign a *type* to resources. To do that, the set Act of actions is partitioned to reflect the types of resources (e.g. $\mathsf{Act} = \mathsf{File} \cup \mathsf{Socket} \cup \cdots$), where each element of the partition contains the actions admissible for the given type (e.g. $\mathsf{File} = \{new_{\mathsf{File}}, open, close, read, write\}$,

where new_{T} represents creating a new resource of type T). The syntax of the ν-constructor is extended with the type, i.e. $\nu n : T.\ U$. Validity should now check that the actions fired on a resource also respect its type. Our model checking technique can be smoothly adapted by using a #-witness for each type (e.g. $\#_{\mathsf{File}}, \#_{\mathsf{Socket}}$, *etc.*), to be dealt with the corresponding "unique witness" automata (e.g. $A_{\#_{\mathsf{File}}}, A_{\#_{\mathsf{Socket}}}$, *etc.*). The time complexity is unaltered.

The language of usages has two simple extensions. The first consists in attaching policies to resources upon creation, similarly to [22]. The construct $\nu n : \varphi.\ U$ is meant to enforce the policy φ on the freshly created resource. An encoding into the existing constructs is possible. First, the whole usage has to be sandboxed with the policy $\varphi_\nu(x)$, obtained from $\varphi(x)$ by adding a new initial state q_ν and an edge labelled $check(x)$ from q_ν to the old initial state. The encoding then transforms $\nu n : \varphi.\ U$ into $\nu n.\ check(n) \cdot U$. The second extension consists in allowing parallel usages $U|V$. Model checking is still possible by transforming usages into Basic Parallel Processes [16] instead of BPAs. However, the time complexity becomes exponential in the number of parallel branches [24].

Another line of investigation will be that of extending our techniques to verify policies of other behavioural types for Web services, e.g. [15, 14, 2].

Related work. Many authors [17, 18, 23, 25] mix static and dynamic techniques to transform programs and make them obey a given policy. Our model allows for local, polyadic policies and events parametrized over dynamically created resources, while the above-mentioned papers only consider global policies and no parametrized events. Polymer [11] is a language for specifying, composing and enforcing (global) security policies. In the lines of *edit automata* [10], a policy can intervene in the program trace, to insert or suppress some events. Policy composition can then be unsound, because the events inserted by a policy may interfere with those monitored by another policy. To cope with that, the programmer must explicitly fix the order in which policies are applied. Being Turing-equivalent, Polymer policies are more expressive than ours, but this gain in expressivity has some disadvantages. First, a policy may potentially be unable to decide whether an event is accepted or not (i.e. it may not terminate). Second, no static guarantee is given about the compliance of a program with the imposed policy. Run-time monitoring is then necessary to enforce the policy, while our model-checking tecnique may avoid this overhead. A typed λ-calculus is presented in [22], with primitives for creating and accessing resources, and for defining their permitted usages. Type safety guarantees that well-typed programs are resource-safe, yet no effective algorithm is given to check compliance of the inferred usages with the permitted ones. The policies of [22] can only speak about the usage of *single* resources, while ours can span over many resources, e.g. a policy requiring that no socket connections can be opened after a local file has been read. A further limitation of [22] is that policies are attached to resources. In mobile code scenarios, e.g. a browser that runs untrusted applets, it is also important that you can impose constraints on how external programs manage the resources created in your local environment. E.g., an applet may create an unbounded number of resources on the browser site, and never re-

lease them, so leading to denial-of-service attacks. Our local policies can deal with this kind of situations. *Shallow history automata* are investigated in [20]. These automata can keep track of the *set* of past access events, rather than the *sequence* of events. Although shallow history automata can express some interesting security properties, they are clearly less expressive than our usage policies. A model for history-based access control is proposed in [26]. It uses control-flow graphs enriched with permissions and a primitive to check them, similarly to [3]. The run-time permissions are the intersection of the static permissions of all the nodes visited in the past. The model-checking technique can decide in EXPTIME (in the number of permissions) if all the permitted traces of the graph respect a given regular property on its nodes. Unlike our local policies, that can enforce any regular policy on traces, the technique of [26] is less general, because there is no way to enforce a policy unless it is encoded as a suitable assignment of permissions to nodes.

Acknowledgments. This research has been partially supported by EU-FETPI Global Computing Project IST-2005-16004 Sensoria (Software Engineering for Service-Oriented Overlay Computers).

References

1. M. Abadi and C. Fournet. Access control based on execution history. In *Proc. 10th Annual Network and Distributed System Security Symposium*, 2003.
2. L. Acciai and M. Boreale. Spatial and behavioral types in the π-calculus. In *Proc. CONCUR*, 2008.
3. M. Bartoletti, P. Degano, and G. L. Ferrari. Static analysis for stack inspection. In *International Workshop on Concurrency and Coordination*, 2001.
4. M. Bartoletti, P. Degano, and G. L. Ferrari. History based access control with local policies. In *Proc. Fossacs*, 2005.
5. M. Bartoletti, P. Degano, and G. L. Ferrari. Types and effects for secure service orchestration. In *Proc. 19th CSFW*, 2006.
6. M. Bartoletti, P. Degano, G. L. Ferrari, and R. Zunino. Types and effects for resource usage analysis. In *Proc. Fossacs*, 2007.
7. M. Bartoletti, P. Degano, G. L. Ferrari, and R. Zunino. Model checking usage policies. Technical Report TR-08-06, Dip. Informatica, Univ. Pisa, 2008.
8. M. Bartoletti, P. Degano, G. L. Ferrari, and R. Zunino. Semantics-based design for secure web services. *IEEE Transactions on Software Engineering*, 34(1), 2008.
9. M. Bartoletti and R. Zunino. LocUsT: a tool for checking usage policies. Technical Report TR-08-07, Dip. Informatica, Univ. Pisa, 2008.
10. L. Bauer, J. Ligatti, and D. Walker. More enforceable security policies. In *Foundations of Computer Security (FCS)*, 2002.
11. L. Bauer, J. Ligatti, and D. Walker. Composing security policies with Polymer. In *Proc. PLDI*, 2005.
12. J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.
13. D. F. C. Brewer and M. J. Nash. The chinese wall security policy. In *Proc. of Symp. on Security and Privacy*, 1989.

14. R. Bruni and L. G. Mezzina. Types and deadlock freedom in a calculus of services, sessions and pipelines. In *Proc. AMAST*, 2008.
15. L. Caires and H. T. Vieira. Typing conversations in the conversation calculus. Technical Report 3/08, DI/FCT/UNL, 2008.
16. S. Christensen. *Decidability and Decomposition in Process Algebras*. PhD thesis, Edinburgh University, 1993.
17. T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Proc. 27th ACM SIGPLAN-SIGACT PoPL*, 2000.
18. Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: a retrospective. In *Proc. 7th New Security Paradigms Workshop*, 1999.
19. J. Esparza. On the decidability of model checking for several μ-calculi and Petri nets. In *Proc. 19th Int. Colloquium on Trees in Algebra and Programming*, 1994.
20. P. W. Fong. Access control by tracking shallow execution history. In *IEEE Symposium on Security and Privacy*, 2004.
21. A. Igarashi and N. Kobayashi. Type reconstruction for linear -calculus with i/o subtyping. *Inf. Comput.*, 161(1):1–44, 2000.
22. A. Igarashi and N. Kobayashi. Resource usage analysis. In *Proc. 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.
23. K. Marriott, P. J. Stuckey, and M. Sulzmann. Resource usage verification. In *Proc. First Asian Programming Languages Symposium*, 2003.
24. R. Mayr. *Decidability and Complexity of Model Checking Problems for Infinite-State Systems*. PhD thesis, Technischen Universität München, 1998.
25. P. Thiemann. Enforcing safety properties using type specialization. In *Proc. ESOP*, 2001.
26. J. Wang, Y. Takata, and H. Seki. HBAC: A model for history-based access control and its model checking. In *Proc. ESORICS*, 2006.