Math. Struct. in Comp. Science (2015), *vol.* 25, *pp.* 710–763. © Cambridge University Press 2014
doi:10.1017/S096012951200093X First published online 10 November 2014

Model checking usage policies[†]

MASSIMO BARTOLETTI[‡], PIERPAOLO DEGANO[§],
GIAN LUIGI FERRARI[§] and ROBERTO ZUNINO[¶]

[‡]*Dipartimento di Matematica e Informatica, Università degli Studi di Cagliari, via Ospedale 72, 09124 Cagliari, Italy*
Email: bart@unica.it
[§]*Dipartimento di Informatica, Università di Pisa, Italy*
[¶]*DISI, Università di Trento and COSBI, Italy*

Received 7 January 2011; revised 21 November 2011

We study *usage automata*, a formal model for specifying policies on the usage of resources. Usage automata extend finite state automata with some additional features, parameters and guards, that improve their expressivity. We show that usage automata are expressive enough to model policies of real-world applications. We discuss their expressive power, and we prove that the problem of telling whether a computation complies with a usage policy is decidable. The main contribution of this paper is a model checking technique for usage automata. The model is that of *usages*, i.e. basic processes that describe the possible patterns of resource access and creation. In spite of the model having infinite states, because of recursion and resource creation, we devise a polynomial-time model checking technique for deciding when a usage complies with a usage policy.

1. Introduction

One of the standard approaches to enforce security policies in computer systems is *access control* (Samarati and de Capitani di Vimercati 2001; Sandhu and Samarati 1994), the process of monitoring the run time accesses to resources in order to guarantee that all and only the authorized accesses can take place. Modern security-aware programming languages place some form of access control mechanism in their run-time environment. These mechanisms can be coupled with static analysis techniques, which can leverage the overhead of dynamic monitoring, and can also enforce certain classes of security policies that are otherwise impossible to achieve by dynamic monitoring alone (Banerjee and Naumann 2004; Fournet and Gordon 2003).

Devising expressive, flexible and efficient mechanisms to control the usage of resources is a major challenge in the design and implementation of security-aware programming languages. The problem becomes crucial by the current programming practice, where it is common to pick from the web some scripts, or plugins, or code snippets, and assemble them into a bigger program, with little or no control about the security of the whole.

[†] This work has been partially supported by EU under grant FP7-257414 (Project ASCENS), and by Aut. Region of Sardinia under grants L.R.7/2007 CRP2-120 (Project TESLA) and CRP-17285 (Project TRICS).

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

A classical approach is that of using an execution monitor to enforce a global invariant that must hold at any point of the execution. This involves monitoring the execution of a program, and taking actions to prevent it from violating the prescribed policy. The events observed by these monitors are accesses to sensible resources, e.g. opening socket connections, reading/writing files, allocating/deallocating memory, *etc*.

Although several models for execution monitors have been proposed over the years, e.g. (Abadi and Fournet 2003; Bauer *et al.* 2002; Edjlali *et al.* 1999; Schneider 2000; Skalka and Smith 2004; Martinelli and Mori 2007), current mainstream programming languages only offer basic and ad hoc mechanisms to specify and enforce custom policies on the usage of resources. For instance, Java and C♯ feature a stack-based access control mechanism, while support for more expressive (history-based) mechanisms is not provided. This makes the functional aspects difficult to keep apart from security, in contrast with the principle of separation of concerns. Indeed, a common programming practice is to implement the enforcement mechanism by explicitly inserting the needed checks into the code. Since forgetting even a single check might compromise the security of the whole application, programmers have to inspect their code very carefully. This may be cumbersome even for small programs, and it may also lead to unnecessary checking.

A relevant issue is finding a good compromise between the expressiveness of policies and the efficiency of the enforcement mechanism. It is also important that the enforcement mechanism can be implemented transparently to programmers, and with a small run-time overhead. Static analysis techniques may be needed to make this overhead tolerable.

1.1. *Contributions of the paper*

In this paper we study *usage automata*, a pure formalism for defining and enforcing policies on the usage of resources. Usage automata extend finite state automata, by allowing edges to carry *variables* and *guards*. Variables represent existentially quantified resources, to be instantiated with actual resources chosen from an *infinite* set. Usage automata are then suitable for expressing policies with parameters ranging over infinite domains (much alike the parameters of a procedure, see e.g. (Shemesh and Francez 1994)). For instance, a usage automaton might state that 'for all files x, you can read or write x only if you have opened x, and not closed it in the meantime'. Guards represent conditions among variables and resources, e.g. 'a file y cannot be read if some other file $x \neq y$ has been read in the past'.

The main contribution of this paper is a model checking technique for usage automata. The structure under consideration is that of *usages*, which are basic processes that describe patterns of resource access and creation. We have shown in previous papers, e.g. (Bartoletti *et al.* 2006; Bartoletti *et al.* 2009b; Bartoletti *et al.* 2009c), how to statically analyse a program to infer a usage that correctly over-approximates its possible run-time traces. Here, we devise a model checking algorithm to decide whether a usage respects a given policy, expressed as a usage automaton. When a program passes this static check, the run-time monitor can be disposed, so leveraging the overhead of access control (see e.g. (Erlingsson and Schneider 1999; Wallach *et al.* 2001) for some statistics on the overhead of stack-based mechanisms).

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

The model checking problem is not completely trivial, since usages are infinite state systems. Actually, they have two different sources of non-finiteness: first, they have a recursion construct; second, they can generate an unbounded number of fresh resources. The proposed technique correctly and completely handles the case in which the number of fresh resources generated by the usage has no bound known at static time. We solve this problem by suitably abstracting resources, so that only a finite number of *witnesses* needs to be considered.

Our model checking algorithm runs in polynomial time on the size of the usage, with an exponential factor in the number of variables in the usage automaton. However, the exponential should not be significant in practice, since many policies, also from real-world applications, have a small number of variables (e.g. this is the case for the policies typically found in bulletin board systems like phpBB (Bartoletti *et al.* 2009a)).

Our technique scales to the case where, instead of a single global policy that spans over the whole program, one has to check the usage against a set of *local* policies (Bartoletti *et al.* 2009c). Local policies generalise both history-based global policies and local checks spread over program code. They exploit a scoping mechanism to allow the programmer to 'sandbox' arbitrary program fragments. Local policies are particularly relevant in service-oriented computing (SOC), since they smoothly allow for safe composition of services (possibly involving mobile code) with different security requirements (Bartoletti *et al.* 2006; Bartoletti *et al.* 2008b). As a pragmatic support for our approach, we have designed and implemented a tool, named LocUsT for 'local usage policy tool', for model checking usage policies (Bartoletti *et al.* 2011) according to the theoretical results presented here.

1.2. *Overview of the paper*

We illustrate our proposal with the help of a simple example. Consider an infinite set of objects $r_1, r_2, \ldots$, that can be dynamically created, accessed and disposed through the actions *new*, *read* and *dispose*, respectively. For instance, the following sequence of events models an execution trace (in this paper, we neglect the actual programs that generate traces):

$$\eta_0 \;=\; new(r_1)\, read(r_1)\, read(r_1)\, new(r_2)\, dispose(r_2).$$

A relevant policy disciplinating the usage of such objects prescribes that disposed objects can no longer be accessed (or disposed); similarly, objects must have been created before they can be accessed or disposed. Additionally we require that, whenever you are accessing an object, it has to be the only one alive (i.e. all the other created objects have been disposed).

This policy is specified through the usage automaton φ in Figure 1. It is a sort of finite state automaton, where the variables x, y in the edges can be instantiated to actual objects, and the guard $y \neq x$ enables the edge from q_1 to *fail*. A trace violates the policy specified by φ if an instantiation of φ exists which leads to the state *fail*. For instance, the trace η_0 above complies with the policy, as well as its extension $\eta_0\, dispose(r_1)$, while the following

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

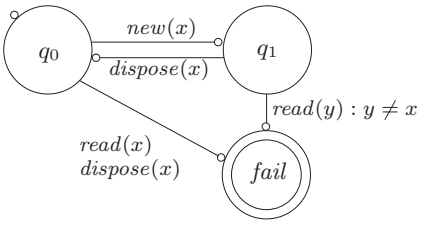


Fig. 1. A usage automaton φ.

traces violate the policy:

$$\eta_1 = \eta_0\, read(r_2) \qquad \text{because } r_2 \text{ has been disposed}$$
$$\eta_2 = \eta_0\, new(r_3) read(r_1) \qquad \text{because both } r_1 \text{ and } r_3 \text{ are alive.}$$

Usage automata will be formally introduced in Section 2, where we also show some further examples. Policy compliance will be defined by instantiating a usage automaton to (an infinite set of) finite state automata. Note that we follow a default-allow approach, that is an event is permitted whenever not explicitly forbidden. When we instantiate a usage automaton, we therefore add to all states a self-loop for each non-forbidden event. A trace violates a policy φ iff it belongs to the language of some instantiations of φ. In Section 3, we will prove policy compliance decidable: indeed, it suffices to consider a *finite* set of such instantiations. In Section 4, we study some expressiveness issues. The main result there is that the expressive power of usage automata increases with the number of variables occurring therein.

The basic calculus of usages is introduced in Section 5. It is a nominal calculus without synchronization. For instance:

$$U_0 = \mu h.\, \big(\varepsilon + \nu n.\, new(n) \cdot read(n) \cdot dispose(n) \cdot h\big)$$

is a usage representing traces of the form:

$$new(r_1)\ read(r_1)\ dispose(r_1)\ new(r_2)\ read(r_2)\ dispose(r_2)\ \ldots$$

Above, μh stands for recursion, ν is a name binder à la π-calculus (Milner *et al.* 1992), ε is the empty usage, $+$ is non-deterministic choice, $\cdot$ is sequential composition.

The model checking problem is specified as follows. Given a usage U and a usage automaton φ, decide whether all the traces denoted by U comply with φ (in model checking terminology, the structure is U and the property is φ). Our model checking technique follows in Section 6, together with our main results, i.e. its correctness and polynomial time complexity. The algorithm consists of several steps. Let $\mathcal{W} = \{\#_1, \ldots, \#_k\}$ be the finite set of witnesses, where k is the number of variables in φ. Then:

1. map the usage U into a Basic Process Algebra term $\mathsf{B}_{\mathcal{W}}(U)$ (Definition 20)
2. construct a finite set $A^1_\varphi, \ldots, A^n_\varphi$ of instantiations of φ (Definition 3)
3. construct the finite state automata $A^1_\varphi \,\mathsf{W}\, \mathsf{A}_{\mathcal{W}}, \ldots, A^n_\varphi \,\mathsf{W}\, \mathsf{A}_{\mathcal{W}}$ (Definitions 34 and 31)
4. model check $\mathsf{B}(U)$ against each of the $A^i_\varphi \,\mathsf{W}\, \mathsf{A}_{\mathcal{W}}$.

The goal of the transformation of the usage U into a basic process algebra (BPA) process (Bergstra and Klop 1985) is to bound the number of resources in the traces of

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

$\mathsf{B}_{\mathcal{W}}(U)$ to a constant, depending on the number of variables occurring in φ. The crucial point is preserving policy compliance: the traces of U comply with φ if and only if, the traces of $\mathsf{B}_{\mathcal{W}}(U)$ do (Theorem 35). The 'only if' part is guaranteed by construction; for the 'if' part we resort to an additional technical construction, i.e. we massage the instantiations of φ to prevent the same witness to be created twice (this is accomplished by the unique witness automaton $\mathsf{A}_{\mathcal{W}}$, composed with A^i_φ through the weak until operator W). The last step can be performed by standard techniques (Esparza 1994).

For instance, our technique correctly detects that (all the traces of) the usage U_0 above complies with φ. As further examples, let:

$$U_1 = \mu h.\,\big(\varepsilon + \nu n.\,new(n)\cdot(\mu h'.\,\varepsilon + read(n)\cdot h')\cdot dispose(n)\cdot h\big)$$
$$U_2 = \mu h.\,\big(\varepsilon + \nu n.\,new(n)\cdot(\mu h'.\,dispose(n) + read(n)\cdot h')\cdot dispose(n)\cdot h\big)$$
$$U_3 = \mu h.\,\big(\varepsilon + \nu n.\,new(n)\cdot(\mu h'.\,\varepsilon + dispose(n) + read(n)\cdot h')\cdot h\big).$$

The usage U_1 is similar to U_0, but now each object can be read multiple times (modelled by the inner recursion $\mu h'$). In U_2, each object is illegally disposed twice, by the last step of the inner recursion and by the *dispose* in the outer recursion. In U_3, the *dispose* can be skipped, so an object can be illegally read when another object is still alive. Our model checking technique correctly detects that U_1 complies with φ, and that U_2 and U_3 do not. Our LocUsT tool supports these facts.

In Section 7, we extend our techniques to the case of local policies. There, we reduce the problem of model checking *local* policies to the problem of checking *global* policies, so making it possible to reuse the machinery developed in Section 6.

In Section 8 we discuss some related work, and in Section 9 we conclude.

All the proofs of our statements are contained either in the main body of the paper, or in Appendix A.

2. Usage automata

We start by introducing the needed syntactic categories and some notation in Table 1. Some additional notation will be introduced later on in Tables 2 and 6. Some of the symbols defined therein will only be used later on.

We assume a denumerable set Res of *resources*, partitioned as follows:

— Res_s is the (finite) set of *static* resources, i.e. those resources that are already available in the environment.
— Res_d is the (infinite) set of *dynamic* resources, i.e. those resources that will be freshly created at run time.
— $\{\#_i\}_{i\in\mathbb{N}}$ is the set of *witness* resources.
— $_$ is the *dummy* resource.

Static resources are kept distinct from dynamic ones, as this slightly increases the expressiveness of usage automata (see Proposition 6). The witness resources and the dummy will only be exploited later on by our model checking machinery, to abstract from the actual identity of dynamic resources. The reason for introducing dummy/witness

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

Table 1. *Notation for usage automata.*

$r, r', \ldots \in \mathsf{Res}$	Resources, disjoint union of:
Res_s	static resources
Res_d	dynamic resources
$\{\#_i\}_{i\in\mathbb{N}}$	witness resources
$_$	dummy resource
$\mathcal{R}$	A set of resources
$\alpha, \alpha', new, \ldots \in \mathsf{Act}$	Actions (a finite set)
$\alpha(\vec{r}), \ldots \in \mathsf{Ev} = \mathsf{Act} \times \mathsf{Res}^*$	Events, with $\lvert\alpha\rvert = k$ if $\vec{r} = (r_1, \ldots, r_k)$
$\eta, \eta', \ldots \in \mathsf{Ev}^*$	Traces (ε is the empty one, $\varepsilon\eta = \eta = \eta\varepsilon$)
$x, x', \ldots \in \mathsf{Var}$	Variables
$\xi, \xi', \ldots \in \mathsf{Res} \cup \mathsf{Var}$	Resources or variables
$g, g', \ldots \in \mathsf{G}$	Guards
$\varphi, \varphi', \ldots$	Usage automata, with arity $\lvert\varphi\rvert = \lvert var(\varphi)\rvert$
$var(g), var(\varphi)$	Set of variables in guards / usage automata
$res(\eta), res(\varphi)$	Set of resources in traces / usage automata
$R(\eta, \varphi)$	Set of resources $res(\eta) \cup res(\varphi) \cup \{\#_1, \ldots, \#_{\lvert\varphi\rvert}\} \setminus \{_\}$
$\sigma, \sigma', \ldots$	Mapping from variables to resources
$A_\varphi(\sigma, \mathcal{R})$	Finite state automaton, instantiation of φ
$\eta \lhd A_\varphi(\sigma, \mathcal{R})$	η is not in the language of $A_\varphi(\sigma, \mathcal{R})$
$\eta \models \varphi$	η respects φ
$\kappa, \kappa', \ldots$	Collapsing (function from Res_d to $\{\#_i\}_{i\in\mathbb{N}} \cup \{_\}$)

resources at this point is purely technical: it allows our definitions and statements to be precise about the range of resource metavariables.

Resources can be used through *events* of the form $\alpha(r_1, \ldots, r_k) \in \mathsf{Ev}$, where $\alpha \in \mathsf{Act}$ is an *action* (of arity $\lvert\alpha\rvert = k$), fired on the target resources $r_1, \ldots, r_k \in \mathsf{Res}$. We assume a special action *new* (of arity 1), that represents the creation of a fresh resource. This means that for each dynamically created resource r, the event $new(r)$ must precede any other action with r among its targets. For notational convenience, when the target resources of an action α are immaterial, we stipulate that α acts on some special (static) resource, and we write just α for such an event.

A *trace* $\eta \in \mathsf{Ev}^*$ is a finite sequence of events, which abstractly represent the usage of resources in a computation. A *usage policy* is a set of traces, which represent those computations that respect some safe pattern of resource usage.

We specify usage policies through usage automata, which extend finite state automata (FSA) to deal with infinite alphabets. The labels on the edges of usage automata may contain variables and guards. Variables represent existentially quantified resources; guards represent conditions among variables and resources.

Before introducing usage automata, we formally define the syntax and semantics of guards.

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

Table 2. *Additional notation for usages and model checking.*

$n, n', \ldots \in \mathsf{Nam}$	Names
$\rho, \rho', \ldots \in \mathsf{Res} \cup \mathsf{Nam}$	Resources or names
$\alpha(\vec{\rho}), \ldots \in \mathsf{Ev}_{\mathsf{Nam}}$	Events, with $\alpha(\vec{\rho}) \in \mathsf{Act} \times (\mathsf{Res} \cup \mathsf{Nam})^*$ and $\lvert\alpha\rvert = \lvert\vec{\rho}\rvert$
$U, U', \ldots$	Usages
$res(U)$	Set of resources occurring in U
$R(U, \varphi)$	Set of resources $res(U) \cup res(\varphi) \cup \{\#_1, \ldots, \#_{\lvert\varphi\rvert}\} \setminus \{_\}$
$P, P', Q, \ldots$	Basic process algebra (BPA) processes
$\mathsf{B}_{\mathcal{W}}(U)_{\Theta}$	BPA process extracted from U
$\mathsf{A}_{\mathcal{W}}$	Unique witness automaton for witnesses $\mathcal{W}$
W	Weak until automaton

Definition 1 (guards). Let $\xi, \xi' \in \mathsf{Res}_s \cup \mathsf{Var}$. We inductively define the set G of guards as follows:

$$\mathsf{G} \ ::= \ \mathit{true} \mid \xi = \xi' \mid \neg\mathsf{G} \mid \mathsf{G} \wedge \mathsf{G}.$$

For all guards g and substitutions $\sigma : var(g) \to \mathsf{Res}$ (where $var(g)$ is the set of variables occurring in g), we say that σ *satisfies* g, in symbols $\sigma \models g$, when:

— $g = \mathit{true}$, or
— $g = (\xi = \xi')$ and $\sigma(\xi) = \sigma(\xi')$, or
— $g = \neg g'$ and it is not the case that $\sigma \models g'$ or
— $g = g' \wedge g''$, and $\sigma \models g'$ and $\sigma \models g''$.

Note that dynamic resources cannot be mentioned in guards (and in usage automata as well), as they are unknown at static time: actually, they will be dealt with suitable instantiations of the variables in $\vec{\xi}$. We feel free to write $\xi \neq \xi'$ for $\neg(\xi = \xi')$, and $g \vee g'$ for $\neg(\neg g \wedge \neg g')$.

We now define the syntax of usage automata. They are similar to FSAs, but with an infinite alphabet and a different acceptance condition. Instead of plain symbols, an edge of a usage automaton is labelled by an event $\alpha(\vec{\xi})$ and by a guard g. As for guards, the elements of the target vector $\vec{\xi}$ can either be static resources or variables.

Definition 2 (usage automata). A *usage automaton* φ is a 5-tuple $\langle S, Q, q_0, F, E\rangle$ where:

— $S \subseteq \mathsf{Act} \times (\mathsf{Res}_s \cup \mathsf{Var})^*$ is the alphabet, such that $\lvert\alpha\rvert = \lvert\vec{\xi}\rvert$ for all $\langle\alpha, \vec{\xi}\rangle \in S$,
— Q is a finite set of states,
— $q_0 \in Q$ is the start, initial state,
— $F \subset Q$ is the set of final 'offending' states,
— $E \subseteq Q \times S \times \mathsf{G} \times Q$ is a finite set of edges.

Graphically, we write an edge $E = \langle q, \langle\alpha, \vec{\xi}\rangle, g, q'\rangle$, as follows:

$$q \xrightarrow{\alpha(\vec{\xi}) : g} q'.$$

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

We denote with $|\varphi|$ the *arity* of φ, i.e. the number of variables occurring in φ. In the graphical representation of usage automata, we stipulate that the symbol $\circ$ without incoming edges marks a state as initial; a double circle marks a state as final. Below in this section, we will show several examples of usage automata.

Once the variables in a usage automaton are instantiated to actual resources, it can be used to accept the traces respecting a usage policy. Unlike FSAs, the final states in usage automata denote *rejection*, i.e. violation of the policy. The formal definition of acceptance will follow in a while. Note that at this point we have two different ways of specifying policies: set-theoretically or by usage automata. While the first is not an *executable* specification, the decidability of policy compliance (Theorem 5) makes usage automata effective as execution monitors.

To define the usage policy denoted by a usage automaton, we must instantiate the variables occurring on its edges. Given a set of resources $\mathcal{R}$, we shall consider *all* the possible instantiations of the variables of φ to resources in $\mathcal{R}$. This operation yields a (possibly infinite) set of automata which have a finite number of states and a possibly infinite number of edges. The usage policy denoted by φ is then the union of the (complemented) languages recognized by the automata resulting from the instantiation of φ. The complement (w.r.t. Ev^*) is needed because the final states of a usage automaton φ denote a violation.

Definition 3 (compliance). Let $\varphi = \langle S, Q, q_0, F, E\rangle$ be a usage automaton, let $\mathcal{R} \subseteq \mathsf{Res}$, and let $\sigma : var(\varphi) \to \mathcal{R} \setminus \{_\}$. We define the automaton $A_\varphi(\sigma, \mathcal{R}) = \langle \Sigma, Q, q_0, F, \delta\rangle$ as follows:

$$\Sigma = \left\{ \alpha(\vec{r}) \,\middle|\, \alpha \in \mathsf{Act} \text{ and } \vec{r} \in \mathcal{R}^{|\alpha|} \right\}$$
$$\delta = \mathcal{X} \cup complete(\mathcal{X}, \mathcal{R})$$

where the sets of edges $\mathcal{X}$ and $complete(\mathcal{X}, \mathcal{R})$ are defined as follows:

$$\mathcal{X} = \left\{ q \xrightarrow{\alpha(\vec{\xi}\sigma)} q' \,\middle|\, q \xrightarrow{\alpha(\vec{\xi}):g} q' \in E \text{ and } \sigma \models g \right\}$$
$$complete(\mathcal{X}, \mathcal{R}) = \left\{ q \xrightarrow{\alpha(\vec{r})} q \,\middle|\, \alpha \in \mathsf{Act},\ \vec{r} \in \mathcal{R}^{|\alpha|}, \text{ and } \nexists q' \in Q : q \xrightarrow{\alpha(\vec{r})} q' \in \mathcal{X} \right\}.$$

Given a trace $\eta \in \mathsf{Ev}^*$, we say that:

— $\eta \lhd A_\varphi(\sigma, \mathcal{R})$, when there are no runs of $A_\varphi(\sigma, \mathcal{R})$ on η leading to a final state.
— η *respects* φ (in symbols, $\eta \models \varphi$) when, for all $\sigma : var(\varphi) \to \mathsf{Res} \setminus \{_\}$, we have that $\eta \lhd A_\varphi(\sigma, \mathsf{Res})$.
— η *violates* φ (in symbols, $\eta \not\models \varphi$) when it is not the case that η respects φ.

The usage policy defined by φ is the set of traces $\{\, \eta \,|\, \eta \models \varphi \,\}$.

To graphically stress the difference between usage automata and their instantiations, we render the transitions of the first by $\multimap$, while for instantiations we use arrows $\to$, like for FSAs. Whenever unambiguous, we use φ for the usage policy it defines.

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

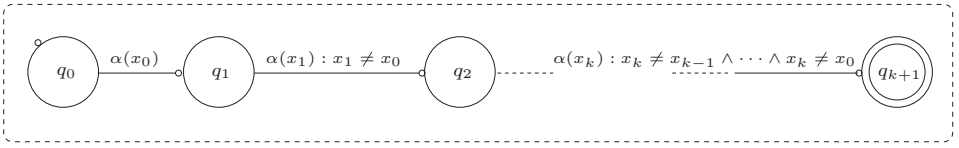


Fig. 2. The usage automaton $\varphi_{\mathrm{DIFF}(k)}$.

Example 1. For all $k \geqslant 0$, let DIFF(k) be the usage policy stating that the action α (of arity 1) can be fired at most on k different resources, while α can be fired many times on the same resource.

$$\mathrm{DIFF}(k) = \mathsf{Ev}^* \setminus \{\, \eta_0\, \alpha(r_0)\, \eta_1 \cdots \eta_k\, \alpha(r_k)\, \eta_{k+1} \mid \forall i \neq j \in 0 \ldots k : r_i \neq r_j \,\}.$$

The usage automaton $\varphi_{\mathrm{DIFF}(k)}$ in Figure 2 specifies the policy DIFF(k).

To show that $\{\, \eta \mid \eta \models \varphi \,\} \subseteq \mathrm{DIFF}(k)$, we proceed contrapositively. We pick up $\eta = \eta_0\, \alpha(r_0)\, \eta_1 \cdots \eta_k\, \alpha(r_k)\, \eta_{k+1} \notin \mathrm{DIFF}(k)$, and we instantiate φ with $\sigma = \{r_0/x_0, \ldots, r_k/x_k\}$, and an offending run is found because $\forall i \neq j \in 0..k : r_i \neq r_j$. To show the converse, assume $\eta \not\models \varphi$. Then, η must be of the form $\eta_0\, \alpha(r_0)\, \eta_1 \cdots \eta_k\, \alpha(r_k)\, \eta_{k+1}$ with all the resources pairwise distinct. Then, a run on η leading to the state q_{k+1} is obtained through be the same σ as above. □

Note that usage automata can be non-deterministic, in the sense that for all states q and input symbols $\alpha(\vec{\xi})$, the set of states:

$$\left\{ q' \;\middle|\; q \xrightarrow{\alpha(\vec{\xi}) : g} q' \right\}$$

is not required to be a singleton. Indeed, Definition 3 requires that *all* the paths of the instances $A_\varphi(\sigma, \mathcal{R})$ driven by a given trace η comply with φ, i.e. they all lead to a non-final state. This is a form of diabolic non-determinism, because one run leading to a final state is enough to classify a trace as offending.

Since, we used final states to represent violations to the policy, usage automata actually adhere to the 'default-accept' paradigm. An alternative approach would be the 'default-deny' one, that allows for specifying the permitted usage patterns, instead of the denied ones. Both approaches have known advantages and drawbacks. On the one hand, the default-deny approach is considered safer, because unpredicted traces are always forbidden. On the other hand, the default-accept approach seems to us more suitable for open systems, where the possible traces are not always known in advance, and the designer only needs to specify the forbidden behaviour. Indeed, we argue that our form of diabolic non-determinism is particularly convenient when specifying policies, since one can focus on the sequences of events that lead to violations, while neglecting those that do not affect the compliance to the policy. All the needed self loops are added by *complete*$(\mathcal{X}, \mathcal{R})$ in Definition 3. This is indeed a crucial task, since it is *not* possible in general to make all the self loops explicit in the edges E (see e.g. Example 8).

Note that an automaton $A_\varphi(\sigma, \mathcal{R})$ always has a finite number of states, but it may have *infinitely* many transitions. In particular, $A_\varphi(\sigma, \mathcal{R})$ always has a finite number of non-self loop transitions, while the number of self loops is infinite whenever the set $\mathcal{R}$ is such.

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

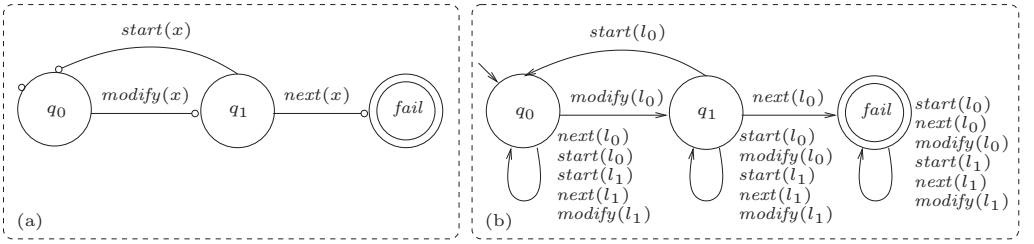


Fig. 3. (a) The usage automaton φ_{List}. (b) The FSA $A_{\varphi_{\text{List}}}(\{x \mapsto l_0\}, \{l_0, l_1\})$.

Indeed, while instantiating φ, the relation

$$\left\{ q \xrightarrow{\alpha(\vec{\xi}\sigma)} q' \;\middle|\; q \xrightarrow{\alpha(\vec{\xi}):g} q' \in E \text{ and } \sigma \models g \right\}$$

only contains finitely many transitions, because such is the set of variables occurring in φ, and because the static resources are not affected by the substitution σ. The completion operation adds self loops for all the events not explicitly mentioned in the policy, by instantiating the variables in all possible ways: these self loops are infinitely many if and only if $\mathcal{R}$ is infinite.

To illustrate our model, we present below some examples.

Example 2 (safe iterators). Consider the implementation of a list data type which allows for iterating over the elements of the list. In this scenario, a relevant usage policy is one that prevents list modifications when the list is being iterated. The relevant events are $start(l)$, for starting the iterator of the list l, $next(l)$, for retrieving the next element of l, and $modify(l)$, that models adding/removing elements from l. For simplicity, we assume that the iterator is always correctly initialized, i.e. we do not require that a $next(l)$ is preceded in the trace by an event $start(l)$. The usage policy is then defined as follows:

$$\text{List} = \mathsf{Ev}^* \setminus \{\, \eta_0\, modify(l)\, \eta_1\, next(l)\, \eta_2 \mid start(l) \notin \eta_1 \,\}.$$

The usage automaton φ_{List} recognizing this policy is depicted in Figure 3a. The automaton φ_{List} has three states: q_0, q_1 and $fail$. In the state q_0, both modifications of the list and iterations are possible. However, a modification (i.e. an add or a remove) done in q_0 will lead to q_1, where it is no longer possible to iterate on l, until a $start(l)$ resets the situation. Consider now: $\eta = start(l_0)\, next(l_0)\, start(l_1)\, next(l_1)\, modify(l_1)\, modify(l_0)\, next(l_0)$. The trace η violates the policy List, because the last event attempts to iterate on the list l_0, after l_0 has been modified. To check that φ_{List} correctly models this behaviour, consider the instantiation $A_0 = A_{\varphi_{\text{List}}}(\{x \mapsto l_0\}, \{l_0, l_1\})$ depicted in Figure 3b. When supplied with the input η, the FSA A_0 reaches the offending state $fail$, therefore by Definition 3 it follows that $\eta \not\models \varphi_{\text{List}}$. □

Example 3. Consider the policy FRESH requiring that the action α (of arity 1) cannot be fired twice on the same resource:

$$\text{FRESH} = \mathsf{Ev}^* \setminus \{\, \eta_0\, \alpha(r)\, \eta_1\, \alpha(r)\, \eta_2 \mid r \in \mathsf{Res} \setminus \{_\},\ \eta_0, \eta_1, \eta_2 \in \mathsf{Ev}^* \,\}.$$

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

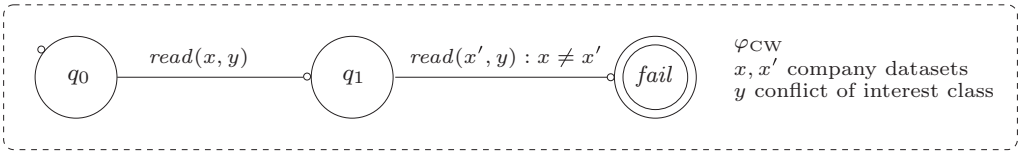


Fig. 4. The usage automaton φ_{CW} (Chinese wall).

The usage policy FRESH is modelled by the usage automaton φ_{FRESH}, with states $\{q_0, q_1, q_2\}$, initial state q_0, offending state q_2 and edges:

$$E = \left\{ q_0 \overset{\alpha(x)}{\longrightarrow\!\!\circ}\ q_1,\ q_1 \overset{\alpha(x)}{\longrightarrow\!\!\circ}\ q_2 \right\}.$$

For instance, the trace $\alpha(r_1)\alpha(r_2)\alpha(r_3)$ respects φ_{FRESH} whenever the three resources r_1, r_2, r_3 are distinct. □

Example 4 (Chinese wall). The Chinese wall policy (Brewer and Nash 1989) is a classical security policy used in commercial and corporate business services. 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 the usage automaton φ_{CW} in Figure 4. The edge from q_0 to q_1 represents accessing the company dataset x in the conflict of interests class y. The label $read(x', y) : x \neq x'$ on the edge leading from q_1 to the offending state *fail* means that a dataset x' different from x has been accessed in the same conflict of interests class y. □

As we will show later on in this paper, *polyadic* usage automata, i.e. having more than one variable, are more expressive than monadic ones. For instance, the Chinese wall policy in Figure 4, cannot be expressed by any usage automata with a single variable. As a matter of fact, we will prove in Section 4 that the expressivity of usage automata increases with the number of its variables.

Example 5 (applet confinement). Consider 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 φ_{AC} in Figure 5. The edge from q_0 to q_1 represents the applet a creating a file f. Accessing f is denied to any applet a' other than a, and this is expressed by the condition $a \neq a'$ guarding both the *read* and the

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

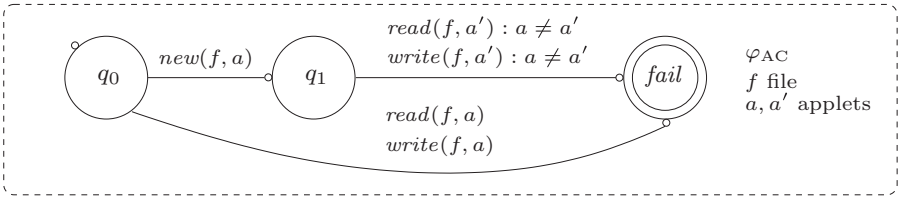


Fig. 5. The usage automaton φ_{AC} (applet confinement).

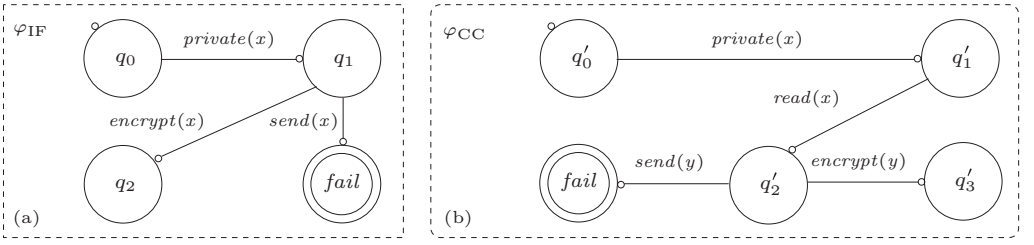


Fig. 6. (a) The usage automaton φ_{IF} (information flow). (b) The usage automaton φ_{CC} (covert channels).

write actions. When the guard is true, the automaton goes from state q_1 to the offending state *fail*. The edge from q_0 to *fail* prohibits accessing local files. □

Example 6 (editor service). Consider a cloud service that allows for editing documents, storing them on a remote site, and sharing them with other users. The front-end of the editor is run by the web 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 (monadic) policy is modelled by the automaton φ_{IF} in Figure 6a. 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 *fail*), then the policy would be violated. Instead, if x is encrypted (edge from q_1 to q_2), then x can be freely transmitted: indeed, the absence of paths from q_2 to an offending state indicates that once in q_2 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 the (diadic) automaton φ_{CC} in Figure 6b. A violation occurs if after some private file x is read (path from q'_0 to q'_2), then some file y is sent (possibly, with $y = x$, edge from q'_2 to the offending state *fail*). □

The following example shows a usage automaton where the final state is not a sink, i.e. it can be left through a suitable event. Indeed, the compliance relation $\eta \models \varphi$ is *not* prefix-closed, so we can define policies which, like in this example, permit to recover from bad states.

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

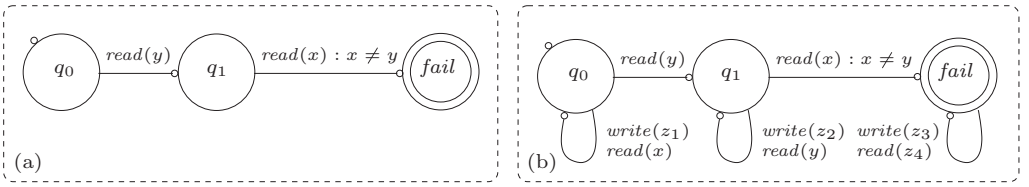


Fig. 7. (a) A usage automaton. (b) A wrong attempt to expose its self-loops.

Example 7. Consider the usage automaton φ_{Loan}, that prevents anyone from taking out a loan if their account is in the red. It is specified as follows:

$$\left\langle \{red, black\}, \{q_0, q_1\}, q_0, \{q_1\}, \left\{ q_0 \overset{red}{\multimap} q_1, q_1 \overset{black}{\multimap} q_0 \right\} \right\rangle.$$

We have that $red\ black \models \varphi_{\text{Loan}}$, while $red \not\models \varphi_{\text{Loan}}$. This technical example shows a policy (φ_{Loan}) which is not prefix-closed. □

Example 8. Consider the policy 'an object x cannot be read if some other object $y \neq x$ has been read in the past'. This is modelled by the usage automaton in Figure 7a, which classifies, e.g. $\eta_0 = read(r_0)write(r_1)write(r_2)read(r_1)$ as offending, while it considers $\eta_1 = read(r_0)write(r_0)read(r_0)$ as permitted. In order to correctly classify the above traces, we rely on $complete(\mathcal{X}, \mathcal{R})$ in Definition 3, which adds the needed self loops, among which those labelled $write(r)$ for all states and resources r.

Now, let us attempt to make all the self loops explicit, without resorting to $complete(\mathcal{X}, \mathcal{R})$. A plausible way to do that is by using the usage automaton in Figure 7b. However, this is a failing attempt, because it incorrectly classifies the trace η_0 as permitted. There is no way to instantiate z_2 so that the edge $write(z_2)$ matches both $write(r_1)$ and $write(r_2)$, therefore the automaton will get stuck on η_0 at some point. As a matter of fact, any attempt to make all the self loops explicit will fail, because it would require an unbounded number of variables in the edges whenever $\mathcal{R}$ is an infinite set. □

As a further reality check, in (Bartoletti *et al.* 2009a) we have specified and enforced through usage automata the policies of a typical bulletin board system, taking inspiration from phpBB, a popular internet forum.

3. Decidability of enforcement

According to Definition 3, to decide if a trace respects a policy we should instantiate a possibly *infinite* number of automata, one for each substitution $\sigma : var(\varphi) \to \mathsf{Res} \setminus \{_\}$. These automata have a finite number of states, but they may have *infinitely* many transitions.

However, the compliance of a trace w.r.t. a usage automaton is a decidable property: Theorem 5 below shows that checking $\eta \models \varphi$ can be done using a *finite* set of finite state automata.

To prove that, we first establish a technical result about usage automata.

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

Notation. Recall that $res(\eta)$ is the set of resources which occur in η, and that $\eta \lhd A_\varphi(\sigma, \mathcal{R})$ if no runs of $A_\varphi(\sigma, \mathcal{R})$ on η lead to a final state (Definition 3). Hereafter, for all traces η and for all usage automata φ, we will denote by $R(\eta, \varphi)$ the set of resources $\big(res(\eta) \cup res(\varphi) \cup \{\#_i\}_{i \in 1,\ldots,|\varphi|}\big) \setminus \{_\}$, and we assume that the witness resources $\#_1, \ldots, \#_{|\varphi|}$ do not occur in η.

Lemma 4. For all usage automata φ, for all traces η, and for all substitutions $\sigma : var(\varphi) \to \mathsf{Res} \setminus \{_\}$:

$$\exists \mathcal{R} \supseteq res(\eta) : \eta \lhd A_\varphi(\sigma, \mathcal{R}) \implies \forall \mathcal{R}' \supseteq res(\eta) : \eta \lhd A_\varphi(\sigma, \mathcal{R}').$$

The underlying intuition is more evident if the statement of Lemma 4 is read contrapositively. If η is recognized as offending by *some* instantiation $A_\varphi(\sigma, \mathcal{R}')$ with $\mathcal{R}' \supseteq res(\eta)$, then η will be offending for *all* the instantiations $A_\varphi(\sigma, \mathcal{R})$, with $\mathcal{R} \supseteq res(\eta)$. In other words, the only relevant fact about the set $\mathcal{R}$ used in instantiations is that $\mathcal{R} \supseteq res(\eta)$, which is always a *finite* set.

The following theorem establishes the decidability of checking whether a trace η respects a policy φ. It states that, rather than checking $\eta \lhd A_\varphi(\sigma, \mathsf{Res})$ for all $\sigma : var(\varphi) \to \mathsf{Res} \setminus \{_\}$ as prescribed by Definition 3, we can restrict ourselves to checking η with respect to a *finite* set of FSAs $A_\varphi(\sigma, \mathcal{R})$. In particular, we can choose $\mathcal{R} = res(\eta)$, which makes the $A_\varphi(\sigma, \mathcal{R})$ a FSA, and we can only consider the substitutions σ with codomain $R(\eta, \varphi)$. Policy compliance is then decidable, because there is only a finite number of such substitutions.

Theorem 5. For all traces η and for all usage automata φ, $\eta \models \varphi$ iff:

$$\forall \sigma : var(\varphi) \to R(\eta, \varphi) \ : \ \eta \lhd A_\varphi(\sigma, res(\eta)).$$

Example 9. This technical example is intended to justify the extra resources $\#_i$ used in Theorem 5. Let φ be the usage automaton in Figure 8a. Consider the trace $\eta = \alpha(r)\alpha(r)$. By Definition 3, it must be $\eta \not\models \varphi$, because η is in the language of the FSA $A_\varphi(\{x \mapsto \#_1\}, \{r\})$, displayed in Figure 8b. Note that Theorem 5 correctly prescribes that the substitution $\sigma = \{x \mapsto \#_1\}$ has be taken into account.

Were Theorem 5 not allowing the range of σ to include any resource except for those in $res(\eta) = \{r\}$, then there would exist a single choice for σ, i.e. $\sigma = \{x \mapsto r\}$. Now, η is not in the language of the FSA $A_\varphi(\{x \mapsto r\}, \{r\})$, displayed in Figure 8c, and so $\eta \lhd A_\varphi(\{x \mapsto r\}, \{r\})$. Since there are not further substitutions to consider, we would incorrectly infer that $\eta \models \varphi$. □

4. Expressiveness results

We start by considering three restricted forms of usage automata, some of which have been put forward in (Bartoletti *et al.* 2005). First, we consider usage automata without static resources, and then those without guards. In both cases, we show these restricted forms are less expressive than unrestricted ones. Finally, we restrict the events $\alpha(\vec{\xi})$ on the

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

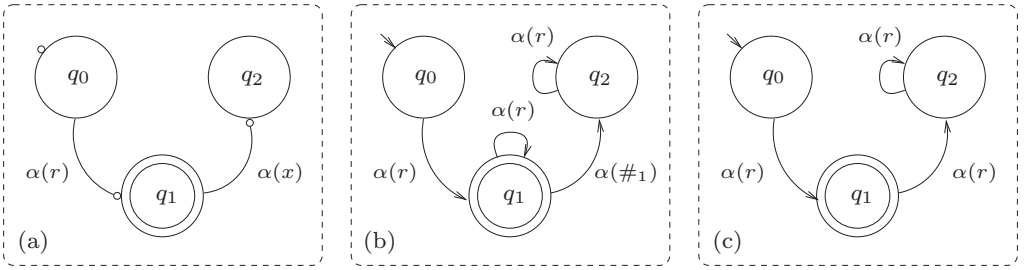


Fig. 8. (a) The usage automaton φ. (b) The FSA $A_\varphi(\{x \mapsto \#_1\}, \{r\})$. (c) The FSA $A_\varphi(\{x \mapsto r\}, \{r\})$.

edges of usage automata to act on a single resource, i.e. $|\vec{\xi}| = 1$. Unlike in the previous case, this restriction does not affect the expressive power of usage automata.

Proposition 6. Removing static resources decreases the expressive power of usage automata.

Proof. There exists no usage automaton *without static resources* that recognizes the policy specified as follows:

$$\left\langle \{\, \alpha(r_i) \,|r_i \in \mathsf{Res}\, \}, \{q_0, fail, q_1\}, q_0, \{fail\}, \left\{ q_0 \xrightarrow{\alpha(x)} fail, fail \xrightarrow{\alpha(r_0)} q_1 \right\} \right\rangle.$$

Indeed, the *fail* state cannot be left if static resources cannot be mentioned in the outcoming edge. □

Proposition 7. Removing guards decreases the expressive power of usage automata.

Proof. There exists no usage automaton *without guards* that recognizes the policy DIFF(1) from Example 1. See Appendix A for more details. □

Proposition 8. Decreasing the arity of events does not decrease the expressive power of usage automata.

We now introduce the notion of *collapsing*. Given a set $\mathcal{R}$ of dynamic resources, a collapsing κ of $\mathcal{R}$ maps the resources in $\mathcal{R}$ into a set of witnesses, while coalescing all the other dynamic resources into the dummy resource $_$. Also, κ acts as the identity on the static resources.

The notion of collapsing will be crucial for our model checking technique. Since a usage automaton φ of arity k can only distinguish among k dynamic resources, to check if a trace η complies with φ we can safely abstract the dynamic resources in η through an injective collapsing to a set of k witnesses (plus the dummy resource $_$). Indeed, an injective collapsing maps distinct resources in $\mathcal{R}$ to distinct witnesses, while all the other dynamic resources become undistinguishable. Reducing the number of resources to a finite set (the static ones, the k witnesses and the dummy) will be one of the key ideas for proving the decidability of model checking.

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

Definition 9 (collapsing). For all $\mathcal{R} \subseteq \mathsf{Res}_d$ and for all $\mathcal{W} \subseteq \{\#_i\}_{i\in\mathbb{N}}$, we say that $\kappa : \mathsf{Res} \to \mathsf{Res}$ is a *collapsing of* $\mathcal{R}$ *onto* $\mathcal{W}$ whenever:

$$\kappa(\mathcal{R}) = \mathcal{W}$$
$$\kappa(\mathsf{Res}_d \setminus \mathcal{R}) = \{_\}$$
$$\kappa \text{ is the identity on } \mathsf{Res} \setminus \mathsf{Res}_d.$$

Moreover, when κ is injective on $\mathcal{R}$, we say it is an *injective collapsing*.

Sometimes, we will not be interested in the set $\mathcal{R}$, in the set $\mathcal{W}$, or in both. In such cases, we will call κ, respectively, a collapsing onto $\mathcal{W}$, a collapsing of $\mathcal{R}$, or simply a collapsing.

Hereafter, we shall write $\eta\kappa$ to denote the homomorphic extension of the function κ to the trace η.

The following lemma shows that the actual identity of the witnesses used by a collapsing is immaterial. One can freely rename them, without affecting policy compliance.

Lemma 10. For traces η, for all collapsings κ, and for all permutations π of witnesses, $\eta\,\kappa \models \varphi$ if and only if $\eta\,\kappa\,\pi \models \varphi$.

The following lemma states that if η violates φ, then there exists an injective collapsing κ such that also $\eta\kappa$ violates φ. Furthermore, κ only needs to collapse a set of $|\varphi|$ dynamic resources to obtain the violation.

Lemma 11. Let η be a trace, let φ be a usage automaton, and let K be the set of injective collapsing such that $|\kappa(\mathsf{Res}_d)| = |\varphi| + 1$. Then:

$$(\forall \kappa \in K : \eta\kappa \models \varphi) \implies \eta \models \varphi.$$

The following lemma states that if there exists an injective collapsing κ such that $\eta\kappa$ violates φ, then it is also the case that η violates φ. The only mandatory condition required about κ (in addition to injectivity) is that it collapses at least $|\varphi|$ dynamic resources to obtain the violation.

Lemma 12. Let η be a trace, let φ be a usage automaton, and let $\bar{K}$ be the set of injective collapsings such that $|\kappa(\mathsf{Res}_d)| \geqslant |\varphi| + 1$ for all $\kappa \in \bar{K}$. Then:

$$\eta \models \varphi \implies (\forall \kappa \in \bar{K} : \eta\kappa \models \varphi).$$

The following theorem puts together the statements of Lemmata 11 and 12. As we shall see in Section 6, it provides us with the basis for constructing our model checking technique for usage policies.

Theorem 13. Let η be a trace, let φ be a usage automaton, and let K be the set of injective collapsings such that $|\kappa(\mathsf{Res}_d)| = |\varphi| + 1$. Then:

$$\eta \models \varphi \iff (\forall \kappa \in K : \eta\,\kappa \models \varphi).$$

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

Theorem 13 can be also exploited as a proof technique to show that some policies are not expressible by any usage automata. An example follows.

Example 10. For all traces η and all actions α, let $\eta\downarrow_\alpha$ be the subtrace of η containing only events of the form $\alpha(r)$, for some r. Let:

$$\mathrm{GEQ} \;=\; \mathsf{Ev}^* \setminus \{\, \eta\,\beta\,\eta' \;|\; |\eta\downarrow_\beta| \geqslant |res(\eta\downarrow_\alpha)| \,\}\,.$$

Intuitively, you can fire as many β as the number of different resources on which you have previously fired α. For instance, we have that:

$$\alpha(r)\beta \in \mathrm{GEQ} \qquad \alpha(r)\beta\beta \notin \mathrm{GEQ}$$
$$\alpha(r)\alpha(r')\beta\beta \in \mathrm{GEQ} \qquad \alpha(r)\alpha(r')\alpha(r)\beta\beta\beta \notin \mathrm{GEQ}.$$

We now prove that GEQ cannot be recognized by any usage automata. By contradiction, assume that there exists φ such that φ recognizes GEQ, and let $n = |\varphi|$. Let $\eta = \alpha(r_1)\cdots\alpha(r_{n+1})\beta^n$, with $r_i \in \mathsf{Res}_d$ for all $i \in 1,\ldots,n+1$, and $r_i \neq r_j$ for all $i \neq j$. Let $\eta_0 = \eta\beta$. Clearly, $\eta_0 \in \mathrm{GEQ}$, so by hypothesis we have that $\eta_0 \models \varphi$. By Theorem 13, it must be that $\eta_0\kappa \models \varphi$ for all injective collapsings κ onto n witnesses $\#_1,\ldots,\#_n$. But this is a contradiction, because $|res((\eta\kappa)\downarrow_\alpha)| = n = |(\eta\kappa)\downarrow_\beta|$, and so $\eta_0\kappa \notin \mathrm{GEQ}$. □

We will now show that each increment of the arity of usage automata widens the set of usage policies they can recognize. To prove that, we shall exploit Theorem 13 for showing that the policy DIFF(n) of Example 1 cannot be expressed by any usage automata with arity $n' < n$.

Proposition 14. The expressive power of usage automata increases with arity.

Proof. Consider the policy DIFF(n) of Example 1. By contradiction, assume that there exists φ with arity $n' < n$ such that $\eta \models \varphi$ if and only if $\eta \in \mathrm{DIFF}(n)$, for all η. W.l.o.g. we can assume that $res(\varphi) = \varnothing$. Let

$$\eta = \alpha(r_0)\alpha(r_1)\cdots\alpha(r_n) \qquad \text{with } r_i \neq r_j \in \mathsf{Res}_d \text{ for all } i \neq j \in 0\ldots n.$$

For all injective collapsings κ onto n' witnesses, $|res(\eta\kappa)| = n' < n$, and so $\eta\kappa \models \varphi$. By Theorem 13, this implies that $\eta \models \varphi$. But this is a contradiction, because $|res(\eta)| = n+1$, and so $\eta \notin \mathrm{DIFF}(n)$. Therefore, DIFF($n$) cannot be expressed by any usage automaton with arity less than n. □

5. A calculus of usages

We now present a basic calculus of usages. We assume a countable set Nam of *names*, ranged over by $n, n', \ldots$, such that $\mathsf{Nam} \cap (\mathsf{Res} \cup \mathsf{Var}) = \varnothing$. Usages $U, U', \ldots$ include ε, representing the empty trace, events $\alpha(\vec{\rho})$ (where $\vec{\rho}$ may include both resources and names), resource creation $\nu n.U$, sequencing $U \cdot U'$, non-deterministic choice $U + U'$, and recursion $\mu h.U$. In $\nu n.\,U$, the free occurrences of the name n in U are bound by ν; similarly, μh binds the free occurrences of the variable h in U. The actual definition follows.

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

Definition 15 (usages). Let $\mathsf{Ev}_{\mathsf{Nam}} = \{\, \alpha(\vec{\rho}) \in \mathsf{Act} \times (\mathsf{Res} \cup \mathsf{Nam})^* \mid |\alpha| = |\vec{\rho}| \,\}$ be the set of events having targets ranging over names and resources. Usages are inductively defined as follows:

U, V	$::= \varepsilon$	empty
	h	variable
	$\alpha(\vec{\rho})$	event, where $\alpha(\vec{\rho}) \in \mathsf{Ev}_{\mathsf{Nam}}$ and $\alpha \neq \mathit{new}$
	$U \cdot V$	sequence
	$U + V$	choice
	$\mu h.\, U$	recursion
	$\nu n.\, U$	resource creation

We stipulate that sequencing $\cdot$ binds more strongly than choice $+$, that in turn has precedence over resource creation νn and recursion μh. Free and bound names and variables are as expected.

We say that a usage is:

— *closed*, when it has no free names and no free variables,
— *initial*, when closed and without dynamic, witness, and dummy resources.

Usages are a polyadic extension of *history expressions* (Bartoletti *et al.* 2005), that in turn extend those in (Skalka and Smith 2004). History expressions have been used, e.g. in (Bartoletti *et al.* 2006; Bartoletti *et al.* 2009b; Bartoletti *et al.* 2009c), to statically approximate the behaviour of terms in λ-calculi extended with primitives for history-based access control. The static analyses developed there take the form of type and effect systems, which associates each typable λ-term with a history expression which safely over-approximates the set of traces obtainable at run time.

Note that usages share much of their constructs with (BPAs) (Bergstra and Klop 1985). In particular, both formalisms feature actions, sequential composition, non-deterministic choice, and recursion. The main differences between usages and BPAs are the following:

— atomic actions in usages may have parameters, which indicate the resources upon which the action is performed. Atomic actions are essentially the events of Section 2, possibly containing names among their targets.
— unlike BPAs, usages feature resource creation $\nu n.\, U$, borrowing from name restriction *à la* π-calculus.
— recursive usages are constructed through the binder μ, while recursive BPAs are obtained through an environment of definitions (see Definition 18).

The semantics of usages is specified through a labelled transition system. The configurations of the transition system are pairs $U, \mathcal{R}$, where U is a usage, and $\mathcal{R} \subseteq \mathsf{Res}_d$ is the (finite) set of dynamic resources generated so far.

Definition 16 (semantics of usages). For all closed usages U, for all $\mathcal{R} \subseteq \mathsf{Res}_d$, we denote with $[\![U]\!]_{\mathcal{R}}$ the set of traces $\eta = a_1 \cdots a_i$ $(i \geqslant 0)$ such that:

$$\exists U', \mathcal{R}' \;\; : \;\; U, \mathcal{R} \xrightarrow{a_1} \cdots \xrightarrow{a_i} U', \mathcal{R}'.$$

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

Table 3. *Labelled transitions semantics of usages.*

$$\varepsilon \cdot U, \mathcal{R} \xrightarrow{\varepsilon} U, \mathcal{R} \qquad \alpha(\vec{r}), \mathcal{R} \xrightarrow{\alpha(\vec{r})} \varepsilon, \mathcal{R} \qquad \mu h.\, U, \mathcal{R} \xrightarrow{\varepsilon} U\{\mu h.\, U/h\}, \mathcal{R}$$

$$\frac{U, \mathcal{R} \xrightarrow{a} U', \mathcal{R}'}{U \cdot V, \mathcal{R} \xrightarrow{a} U' \cdot V, \mathcal{R}'} \qquad \frac{U_0, \mathcal{R} \xrightarrow{a} U_0', \mathcal{R}'}{U_0 + U_1, \mathcal{R} \xrightarrow{a} U_0', \mathcal{R}'} \qquad \frac{U_1, \mathcal{R} \xrightarrow{a} U_1', \mathcal{R}'}{U_0 + U_1, \mathcal{R} \xrightarrow{a} U_1', \mathcal{R}'}$$

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

The transition relation $\xrightarrow{a}$, where $a \in \mathsf{Ev} \cup \{\varepsilon\}$, is the least relation satisfying the axioms and the inference rule in Table 3.

The rules of the operational semantics are pretty standard, except for the one dealing with resource creation. The set $\mathcal{R}$ in configurations accumulates the resources created at run time so far. Firing a νn amounts to pick up a new dynamic resource not in $\mathcal{R}$, and in binding it to the name n. The side condition ensures that no resource can be 'created' twice. The label $new(r)$ records in the trace the creation of the resource r.

Example 11. Let $U = \mu h.\, \nu n.\, (\varepsilon + \alpha(n) \cdot h)$. For all $\mathcal{R}$, the traces in $[\![U]\!]_{\mathcal{R}}$ are the strings of the form $\eta = new(r_1)\alpha(r_1) \cdots new(r_k)$ or $\eta\, \alpha(r_k)$ for all $k \geqslant 0$ and pairwise distinct resources r_i such that $r_i \notin \mathcal{R}$. □

We introduce below some requirements on traces, that characterize when they faithfully reflect well-formed computations.

Definition 17 (well-formed traces). A trace is *well-formed* when it is never the case that one of the following holds:

1. a static resource r is the target of a $new(r)$ event;
2. for some resource r, the $new(r)$ event is fired more than once;
3. an event $\alpha(\vec{r})$, with $r' \in \vec{r} \cap \mathsf{Res}_d$ and $\alpha \neq new$, is fired before a $new(r')$.

Hereafter, we shall only consider usages U which denote well-formed traces. Indeed, the operational semantics in Definition 16 guarantees all the items required by Definition 17.

6. Model checking usage policies

The run-time enforcement of usage policies may reduce the overall performance of systems. Static analysis may allow to dispose of the run-time monitor, thus reducing such overhead, see e.g. (Erlingsson and Schneider 1999; Wallach *et al.* 2001). The static analysis we are proposing requires a couple of steps, informally described below, and made precise in the rest of this section.

To statically verify whether a usage U respects a usage policy φ, we first transform U into a BPA term. Then, we model-check the resulting BPA against a finite set of FSAs, obtained by suitable instantiations of φ.

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

The problem of checking that the traces of a BPA P are included in the language accepted by a FSA A is known to be decidable (Esparza 1994). The decision procedure amounts to first transform P into a pushdown automaton A_P, to construct then the intersection $A_P \times \bar{A}$ of A_P and the complement $\bar{A}$ of A, and finally to check the emptiness of the language accepted by $A_P \times \bar{A}$, which is still a pushdown automaton. Since, such a language is context-free, checking its emptiness is decidable. Several algorithms and tools show this approach feasible, see e.g. (Baier and Katoen 2008).

Note that the above-mentioned decision procedure cannot be directly applied to usages. Indeed, when resource creation occurs within the body of a recursion, like e.g. in $U = \mu h.\, \nu n.\, \alpha(n) \cdot h$, the resulting traces will have an unbounded number of resources. According to Theorem 5, one should then consider every possible substitution $\sigma : var(\varphi) \to R(\eta, \varphi)$, for all $\eta \in [\![U]\!]_\varnothing$. The number of such substitutions grows unboundedly as new resources are created in η: thus, we would have to intersect an *infinite* number of FSAs, which is clearly unfeasible.

To solve this problem, in Definition 20 below we shall define a mapping from usages to BPAs that collapses the dynamic resources, while preserving policy compliance. The intuition is that any usage automaton φ can only distinguish among the resources that will be bound to the variables of φ. Thus, it is sound to consider only $|\varphi| + 1$ classes of representatives of dynamic resources: the resources $\{\#_i\}_{i \in 1 \ldots |\varphi|}$, that are witnesses for the dynamic resources bound to the variables in $var(\varphi)$, and the dummy resource $_$ that is the witness for all the other dynamic resources.

Theorems 28 and 30 will establish the soundness and completeness of our translation. We shall then exploit Theorem 13 to show that such a translation also preserves policy compliance. The overall result about our technique for model checking usage policies is stated in Theorem 35. We now proceed with the needed definitions.

The BPA processes contain the terminated process 0, events β, the operators $\cdot$ and $+$ that denote sequential composition and (non-deterministic) choice, and variables $X, Y, \ldots$. To allow for recursion, a BPA is then defined as a process p and a set of definitions Δ for the variables X that occur therein.

Definition 18 (syntax of basic process algebras). A *BPA process* is given by the following syntax, where $\beta \in \mathsf{Ev}_{\mathsf{Nam}}$:

$$p, q \ ::= \ 0 \ \mid \ \beta \ \mid \ p \cdot q \ \mid \ p + q \ \mid \ X.$$

A *BPA definition* has the form $X \triangleq p$. A set Δ of BPA definitions is *coherent* when $(X \triangleq p) \in \Delta$ and $(X \triangleq p') \in \Delta$ imply $p = p'$.

A *BPA* is a pair $\langle p, \Delta \rangle$ such that (i) Δ is finite and coherent, and (ii) for all X occurring in p or Δ, there exists some q such that $X \triangleq q \in \Delta$.

We write $\Delta + \Delta'$ for $\Delta \cup \Delta'$, when coherent (otherwise, $\Delta + \Delta'$ is undefined).

Let $P = \langle p, \Delta \rangle$ and $P' = \langle p', \Delta' \rangle$. We write $P \cdot P'$ for $\langle p \cdot p', \Delta + \Delta' \rangle$, and $P + P'$ for $\langle p + p', \Delta + \Delta' \rangle$. We assume that $+$ is commutative and associative, and so we write $\sum_{i \in I} P_i$ for the choice among a finite set of BPAs (0 if $I = \varnothing$).

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

Table 4. *Labelled transition semantics of BPAs.*

$$0 \cdot p \xrightarrow{\varepsilon} p \qquad \beta \xrightarrow{\beta} 0 \qquad \frac{p \xrightarrow{a} p'}{p \cdot q \xrightarrow{a} p' \cdot q}$$

$$\frac{p_0 \xrightarrow{a} p_0'}{p_0 + p_1 \xrightarrow{a} p_0'} \qquad \frac{p_1 \xrightarrow{a} p_1'}{p_0 + p_1 \xrightarrow{a} p_1'} \qquad \frac{X \triangleq p \in \Delta}{X \xrightarrow{\varepsilon} p}$$

Table 5. *Mapping usages to BPAs.*

$$\mathsf{B}_{\mathcal{W}}(\varepsilon)_\Theta = \langle 0, \varnothing \rangle \qquad \mathsf{B}_{\mathcal{W}}(h)_\Theta = \langle \Theta(h), \varnothing \rangle \qquad \mathsf{B}_{\mathcal{W}}(\alpha(\vec{\rho}))_\Theta = \langle \alpha(\vec{\rho}), \varnothing \rangle$$

$$\mathsf{B}_{\mathcal{W}}(U \cdot V)_\Theta = \mathsf{B}_{\mathcal{W}}(U)_\Theta \cdot \mathsf{B}_{\mathcal{W}}(V)_\Theta \qquad \mathsf{B}_{\mathcal{W}}(U_0 + U_1)_\Theta = \mathsf{B}_{\mathcal{W}}(U_0)_\Theta + \mathsf{B}_{\mathcal{W}}(U_1)_\Theta$$

$$\mathsf{B}_{\mathcal{W}}(\nu n.U)_\Theta = new(_) \cdot \mathsf{B}_{\mathcal{W}}(U\{_/n\})_\Theta + \sum_{\#_i \in \mathcal{W}} new(\#_i) \cdot \mathsf{B}_{\mathcal{W}\setminus\{\#_i\}}(U\{\#_i/n\})_\Theta$$

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

BPA processes up to α-conversion of their variables are identified.

The semantics of BPAs is given by the labelled transition system in Definition 19. Note that there is no need to consider infinite computations, because if a trace η violates a usage policy φ, then there exists a finite prefix of η which violates φ, too.

Definition 19 (semantics of basic process algebras). The semantics $[\![P]\!]$ of a BPA $P = \langle p_0, \Delta \rangle$ is the set of the traces labelling finite computations:

$$\left\{ \eta = a_1 \cdots a_i \,\middle|\, p_0 \xrightarrow{a_1} \cdots \xrightarrow{a_i} p_i \text{ and } i \geqslant 0 \right\}$$

where the relation $\xrightarrow{a}$, for $a \in \mathsf{Ev}_{\mathsf{Nam}} \cup \{\varepsilon\}$, is inductively defined in Table 4.

We now specify a transformation from usages into BPAs.

Definition 20. Let U be a usage, let $\mathcal{W} \subset \{\#_i\}_{i \in \mathbb{N}}$ be a finite set of witnesses, and let Θ be a mapping from usage variables to BPA processes such that $dom(\Theta)$ contains the free variables of U. We inductively define the BPA $\mathsf{B}_{\mathcal{W}}(U)_\Theta$ in Table 5. Hereafter, we shall omit Θ when empty.

Events, variables, concatenation and choice are mapped 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 resource creation $\nu n.\, U$, for which we generate a choice between two BPA processes. In the first branch of the choice, the name n is replaced by the dummy resource $_$. In the second branch, a choice itself, n is replaced in each sub-branch by the witness resource $\#_i$, for all witnesses in $\mathcal{W}$. The

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

item $\#_i$ is then removed from $\mathcal{W}$, meaning that we have already witnessed it. When there are no witnesses left, only $_$ is used.

Our next step is proving the correspondence between usages and BPAs. We shall prove that the traces of a usage are all and only those strings that label the computations of the associated BPA, under a suitable collapsing of resources. Some auxiliary definitions and results are needed.

The following two definitions are quite standard.

Definition 21 (simulation). Let $\mathcal{S}$ be a binary relation over BPAs. We say that $\mathcal{S}$ is a *simulation* if and only if, whenever $P\,\mathcal{S}\,Q$:

— if $Q \xrightarrow{a} Q'$, then there exists P' such that $P \xrightarrow{a} P'$ and $P'\,\mathcal{S}\,Q'$
— if $Q \not\rightarrow$, then $P \not\rightarrow$.

We say that P *simulates* Q ($P \succ Q$ in symbols) when there exists a simulation $\mathcal{S}$ such that $P\,\mathcal{S}\,Q$.

Definition 22 (bisimulation). Let $\mathcal{S}$ be a binary relation over BPAs. We say that $\mathcal{S}$ is a *bisimulation* if and only if, whenever $P\,\mathcal{S}\,Q$:

— if $P \xrightarrow{a} P'$, then there exists Q' such that $Q \xrightarrow{a} Q'$ and $P'\,\mathcal{S}\,Q'$, and
— if $Q \xrightarrow{a} Q'$, then there exists P' such that $P \xrightarrow{a} P'$ and $P'\,\mathcal{S}\,Q'$.

We say that P *is bisimilar to* Q ($P \sim Q$ in symbols) when there exists a bisimulation $\mathcal{S}$ such that $P\,\mathcal{S}\,Q$.

Definition 23 (coherent renaming). Let ζ be a substitution from BPA variables to BPA variables, and let $P = \langle p, \Delta \rangle$ be a BPA. We say that ζ is a *coherent renaming of* P when $\Delta\zeta$ is coherent.

Lemma 24. Let ζ be a coherent renaming of a BPA P. Then, $P \sim P\zeta$.

Proof. It is straightforward to check that the relation $\{\, \langle P, P\zeta \rangle \mid P \text{ is a BPA} \,\}$ is a bisimulation. □

We now state a lemma which, roughly, allows substitution of usages for recursion variables to commute with the mapping $\mathsf{B}(U)_\Theta$. More precisely, given a substitution $U\{U'/h'\}$, it allows to move the BPA associated to U' to the environment Θ. This will be crucial in the proofs later on, when dealing with the case $U' = \mu h.U$.

Lemma 25. For all $\mathcal{W}, U, U', h'$ and Θ, there exists a coherent renaming ζ of $\mathsf{B}_\mathcal{W}(U\{U'/h'\})_\Theta$ such that ζ is the identity on the BPA variables in Θ, and

$$\mathsf{B}_\mathcal{W}(U\{U'/h'\})_\Theta\,\zeta \;=\; \mathsf{B}_\mathcal{W}(U)_{\Theta\{\mathsf{B}_\mathcal{W}(U')_\Theta/h'\}}$$

where $\mathsf{B}_\mathcal{W}(U)_{\Theta\{\langle p',\Delta'\rangle/h'\}}$ stands for $\langle p, \Delta + \Delta' \rangle$ if $\mathsf{B}_\mathcal{W}(U)_{\Theta\{p'/h'\}} = \langle p, \Delta \rangle$.

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

Example 12. To illustrate the need for the coherent renaming ζ in Lemma 25, let $U = h' \cdot h'$ and $U' = \mu h.\, \alpha \cdot h$. We have that:

$$\mathsf{B}_{\mathcal{W}}(U\{U'/h'\})_{\Theta} = \langle X \cdot Y, \{X \triangleq \alpha \cdot X, Y \triangleq \alpha \cdot Y\}\rangle$$
$$\mathsf{B}_{\mathcal{W}}(U)_{\Theta\{\mathsf{B}_{\mathcal{W}}(U')_{\Theta}/h'\}} = \langle X \cdot X, \{X \triangleq \alpha \cdot X\}\rangle.$$

To recover the equality stated in the lemma, it suffices to choose the renaming $\zeta = \{Y \mapsto X\}$, which is coherent for $\{X \triangleq \alpha \cdot X, Y \triangleq \alpha \cdot Y\}$. □

The following lemma states that if you feed $\mathsf{B}_{\mathcal{W}}(U)$ with more witnesses, you obtain a BPA that simulates the original one. This fact will be needed in the proofs later on, when dealing with the inductive case $U_0 \cdot U_1$.

Lemma 26. For all usages U, and for all $\mathcal{W}, \mathcal{W}', \Theta$:

$$\mathcal{W} \supseteq \mathcal{W}' \implies \mathsf{B}_{\mathcal{W}}(U)_{\Theta} \succ \mathsf{B}_{\mathcal{W}'}(U)_{\Theta}.$$

The following lemma is the basic building block for showing the soundness of the transformation $\mathsf{B}_{\mathcal{W}}(U)$. It states that, for all suitable injective collapsings κ, a transition $\xrightarrow{a}$ performed by a usage U can be replayed by the BPA $\mathsf{B}_{\mathcal{W}}(U)$, modulo κ. The set $\mathcal{W}$ must include all the witnesses used by κ to collapse the dynamic resources not generated yet. Hereafter, given a BPA P and a collapsing κ, the BPA $P\kappa$ will denote P under the homomorphic extension of κ to BPAs.

Lemma 27. Let U be a closed usage, let $\mathcal{R} \subset \mathsf{Res}_d$ be a finite set of resources, let $\mathcal{W}$ be a finite set of witnesses, and let κ be an injective collapsing such that $\mathcal{W} \supseteq \kappa(\mathsf{Res}_d \setminus \mathcal{R}) \setminus \{_\}$. Then, for all $a, U', \mathcal{R}'$ such that

$$U, \mathcal{R} \xrightarrow{a} U', \mathcal{R}'$$

the following holds:

$$\forall P \succ \mathsf{B}_{\mathcal{W}}(U)\kappa \;:\; \exists \mathcal{W}' \supseteq \kappa(\mathsf{Res}_d \setminus \mathcal{R}'),\ \exists P' \succ \mathsf{B}_{\mathcal{W}'}(U')\kappa \;:\; P \xrightarrow{a\kappa} P'.$$

The following theorem guarantees that a trace η of an initial usage U can be mimicked by the BPA $\mathsf{B}_{\mathcal{W}}(U)$, provided that $\mathcal{W}$ has enough witnesses.

Theorem 28. Let U be an initial usage, let κ be an injective collapsing, and let $\mathcal{W}$ be a finite set of witnesses such that $\mathcal{W} \supseteq \kappa(\mathsf{Res}_d) \setminus \{_\}$. Then, for all η:

$$\eta \in [\![U]\!]_{\varnothing} \implies \eta\kappa \in [\![\mathsf{B}_{\mathcal{W}}(U)]\!].$$

Proof. By induction on the length of η, applying at each step Lemma 27. Note that, since U is initial, at the first step we have $\mathsf{B}_{\mathcal{W}}(U)\kappa = \mathsf{B}_{\mathcal{W}}(U)$. □

Example 13. Let $U = \mu h.\, \nu n.\, \alpha(n) \cdot h$, and recall the policy DIFF(k) from Example 1. We have that $U \not\models \varphi_{\mathrm{DIFF}(1)}$, e.g. because of the trace:

$$\eta = new(r_0)\alpha(r_0)new(r_1)\alpha(r_1)new(r_2)\alpha(r_2)$$

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

Let now $\mathcal{W} = \{\#_1, \#_2\}$. We have $\mathsf{B}_{\mathcal{W}}(U) = \langle X, \{X \triangleq p\}\rangle$, where

$$p \;=\; new(_) \cdot \alpha(_) \cdot X + new(\#_1) \cdot \alpha(\#_1) \cdot X + new(\#_2) \cdot \alpha(\#_2) \cdot X.$$

Let κ be the injective collapsing such that $\kappa(r_1) = \#_1$ and $\kappa(r_2) = \#_2$. Then,

$$\eta\kappa = new(_)\alpha(_)new(\#_1)\alpha(\#_1)new(\#_2)\alpha(\#_2)$$

is a trace of $\mathsf{B}_{\mathcal{W}}(U)$, as correctly predicted by Theorem 28.

Note that $\eta\kappa \not\models \varphi_{\mathrm{DIFF}(1)}$, while any collapsing κ' onto some $\mathcal{W}'$ with $|\mathcal{W}'| = 1$ would have unsoundly lead to $\eta\kappa' \models \varphi_{\mathrm{DIFF}(1)}$. □

The following lemma is the basic building block for showing the completeness of the transformation $\mathsf{B}_{\mathcal{W}}(U)$. It states that, given any suitable injective collapsing κ, a transition $\xrightarrow{a}$ of the BPA $\mathsf{B}_{\mathcal{W}}(U)$ can be replayed by the usage U, modulo κ. If the transition taken by the BPA models the creation of a fresh resource, i.e. if $a = new(\#_i)$, then there must exist a dynamic resource r, not already used, such that $\kappa(r) = \#_i$.

Lemma 29. Let U be a closed usage, let $\mathcal{R} \subseteq \mathsf{Res}_d$ be a finite set of resources, let $\mathcal{W}$ be a finite set of witnesses, and let κ be a collapsing. Assume that

$$P \;\prec\; \mathsf{B}_{\mathcal{W}}(U)\kappa \qquad\qquad P \;\xrightarrow{a}\; P'$$

and, if $a = new(\#_i)$, then there exists some $r \in \mathsf{Res}_d \setminus \mathcal{R}$ such that $\kappa(r) = \#_i$.

Then, there exist $U', \mathcal{R}'$, and b such that

$$U, \mathcal{R} \xrightarrow{b} U', \mathcal{R}' \qquad\qquad a = b\,\kappa \qquad\qquad P' \prec \mathsf{B}_{\mathcal{W}}(U')\kappa.$$

The following theorem guarantees that, for all suitable collapsings κ, an initial usage U can perform all the computations that the BPA $\mathsf{B}_{\mathcal{W}}(U)$ can perform, modulo κ. Similarly to Lemma 29, if the trace η of the BPA has n occurrences of $new(\#_i)$, then there must exist n dynamic resources mapped by κ to $\#_i$. To formalize this condition, for all traces η and for all $i \in \mathbb{N}$ we define $W_i(\eta)$ as the number of the occurrences of the event $new(\#_i)$ in η.

Theorem 30. Let U be an initial usage, let $\mathcal{W}$ be a finite set of witnesses, let η' be a trace, and let κ be a collapsing such that $|\kappa^{-1}(\#_i)| \;=\; W_i(\eta')$, for all $\#_i \in \mathcal{W}$. Then, for all traces η':

$$\eta' \in [\![\mathsf{B}_{\mathcal{W}}(U)]\!] \quad\Longrightarrow\quad \exists \eta \;:\; \eta \in [\![U]\!]_{\varnothing} \text{ and } \eta' = \eta\,\kappa.$$

Proof. By induction on the length of η, applying at each step Lemma 29. Note that, since U is initial, at the first step we have $\mathsf{B}_{\mathcal{W}}(U)\kappa = \mathsf{B}_{\mathcal{W}}(U)$. Also, note that the assumption on κ, i.e. for each $new(\#_i)$ there exist an r such that $\kappa(r) = \#_i$, allows to maintain the invariant required by Lemma 29. □

Example 14. Let $U = (\nu n.\, \alpha(n))\cdot(\nu n.\, \alpha(n))$, and recall the policy FRESH(k) from Example 3. We have that $U \models \mathrm{FRESH}(2)$, because the two resources that are the targets of α are

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

distinct. Let $\mathcal{W} = \{\#_1, \#_2\}$. We have that

$$\begin{aligned}\mathsf{B}_{\mathcal{W}}(U) \;= &\big(new(_)\cdot\alpha(_) + new(\#_1)\cdot\alpha(\#_1) + new(\#_2)\cdot\alpha(\#_2)\big)\cdot\\ &\big(new(_)\cdot\alpha(_) + new(\#_1)\cdot\alpha(\#_1) + new(\#_2)\cdot\alpha(\#_2)\big).\end{aligned}$$

Note that the BPA $\mathsf{B}_{\mathcal{W}}(U)$ violates the policy FRESH(2), e.g. because of the trace $\eta = new(\#_1)\alpha(\#_1)new(\#_1)\alpha(\#_1)$. This would seem to suggest a possible source of incompleteness of our verification technique. However, note that there is something odd in the trace η above: actually, the action *new* is fired twice on the same witness $\#_1$. We will exploit this insight to recover completeness for our verification technique. This will be done by composing the instantiated usage automaton with the unique witness automaton (Definition 31) by a weak until operator (Definition 34). □

As shown in Example 14, the fact that U respects φ does not always imply that $\mathsf{B}_{\mathcal{W}}(U)$ respect φ, so leading to a sound but incomplete decision procedure. The problem is that $\mathsf{B}_{\mathcal{W}}(U)$ may use the same witness for different resources created by U. This may lead to violations of policies, e.g. those that prevent some actions from being performed twice (or more) on the same resource. To recover a (sound and) complete decision procedure, it suffices to check any trace of $\mathsf{B}_{\mathcal{W}}(U)$ only *until* the second witness $\#_i$ is generated for some $\#_i \in \mathcal{W}$, i.e. before the second occurrence of $new(\#_i)$. This is accomplished by composing $\mathsf{B}_{\mathcal{W}}(U)$ with the 'unique witness' automaton through a weak until operator, defined below.

The unique witness $\mathsf{A}_{\mathcal{W}}$ is a finite state automaton that reaches an offending state on those traces containing more than one $new(\#_i)$ event, for some $\#_i \in \mathcal{W}$. This is sound, because these traces are non-well-formed and will never be generated by a usage – indeed, they are spuriously introduced by the mapping in Definition 20. We write $A \vee B$ for the FSA that recognizes the union of the languages of the FSAs A and B.

Definition 31 (unique witness). Let $\mathcal{W}$ be a finite set of witnesses. The *unique witness* automaton $\mathsf{A}_{\mathcal{W}}$ is defined as follows:

$$\mathsf{A}_{\mathcal{W}} \;=\; \bigvee_{\#_i \in \mathcal{W}} \mathsf{A}_{\#_i}$$

where the automata $\mathsf{A}_{\#_i} = \langle \Sigma, \{q_0^i, q_1^i, q_2^i\}, q_0^i, \{q_2^i\}, \rho_{\#_i}\rangle$ are defined as follows:

$$\begin{aligned}\rho_{\#_i} = &\{q_0^i \xrightarrow{new(\#_i)} q_1^i,\; q_1^i \xrightarrow{new(\#_i)} q_2^i\}\\ &\cup \left\{q_0 \xrightarrow{\vartheta} q_0,\; q_1 \xrightarrow{\vartheta} q_1 \,\middle|\, \vartheta \in \Sigma \setminus \{new(\#_i)\}\right\} \cup \left\{q_2 \xrightarrow{\vartheta} q_2 \,\middle|\, \vartheta \in \Sigma\right\}.\end{aligned}$$

Lemma 32. For all histories η and for all sets of witnesses $\mathcal{W}$:

$$\eta \lhd \mathsf{A}_{\mathcal{W}} \quad\Longrightarrow\quad \forall \#_i \in \mathcal{W} \;:\; W_i(\eta) = 1$$

Proof. Straightforward from Definition 31. □

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

The following corollary of Theorem 30 allows us to reconcile completeness of the transformation $\mathsf{B}_{\mathcal{W}}(U)$ with policy compliance. For all the traces η' of $\mathsf{B}_{\mathcal{W}}(U)$ that respect the unique witness automaton $\mathsf{A}_{\mathcal{W}}$, there exists a corresponding trace of the usage U, modulo κ. Unlike in Theorem 30, now we can choose an *injective* collapsing κ, and so Lemma 12 guarantees that policy compliance will be preserved.

Corollary 33. Let U be an initial usage, let $\mathcal{W}$ be a finite set of witnesses, and let κ be an injective collapsing such that $\kappa(\mathsf{Res}_d) \supseteq \mathcal{W}$. For all traces η', if

$$\eta' \in [\![\mathsf{B}_{\mathcal{W}}(U)]\!] \qquad\qquad \eta' \lhd \mathsf{A}_{\mathcal{W}}$$

then there exists a trace η such that

$$\eta \in [\![U]\!]_{\varnothing} \qquad\qquad \eta' = \eta\,\kappa.$$

Proof. Straightforward from Theorem 30 and Lemma 32. □

We are almost ready to prove our main result, that is a sound and complete method for statically checking policy compliance of usages. Still, an auxiliary definition is needed. For all usages U and for all usage automata φ, let the set $R(U,\varphi)$ denote $res(U) \cup res(\varphi) \cup \{\#_i\}_{i\in 1..|\varphi|}$, where $res(U)$ comprises all the static resources mentioned in U. In Definition 34 we set up the weak until operator W. The main result eventually follows in Theorem 35.

Definition 34 (weak until). Let $A_0 = \langle S, Q_0, q_0, F_0, \delta_0\rangle$ and $A_1 = \langle S, Q_1, q_1, F_1, \delta_1\rangle$ be complete automata, i.e. for each state q and $\alpha \in S$, there exists a transition from q labelled by α. We define the *weak until* automaton $A_0\ \mathsf{W}\ A_1 = \langle S, Q, q, F, \delta\rangle$ as follows:

$$\begin{aligned}
Q &= Q_0 \times Q_1\\
F &= F_0 \times (Q_1 \setminus F_1)\\
q &= q_0 \times q_1\\
\delta &= \{\langle q_i, q_j\rangle \xrightarrow{\vartheta} \langle q_i', q_j'\rangle \mid q_i \xrightarrow{\vartheta} q_i' \in \delta_0, q_j \xrightarrow{\vartheta} q_j' \in \delta_1, q_j \in Q_1 \setminus F_1\}\\
&\quad \cup \{\langle q_i, q_j\rangle \xrightarrow{\vartheta} \langle q_i, q_j\rangle \mid q_j \in F_1\}.
\end{aligned}$$

Theorem 35. Let U be an initial usage, let φ be a usage automaton, and let $\mathcal{W} = \{\#_i\}_{i\in 1..|\varphi|}$. Then, $U \models \varphi$ if and only if

$$\forall \sigma : var(\varphi) \to R(U,\varphi)\ :\ [\![\mathsf{B}_{\mathcal{W}}(U)]\!] \lhd A_\varphi(\sigma, res(U) \cup \mathcal{W} \cup \{_\})\ \mathsf{W}\ \mathsf{A}_{\mathcal{W}}.$$

Proof. For the 'if' case (soundness), we prove the contrapositive. Assume that $U \not\models \varphi$. So, pick a trace η such that

$$\eta \in [\![U]\!]_{\varnothing} \qquad\qquad \eta \not\models \varphi.$$

By Lemma 11, there exists an injective collapsing κ such that

$$\kappa(\mathsf{Res}_d) \subseteq \mathcal{W} \cup \{_\} \qquad\qquad \eta\kappa \not\models \varphi.$$

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

By Theorem 5, there exists $\sigma : var(\varphi) \to R(\eta\kappa, \varphi)$ such that $\eta\kappa \not\triangleleft A_\varphi(\sigma, res(\eta\kappa))$. Since $res(\eta\kappa) \subseteq (res(\eta) \cap \mathsf{Res}_s) \cup \mathcal{W} \cup \{_\} \subseteq res(U) \cup \mathcal{W} \cup \{_\}$, then by Lemma 4:

$$\eta\kappa \not\triangleleft A_\varphi(\sigma, res(U) \cup \mathcal{W} \cup \{_\}).$$

By Theorem 28, we have that

$$\eta\kappa \in [\![\mathsf{B}_\mathcal{W}(U)]\!].$$

Since U is initial, it can only generate well-formed traces: so, η is well-formed. Furthermore, since κ is an injective collapsing into $\mathcal{W}$, then $\eta\kappa \lhd \mathsf{A}_\mathcal{W}$.

Summing up, we have found a substitution $\sigma : var(\varphi) \to R(U, \varphi)$ such that

$$[\![\mathsf{B}_\mathcal{W}(U)]\!] \ni \eta\kappa \not\triangleleft A_\varphi(\sigma, res(U) \cup \mathcal{W} \cup \{_\}) \mathsf{W} \mathsf{A}_\mathcal{W}$$

which is the desired contradiction.

For the 'only if' part (completeness) we prove the contrapositive. Pick a trace η' and a substitution $\sigma : var(\varphi) \to R(U, \varphi)$ such that

$$\eta' \in [\![\mathsf{B}_\mathcal{W}(U)]\!] \qquad \eta' \not\triangleleft A_\varphi(\sigma, res(U) \cup \mathcal{W} \cup \{_\}) \qquad \eta' \lhd \mathsf{A}_\mathcal{W}.$$

Let κ be an injective collapsing such that $\kappa(\mathsf{Res}_d) \supseteq \mathcal{W}$. Since U is initial and $\eta' \lhd \mathsf{A}_\mathcal{W}$, then by Corollary 33 there exists η such that

$$\eta \in [\![U]\!]_\varnothing \qquad \eta' = \eta\,\kappa.$$

Since $res(\eta\kappa) \subseteq (res(\eta) \cap \mathsf{Res}_s) \cup \mathcal{W} \cup \{_\} \subseteq res(U) \cup \mathcal{W} \cup \{_\}$, then by Lemma 4:

$$\eta\kappa \not\triangleleft A_\varphi(\sigma, res(\eta\kappa)).$$

By Theorem 5, the above implies that $\eta\kappa \not\models \varphi$. Therefore, we can apply Lemma 12 to deduce that $\eta \not\models \varphi$ – the desired contradiction. □

We now show that our method for verifying the validity of a usage U can be efficiently implemented. Indeed, we first associate with U a BPA $\mathsf{B}(U)$, and then we model-check $\mathsf{B}(U)$ against a finite set of finite state automata. Both phases require a polynomial number of steps in the size $|U|$ of U (the number of nodes of the a abstract syntax tree of U). However, it requires an exponential factor in the arity of the given usage automaton. There is a hidden exponential factor in the size (number of states) of the usage automaton, needed to make the FSA deterministic. These exponential factors should not have a significant impact in practice, since one expects that real-world policies to be expressed with a small number of states and variables.

Theorem 36. The computational complexity of verifying that $U \models \varphi$ is

$$\mathcal{O}(|U|^{|\varphi|+1}).$$

Proof. (outline). Let $n = |U|$, and let $k = |\varphi|$, assuming that $n \gg k$. We estimate the number of nodes of $\mathsf{B}(U)$ through the following recursive equation $F(n,k)$. If n or k are 0, then there is only the root, so $F(n,k) = 1$. For the general case, we can over-approximate F as follows:

$$F(n,k) \leqslant F(n-1,k) + k \times F(n-1,k-1) + k + 2$$

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

which corresponds to the case of Definition 20:

$$\mathsf{B}_{\mathcal{W}}(\nu n.U)_{\Theta} = new(_) \cdot \mathsf{B}_{\mathcal{W}}(U\{_/n\})_{\Theta} + \sum_{\#_i \in \mathcal{W}} new(\#_i) \cdot \mathsf{B}_{\mathcal{W}\setminus\{\#\}_i}(U\{\#_i/n\})_{\Theta}.$$

We now prove that $F(n,k) \leqslant n^{k+1} + 1$. We proceed by induction on n. The base case is trivial. For the inductive case, by the induction hypothesis and the binomial theorem it follows that

$$\begin{aligned} F(n,k) &\leqslant F(n-1,k) + k \times F(n-1,k-1) + k + 2 \\ &\leqslant (n-1)^{k+1} + 1 + k \times ((n-1)^k + 1) + k + 2 \\ &\leqslant ((n-1)+1)^{k+1} + 1 \\ &= n^{k+1} + 1. \end{aligned}$$

Therefore, $F(n,k) = \mathcal{O}(n^{k+1})$. □

7. Local policies

In the previous section we have devised a model checking technique which is capable of statically enforcing a *global* policy on a given usage.

However, in real-world scenarios, e.g. in SOC, one has to deal with a set of interacting services, each with its own *local* requirements on the usage of resources. The usage at hand should then abstract from the whole service composition, and it should contain information about which policies have to be enforced on each sub-component.

To cope with this situation, we proposed in (Bartoletti *et al.* 2005) *local policies*, that formalize and enhance the concept of sandbox. Traces and usages were extended there with special events that denote entering/leaving a sandbox.

The idea is the following. Assume you have a usage $U = U_0 \cdot U_1 \cdot U_2$ and a policy φ to be enforced locally on U_1. Then, you can put U_1 within a sandbox for φ. The new usage is written as $U' = U_0 \cdot [_{\varphi} \cdot U_1 \cdot]_{\varphi} \cdot U_2$, where the *framing events* $[_{\varphi}$ and $]_{\varphi}$ denote entering the sandbox for φ and leaving it, respectively. Roughly, all the traces produced by U_1 must respect the policy φ. Since φ might be legitimately interested in what happened before the sandbox was entered, we allow it to inspect also the trace generated before $[_{\varphi}$. However, we can abort executions which violate φ only *inside* a sandbox for φ, i.e. when φ is *active*.

To deal with local policies, we extend the set Act of actions with framing events $[_{\varphi}$, $]_{\varphi}$ for all φ. These events can be used both in traces and in usages. For example, a trace $\alpha\,[_{\varphi}\,\alpha'\,]_{\varphi}\alpha''$ represents a computation that (i) generates an event α, (ii) enters the scope of φ, (iii) generates α' within the scope of φ, (iv) leaves the scope of φ and (v) generates an event α'' outside the scope of φ. Intuitively, ε, α and $\alpha\alpha'$ must respect φ, while $\alpha\alpha'\alpha''$ will not be required to. Usages U can only use framing events in a disciplined manner, that is through the construction $[_{\varphi} \cdot U \cdot]_{\varphi}$, called φ-*framing* of U and often abbreviated as $\varphi[U]$.

We say that η has balanced framings when each opening framing event $[_{\varphi}$ has a well-nested, corresponding closing $]_{\varphi}$. For example, $\alpha[_{\varphi}\alpha'[_{\varphi'}\alpha'']_{\varphi'}]_{\varphi}$ is balanced, while $\alpha[_{\varphi}\alpha'[_{\varphi'}\alpha'']_{\varphi}$ is not. Herewith, we shall only consider traces that are prefixes of balanced histories. This property is always guaranteed for the traces generated by usages.

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

Table 6. *Additional notation for local policies.*

$[_\varphi, [_{\varphi'}, \ldots$	Opening framing events
$]_\varphi,]_{\varphi'}, \ldots$	Closing framing events
$act(\eta)$	Multiset of active policies in η
$\Phi(\eta)$	Set of policies occurring in η
$\Phi(U)$	Set of policies occurring in U
$\eta^{-[]}$	Trace η with framing events removed
$\models \eta$	η is valid
$\varphi_{[]}$	Framed usage automaton

We now define the multiset of policies which are active for a trace.

Definition 37 (active policies). The multiset $act(\eta)$ of the active policies of a trace η is defined as follows:

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

Traces and usages are *valid* when they respect all the relevant active policies. To give some intuition, recall from Example 6 the policy φ_{IF}, saying that private files cannot be sent in plain text. Let $\eta = private(f)\,[_{\varphi_{\mathrm{IF}}}\,send(f)\,]_{\varphi_{\mathrm{IF}}}$. Then, η is *not* valid, because $private(f)\,send(f)$ does not obey the active policy φ_{IF}. Note that the ability of checking the *whole* trace was crucial: if one were not able to observe the events before the opening of the framing, he would unsafely deduce that η is valid.

Definition 38 (validity of traces and usages). 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 $\eta^{-[]}$, i.e. η with all the framing events removed, is defined as follows:

$$\varepsilon^{-[]} = \varepsilon \qquad \alpha(\vec{r})^{-[]} = \alpha(\vec{r}) \qquad (\,\eta[_\varphi\,)^{-[]} = (\,\eta\,]_\varphi\,)^{-[]} = \eta^{-[]}.$$

A usage U is *valid* when, for all $U', \mathcal{R}, \mathcal{R}'$ and η, if $U, \mathcal{R} \xrightarrow{\eta} U', \mathcal{R}'$ then $\models \eta$.

Example 15. Let $U = \alpha_0 \cdot \varphi[\alpha_1 \cdot \varphi'[\alpha_2] \cdot \alpha_3]$. The following is a trace of U:

$$\eta = \alpha_0\,[_\varphi\,\alpha_1\,[_{\varphi'}\,\alpha_2\,]_{\varphi'}\,\alpha_3.$$

We have that η is valid if and only if: $\varepsilon \models \varphi$, $\alpha_0 \models \varphi$, $\alpha_0\alpha_1 \models \varphi$, $\alpha_0\alpha_1\alpha_2 \models \varphi$, $\alpha_0\alpha_1\alpha_2\alpha_3 \models \varphi$, and $\alpha_0\alpha_1 \models \varphi'$, $\alpha_0\alpha_1\alpha_2 \models \varphi'$. □

Validity of traces is a safety property (Schneider 2000). Also, validity is not compositional in general, in the sense that the validity of two traces does not imply the validity of their composition. This is a consequence of the assumption that no past events can be hidden. Note in passing that, differently from validity, policy compliance $\eta \models \varphi$ of

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

Definition 3 is *not* prefix-closed. This allows for defining policies which permit to recover from offending states, see e.g. Example 7.

Lemma 39. For all traces η and η':

$$\models \eta\,\eta' \implies \models \eta \qquad \text{(prefix-closed)}$$
$$\models \eta \text{ and } \models \eta' \not\Longrightarrow \models \eta\,\eta' \qquad \text{(not compositional).}$$

Proof. The proof of the prefix-closedness is immediate, by definition. The fact that validity is not compositional follows from the following example. Consider the trace $\eta = \alpha[_\varphi\,\alpha\,]_\varphi\,\alpha$, where φ requires that α cannot be fired more than twice. 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'. We have that $act(\bar{\eta}) = \{\!|\varphi|\!\}$, but $\bar{\eta}^{-[]} = \alpha\alpha\alpha \not\models \varphi$. This shows that validity is not compositional. □

Example 16. Recall from Example 7 the usage automaton φ_{Loan}, that prevents anyone from taking out a loan if their account is in the red. We have that *red black* $[_{\varphi_{\text{Loan}}}$ is valid. Indeed, *red black* $\models \varphi_{\text{Loan}}$ — 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). □

We specify in Definition 40 below a transformation of usage automata that enables them to recognize valid traces. This is done by instrumenting a usage automaton φ with framing events, resulting in a *framed* usage automaton $\varphi_{[]}$ that will recognize those traces that are valid with respect to the policy φ.

Definition 40 (framed usage automata). Let $\varphi = \langle S, Q, q_0, F, E\rangle$ be a usage automaton. We define the *framed* usage automaton $\varphi_{[]} = \langle S', Q', q_0, F', E'\rangle$ as:

$$\begin{aligned}
S' &= S \cup \{[_\varphi,]_\varphi, [_{\varphi'},]_{\varphi'}, \ldots\}\\
Q' &= Q \cup \{\,\dot{q}\,|q \in Q\,\}\\
F' &= \{\,\dot{q}\,|q \in F\,\}\\
E' &= E \cup \left\{ q \xrightarrow{[_\varphi} \dot{q} \,\middle|\, q \in Q \right\} \cup \left\{ \dot{q} \xrightarrow{]_\varphi} q \,\middle|\, q \in Q \setminus F \right\}\\
&\qquad \cup \left\{ \dot{q} \xrightarrow{\alpha(\vec{\rho}):g} \dot{q}' \,\middle|\, q \xrightarrow{\alpha(\vec{\rho}):g} q' \in E \text{ and } q \in Q \setminus F \right\}.
\end{aligned}$$

Intuitively, the usage automaton $\varphi_{[]}$ is partitioned into two layers. Both are copies of φ, but all the states in the first layer of $\varphi_{[]}$ are made non-final. This represents being outside the φ-framing. 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, offending state. The transitions

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

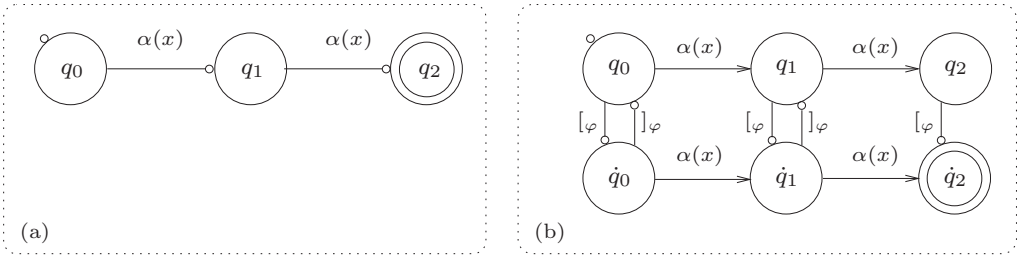


Fig. 9. (a) Usage automaton φ. (b) Framed usage automaton $\varphi_{[]}$.

in the second layer are copies of those in φ, the only difference being that the final states are sinks. The final states in the second layer are exactly those final in φ.

Example 17. Recall from Example 3 the policy FRESH, which prevents from firing the action α twice on the same resource. The usage automaton φ that models FRESH is displayed in the left-hand side of Figure 9. The framed version $\varphi_{[]}$ is displayed in the right-hand side of the same figure. □

In the rest of this section, we shall only consider traces without 'redundant' framings, i.e. we will discard traces of the form $\eta\,[_\varphi\,\eta'\,[_\varphi$ with $]_\varphi \notin \eta'$. We do this without loss of generality. Actually, in (Bartoletti *et al.* 2005) we defined a static transformation of usages that removes these redundant framings, while preserving validity. 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 this transformation has always been applied to usages.

We will relate framed usage automata with validity in Lemma 43. A trace η (which has no redundant framings, after the assumed transformation) is valid if and only if it complies with the framed automata $\varphi_{[]}$, for all the policies φ spanning over η. Before proving Lemma 43, a couple of auxiliary results are needed. Lemma 41 relates the instantiations of usage automata with the instantiations of their framed versions.

Lemma 41. Let η be a trace (without redundant framings), let φ be a usage automaton, let $\mathcal{R} \subseteq \mathsf{Res}$, and let $\sigma : var(\varphi) \to \mathcal{R} \setminus \{_\}$. Then

$$(a) \quad \eta^{-[]} \lhd A_\varphi(\sigma,\mathcal{R}) \implies \eta \lhd A_{\varphi_{[]}}(\sigma,\mathcal{R})$$
$$(b) \quad \eta \lhd A_{\varphi_{[]}}(\sigma,\mathcal{R}) \implies \eta^{-[]} \lhd A_\varphi(\sigma,\mathcal{R}) \text{ or } \varphi \notin act(\eta)$$

Lemma 42 connects the active policies of η with the compliance between η and $\varphi_{[]}$.

Lemma 42. For all traces η (without redundant framings) and for all φ:

$$\eta \not\models \varphi_{[]} \implies \varphi \in act(\eta).$$

The following lemma characterizes trace validity in terms of policy compliance. Below, we denote with $\Phi(\eta)$ the set of usage automata φ such that $[_\varphi$ or $]_\varphi$ occur in η. Note that, since policy compliance is decidable (Theorem 5), then the following lemma states that also trace validity is decidable.

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

Lemma 43. A trace η (without redundant framings) is valid if and only if

$$\forall \varphi \in \Phi(\eta) \ : \ \eta \models \varphi_{[]}.$$

Our last theorem characterizes usage validity in terms of policy compliance. Together with Theorem 35, it provides us with a sound and complete model checking technique for verifying the validity of usages. Recall that we have assumed the usage at hand transformed with the algorithm in (Bartoletti *et al.* 2005), which guarantees the absence of redundant framings. Below, we denote with $\Phi(U)$ the set of usage automata φ such that $[_\varphi$ or $]_\varphi$ occur in U.

Theorem 44. For all initial usages U, U is valid if and only if

$$\forall \varphi \in \Phi(U) \ : \ U \models \varphi_{[]}.$$

Proof. Straightforward from Lemma 43. □

Example 18. Consider the following usage:

$$U = \varphi[\nu n.\ \varphi_{read1}\,[\mu h.(write(n) \cdot h + read(n) \cdot dispose(n))] \cdot write(n) \cdot read(n)]$$

where φ is the usage automaton displayed in Figure 1, while φ_{read1} is the usage automaton displayed in Figure 7, where α has been replaced with *read*.

Intuitively, U models a usage pattern where a new object n is created, it may be written to several times (the loop is modelled by the recursion construct μh), and, on leaving the loop, n is read and finally disposed off. The policy φ_{read1} is active throughout the loop. Once the scope of φ_{read1} is left, the object n is written to and read again. The scope of the policy φ is the whole usage.

According to Theorem 44, U is valid if and only if $U \models \varphi_{[]}$ and $U \models \varphi_{read1\,[]}$. By Theorem 35, the translation of U into BPA $\mathsf{B}_{\mathcal{W}}(U)$ requires only two witnesses ($\mathcal{W} = \{\#_1, \#_2\}$). Roughly, here the dynamically generated resource n is replaced by the witnesses $\#_1$ and $\#_2$ in two different branches (the third branch with the dummy resource $_$ is immaterial in this case, since there is only one ν). When checking the BPA against $\varphi_{read1\,[]}$, say on $\#_1$, we discover that: (1) entering the policy framing makes the automaton move to the 'active' layer; (2) the first $read(\#_1)$ performed by the BPA makes the automaton reach state q_1; (3) leaving the policy framing makes the automaton return to the 'inactive' layer; (4) the second $read(\#_1)$ performed by the BPA makes the automaton reach state *fail*, but since we are in the 'inactive' layer, this is not an offending state. Overall, we have that U does not violate φ_{read1}. This is correct because, within the scope of φ_{read1}, only one *read* can be performed.

By comparison, when checking U against $\varphi_{[]}$ instantiated with $x = \#_1$ and $y = \#_2$, we discover that: (1) entering the policy framing makes the automaton move to the 'active' layer, where (2) the $new(\#_1)$ performed by the BPA makes the automaton reach state q_1; (3) writes are ignored by this automaton because of the self-loops; (4) $read(\#_1)$ is ignored in q_1; (5) $dispose(\#_1)$ makes the automaton move to q_0; (6) the final $read(\#_1)$ makes the automaton move to *fail*. This last state is offending, since it belongs to the 'active'layer in the framed automaton. Hence, there is a trace of U

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

causing a violation of φ. Indeed, within the scope of φ the object n is accessed after it is disposed.

8. Related work

Usage automata constitute a class of automata over infinite alphabets. The techniques developed in this paper provide an effective finite-state representation of this class of automata. The issue of extending finite-state techniques to automata over infinite alphabets has been previously considered, e.g. in (Grumberg *et al.* 2010; Kaminski and Francez 1994; Segoufin 2006; Shemesh and Francez 1994).

Variable finite automata (VFA) (Grumberg *et al.* 2010) are a class of automata where the alphabet consists of constant symbols, as well as variables ranging over an infinite alphabet. Variables comprise a finite set of bounded variables, and a single free variable. Each bounded variable is assigned to a different symbol, while the free variable is a sort of wildcard, which can be associated to different symbols in each occurrence. Some recent results show that VFA are strictly more expressive than usage automata (Degano *et al.* 2011). This seems related to the free variable, which has no counterpart in usage automata. We conjecture that a similar wildcard construct can be added to usage automata, while preserving the model checking result. Our model checking technique can be extended as in (Degano *et al.* 2011) to construct an algorithm for model checking usages against VFA.

Register automata (Kaminski and Francez 1994; Shemesh and Francez 1994) extend finite-state automata with a finite set of registers. Each register can store a symbol taken from an infinite alphabet. Register automata have been proved in (Ciancia and Tuosto 2009) to have the same expressive power as history-dependent automata, a special class of automata which can manage fresh name generation (Montanari and Pistore 2005). Register automata are more expressive than our usage automata. Intuitively, our witnesses play the same role of the content of registers, but while registers can be updated during a run, witnesses are constant entities. Register automata are not simple to use – quoting from (Grumberg *et al.* 2010): 'the formalisms of register, pebble and data automata all fail hard the simplicity criterion'. We think usage automata are a simpler specification formalism because they are more declarative in style, since each variable is bound to the same resource for the whole lifetime of any instantiation.

A characterization of the class of policies enforceable through run-time monitoring systems is given in (Alpern and Schneider 1987; Schneider 2000). There, the policy language is that of *security automata*, a class of Büchi automata that recognize safety properties. To handle the case of parameters ranging over infinite domains, security automata resort to a countable set of states and input symbols. The transition relation of security automata needs not even be computable, therefore, they are strictly more expressive than usage automata. Several other mechanisms, e.g. (Bauer *et al.* 2005; Martinelli and Mori 2007; Pandey and Hashii 1999) are Turing-equivalent, and so able to recognize more policies than usage automata. Yet, this flexibility comes at a cost. First, the process of deciding whether an action must be denied or not might not terminate. Second, non-trivial static optimizations are unfeasible, unlike in our approach. In particular, no

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

precise static guarantee can be given about the compliance of a program with the imposed policy: run-time monitoring is then needed to enforce the policy, while our usage automata are model-checkable, so possibly avoiding this overhead.

In (Igarashi and Kobayashi 2002), policies are modelled as sets of permitted usage patterns, to be attached to resources upon creation. Our usage automata are not hard-wired to resources, yet they are *parametric* over resources. For instance, a usage automaton φ with variables x, y means that, for all the possible instantiations of x and y to actual resources, 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.

The usage automata and the model checking technique presented in this paper extend and improve previous versions appeared in (Bartoletti 2009; Bartoletti *et al.* 2008a; Bartoletti *et al.* 2009c). The main improvement w.r.t. past approaches is the ability to deal with an arbitrary number of variables, while retaining a polynomial-time model checking algorithm. Also, the current paper features a neater technical apparatus, which builds upon an indistinguishability result (Theorem 13) to devise a model checking algorithm for global policies, which smoothly scales to the general case of local policies.

9. Conclusions

We proposed a model for policies which control the usage of resources. Usage automata follow the history-based approach to security, and they can be enforced through finite state automata. A basic calculus of usages was presented to describe the patterns of resource access and creation, and obligations to respect usage policies, possibly within a local scope.

In spite of the augmented flexibility given by resource creation and by policy parametrization, we devised a complete and efficient (essentially polynomial-time) model checking technique for deciding when a usage complies with *all* the usage policies it is subject to, within their local scope. 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 usage properties of programs. When there are policies that a usage does not obey, only these have to be monitored at run time, essentially also requiring polynomial-time. We have shown the applicability of our theory by modelling a variety of relevant policies and applications.

The theory presented in this paper has been used for two implementations:

— Jalapa (Bartoletti *et al.* 2009a), an extension to the security model of Java. It features history-based usage policies, specified through usage automata. Programmers can define their scope through a 'sandbox' construct. The enforcement of policies is based on bytecode rewriting, and it is transparent to programmers (they need not insert explicit policy checks).

— LocUsT('local usage policies tool'), a tool which model checks usages U against usage automata φ. This implements the model checking technique of Sections 6 and 7, mainly exploiting Theorems 35 and 44.

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

We now briefly comment on the LocUsT implementation. Roughly, given a usage U and a usage automaton φ, it (1) regularizes U through the algorithm described in (Bartoletti *et al.* 2005), (2) converts U into a BPA following Table 5, (3) computes a finite number of framed instantiations of φ (one for each σ considered in Theorem 35), (4) composes each instantiation with the "unique witness" automaton (as required by the same theorem) and (5) model checks the BPA of step (2) against each of the FSAs obtained at step (4).

More in detail, in the last step LocUsT checks whether the set of (prefixes of) traces of the BPA and the language of the FSA have a non-empty intersection. This is implemented through a fixpoint algorithm, briefly described below. For all FSA states q and all BPA named processes X we compute two sets: (a) the set of FSA states reachable from q via a (finite) trace of X, and (b) the set of FSA states reachable from q via a prefix of a (possibly infinite) trace of X. This leads to defining a monotonic function over a domain which satisfies the ascending chain condition, because we only have a finite number of sets to compute, and each one is bounded from above by the set of all the states of the FSA. Hence, the fixpoint is computable (in PTIME). To check for language intersection we then test if from the initial state q_0 it is possible to reach a final (offending) state using a prefix of a trace of the initial process X_0.

Some relevant extensions to our model are possible. A first simple extension 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. $\mathsf{File} = \{open, close, read, write\}$). The syntax of the ν-constructor is extended with the type τ, i.e. $\nu n : \tau.\ U$. 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 pairwise disjoint sets of witnesses, each for each type (e.g. $T_{\mathsf{File}} = \{\#_{F1}, \#_{F2}, ...\}$, T_{Socket}, *etc.*), to be dealt with by the corresponding 'unique witness' automata. The overall time complexity is essentially unaltered; actually, partitioning resources according to their type might even reduce the impact of the exponential factor.

The language of usages has two simple extensions. The first consists in attaching policies to resources upon creation, similarly to (Igarashi and Kobayashi 2002). 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 (Christensen 1993) instead of BPAs. However, the time complexity becomes exponential in the number of parallel branches (Mayr 1998). A less immediate extension concerns adding quantitative information to both policies and operations on resources, so leading to a notion of *quantitative* validity of usages, see e.g. (Degano *et al.* 2011). It would be interesting to compare this notion with the more classical ones, based on a Markovian approach, e.g. (Hillston 1996), and to apply stochastic model checking to our case, e.g. with PRISM (Kwiatkowska *et al.* 2009).

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

Acknowledgements

We thank Roberto Grossi for pleasant discussion and the anonymous referees for their insightful comments.

Appendix A. Proofs

Proofs for Section 3

Lemma 4. For all usage automata φ, for all traces η, and for all substitutions $\sigma : var(\varphi) \to \mathsf{Res} \setminus \{_\}$:

$$\exists \mathcal{R} \supseteq res(\eta) : \eta \lhd A_\varphi(\sigma, \mathcal{R}) \implies \forall \mathcal{R}' \supseteq res(\eta) : \eta \lhd A_\varphi(\sigma, \mathcal{R}').$$

Proof. We prove the contrapositive. Assume there exists $\mathcal{R}' \supseteq res(\eta)$ such that $\eta \not\lhd A_\varphi(\sigma, \mathcal{R}')$. Then, there is a run $q_0 \xrightarrow{\eta} q_k$ of $A_\varphi(\sigma, \mathcal{R}')$ where $q_k \in F$. For all $\mathcal{R} \supseteq res(\eta)$, we reconstruct an offending run of $A_\varphi(\sigma, \mathcal{R})$ on η.

Let $\delta_{\mathcal{R}}$ and $\delta_{\mathcal{R}'}$ be the transition relations of $A_\varphi(\sigma, \mathcal{R})$ and $A_\varphi(\sigma, \mathcal{R}')$, respectively, let $\delta^*_{\mathcal{R}}$ and $\delta^*_{\mathcal{R}'}$ be their (labelled) reflexive and transitive closures, and let $\mathcal{X}_{\mathcal{R}}$ and $\mathcal{X}_{\mathcal{R}'}$ be the sets of edges instantiated without completion.

We will prove by induction on the length of η that $\delta^*_{\mathcal{R}'}(\{q_0\}, \eta) \subseteq \delta^*_{\mathcal{R}}(\{q_0\}, \eta)$.

The base case $\eta = \varepsilon$ is trivial.

For the inductive case, let $\eta = \eta' \alpha(\vec{r})$. By the induction hypothesis, we have that $\delta^*_{\mathcal{R}'}(\{q_0\}, \eta') \subseteq \delta^*_{\mathcal{R}}(\{q_0\}, \eta')$. Let now $q \in \delta^*_{\mathcal{R}'}(\{q_0\}, \eta')$, and assume that $q' \in \delta_{\mathcal{R}'}(q, \alpha(\vec{r}))$. Only one of the following two cases may occur.

— $q \xrightarrow{\alpha(\vec{r})} q' \in \mathcal{X}_{\mathcal{R}'}$.

Note that both $\mathcal{R}$ and $\mathcal{R}'$ give rise to the same set:

$$\mathcal{X}_{\mathcal{R}} = \mathcal{X}_{\mathcal{R}'} = \left\{ q \xrightarrow{\alpha(\vec{\xi}\sigma)} q' \;\middle|\; q \xrightarrow{\alpha(\vec{\xi}) : g} q' \in E \text{ and } \sigma \models g \right\}$$

Indeed, the sets $\mathcal{R}$ and $\mathcal{R}'$ only affect the self loops added by $complete(\mathcal{X}, \mathcal{R})$ and $complete(\mathcal{X}, \mathcal{R}')$, respectively.

— $q \xrightarrow{\alpha(\vec{r})} q' \in complete(\mathcal{X}_{\mathcal{R}'}, \mathcal{R}')$, with $q' = q$.

Since $r \in res(\eta)$ and both $\mathcal{R} \supseteq res(\eta)$ and $\mathcal{R}' \supseteq res(\eta)$, it follows that $q \xrightarrow{\alpha(\vec{r})} q' \in complete(\mathcal{X}, \mathcal{R})$ if and only if $q \xrightarrow{\alpha(\vec{r})} q' \in complete(\mathcal{X}, \mathcal{R}')$, which implies the thesis.

□

Theorem 5. For all traces η and for all usage automata φ, $\eta \models \varphi$ iff

$$\forall \sigma : var(\varphi) \to R(\eta, \varphi) \;:\; \eta \lhd A_\varphi(\sigma, res(\eta)).$$

Proof. For the 'only if' part, assume that $\eta \models \varphi$, i.e. that $\eta \lhd A_\varphi(\sigma', \mathsf{Res})$, for all $\sigma' : var(\varphi) \to \mathsf{Res} \setminus \{_\}$. Since $R(\eta, \varphi) \subseteq \mathsf{Res} \setminus \{_\}$, then it is also the case that $\eta \lhd A_\varphi(\sigma, \mathsf{Res})$. By Lemma 4, it follows that $\eta \lhd A_\varphi(\sigma, res(\eta))$.

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

For the 'if' part, we prove the contrapositive. Assume that $\eta \not\models \varphi$, i.e. $\eta \not\triangleleft A_\varphi(\sigma', \mathsf{Res})$ for some $\sigma' : var(\varphi) \to \mathsf{Res} \setminus \{_\}$. We will prove that there exists some $\sigma : var(\varphi) \to R(\eta, \varphi)$ such that $\eta \not\triangleleft A_\varphi(\sigma, res(\eta))$. We write $ran(\sigma)$ for the image of σ on its domain. Let

$$\mathcal{R} = ran(\sigma') \setminus R(\eta, \varphi) \qquad \text{(i.e. resources in } ran(\sigma') \text{ unavailable for } \sigma)$$
$$\mathcal{W} = \{\#_i\}_{i \in 1..|\varphi|} \setminus ran(\sigma') \qquad \text{(i.e. witnesses available for } \sigma).$$

Since $|\mathcal{R}| \leqslant |\mathcal{W}|$ holds by construction, then there exists an injective function $\omega : \mathcal{R} \to \mathcal{W}$. We exploit this fact to define the substitution $\sigma : var(\varphi) \to R(\eta, \varphi)$ needed to prove that $\eta \not\triangleleft A_\varphi(\sigma, res(\eta))$. We map each $x \in var(\varphi)$ into $\sigma'(x)$, if $\sigma'(x)$ belongs to the admissible codomain for σ, i.e. $R(\eta, \varphi)$. Otherwise, if $\sigma'(x)$ is a resource in $\mathcal{R}$, then σ maps x into an available witness in $\mathcal{W}$. In symbols

$$\sigma(x) = \begin{cases} \omega(\sigma'(x)) & \text{if } \sigma'(x) \in \mathcal{R} \\ \sigma'(x) & \text{otherwise.} \end{cases} \tag{A1}$$

We will first prove an auxiliary result. For all guards g:

$$\sigma \models g \iff \sigma' \models g. \tag{A2}$$

We proceed by induction on the structure of g. The base case $g = true$ is trivial. The other base case is when g has the form $\xi_0 = \xi_1$. Exactly one of the following four subcases holds.

— $\xi_0, \xi_1 \in \mathsf{Res}_s$. The thesis follows from $\sigma(\xi_0) = \xi_0 = \sigma'(\xi_0)$ and $\sigma(\xi_1) = \xi_1 = \sigma'(\xi_1)$.
— $\xi_0 \in \mathsf{Var}, \xi_1 \in \mathsf{Res}_s$. Then, $\sigma(\xi_1) = \xi_1 = \sigma'(\xi_1)$. By (A1), we have that

$$\sigma(\xi_0) = \begin{cases} \omega(\sigma'(\xi_0)) & \text{if } \sigma'(\xi_0) \in \mathcal{R} \qquad \text{(a1)} \\ \sigma'(\xi_0) & \text{otherwise} \qquad \text{(a2).} \end{cases}$$

($\Longleftarrow$) Assume that $\sigma' \models \xi_0 = \xi_1$. We are in the case (a2), because $\xi_1 \in res(\varphi)$ implies that $\xi_1 \notin \mathcal{R}$. Therefore, $\sigma(\xi_0) = \sigma'(\xi_0) = \xi_1 = \sigma(\xi_1)$.

($\Longrightarrow$) Assume that $\sigma \models \xi_0 = \xi_1$. Again, we are in the case (a2), because $\xi_1 \notin \mathcal{R}$. Therefore, $\sigma'(\xi_0) = \sigma(\xi_0) = \sigma(\xi_1) = \xi_1 = \sigma'(\xi_1)$.

— $\xi_0 \in \mathsf{Res}_s, \xi_1 \in \mathsf{Var}$. Symmetric to the previous case.
— $\xi_0, \xi_1 \in \mathsf{Var}$. By (A1), we have that

$$\sigma(\xi_0) = \begin{cases} \omega(\sigma'(\xi_0)) & \text{if } \sigma'(\xi_0) \in \mathcal{R} \qquad \text{(a1)} \\ \sigma'(\xi_0) & \text{otherwise} \qquad \text{(a2)} \end{cases}$$

$$\sigma(\xi_1) = \begin{cases} \omega(\sigma'(\xi_1)) & \text{if } \sigma'(\xi_1) \in \mathcal{R} \qquad \text{(b1)} \\ \sigma'(\xi_1) & \text{otherwise.} \qquad \text{(b2)} \end{cases}$$

($\Longleftarrow$) Assume that $\sigma' \models \xi_0 = \xi_1$. Let $r = \sigma'(\xi_0)$. If $r \notin \mathcal{R}$, then we are in the cases (a2,b2), and so $\sigma(\xi_0) = \sigma'(\xi_0) = r = \sigma'(\xi_1) = \sigma(\xi_1)$. Otherwise, if $r \in \mathcal{R}$, then we are in the cases (a1,b1), and so $\sigma(\xi_0) = \omega(r) = \sigma(\xi_1)$. In both cases, we conclude that $\sigma \models \xi_0 = \xi_1$.

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

($\Longrightarrow$) Assume that $\sigma \models \xi_0 = \xi_1$, i.e. $\sigma(\xi_0) = r = \sigma(\xi_1)$ for some r. We have that either $r \in \mathcal{W}$, and so the cases (a1,b1) apply, or $r \in ran(\sigma')\backslash\mathcal{R}$, and since $(ran(\sigma')\backslash\mathcal{R})\cap\mathcal{W} = \varnothing$, then the cases (a2,b2) apply.
In the cases (a1,b1), we have that $\sigma(\xi_0) = \omega(\sigma'(\xi_0)) = \omega(\sigma'(\xi_1)) = \sigma(\xi_1)$. Since ω is injective, then we conclude $\sigma'(\xi_0) = \sigma'(\xi_1)$.
In the cases (a2,b2), we have that $\sigma'(\xi_0) = \sigma(\xi_0) = \sigma(\xi_1) = \sigma'(\xi_1)$.
In both cases, we have proved that $\sigma' \models \xi_0 = \xi_1$.

This concludes the proof of (A2) in the base case $\xi_0 = \xi_1$. The proof of the inductive cases $\neg g$ and $g_0 \wedge g_1$ is straightforward.

Back to the main statement, since $\eta \not\triangleleft A_\varphi(\sigma', \mathsf{Res})$, then by Definition 3 there exists a run $q_0 \xrightarrow{\eta} q_k$ of $A_\varphi(\sigma', \mathsf{Res})$ on η such that $q_k \in F$. We will reconstruct an offending run of $A_\varphi(\sigma, res(\eta))$ on η. Let δ and δ' be the transition relations of $A_\varphi(\sigma, res(\eta))$ and $A_\varphi(\sigma', \mathsf{Res})$, respectively, let δ^* and δ'^* be their (labelled) reflexive and transitive closures, and let $\mathcal{X}$ and $\mathcal{X}'$ be the sets of edges instantiated without completion.

We prove by induction on the length of η:

$$\delta'^*(\{q_0\}, \eta) \subseteq \delta^*(\{q_0\}, \eta).$$

The base case $\eta = \varepsilon$ is trivial.

For the inductive case, let $\eta = \eta'\alpha(\vec{r})$. By the induction hypothesis, we have that $\delta'^*(\{q_0\}, \eta') \subseteq \delta^*(\{q_0\}, \eta')$. Let now $q \in \delta'^*(\{q_0\}, \eta')$, and assume that $q' \in \delta'(q, \alpha(\vec{r}))$. Only one of the following two cases may occur.

1. $q \xrightarrow{\alpha(\vec{r})} q' \in \mathcal{X}'$.

   Then, there exists an edge $q \xrightarrow{\alpha(\vec{\xi}):g} q'$ in φ such that

   $$\sigma' \models g \qquad\qquad \vec{\xi}\,\sigma' = \vec{r}.$$

   By (A2), it follows that $\sigma \models g$. Also, since each component of $\vec{r}$ is in $res(\eta)$, and $res(\eta) \cap \mathcal{R} = \varnothing$, then by (A1) we have that $\vec{\xi}\sigma = \vec{\xi}\sigma' = \vec{r}$. Therefore, the edge $q \xrightarrow{\alpha(\vec{\xi}):g} q'$ in φ can also be instantiated to the transition $q \xrightarrow{\alpha(\vec{r})} q' \in \mathcal{X}$, which proves the needed inclusion.

2. $q \xrightarrow{\alpha(\vec{r})} q' \in complete(\mathcal{X}', \mathsf{Res})$.

   Then, $q' = q$, and there exists no q'' such that $q \xrightarrow{\alpha(\vec{r})} q'' \in \mathcal{X}'$. We must prove that there also exists no q'' such that $q \xrightarrow{\alpha(\vec{r})} q'' \in \mathcal{X}$. By contradiction, assume that $q \xrightarrow{\alpha(\vec{r})} q'' \in \mathcal{X}$.

   Then, there would exist an edge $q \xrightarrow{\alpha(\vec{\xi}\,'):g'} q''$ in φ such that

   $$\sigma \models g' \qquad\qquad \vec{\xi}\,'\sigma = \vec{r}.$$

   By (A2), it follows that $\sigma' \models g'$. Similarly to the previous case, we also have that $\vec{\xi}\,'\sigma' = \vec{\xi}\,'\sigma = \vec{r}$. Therefore, $q \xrightarrow{\alpha(\vec{r})} q'' \in \mathcal{X}'$ – which is the desired contradiction.

□

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

Proofs for Section 4

Proposition 7. Removing guards decreases the expressive power of usage automata.

Proof. Recall from Example 1 the usage policy DIFF(k), that prevents the action α to be fired on more than k distinct resources. We shall prove that there exists no usage automaton *without guards* that recognizes DIFF(1).

By contradiction, assume that φ has no guards and recognizes DIFF(1). Let

$$\eta = \alpha(r)\alpha(r') \qquad \text{where } r, r' \notin res(\varphi) \text{ and } r \neq r'.$$

Since $\eta \notin \text{DIFF}(1)$, by Theorem 5 it follows that $\eta \not\lhd A_\varphi(\sigma, \{r, r'\})$ for some $\sigma : var(\varphi) \to R(\eta, \varphi)$. Also, since $\alpha(r) \in \text{DIFF}(1)$, then $\alpha(r) \lhd A_\varphi(\sigma, \{r, r'\})$. Any run of $A_\varphi(\sigma, \{r, r'\})$ on η will then have the form:

$$q_0 \xrightarrow{\alpha(r)} q_1 \xrightarrow{\alpha(r')} q_2 \qquad \text{where } q_0, q_1 \notin F \text{ and } q_2 \in F.$$

Since $q_1 \notin F$ and $q_2 \in F$, the transition from q_1 to q_2 cannot be a self-loop, and so there exists $q_1 \xrightarrow{\alpha(x)\!\!\circ} q_2 \in E$ such that $\sigma(x) = r'$. Also, it must be $q_0 \neq q_1$, because of the following. By contradiction, assume that $q_0 = q_1$.

Since $q_0 = q_1 \xrightarrow{\alpha(x)\!\!\circ} q_2 \in E$, then the trace $\alpha(r)$ would violate φ, which contradicts the hypothesis that φ recognizes DIFF(1).

Thus, $q_0 \neq q_1$, and so there exists $q_0 \xrightarrow{\alpha(y)\!\!\circ} q_1 \in E$ such that $\sigma(y) = r$. Consider now the trace:

$$\eta' = \alpha(r)\alpha(r) \in \text{DIFF}(1).$$

We have that $\eta' \not\lhd A_\varphi(\{x \mapsto r, y \mapsto r\}, \{r\})$, and so by Theorem 5 it follows that $\eta' \not\models \varphi$ – contradiction, because we assumed φ to recognize DIFF(1). □

Proposition 8. Decreasing the arity of events does not decrease the expressive power of usage automata.

Proof. Given a usage automaton φ, we show that restricting its events to act on a single target preserves the policy defined by φ, modulo an injective transformation of traces.

Let Act_1 be the set of actions defined as follows:

$$\mathsf{Act}_1 = \left\{ \alpha^1, \ldots, \alpha^{|\alpha|} \,\middle|\, \alpha \in \mathsf{Act} \right\} \qquad \mathsf{Act}_1 \cap \mathsf{Act} = \varnothing.$$

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

We define the following transformation, called *slicing*. For each usage automaton $\varphi = \langle S, Q, q_0, F, E\rangle$, we define the usage automaton $slice(\varphi) = \langle S', Q', q_0, F, E'\rangle$ as follows:

$$S' = \left\{ \alpha^1(\xi_1), \ldots, \alpha^k(\xi_k) \middle| \alpha(\xi_1, \ldots, \xi_k) \in S \right\}$$

$$Q' = Q \cup \left\{ q_e^i \middle| e = \left(q \xrightarrow{\alpha(\tilde{\xi}):q} q' \right) \in E,\ i \in 1..|\alpha| - 1 \right\}$$

$$E' = \left\{ q \xrightarrow{\alpha^1(\xi_1):g} q_e^1,\ \ q_e^{i-1} \xrightarrow{\alpha^i(\xi_i):g} q_e^i,\ \ q_e^{k-1} \xrightarrow{\alpha^k(\xi_k):g} q' \ \middle|\ e = \left(q \xrightarrow{\alpha(\xi_1,\ldots,\xi_k):g} q' \right) \in E,\ i \in 2..k-1 \right\}.$$

We then define the injective transformation $slice : \mathsf{Ev}^* \to (\mathsf{Act}_1 \times \mathsf{Res})^*$, that maps the events on many resources into sequences of events on a single resource:

$$slice(\varepsilon) = \varepsilon \qquad slice(\eta\, \alpha(r_1, \ldots, r_k)) = slice(\eta)\, \alpha^1(r_1) \cdots \alpha^k(r_k).$$

It is straightforward to prove, by induction on the length of η, that for all usage automata φ and for all traces η:

$$\eta \models \varphi \iff slice(\eta) \models slice(\varphi).$$

□

Lemma 10. For traces η, for all collapsings κ, and for all permutations π of witnesses, $\eta\,\kappa \models \varphi$ if and only if $\eta\,\kappa\,\pi \models \varphi$.

Proof. Straightforward induction on the length of traces. □

Lemma 11. Let η be a trace, let φ be a usage automaton, and let K be the set of injective collapsing such that $|\kappa(\mathsf{Res}_d)| = |\varphi| + 1$. Then

$$(\forall \kappa \in K : \eta\kappa \models \varphi) \implies \eta \models \varphi.$$

Proof. We prove the contrapositive. Assume that $\eta \not\models \varphi$. Then, by Theorem 5, there exists $\sigma : var(\varphi) \to R(\eta, \varphi)$ such that $\eta \not\lhd A_\varphi(\sigma, res(\eta))$. Let $ran(\sigma) = \{r_1, \ldots, r_h\}$, where $r_1, \ldots, r_h$ are pairwise distinct, and let $r_{h+1}, \ldots, r_{|\varphi|}$ be distinct dynamic resources not occurring in η.

Let κ be the collapsing defined as follows, for all $r \in \mathsf{Res}_d$:

$$\kappa(r) = \begin{cases} \#_i & \text{if } r_i = r \text{ for some } i \in 1 \ldots |\varphi| \\ _ & \text{otherwise} \end{cases} \tag{A3}$$

Note that κ is injective on $\kappa^{-1}(\{\#_i\}_{i \in 1\ldots|\varphi|})$, and $\kappa(\mathsf{Res}_d) = \{\#_i\}_{i \in 1\ldots|\varphi|} \cup \{_\}$. Thus, we have that $\kappa \in K$.

By Definition 3, since $\eta \not\lhd A_\varphi(\sigma, res(\eta))$, then there is a run $q_0 \xrightarrow{\eta} q_k$ of $A_\varphi(\sigma, res(\eta))$ such that $q_k \in F$. Let $\sigma' = \sigma\kappa$. We will reconstruct an offending run of $A_\varphi(\sigma', res(\eta\kappa))$ on $\eta\kappa$.

We first prove an auxiliary result. For all guards g occurring in φ:

$$\sigma \models g \iff \sigma' \models g. \tag{A4}$$

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

We proceed by induction on the structure of g. The base case $g = \mathit{true}$ is trivial. The other base case is when g has the form $\xi_0 = \xi_1$. We only deal with the subcase $\xi_0, \xi_1 \in \mathsf{Var}$ (the other subcases are straightforward).

($\Longrightarrow$) We have that $\xi_0\sigma' = \xi_0\sigma\kappa = \xi_1\sigma\kappa = \xi_1\sigma'$.

($\Longleftarrow$) Let $\xi_0\sigma' = r = \xi_1\sigma'$. Since $ran(\sigma') = ran(\sigma)\kappa \subseteq \mathsf{Res}_s \cup \{\#_i\}_{i \in 1..|\varphi|} \cup \{_\}$, there are the following exhaustive cases:

— $r \in \mathsf{Res}_s$. The thesis follows by the fact that κ is the identity on static resources.
— $r = \#_i$, for some $i \in 1..|\varphi|$. By (A3), this implies that $\xi_0\sigma = r_i = \xi_1\sigma$.
— $r = _$. This case cannot occur, as (A3) would imply that $\xi_0\sigma, \xi_1\sigma \notin ran(\sigma)$ – a contradiction.

This concludes the proof of (A4) in the base case $\xi_0 = \xi_1$. The proof of the inductive cases $\neg g$ and $g_0 \wedge g_1$ is straightforward.

Let δ and δ_κ be the transition relations of $A_\varphi(\sigma, res(\eta))$ and $A_\varphi(\sigma', res(\eta\kappa))$, respectively, let δ^* and δ_κ^* be their (labelled) reflexive and transitive closures, and let X and X_κ be the sets of edges instantiated without completion.

We prove by induction on the length of η that:

$$\delta^*(\{q_0\}, \eta) \subseteq \delta_\kappa^*(\{q_0\}, \eta\kappa).$$

The base case $\eta = \varepsilon$ is trivial.

For the inductive case, let $\eta = \eta'\alpha(\vec{r})$. By the induction hypothesis, we have that $\delta^*(\{q_0\}, \eta') \subseteq \delta_\kappa^*(\{q_0\}, \eta'\kappa)$. Let now $q \in \delta^*(\{q_0\}, \eta')$, and assume that $q \xrightarrow{a} q'$. Only one of the following two cases may occur.

1. $q \xrightarrow{a} q' \in X$.

   Then, there exists an edge $q \xrightarrow{\alpha(\vec{\xi}):g} q'$ in φ such that

   $$\sigma \models g \qquad\qquad \alpha(\vec{\xi}\,\sigma) = a.$$

   By (A4), it follows that $\sigma' \models g$. Also, we have that $\alpha(\vec{\xi}\sigma') = a\kappa$. Therefore, the edge $q \xrightarrow{\alpha(\vec{\xi}):g} q'$ in φ can also be instantiated to the transition $q \xrightarrow{a\kappa} q' \in X_\kappa$, which proves the needed inclusion.

2. $q \xrightarrow{a} q' \in \mathit{complete}(X, res(\eta))$.

   Then, $q' = q$, and there exists no q'' such that $q \xrightarrow{a} q'' \in X$. We must prove that there also exists no q'' such that $q \xrightarrow{a\kappa} q'' \in X_\kappa$. By contradiction, assume that $q \xrightarrow{a\kappa} q'' \in X_\kappa$.

   Then, there would exist an edge $q \xrightarrow{\alpha(\vec{\xi}\,'):g'} q''$ in φ such that

   $$\sigma' \models g' \qquad\qquad \alpha(\vec{\xi}\,'\sigma') = a\kappa.$$

   By (A4), it follows that $\sigma \models g'$. Since $\alpha(\vec{\xi}\,'\sigma') = a\kappa$ and κ is bijective on $ran(\sigma)$, then we have $\alpha(\vec{\xi}\,'\sigma) = a$. Therefore, $q \xrightarrow{a} q'' \in X$ – which is the desired contradiction.

□

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

Lemma 12. Let η be a trace, let φ be a usage automaton, and let $\bar{K}$ be the set of injective collapsings such that $|\kappa(\mathsf{Res}_d)| \geqslant |\varphi| + 1$ for all $\kappa \in \bar{K}$. Then

$$\eta \models \varphi \implies (\forall \kappa \in \bar{K} : \eta\kappa \models \varphi).$$

Proof. We prove the contrapositive. Assume that there exists an injective collapsing $\kappa' \in \bar{K}$ such that $\eta\kappa' \not\models \varphi$. Let $\kappa'^{-1}(\{\#_i\}_{i\in\mathbb{N}}) = \{r_1, \ldots, r_h\}$. Since $\kappa' \in \bar{K}$, then $h \geqslant |\varphi|$. By Lemma 10, we can rename the witnesses used by κ', to obtain an injective collapsing κ onto $\{\#_1, \ldots, \#_h\}$ such that $\eta\kappa \not\models \varphi$. Let $\bar{\kappa}$ be the inverse of κ on $\kappa^{-1}(\{\#_i\}_{i\in\mathbb{N}})$, that is:

$$\bar{\kappa}(r) = \begin{cases} r_i & \text{if } r = \#_i \text{ and } \kappa(r_i) = \#_i \\ r & \text{otherwise.} \end{cases} \tag{A5}$$

Note that $\bar{\kappa}$ is well-defined, because κ is injective on $\kappa^{-1}(\{\#_i\}_{i\in\mathbb{N}})$.

By Theorem 5, there exists $\sigma' : var(\varphi) \to R(\eta\kappa, \varphi)$ such that

$$\eta\kappa \not\Vdash A_\varphi(\sigma', res(\eta\kappa)).$$

By Definition 3, there is a run $q_0 \xrightarrow{\eta\kappa} q_k$ of $A_\varphi(\sigma', res(\eta\kappa))$ such that $q_k \in F$. Let $\sigma = \sigma'\bar{\kappa}$. We will reconstruct an offending run of $A_\varphi(\sigma, res(\eta))$ on η.

The following auxiliary result will be needed.

$$\forall \xi \in \mathsf{Res}_s \cup \mathsf{Var} \ : \ \xi\sigma' = \xi\sigma\kappa. \tag{A6}$$

We now prove (A6). If $\xi \in \mathsf{Res}_s$, there is nothing to prove, since σ, σ' and κ are the identity on static resources. If $\xi \in \mathsf{Var}$, the only relevant case is when $\xi \in var(\varphi)$. By (A5), we have $\xi\sigma\kappa = \xi\sigma'\bar{\kappa}\kappa$. Since $ran(\sigma') \subseteq R(\eta\kappa, \varphi) = res(\eta\kappa) \cup res(\varphi) \cup \{\#_i\}_{i\in 1\ldots|\varphi|} \setminus \{_\}$, there are the following exhaustive cases:

— $\xi\sigma' \in \mathsf{Res}_s$. The thesis follows by the fact that $\bar{\kappa}, \kappa$ are the identity on static resources.
— $\xi\sigma' = \#_i$, for some $i \in 1\ldots|\varphi|$. By hypothesis, $\#_i \in \kappa(\mathsf{Res}_d)$, so there exists some $r \in \mathsf{Res}_d$ such that $\kappa(r) = \#_i$. By (A5), $\bar{\kappa}(\#_i) = r$. Then, $\xi\sigma'\bar{\kappa}\kappa = \#_i\bar{\kappa}\kappa = r\kappa = \#_i$.
— $\xi\sigma' = _$. This case cannot occur, since $_ \notin R(\eta\kappa, \varphi)$.

This concludes the proof of (A6).

We now prove a further auxiliary result. For all guards g occurring in φ:

$$\sigma \models g \iff \sigma' \models g. \tag{A7}$$

We proceed by induction on the structure of g. The base case $g = true$ is trivial. The other base case is when g has the form $\xi_0 = \xi_1$. We only deal with the subcase $\xi_0, \xi_1 \in \mathsf{Var}$ (the other subcases are straightforward).

($\Longrightarrow$) By (A6), we have that $\xi_0\sigma' = \xi_0\sigma\kappa = \xi_1\sigma\kappa = \xi_1\sigma'$.

($\Longleftarrow$) Let $\xi_0\sigma' = \xi_1\sigma' = r$. Then, $\xi_0\sigma = \xi_0\sigma'\bar{\kappa} = r\bar{\kappa} = \xi_1\sigma'\bar{\kappa} = \xi_1\sigma$.

This concludes the proof of (A7) in the base case $\xi_0 = \xi_1$. The proof of the inductive cases $\neg g$ and $g_0 \wedge g_1$ is straightforward.

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

Let δ and δ_κ be the transition relations of $A_\varphi(\sigma, res(\eta))$ and $A_\varphi(\sigma', res(\eta\kappa))$, respectively, let δ^* and δ_κ^* be their (labelled) reflexive and transitive closures, and let X and X_κ be the sets of edges instantiated without completion.

We prove by induction on the length of η that

$$\delta_\kappa^*(\{q_0\}, \eta\kappa) \subseteq \delta^*(\{q_0\}, \eta).$$

The base case $\eta = \varepsilon$ is trivial.

For the inductive case, let $\eta = \eta'\alpha(\vec{r})$. By the induction hypothesis, we have that $\delta_\kappa^*(\{q_0\}, \eta'\kappa) \subseteq \delta^*(\{q_0\}, \eta')$. Let now $q \in \delta_\kappa^*(\{q_0\}, \eta'\kappa)$, and assume that $q \xrightarrow{a\kappa} q'$. Only one of the following two cases may occur.

1. $q \xrightarrow{a\kappa} q' \in X_\kappa$.

   Then, there exists an edge $q \xrightarrow{\alpha(\vec{\xi}):g} q'$ in φ such that

   $$\sigma' \models g \qquad\qquad \alpha(\vec{\xi}\,\sigma') = a\kappa.$$

   By (A7), it follows that $\sigma \models g$. Also, we have that

   $$\alpha(\vec{\xi}\sigma) \;=\; \alpha(\vec{\xi}\sigma'\,\bar{\kappa}) \;=\; \alpha(\vec{\xi}\sigma')\,\bar{\kappa} \;=\; a\kappa\bar{\kappa} \;=\; a.$$

   Combining the above, we have proved that $\sigma \models g$ and $\alpha(\vec{\xi}\,\sigma) = a$.

   Therefore, the edge $q \xrightarrow{\alpha(\vec{\xi}):g} q'$ in φ can also be instantiated to the transition $q \xrightarrow{a} q' \in X$, which proves the needed inclusion.

2. $q \xrightarrow{a\kappa} q' \in complete(X_\kappa, res(\eta\kappa))$.

   Then, $q' = q$, and there exists no q'' such that $q \xrightarrow{a\kappa} q'' \in X_\kappa$. We must prove that there also exists no q'' such that $q \xrightarrow{a} q'' \in X$. By contradiction, assume that $q \xrightarrow{a} q'' \in X$.

   Then, there would exist an edge $q \xrightarrow{\alpha(\vec{\xi}\,'):g'} q''$ in φ such that

   $$\sigma \models g' \qquad\qquad \alpha(\vec{\xi}\,'\sigma) = a.$$

   By (A7), it follows that $\sigma' \models g'$. Furthermore, by (A6) we have that $\alpha(\vec{\xi}\,'\sigma') = \alpha(\vec{\xi}\,'\sigma\kappa) = a\kappa$. Therefore, $q \xrightarrow{a\kappa} q'' \in X_\kappa$ – which is the desired contradiction.

□

Proofs for Section 6

Lemma A.1. For all U, $\mathcal{W}$, Θ, and for all substitutions $\sigma : \mathsf{Nam} \to \mathsf{Res}$:

$$\mathsf{B}_{\mathcal{W}}(U\sigma)_\Theta \;=\; \mathsf{B}_{\mathcal{W}}(U)_{\Theta\sigma}\,\sigma.$$

Proof. Straightforward by induction on the structure of U. □

Lemma 25. For all $\mathcal{W}, U, U', h'$ and Θ, there exists a coherent renaming ζ of $\mathsf{B}_{\mathcal{W}}(U\{U'/h'\})_\Theta$ such that ζ is the identity on the BPA variables in Θ, and

$$\mathsf{B}_{\mathcal{W}}(U\{U'/h'\})_\Theta\,\zeta \;=\; \mathsf{B}_{\mathcal{W}}(U)_{\Theta\{\mathsf{B}_{\mathcal{W}}(U')_\Theta/h'\}}$$

where $\mathsf{B}_{\mathcal{W}}(U)_{\Theta\{\langle p',\Delta'\rangle/h'\}}$ stands for $\langle p, \Delta + \Delta'\rangle$, if $\mathsf{B}_{\mathcal{W}}(U)_{\Theta\{p'/h'\}} = \langle p, \Delta\rangle$.

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

Proof. By induction on the structure of U. The base cases $U = \varepsilon, \alpha(\vec{r}), h$ and the inductive cases $U = U_0 \cdot U_1$, $U = U_0 + U_1$, and $U = \nu n.\, U''$ are straightforward. Consider now the case $U = \mu h.\, U''$. If $h = h'$, trivial. Otherwise

$$\mathsf{B}_{\mathcal{W}}(U\{U'/h'\})_{\Theta} = \langle X, \{X \triangleq p\} + \Delta\rangle \quad \text{where } \langle p, \Delta\rangle = \mathsf{B}_{\mathcal{W}}(U''\{U'/h'\})_{\Theta\{X/h\}}$$
$$\mathsf{B}_{\mathcal{W}}(U)_{\Theta\{\mathsf{B}_{\mathcal{W}}(U')_{\Theta}/h'\}} = \langle X, \{X \triangleq p'\} + \Delta'\rangle \quad \text{where } \langle p', \Delta'\rangle = \mathsf{B}_{\mathcal{W}}(U'')_{\Theta\{X/h, \mathsf{B}_{\mathcal{W}}(U')_{\Theta}/h'\}}$$

(note that we can choose the fresh variable X in both the equations). By the induction hypothesis, there exists a renaming ζ coherent with $\mathsf{B}_{\mathcal{W}}(U''\{U'/h'\})_{\Theta\{X/h\}}$, such that ζ leaves the variables in $\Theta\{X/h\}$ untouched (thus $X\zeta = X$), and

$$\mathsf{B}_{\mathcal{W}}(U''\{U'/h'\})_{\Theta\{X/h\}}\,\zeta \;=\; \mathsf{B}_{\mathcal{W}}(U'')_{\Theta\{X/h, \mathsf{B}_{\mathcal{W}}(U')_{\Theta}/h'\}}$$

i.e. $p\zeta = p'$ and $\Delta\zeta = \Delta'$. Therefore, it follows that

$$\begin{aligned}\mathsf{B}_{\mathcal{W}}(U\{U'/h'\})_{\Theta}\,\zeta &= \langle X, \{X \triangleq p\} + \Delta\rangle\,\zeta \\ &= \langle X\zeta, \{X\zeta \triangleq p\zeta\} + \Delta\zeta\rangle \\ &= \langle X, \{X \triangleq p'\} + \Delta'\rangle \\ &= \mathsf{B}_{\mathcal{W}}(U)_{\Theta\{\mathsf{B}_{\mathcal{W}}(U')_{\Theta}/h'\}}\end{aligned}$$

which concludes the proof. □

The following lemma states some well-known properties of simulation and bisimulation.

Lemma A.2. The similarity relation $\succ$ between BPAs is a simulation, and a preorder. The bisimilarity relation $\sim$ is a bisimulation, and an equivalence relation. Moreover, for all BPAs P, Q, R and substitutions σ:

$$\begin{aligned}&P \succ Q \implies P \cdot R \succ Q \cdot R \;\text{ and }\; R \cdot P \succ R \cdot Q \\ &P \sim Q \implies P \succ Q \\ &P \sim Q \implies P\sigma \sim Q\sigma \\ &P \sim Q \implies P \cdot R \sim Q \cdot R.\end{aligned}$$

Lemma 26. For all usages U, and for all $\mathcal{W}, \mathcal{W}', \Theta$:

$$\mathcal{W} \supseteq \mathcal{W}' \implies \mathsf{B}_{\mathcal{W}}(U)_{\Theta} \succ \mathsf{B}_{\mathcal{W}'}(U)_{\Theta}.$$

Proof. By induction on the size of U. The only non-trivial case is $U = \nu n.U'$, for which we have the following:

$$\begin{aligned}\mathsf{B}_{\mathcal{W}}(\nu n.\, U') &= new(_) \cdot \mathsf{B}_{\mathcal{W}}(U'\{_/n\}) + \sum_{\#_i \in \mathcal{W}} new(\#_i) \cdot \mathsf{B}_{\mathcal{W}}(U'\{\#_i/n\}) \\ &\succ new(_) \cdot \mathsf{B}_{\mathcal{W}'}(U'\{_/n\}) + \sum_{\#_i \in \mathcal{W}'} new(\#_i) \cdot \mathsf{B}_{\mathcal{W}'}(U'\{\#_i/n\}) \\ &= \mathsf{B}_{\mathcal{W}'}(\nu n.\, U')\end{aligned}$$

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

where in the second step we have applied Lemma A.2, and the induction hypothesis $1 + |\mathcal{W} \setminus \mathcal{W}'|$ times. □

Lemma 27. Let U be a closed usage, let $\mathcal{R} \subset \mathsf{Res}_d$ be a finite set of resources, let $\mathcal{W}$ be a finite set of witnesses, and let κ be an injective collapsing such that $\mathcal{W} \supseteq \kappa(\mathsf{Res}_d \setminus \mathcal{R}) \setminus \{_\}$. Then, for all $a, U', \mathcal{R}'$ such that

$$U, \mathcal{R} \xrightarrow{a} U', \mathcal{R}'$$

the following holds:

$$\forall P \succ \mathsf{B}_{\mathcal{W}}(U)\kappa \;\; : \;\; \exists \mathcal{W}' \supseteq \kappa(\mathsf{Res}_d \setminus \mathcal{R}'),\ \exists P' \succ \mathsf{B}_{\mathcal{W}'}(U')\kappa \;\; : \;\; P \xrightarrow{a\kappa} P'.$$

Proof. By induction on the depth of the derivation of $U, \mathcal{R} \xrightarrow{a} U', \mathcal{R}'$. In all the cases below, we will fulfill the statement by choosing $\mathcal{W}' = \mathcal{W} \setminus \kappa(\mathcal{R}' \setminus \mathcal{R})$. Indeed, we have that

$$\begin{aligned}
\mathcal{W}' &= \mathcal{W} \setminus \kappa(\mathcal{R}' \setminus \mathcal{R}) \\
&\supseteq \kappa(\mathsf{Res}_d \setminus \mathcal{R}) \setminus (\kappa(\mathcal{R}' \setminus \mathcal{R}) \cup \{_\}) && \text{as } \mathcal{W} \supseteq \kappa(\mathsf{Res}_d \setminus \mathcal{R}) \setminus \{_\} \\
&\supseteq \kappa(\mathsf{Res}_d) \setminus (\kappa(\mathcal{R}) \cup \kappa(\mathcal{R}' \setminus \mathcal{R}) \cup \{_\}) \\
&= \kappa(\mathsf{Res}_d) \setminus (\kappa(\mathcal{R}') \cup \{_\}) && \text{as } \mathcal{R}' \supseteq \mathcal{R} \\
&= \kappa(\mathsf{Res}_d \setminus \mathcal{R}') \setminus \{_\} && \text{as } \kappa \text{ is an injective collapsing.}
\end{aligned}$$

Base cases

— $\alpha(\vec{r}), \mathcal{R} \xrightarrow{\alpha(\vec{r})} \varepsilon, \mathcal{R}$.

By Definitions 19 and 20:

$$P \succ \mathsf{B}_{\mathcal{W}}(\alpha(\vec{r}))\kappa = \langle \alpha(\vec{r}), \varnothing \rangle \kappa \xrightarrow{\alpha(\vec{r})\kappa} \langle 0, \varnothing \rangle = \mathsf{B}_{\mathcal{W}}(\varepsilon)\kappa$$

from which the thesis follows directly.

— $\nu n.\, U'', \mathcal{R} \xrightarrow{new(r)} U''\{r/n\}, \mathcal{R} \cup \{r\}$, with $r \notin \mathcal{R}$.

$$P \succ \mathsf{B}_{\mathcal{W}}(\nu n.U'')\kappa = new(_) \cdot \mathsf{B}_{\mathcal{W}}(U''\{_/n\})\kappa + \sum_{w \in \mathcal{W}} new(w) \cdot \mathsf{B}_{\mathcal{W} \setminus \{w\}}(U''\{w/n\})\kappa.$$

Two cases are possible, depending on whether $\kappa(r) = _$ or $\kappa(r) \in \{\#_i\}_{i \in \mathbb{N}}$.

In the first case, $\kappa(r) = _$, since

$$P \succ \mathsf{B}_{\mathcal{W}}(\nu n.U'')\kappa \xrightarrow{new(_)} \mathsf{B}_{\mathcal{W}}(U''\{_/n\})\kappa$$

then there exists $P' \succ \mathsf{B}_{\mathcal{W}}(U''\{_/n\})\kappa$ such that $P \xrightarrow{new(_)} P'$.

Combining the above:

$$\begin{aligned}
P' &\succ \mathsf{B}_{\mathcal{W}}(U''\{_/n\})\kappa \\
&= \mathsf{B}_{\mathcal{W}}(U'')\{_/n\}\kappa && \text{by Lemma A.1} \\
&= \mathsf{B}_{\mathcal{W}}(U'')\{r/n\}\kappa && \text{as } \kappa(r) = _ \\
&= \mathsf{B}_{\mathcal{W}}(U''\{r/n\})\kappa && \text{by Lemma A.1} \\
&= \mathsf{B}_{\mathcal{W}'}(U')\kappa && \text{as } \mathcal{W}' = \mathcal{W} \setminus \{\kappa(r)\} = \mathcal{W}.
\end{aligned}$$

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

In the second case, we have $\kappa(r) = \#_j$, for some j. Since $r \in \mathsf{Res}_d \setminus \mathcal{R}$, then $\kappa(r) \in \kappa(\mathsf{Res}_d \setminus \mathcal{R}) \subseteq \mathcal{W}$. Since

$$\mathsf{B}_{\mathcal{W}}(\nu n.U'')\kappa \xrightarrow{new(\#_j)} \mathsf{B}_{\mathcal{W}\setminus\{\#_j\}}(U''\{\#_j/n\})\kappa$$

then there exists $P' \succ \mathsf{B}_{\mathcal{W}\setminus\{\#_j\}}(U''\{\#_j/n\})\kappa$ such that $P \xrightarrow{new(\#_j)} P'$.
Combining the above:

$$\begin{aligned} P' &\succ \mathsf{B}_{\mathcal{W}\setminus\{\#_j\}}(U''\{\#_j/n\})\kappa \\ &= \mathsf{B}_{\mathcal{W}\setminus\{\#_j\}}(U'')\{\#_j/n\}\kappa && \text{by Lemma A.1} \\ &= \mathsf{B}_{\mathcal{W}\setminus\{\#_j\}}(U'')\{r/n\}\kappa && \text{as } \kappa(r) = \#_j \\ &= \mathsf{B}_{\mathcal{W}\setminus\{\#_j\}}(U''\{r/n\})\kappa && \text{by Lemma A.1} \\ &= \mathsf{B}_{\mathcal{W}'}(U')\kappa && \text{as } \mathcal{W}' = \mathcal{W} \setminus \{\kappa(r)\} = \mathcal{W} \setminus \{\#_j\} \end{aligned}$$

— $U = \varepsilon \cdot U_1, \mathcal{R} \xrightarrow{\varepsilon} U_1, \mathcal{R}$.

$$P \succ \mathsf{B}_{\mathcal{W}}(\varepsilon \cdot U_1)\kappa \;=\; \mathsf{B}_{\mathcal{W}}(\varepsilon)\kappa \cdot \mathsf{B}_{\mathcal{W}}(U_1)\kappa \;=\; \langle 0, \varnothing\rangle \cdot \mathsf{B}_{\mathcal{W}}(U_1)\kappa \xrightarrow{\varepsilon} \mathsf{B}_{\mathcal{W}}(U_1)\kappa$$

from which the thesis follows directly.

— $\mu h.\, U'', \mathcal{R} \xrightarrow{\varepsilon} U''\{U/h\}, \mathcal{R}$

$$\mathsf{B}_{\mathcal{W}}(\mu h.\, U'')\kappa = \langle X, \{X \triangleq p\} + \Delta\rangle\kappa \quad \text{where } \langle p, \Delta\rangle = \mathsf{B}_{\mathcal{W}}(U'')_{\{X/h\}}.$$

Let $P \succ \mathsf{B}_{\mathcal{W}}(\mu h.\, U'')\kappa$. Since $\langle X, \{X \triangleq p\} + \Delta\rangle\kappa \xrightarrow{\varepsilon} \langle p, \{X \triangleq p\} + \Delta\rangle\kappa$, then there exist P' such that $P \xrightarrow{\varepsilon} P' \succ \langle p, \{X \triangleq p\} + \Delta\rangle\kappa$.
By Lemma 25, it follows that

$$\mathsf{B}_{\mathcal{W}}(U''\{U/h\})\zeta \;=\; \mathsf{B}_{\mathcal{W}}(U'')_{\{\mathsf{B}_{\mathcal{W}}(U)/h\}} \;=\; \langle p, \{X \triangleq p\} + \Delta\rangle$$

for some renaming ζ coherent with $\mathsf{B}_{\mathcal{W}}(U''\{U/h\})$.
Then, by Lemma 24:

$$\mathsf{B}_{\mathcal{W}}(U''\{U/h\})\zeta \;\sim\; \mathsf{B}_{\mathcal{W}}(U''\{U/h\}) \;=\; \mathsf{B}_{\mathcal{W}}(U').$$

Hence, since a bisimulation is also a simulation,

$$\mathsf{B}_{\mathcal{W}}(U''\{U/h\})\zeta \;\succ\; \mathsf{B}_{\mathcal{W}}(U').$$

Combining the above, we conclude that

$$P' \;\succ\; \langle p, \{X \triangleq p\} + \Delta\rangle\kappa \;\succ\; \mathsf{B}_{\mathcal{W}}(U')\kappa.$$

Inductive cases

— $U_0 + U_1, \mathcal{R} \xrightarrow{a} U_0', \mathcal{R}'$, with $U_0, \mathcal{R} \xrightarrow{a} U_0', \mathcal{R}'$ (the symmetric case is similar).
By Definition 20, we have

$$P \;\succ\; \mathsf{B}_{\mathcal{W}}(U_0 + U_1)\kappa \;=\; \mathsf{B}_{\mathcal{W}}(U_0)\kappa + \mathsf{B}_{\mathcal{W}}(U_1)\kappa \;\succ\; \mathsf{B}_{\mathcal{W}}(U_0)\kappa.$$

Since $P \succ \mathsf{B}_{\mathcal{W}}(U_0)\kappa$ and $U_0, \mathcal{R} \xrightarrow{a} U_0', \mathcal{R}'$, by the induction hypothesis it follows that there exists P' and $\mathcal{W}' = \mathcal{W} \setminus \kappa(\mathcal{R}' \setminus \mathcal{R})$ such that

$$P \xrightarrow{a\kappa} P' \;\succ\; \mathsf{B}_{\mathcal{W}'}(U_0')\kappa$$

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

and the thesis follows immediately.

— $U_0 \cdot U_1, \mathcal{R} \xrightarrow{a} U_0' \cdot U_1, \mathcal{R}'$, with $U_0, \mathcal{R} \xrightarrow{a} U_0', \mathcal{R}'$.
By Definition 20, we have the following:

$$P \succ \mathsf{B}_{\mathcal{W}}(U)\kappa = \mathsf{B}_{\mathcal{W}}(U_0 \cdot U_1)\kappa = \mathsf{B}_{\mathcal{W}}(U_0)\kappa \cdot \mathsf{B}_{\mathcal{W}}(U_1)\kappa.$$

Let $Q = \mathsf{B}_{\mathcal{W}}(U_0)\kappa$. By the induction hypothesis, there exist Q' and $\mathcal{W}' = \mathcal{W} \setminus \kappa(\mathcal{R}' \setminus \mathcal{R})$ such that

(A8) $$Q \xrightarrow{a\kappa} Q' \succ \mathsf{B}_{\mathcal{W}'}(U_0')\kappa.$$

Since $P \succ Q \cdot \mathsf{B}_{\mathcal{W}}(U_1)\kappa$ and $Q \xrightarrow{a\kappa} Q'$, then there exists P' such that

(A9) $$P \xrightarrow{a\kappa} P' \succ Q' \cdot \mathsf{B}_{\mathcal{W}}(U_1)\kappa.$$

Therefore, we have that

$$\begin{aligned} P' &\succ Q' \cdot \mathsf{B}_{\mathcal{W}}(U_1)\kappa && \text{by (A9)} \\ &\succ Q' \cdot \mathsf{B}_{\mathcal{W}'}(U_1)\kappa && \text{by Lemma 26 } (\mathcal{W}' \subseteq \mathcal{W}) \\ &\succ \mathsf{B}_{\mathcal{W}'}(U_0')\kappa \cdot \mathsf{B}_{\mathcal{W}'}(U_1)\kappa && \text{by (A8)} \\ &= \mathsf{B}_{\mathcal{W}'}(U')\kappa. \end{aligned}$$

This concludes the proof of the lemma. □

Lemma 29. Let U be a closed usage, let $\mathcal{R} \subseteq \mathsf{Res}_d$ be a finite set of resources, let $\mathcal{W}$ be a finite set of witnesses, and let κ be a collapsing. Assume that

$$P \prec \mathsf{B}_{\mathcal{W}}(U)\kappa \qquad P \xrightarrow{a} P'$$

and, if $a = new(\#_i)$, then there exists some $r \in \mathsf{Res}_d \setminus \mathcal{R}$ such that $\kappa(r) = \#_i$.

Then, there exist $U', \mathcal{R}'$, and b such that

$$U, \mathcal{R} \xrightarrow{b} U', \mathcal{R}' \qquad a = b\,\kappa \qquad P' \prec \mathsf{B}_{\mathcal{W}}(U')\kappa.$$

Proof. By induction on the structure of U. We have the following cases:

— $U = \varepsilon$.
In this case, the conditions $P \prec \mathsf{B}_{\mathcal{W}}(U)\kappa = 0$ and $P \xrightarrow{a} P'$ cannot be met together, since 0 cannot take any transitions.

— $U = \alpha(\vec{r})$.
Let $P \prec \mathsf{B}_{\mathcal{W}}(U)\kappa = \alpha(\vec{r})\kappa$. Since $\alpha(\vec{r})\kappa$ can only take the transition

$$\alpha(\vec{r})\kappa \xrightarrow{\alpha(\vec{r})\kappa} 0$$

then it must be the case that $a = \alpha(\vec{r})\kappa$ and $P' \prec 0$. The thesis holds with $U' = \varepsilon$, $b = \alpha(\vec{r})$, $\mathcal{R}' = \mathcal{R}$, which give $\mathsf{B}_{\mathcal{W}}(U')\kappa = 0 \succ P'$.

— $U = \nu n.\, U''$.

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

Let $P \prec \mathsf{B}_{\mathcal{W}}(\nu n.\, U'')\kappa$. Then, by Definition 20:

$$P \prec new(_) \cdot \mathsf{B}_{\mathcal{W}}(U''\{_/n\})\kappa + \sum_{w \in \mathcal{W}} new(w) \cdot \mathsf{B}_{\mathcal{W}\setminus\{w\}}(U''\{w/n\})\kappa.$$

Note that, since the set of witnesses $\mathcal{W}$ is finite, there is a finite set of processes Q such that $\mathsf{B}_{\mathcal{W}}(\nu n.\, U'')\kappa \xrightarrow{a} Q$. Also, any such transitions must fall within one of the following two cases:

1. $Q = \mathsf{B}_{\mathcal{W}\setminus\{\#_i\}}(U''\{\#_i/n\})\kappa$, for some $\#_i \in \mathcal{W}$, and $a = new(\#_i)$.
2. $Q = \mathsf{B}_{\mathcal{W}}(U''\{_/n\})\kappa$, and $a = new(_)$.

In the case (1), we have that $P' \prec \mathsf{B}_{\mathcal{W}\setminus\{\#_i\}}(U''\{\#_i/n\})\kappa$. By hypothesis, we can choose a fresh $r \in \mathsf{Res}_d \setminus \mathcal{R}$ such that $\kappa(r) = \#_i$.
We have the following transition:

$$\nu n.U'', \mathcal{R} \xrightarrow{new(r)} U''\{r/n\}, \mathcal{R} \cup \{r\}.$$

Now let $U' = U''\{r/n\}$, let $b = \varepsilon$, and let $\mathcal{R}' = \mathcal{R} \cup \{r\}$. We have that

$$\begin{aligned}
P' &\prec \mathsf{B}_{\mathcal{W}\setminus\{\#_i\}}(U''\{\#_i/n\})\kappa \\
&= \mathsf{B}_{\mathcal{W}\setminus\{\#_i\}}(U'')\{\#_i/n\}\kappa && \text{by Lemma A.1} \\
&= \mathsf{B}_{\mathcal{W}\setminus\{\#_i\}}(U'')\{r/n\}\kappa && \text{as } \kappa(r) = \#_i \\
&= \mathsf{B}_{\mathcal{W}\setminus\{\#_i\}}(U''\{r/n\})\kappa && \text{by Lemma A.1} \\
&\prec \mathsf{B}_{\mathcal{W}}(U')\kappa && \text{by Lemma 26}
\end{aligned}$$

Case (2) is similar. We have $P' \prec \mathsf{B}_{\mathcal{W}}(U''\{_/n\})\kappa$, and we choose a fresh $r \in \mathsf{Res}_d \setminus \mathcal{R}$ such that $\kappa(r) = _$ (this is always possible, because $\mathcal{R}$ is finite). Then, we let $U' = U''\{r/n\}$ and $\mathcal{R}' = \mathcal{R} \cup \{r\}$.

— $U = U_0 \cdot U_1$.
Let $P \prec \mathsf{B}_{\mathcal{W}}(U_0 \cdot U_1)\kappa$. Then, by Definition 20:

$$P \prec \mathsf{B}_{\mathcal{W}}(U_0)\kappa \cdot \mathsf{B}_{\mathcal{W}}(U_1)\kappa.$$

Now, we have two subcases depending on the structure of U_0.

1. $U_0 = \varepsilon$. In this case $\mathsf{B}_{\mathcal{W}}(U_0)\kappa = 0$, hence we have the transition

$$0 \cdot \mathsf{B}_{\mathcal{W}}(U_1)\kappa \xrightarrow{\varepsilon} \mathsf{B}_{\mathcal{W}}(U_1)\kappa.$$

So, it must be $a = \varepsilon$ and $P' \prec \mathsf{B}_{\mathcal{W}}(U_1)\kappa$. The thesis follows directly, by choosing $U' = U_1$, $b = a$, and $\mathcal{R}' = \mathcal{R}$.

2. $U_0 \neq \varepsilon$. Any transition from $\mathsf{B}_{\mathcal{W}}(U)\kappa$ has been derived as follows:

$$\frac{\mathsf{B}_{\mathcal{W}}(U_0)\kappa \xrightarrow{a} Q'}{\mathsf{B}_{\mathcal{W}}(U_0)\kappa \cdot \mathsf{B}_{\mathcal{W}}(U_1)\kappa \xrightarrow{a} Q' \cdot \mathsf{B}_{\mathcal{W}}(U_1)\kappa}.$$

So, we have $P' \prec Q' \cdot \mathsf{B}_{\mathcal{W}}(U_1)\kappa$. Since $\mathsf{B}_{\mathcal{W}}(U_0)\kappa \prec \mathsf{B}_{\mathcal{W}}(U_0)\kappa$, by the induction hypothesis there exist $U_0', \mathcal{R}'$ and b such that

$$U_0, \mathcal{R} \xrightarrow{b} U_0', \mathcal{R}' \qquad a = b\,\kappa \qquad Q' \prec \mathsf{B}_{\mathcal{W}}(U_0')\kappa.$$

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

Let $U' = U'_0 \cdot U_1$. We have that

$$\frac{U_0, \mathcal{R} \xrightarrow{b} U'_0, \mathcal{R}'}{U_0 \cdot U_1, \mathcal{R} \xrightarrow{b} U'_0, \cdot U_1, \mathcal{R}'}.$$

Also, we have

$$P' \prec Q' \cdot \mathsf{B}_{\mathcal{W}}(U_1)\kappa \prec \mathsf{B}_{\mathcal{W}}(U'_0)\kappa \cdot \mathsf{B}_{\mathcal{W}}(U_1)\kappa = \mathsf{B}_{\mathcal{W}}(U')\kappa.$$

— $U = U_0 + U_1$.
Let $P \prec \mathsf{B}_{\mathcal{W}}(U_0 + U_1)\kappa$. Then, by Definition 20:

$$P \prec \mathsf{B}_{\mathcal{W}}(U_0)\kappa + \mathsf{B}_{\mathcal{W}}(U_1)\kappa.$$

Only two transitions are possible. Consider now the case where the left-hand side moves (the other case is symmetric). We have that

$$\frac{\mathsf{B}_{\mathcal{W}}(U_0)\kappa \xrightarrow{a} Q'}{\mathsf{B}_{\mathcal{W}}(U_0)\kappa + \mathsf{B}_{\mathcal{W}}(U_1)\kappa \xrightarrow{a} Q'}.$$

So, it must be the case that $P \xrightarrow{a} P' \prec Q'$. Since $\mathsf{B}_{\mathcal{W}}(U_0)\kappa \xrightarrow{a} Q'$, by the induction hypothesis there exist $U'_0, \mathcal{R}'$ and b such that

$$U_0, \mathcal{R} \xrightarrow{b} U'_0, \mathcal{R}' \qquad a = b\,\kappa \qquad Q' \prec \mathsf{B}_{\mathcal{W}}(U'_0)\kappa.$$

Then, the thesis follows by $P' \prec Q' \prec \mathsf{B}_{\mathcal{W}}(U'_0)\kappa$ and

$$\frac{U_0, \mathcal{R} \xrightarrow{b} U'_0, \mathcal{R}'}{U_0 + U_1, \mathcal{R} \xrightarrow{b} U'_0, \mathcal{R}'}.$$

The other case is similar.

— $U = \mu h.\, U''$.
Let $P \prec \mathsf{B}_{\mathcal{W}}(\mu h.\, U'')\kappa$. Then, by Definition 20:

$$P \prec \langle X, \Delta + \{X \triangleq p\}\rangle\kappa \quad \text{where } \mathsf{B}_{\mathcal{W}}(U'')_{\Theta\{X/h\}} = \langle p, \Delta\rangle.$$

By construction, there is only one transition from $\langle X, \{X \triangleq p\} + \Delta\rangle\kappa$:

$$\langle X, \{X \triangleq p\} + \Delta\rangle\kappa \xrightarrow{\varepsilon} \langle p, \{X \triangleq p\} + \Delta\rangle\kappa.$$

Then, we must have $P' \prec \langle p, \Delta + \{X \triangleq p\}\rangle\kappa$, and $a = \varepsilon$.
We have that

$$\mu h.\, U'', \mathcal{R} \xrightarrow{\varepsilon} U''\{U/h\}, \mathcal{R}.$$

So, let $U' = U''\{U/h\}$, let $b = \varepsilon$, and let $\mathcal{R}' = \mathcal{R}$.
By Lemma 25, there is some coherent renaming ζ such that

$$\mathsf{B}_{\mathcal{W}}(U''\{U/h\})\zeta = \mathsf{B}_{\mathcal{W}}(U'')_{\{\mathsf{B}_{\mathcal{W}}(U)/h\}}.$$

By Lemma 24, $\mathsf{B}_{\mathcal{W}}(U''\{U/h\}) \sim \mathsf{B}_{\mathcal{W}}(U''\{U/h\})\zeta$.

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

Combining the above:

$$\begin{aligned} P' &\prec \langle p, \{X \triangleq p\} + \Delta\rangle\kappa \\ &= \langle p, \{X \triangleq p\} + \Delta + \{X \triangleq p\} + \Delta\rangle\kappa \\ &= \mathsf{B}_{\mathcal{W}}(U'')_{\{\langle X, \{X \triangleq p\} + \Delta\rangle/h\}}\kappa \\ &= \mathsf{B}_{\mathcal{W}}(U'')_{\{\mathsf{B}_{\mathcal{W}}(U)/h\}}\kappa \\ &\sim \mathsf{B}_{\mathcal{W}}(U''\{U/h\})\kappa \\ &= \mathsf{B}_{\mathcal{W}}(U')\kappa. \end{aligned}$$

□

Proofs for Section 7

Lemma 41. Let η be a trace (without redundant framings), let φ be a usage automaton, let $\mathcal{R} \subseteq \mathsf{Res}$, and let $\sigma : var(\varphi) \to \mathcal{R} \setminus \{_\}$. Then

$$\begin{array}{lll} (a) & \eta^{-[]} \lhd A_{\varphi}(\sigma, \mathcal{R}) & \Longrightarrow \quad \eta \lhd A_{\varphi_{[]}}(\sigma, \mathcal{R}) \\ (b) & \eta \lhd A_{\varphi_{[]}}(\sigma, \mathcal{R}) & \Longrightarrow \quad \eta^{-[]} \lhd A_{\varphi}(\sigma, \mathcal{R}) \ \text{ or } \ \varphi \notin act(\eta). \end{array}$$

Proof. We first introduce some notation. We denote by Q and $\dot{Q}$ the states in the first and in the second layer of $A_{\varphi_{[]}}(\sigma, \mathcal{R})$, respectively. We denote with F and $\dot{F}$ the final states of $A_{\varphi}(\sigma, \mathcal{R})$ and $A_{\varphi_{[]}}(\sigma, \mathcal{R})$, respectively. Also, for each state q in $A_{\varphi}(\sigma, \mathcal{R})$, we denote with $q@Q$ and $q@\dot{Q}$ the projections of q on the first and on second layer of $A_{\varphi_{[]}}(\sigma, \mathcal{R})$.

For (a), we prove the contrapositive. Let $\eta = \beta_1 \cdots \beta_n$, and assume that $\eta \not\lhd A_{\varphi_{[]}}(\sigma, \mathcal{R})$. By Definition 3, there exists an offending run:

$$q^{(0)} \xrightarrow{\eta} q^{(n)} \in \dot{F} \qquad \text{run of } A_{\varphi_{[]}}(\sigma, \mathcal{R}) \text{ on } \eta.$$

We shall now construct an offending run:

$$\bar{q}^{(0)} \xrightarrow{\eta^{-[]}} \bar{q}^{(\bar{n})} \in F \qquad \text{run of } A_{\varphi}(\sigma, \mathcal{R}) \text{ on } \eta^{-[]}.$$

At the first step of this construction, let $\bar{q}^{(0)} = q^{(0)}$. For the remaining steps, we shall scan the trace η with the index k, the trace $\eta^{-[]}$ with $\bar{k}$, and at each step we shall define $\bar{q}^{(\bar{k}+1)}$. At the kth step, if β_k is a framing event then we just let $\bar{q}^{(\bar{k}+1)} = \bar{q}^{(\bar{k})}$, thanks to the self-loop of $A_{\varphi}(\sigma, \mathcal{R})$ on β_k.

For each transition $q^{(k)} \xrightarrow{\beta_k} q^{(k+1)}$ with $\beta_k \in \mathsf{Ev}$, we let $\bar{q}^{(\bar{k}+1)} = q^{(k+1)}@Q$. By Definitions 3 and 40, $A_{\varphi}(\sigma, \mathcal{R})$ has the transition $\bar{q}^{(\bar{k})} \xrightarrow{\beta_{\bar{k}}} \bar{q}^{(\bar{k}+1)}$.

For (b), we prove the contrapositive. Assume that $\eta^{-[]} \not\lhd A_{\varphi}(\sigma, \mathcal{R})$ and $\varphi \in act(\eta')$. By Definition 3, there is an offending run:

$$\bar{q}^{(0)} \xrightarrow{\eta^{-[]}} \bar{q}^{(\bar{n})} \in F \qquad \text{run of } A_{\varphi}(\sigma, \mathcal{R}) \text{ on } \eta^{-[]}.$$

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

We will show how to construct an offending run:

$$q^{(0)} \xrightarrow{\eta} q^{(n+1)} \in \dot{F} \qquad\qquad \text{run of } A_{\varphi_{[]}}(\sigma,\mathcal{R}) \text{ on } \eta.$$

At the first step of our construction, let $q^{(0)} = \bar{q}^{(0)}$. For the remaining steps, we shall scan the trace η with the index k, the trace $\eta'^{-[]}$ with $\bar{k}$, and at each step we shall define $q^{(k+1)}$. For the kth step, there are the following cases.

— If $\beta_k = [_\varphi$, then $q^{(k)} \in Q$, because η has no redundant framings. Let $q^{(k+1)} = q^{(k)}@\dot{Q}$. By Definitions 40 and 3, $A_{\varphi_{[]}}(\sigma,\mathcal{R})$ has a transition $q^{(k)} \xrightarrow{[_\varphi} q^{(k+1)}$.

— If $\beta_k =]_\varphi$, then $q^{(k)} \in \dot{Q}$, because η has no redundant framings. Moreover, $\bar{q}^{(\bar{k})} \notin F$, otherwise we have found a (strict) prefix of η that is offending for $A_{\varphi_{[]}}(\sigma,\mathcal{R})$. Let $q^{(k+1)} = q^{(k)}@Q$. Then, $A_{\varphi_{[]}}(\sigma,\mathcal{R})$ can take the transition $q^{(k)} \xrightarrow{]_\varphi} q^{(k+1)}$.

— If $\beta_k = [_{\varphi'}$ or $\beta_k =]_{\varphi'}$ for some $\varphi' \neq \varphi$, then let $q^{(k+1)} = q^{(k)}$.

— Otherwise, if $\beta_k \in \mathsf{Ev}$, there are the following two subcases.

  – If $q^{(k)} \in Q$, then choose $q^{(k+1)} = \bar{q}^{(\bar{k}+1)}$.

  – If $q^{(k)} \in \dot{Q}$, then $\bar{q}^{(\bar{k})} \notin F$: otherwise, we have found a (strict) prefix of η that is offending for $A_{\varphi_{[]}}(\sigma,\mathcal{R})$. So, $A_\varphi(\sigma,\mathcal{R})$ has a transition $\bar{q}^{(\bar{k})} \xrightarrow{\beta_{\bar{k}}} \bar{q}^{(\bar{k}+1)}$. Let $q^{(k+1)} = \bar{q}^{(\bar{k}+1)}@\dot{Q}$. Then, $A_{\varphi_{[]}}(\sigma,\mathcal{R})$ has the transition $q^{(k)} \xrightarrow{\beta_k} q^{(k+1)}$.

We have then constructed a run $q^{(0)} \xrightarrow{\eta} q^{(n+1)}$ of $A_{\varphi_{[]}}(\sigma,\mathcal{R})$ on η. Since $\varphi \in act(\eta)$, then in η the scope of φ has been entered but not exited. Thus, $q^{(n+1)}$ is in the second layer of $A_{\varphi_{[]}}(\sigma,\mathcal{R})$. The state $q^{(n+1)}$ is offending, because $\bar{q}_k \in F$. This yields the expected contradiction. □

Lemma 42. For all traces η (without redundant framings) and for all φ:

$$\eta \not\models \varphi_{[]} \implies \varphi \in act(\eta).$$

Proof. Straightforward by induction on the length of η. □

Lemma 43. A trace η (without redundant framings) is valid if and only if

$$\forall \varphi \in \Phi(\eta) \ : \ \eta \models \varphi_{[]}.$$

Proof. We proceed by induction on the length of η. The base case, $\eta = \varepsilon$, is trivial. For the inductive case, assume that $\eta = \eta'\beta$.

For the 'only if' part, assume that η is valid, and let $\varphi \in \Phi(\eta)$. There are the following two subcases.

— If $\varphi \notin act(\eta)$, then by Lemma 42 it follows that $\eta \models \varphi_{[]}$.

— If $\varphi \in act(\eta)$, then by Definition 38 it follows that $\eta^{-[]} \models \varphi$. By contradiction, assume that $\eta \not\models \varphi_{[]}$. By Theorem 5, it follows that there exists $\sigma : var(\varphi) \to R(\eta,\varphi)$ such that

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

$\eta \not\triangleleft A_{\varphi_{[]}}(\sigma, res(\eta))$. Therefore, by Lemma 41(a), we have that $\eta^{-[]} \not\triangleleft A_{\varphi}(\sigma, res(\eta))$. By Theorem 5, we finally obtain $\eta^{-[]} \not\models \varphi$ – contradiction.

For the 'if' part, we prove the contrapositive. Assume that η is not valid. By Definition 38, either η' is not valid, or $\eta^{-[]} \not\models \varphi$ for some $\varphi \in act(\eta)$.

— If η' is not valid, by the induction hypothesis it follows that $\eta' \not\models \varphi_{[]}$ for some $\varphi \in \Phi(\eta') = \Phi(\eta)$. Since all the offending states of $\varphi_{[]}$ are sinks, it must also be the case that $\eta \not\models \varphi_{[]}$.

— If $\eta^{-[]} \not\models \varphi$ for some $\varphi \in act(\eta)$, then by Theorem 5 it follows that there exists $\sigma : var(\varphi) \to R(\eta, \varphi)$ such that $\eta^{-[]} \not\triangleleft A_{\varphi}(\sigma, res(\eta))$. Therefore, by Lemma 41(b), we have that $\eta \not\triangleleft A_{\varphi_{[]}}(\sigma, res(\eta))$. Theorem 5 then implies the thesis $\eta \not\models \varphi_{[]}$.

□

References

Abadi, M. and Fournet, C. (2003) Access control based on execution history. In: *Proceedings of 10th Annual Network and Distributed System Security Symposium*, The Internet Society.

Alpern, B. and Schneider, F. B. (1987) Recognizing safety and liveness. *Distributed Computing* **2** (3) 117–126.

Baier, C. and Katoen, J.-P. (2008) *Principles of Model Checking*, MIT Press.

Banerjee, A. and Naumann, D. A. (2004) History-based access control and secure information flow. In: Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Cards (CASSIS). *Springer Lecture Notes in Computer Science* **3362** 27–48.

Bartoletti, M. (2009) Usage automata. In: Proceedings of the ARSPA-WITS. *Springer Lecture Notes in Computer Science* **5511** 52–69.

Bartoletti, M., Caires, L., Lanese, I., Mazzanti, F., Sangiorgi, D., Vieira, H. T. and Zunino, R. (2011) Tools and verification. In: M. Wirsing and M. Hölzl (eds.) Rigorous Software Engineering for Service-Oriented Systems - Results of the SENSORIA project on Software Engineering for Service-Oriented Computing. *Lecture Notes in Computer Science* **6582** 408–427.

Bartoletti, M., Costa, G., Degano, P., Martinelli, F. and Zunino, R. (2009a) Securing Java with local policies. *Journal of Object Technology* **8** (4) 5–32. Tool demonstration at BYTECODE 2009. *Electronic Notes in Theoretical Computer Science* **253** (5) 145–151. Available at `http://jalapa.sourceforge.net`

Bartoletti, M., Degano, P. and Ferrari, G. (2009b) Planning and verifying service composition. Extended version of Proceedings of 18th Computer Security Foundations Workshop (CSFW'05) . *Journal of Computer Security* **17** (5) 799–837.

Bartoletti, M., Degano, P. and Ferrari, G.-L. (2005) History based access control with local policies. In: Proceedings of 8th International Conference on Foundations of Software Science and Computational Structures, (FOSSACS). *Springer Lecture Notes in Computer Science* **3441** 316–332.

Bartoletti, M., Degano, P. and Ferrari, G.-L. (2006) Types and effects for secure service orchestration. In: *Proceedings of 19th CSFW*, IEEE Computer Society 57–69.

Bartoletti, M., Degano, P., Ferrari, G.-L. and Zunino, R. (2008a) Model checking usage policies. In: Proceedings of Trustworthy Global Computing (TGC). *Springer Lecture Notes in Computer Science* **5474** 19–35.

Bartoletti, M., Degano, P., Ferrari, G.-L. and Zunino, R. (2008b) Semantics-based design for secure web services. *IEEE Transactions on Software Engineering* **34** (1) 33–49.

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

Bartoletti, M., Degano, P., Ferrari, G. L. and Zunino, R. (2009c) Local policies for resource usage analysis. Extended version of Proceedings of FOSSACS'07. *ACM Transactions on Programming Languages and Systems* **31** (6) 23:1–23:43.

Bauer, L., Ligatti, J. and Walker, D. (2002). More Enforceable Security Policies. In: *More Enforceable Security Policies: proceedings of the FLoC'02 workshop on Foundations of Computer Security*. Iliano Cervesato (editor). Copenhagen, Denmark: DIKU Technical Report, 95–104. Available at `http://www.cs.princeton.edu/ jligatti/papers/mesp.pdf`

Bauer, L., Ligatti, J. and Walker, D. (2005) Composing security policies with Polymer. In: *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, ACM Press 305–314.

Bergstra, J. A. and Klop, J. W. (1985) Algebra of communicating processes with abstraction. *Theoretical Computer Science* **37** 77–121.

Brewer, D. F. C. and Nash, M. J.(1989) The Chinese wall security policy. In: *Proceedings of IEEE Symposium on Security and Privacy* 206–214.

Christensen, S. (1993) *Decidability and Decomposition in Process Algebras*, Ph.D. thesis, Edinburgh University.

Ciancia, V. and Tuosto, E. (2009) A novel class of automata for languages on infinite alphabets. *Technical Report* CS-09-003, Department of Computer Science, University of Leicester.

Degano, P., Ferrari, G. L. and Mezzetti, G. (2011) On quantitative security policies. In: Proceedings of PaCT. *Springer Lecture Notes in Computer Science* **6873** 23–39.

Degano, P., Ferrari G. L. and Mezzetti, G. (2012) Nominal Automata for Resource Usage Control. *CIAA* 125–137. Available at `http://dx.doi.org/10.1007/978-3-642-31606-7_11`

Edjlali, G., Acharya, A. and Chaudhary, V. (1999) History-based access control for mobile code. In: Secure Internet Programming. *Lecture Notes in Computer Science* **1603** 38–48.

Erlingsson, Ú. and Schneider, F. B. (1999) SASI enforcement of security policies: A retrospective. In: Proceedings of 7th New Security Paradigms Workshop 87–95.

Esparza, J. (1994) On the decidability of model checking for several μ-calculi and Petri nets. In: Proceedings of 19th International Colloquium on Trees in Algebra and Programming. *Springer Lecture Notes in Computer Science* **787** 115–129.

Fournet, C. and Gordon, A. D. (2003) Stack inspection: Theory and variants. *ACM Transactions on Programming Languages and Systems* **25** (3) 360–399.

Grumberg, O., Kupferman, O. and Sheinvald, S. (2010) Variable automata over infinite alphabets. In: Language and Automata Theory and Applications. *Springer Lecture Notes in Computer Science* **6031** 561–572.

Hillston, J. (1996) *A Compositional Approach to Performance Modelling*, Cambridge University Press.

Igarashi, A. and Kobayashi, N. (2002) Resource usage analysis. In: *Proceedings of 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, ACM Press.

Kaminski, M. and Francez, N. (1994) Finite-memory automata. *Theoretical Computer Science* **134** (2) 329–363.

Kwiatkowska, M., Norman, G. and Parker, D. (2009) PRISM: Probabilistic model checking for performance and reliability analysis. *ACM SIGMETRICS Performance Evaluation Review* **36** (4) 40–45.

Martinelli, F. and Mori, P. (2007) Enhancing Java security with history based access control. In: Foundations of Security Analysis and Design (FOSAD) Tutorial Lectures. *Springer Lecture Notes in Computer Science* **4677** 135–159.

Mayr, R. (1998) *Decidability and Complexity of Model Checking Problems for Infinite-State Systems*, Ph.D. thesis, Technische Universität München.

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press

Milner, R., Parrow, J. and Walker, D. (1992) A calculus of mobile processes, I and II', *Information and Computation* **100** (1) 1–77.

Montanari, U. and Pistore, M. (2005) History-dependent automata: An introduction. In: Proceedings of SFM. *Springer Lecture Notes in Computer Science* **3465** 1–28.

Pandey, R. and Hashii, B. (1999) Providing fine-grained access control for Java programs. In: Proceedings of 13th European Conference on Object-Oriented Programming (ECOOP). *Springer Lecture Notes in Computer Science* **1628** 449–473.

Samarati, P. and de Capitani di Vimercati, S. (2001) Access control: Policies, models, and mechanisms. In: Foundations of Security Analysis and Design (FOSAD) Tutorial Lectures. *Springer Lecture Notes in Computer Science* **2171** 137–196.

Sandhu, R. and Samarati, P. (1994) Access control: Principles and practice. *IEEE Communications Magazine* **32** 40–48.

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

Segoufin, L. (2006) Automata and logics for words and trees over an infinite alphabet In: Proceedings of 20th International Workshop on Computer Science Logic (CSL). *Springer Lecture Notes in Computer Science* **4207** 41–57.

Shemesh, Y. and Francez, N. (1994) Finite-state unification automata and relational languages. *Information and Computation* **114** (2) 192–213.

Skalka, C. and Smith, S. (2004) History effects and verification. In: Proceedings of Asian Symposium on Programming Languages and Systems (APLAS). *Springer Lecture Notes in Computer Science* **3302** 107–128.

Wallach, D.S., Appel, A.W. and Felten, E.W. (2001) SAFKASI: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)* **9** (4) 341–378.

https://doi.org/10.1017/S096012951200093X Published online by Cambridge University Press