Nominal Automata for Resource Usage Control[⋆]

Pierpaolo Degano, Gian-Luigi Ferrari, and Gianluca Mezzetti

{degano,giangi,mezzetti}@di.unipi.it
Dipartimento di Informatica — Università di Pisa

Abstract. Two classes of nominal automata, namely Usage Automata (UAs) and Variable Finite Automata (VFAs) are considered to express resource control policies over program execution traces expressed by a nominal calculus (`Usages`). We first analyse closure properties of UAs, and then show UAs less expressive than VFAs. We finally carry over to VFAs the symbolic technique for model checking `Usages` against UAs, so making it possible to verify the compliance of a program with a larger class of security properties.

Introduction

Computational models based on finite alphabets seem insufficient to accurately describe programs that adapt their behaviour when plugged inside mutable operational environments, and that therefore offer a multiplicity of dynamic entities. Ubiquitous computing is an illustrative example of these phenomena. Since we cannot predict the actual identities of all the entities that programs may plug in, we can abstractly represent such mutable operational environments as issuing stimuli taken from infinite alphabets. The challenge is therefore developing structures to formally manage infinite alphabets.

In this paper, we exploit *nominal techniques* [14] to deal with these alphabets, the elements of which are called *urelements*. Urelements are atomic objects that are indistinguishable: we can always substitute one for another. The only thing that characterises an object made of urelements is its shape, rather than the actual urelements it is made of. There are many instances of nominal models in the real word. E.g. XML files may contain URL links coming from the web but, roughly, an XML Schema Definition can validate XML files ignoring the specific actual content of these links.

Nominal techniques have been fruitful considered in several fields. Nominal process calculi, namely calculi with dynamic name creations and name-passing, have been shown effective to deal with security and mobility [15]. Nominal automata, that recognise languages over an infinite alphabet, have been developed over the years [9, 5, 16, 18, 8, 24, 11, 10]. Some of these formalisms, among which [9, 8, 7], operate on *data words*, i.e. strings of operations acting on data objects, e.g. reading a file or invoking a remote server. We refer to [22, 20] for a detailed survey and some comparisons. The motivating application of some nominal automata, for example Finite Memory Automata (FMA) [18] and Variable Finite Automata (VFAs) [16], is to express properties of XML and Datalog data. Also, HD-Automata [19] and Fresh Register Automata [24] can decide bisimulation properties of a finite control restriction of the π-calculus [21].

This paper builds over the nominal technique introduced in [6]. There, a basic nominal process calculus, now called `Usages`, is proposed to abstractly represent the

[⋆] This work has been partially supported by IST-FP7-FET open-IP project ASCENS and Regione Autonoma Sardegna, L.R. 7/2007, project TESLA.

behaviour of programs that dynamically generate and operate over resources, through actions, so generating data words. `Usages` encompass full sequentialization, general recursive definitions, and a dynamic name generation operator, much like the π-calculus. Instead, `Usages` do not include name passing facilities.

Usage Automata, UAs for short, have been introduced [5, 6] to specify and enforce security policies of a system, the behaviour of which is abstractly represented by the language of a usage. The security policies considered are actually safety properties, expressing that nothing bad will occur during an execution. To show that a usage U respects a policy φ, represented by a UA, [5] resorted to model checking, in spite of the possible infinite resources the usage U can generate. The verification of security is reduced to the emptiness problem of the intersection between a pushdown and a finite state automaton. The first, i.e. the model, comes from U, the second, i.e. the property, from the UA φ. Indeed, this is the classical automata based model checking technique by Vardi and Volper [25].

This paper aims at a further development of nominal models. In particular we show that classical automata techniques can contribute to assess and better evaluate the expressiveness and the exploitation of nominal models in practice. To this purpose we compare the expressivity of the `Usages` with other nominal models considered in the literature (Section 2). We will show `Usages` to have an expressivity never considered before.

Then we establish some closure properties of the UAs, by studying them as nominal automata from a language theoretic viewpoint (Section 2). In particular, we show that UAs are closed under intersection and union, but not under complement and Kleene star. When seen as policies, this result amounts to saying that we can impose two policies together or be happy if one out of two is obeyed. Instead, to consider secure only those traces that violate a given policy, one has to explicitly build a new UA, which is not guaranteed to exist at all.

Then we move to VFAs. We conservatively extend VFAs to deal with data words. We also show that this smooth extension of VFAs yields automata that are more expressive than UAs (in the spirit of the taxonomy of nominal automata developed in [22, 20].

To conclude the paper, we express through VFAs a larger class of security policies on `Usages`. The crucial point is whether a usage U can be model checked against a VFA (Section 3). We face this problem by rephrasing the technique of [5] and by defining *symbolic* VFAs, that represent the languages of VFAs under collapsing. In this way, VFAs become standard Finite State Automata, thus making model checking possible.

We omit the proofs because of lack of space; they can be found in [12]. Occasionally, we will sketch our arguments to give additional insights on our results.

1 Preliminaries

We recall the notion of `Usages` [6, 2, 5, 3] and Usage Automata [5] and extend VFAs [16] to work on data words. These formalisms will be used to represent execution traces of programs that dynamically generate new resources, and to express properties of sets of traces.

We assume hereafter a finite set Act of actions that operate on a given infinite set *Res* of resources. The actions comprise a special action new that represent the generation of a fresh resource. An event is a pair $\alpha(r)$ with $\alpha \in \mathsf{Act}$ and $r \in \mathit{Res}$, that represents the firing of the action α on the resource r. Then, an execution trace is a sequence of events, i.e. a *data word*. We only consider here `Usages` and Usage Automata with actions on a single resource and we refer the interested reader to [5] for the polyadic version.

The set of resources *Res* is partitioned into two subsets: Res_s and Res_d. Res_s is a finite set of static resources, typically the one that are hard-coded in the program. Instead, Res_d is a countable infinite set of dynamic resources, i.e. the urelements, that are dynamically created. Indeed, whenever a program can generate an execution trace $\alpha(a)\beta(d)\alpha(d)\alpha(d')\beta(a)$ with $a \in Res_s, d, d' \in Res_d$ it can also generate $\alpha(a)\beta(d')\alpha(d')\alpha(d)\beta(a)$, since they only differ on the urelements d, d' that appear in them. Indeed, d and d' are simply exchanged without confusing their identities.

1.1 Usages

`Usages` are a simple nominal calculus designed to abstractly represent the behaviour of programs that create and use resources dynamically [23, 4]. In [4], e.g., these safe approximations are mechanically derived by a type and effect system from the expressions of a λ-calculus suitably extended to specify web services in a secure manner (e.g. featuring a call-by-contract primitive and security policies). For more details and examples the reader is referred to [6].

The syntax of `Usages` follows.

Definition 1.1 (`Usages`). *Let* Nam *be a countable set of names such that* $\mathsf{Nam} \cap Res = \emptyset$. `Usages` *are inductively defined as follows:*

$U, V ::=$	ε	*empty*
	h	*recursion variable*
	$\alpha(r)$	$\alpha(r) \in \mathsf{Act} \times (Res \cup \mathsf{Nam}), \alpha \neq \mathsf{new}$
	$U \cdot V$	*sequence*
	$U + V$	*choice*
	$\mu h.U$	*recursion*
	$\nu n.U$	*resource creation,* $n \in \mathsf{Nam}$

The operators of the calculus are similar to those of the π-calculus, but we have full sequentialization, general recursion and no parallel operator; μh and νn are binders, the first one on recursion variables, the second on names.

A usage is *closed* when it has no free names and no free variables; it is *initial* when it is closed and with no dynamic resources, i.e. it is never the case that a resource $r \in Res_d$ appear as parameter of an action.

The semantics of `Usages` is specified by the labelled transition system in Table 1. We associate with a usage the language consisting of all the prefixes of the traces labelling its computations. The configurations of the transition system are pairs $(U, \mathcal{R})$, where U is a usage and $\mathcal{R} \subseteq Res_d$ is the set of dynamic resources generated so far.

Definition 1.2 (Semantics of `Usages`). *Given a closed usage U we denote with* $[\![U]\!]$ *the set of traces* $\eta = w_1 \dots w_n (w_i \in (\mathsf{Act} \times Res) \cup \{\varepsilon\}, 1 \leq i \leq n)$ *such that:*

$$\exists U', \mathcal{R}'.\, U, \emptyset \xrightarrow{w_1} \cdots \xrightarrow{w_n} U', \mathcal{R}'$$

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

$$\frac{U, \mathcal{R} \xrightarrow{\alpha(r)} U', \mathcal{R}'}{U \cdot V, \mathcal{R} \xrightarrow{\alpha(r)} U' \cdot V, \mathcal{R}'} \qquad \frac{U, \mathcal{R} \xrightarrow{\alpha(r)} U', \mathcal{R}'}{U + V, \mathcal{R} \xrightarrow{\alpha(r)} U', \mathcal{R}'} \qquad \frac{V, \mathcal{R} \xrightarrow{\alpha(r)} V', \mathcal{R}'}{U + V, \mathcal{R} \xrightarrow{\alpha(r)} V', \mathcal{R}'}$$

$$\frac{}{\nu n.U, \mathcal{R} \xrightarrow{\mathsf{new}(r)} U\{r/n\}, \mathcal{R} \cup \{r\}} \quad \text{if } r \in Res_d \setminus \mathcal{R}$$

Table 1. Operational semantics of `Usages`.

The following definition is technical and will be used in Section 3.

Definition 1.3 (Well formedness). *A trace* η *is* well-formed *if it is never the case that:*

1. $\eta = \eta' \mathsf{new}(r) \eta''$ *for some* η', η'' *with* $r \in Res_s$ *or*
2. $\eta = \eta' \mathsf{new}(r) \eta'' \mathsf{new}(r) \eta'''$ *for some* $\eta', \eta'', \eta''', r$ *or*
3. $\eta = \eta' \alpha(r) \eta'' \mathsf{new}(r) \eta'''$ *for some* $\eta', \eta'', \eta''', \alpha$

1.2 Usage Automata

Usage Automata (UAs) [5] are nominal automata over data words. The motivating application of UAs is to express policies for controlling the usage of resources. Policies are regular sets of traces [17]. The UAs express security policies relying on the so-called default-accept approach thus recognizing unwanted traces. Before going into the formal definitions of UAs, consider the example in Fig. 1. The automaton describes the usage policy for opening, reading and writing files. Essentially, it amounts to saying that a file f must be opened before being used (f is a variable standing for a generic file). Starting from q_0, performing the action open on f brings the automaton to q_1, so allowing the file f to be read and written. Instead, an attempt of reading or writing a different file f' or an un-opened file brings to the offending state q_2. For instance, the string open(foo.txt) read(bar.txt) is offending, while open(foo.txt) read(foo.txt) is legal. We assume that a UA remains in the same state by recognising an action that does not match any of the labels of the outgoing edges. E.g. in Fig. 1, the self-loops in q_1 are redundant and are only displayed for readability.

To define UAs, it is convenient to assume a countable infinite set of variables Var; from now onwards, let $V \subset \mathsf{Var}$. We start with a couple of auxiliary definitions.

Definition 1.4 (Substitution). *A* substitution for V *is a function* $\sigma : V \to R, R \subseteq Res.$

Hereafter a substitution σ is considered to be trivially extended on Res_s so that $\sigma(a) = a$ for all $a \in Res_s$. Hence, if $\sigma : V \to R$, the set R contains at least Res_s.

Below, we recall the syntax and the semantics of *guards*.

Definition 1.5 (Guards). *Given a set V of variables we inductively define the set G of guards on* $Res_s \cup V$*, ranged over by* ζ, ζ'*, as follows:*

$G_1, G_2 := true \mid \zeta = \zeta' \mid \neg G_1 \mid G_1 \wedge G_2$

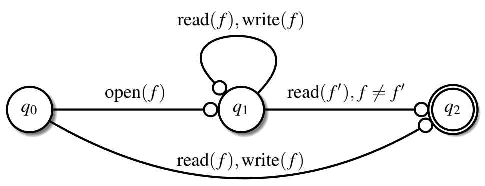

Fig. 1. An example of UA that describe the usage policy for opening, reading and writing files.

A given substitution $\sigma : V \to R$ satisfies a guard g, in symbols $\sigma \models g$, if and only if: $(g = true)$ or $(g = (\zeta = \zeta')$ and $\sigma(\zeta) = \sigma(\zeta'))$ or $(g = \neg g'$ and it is not the case that $\sigma \models g')$ or $(g = g' \wedge g''$ and $\sigma \models g'$ and $\sigma \models g'')$.

We write $(\zeta \neq \zeta')$ for $\neg(\zeta = \zeta')$, $g_1 \vee g_2$ for $\neg(\neg g_1 \wedge \neg g_2))$ and $g_1 \to g_2$ for $\neg g_1 \vee g_2$.

Definition 1.6 (Usage Automata). *A* Usage Automaton (UA) *φ is $\langle S, Q, q_0, F, E\rangle$. The finite set $S \subseteq \mathsf{Act} \times (Res_s \cup Var)$ is its alphabet; Q is its finite set of states; q_0 its initial state; $F \subseteq Q$ the set of its final states; $E \subseteq Q \times S \times G \times Q$ is its finite set of edges with G set of guards on resources and variables in S.*

Given a UA φ, we will refer to the variables occurring in S with $Var(\varphi)$.

Definition 1.7 (Instantiation of UAs). *Let $\varphi = \langle S, Q, q_0, F, E\rangle$ be a UA and $\sigma : Var(\varphi) \to R$ be a substitution. The* instantiation *of φ under σ is the automaton $\varphi_\sigma = \langle R, Q, q_0, F, \delta_\sigma\rangle$, where $\delta_\sigma = X_\sigma \cup \mathrm{Comp}_\sigma(X_\sigma)$ with*

$$X_\sigma = \{(q, \alpha(\sigma(v)), q') \mid (q, \alpha(v), g, q') \in E \text{ and } \sigma \models g\}$$
$$\mathrm{Comp}_\sigma(X_\sigma) = \{(q, \alpha(r), q) \mid \alpha \in Act, r \in R \text{ and } \nexists q' \in Q.(q, \alpha(r), q') \in X_\sigma\}$$

Note that the completion $\mathrm{Comp}_\sigma(X_\sigma)$ may possibly contain infinitely many self-loops of the form $(q, \alpha(r), q)$ when $r \in Res_d$.

Language recognizability by an automaton with infinitely many edges is defined much like that for standard Finite State Automata (FSA, for short): $\eta \in L(\varphi_\sigma)$ if there exists a finite path in φ_σ from q_0 to a $q' \in F$ labelled with η.

Definition 1.8 (Language of UAs). *The string $\eta \in L(\varphi)$ iff there exists a substitution $\sigma : Var(\varphi) \to R$ for some $R \subseteq Res$ such that $\eta \in L(\varphi_\sigma)$.*

1.3 Variable Finite Automata on Data Words

In this section we conservatively extend Variable Finite Automata [16] to work over data words. To simplify notation we overload Act to also denote the actions of VFA.

Definition 1.9 (Variable Finite Automata). *The tuple $\mathcal{A} = \langle \mathsf{Act}, \Omega, \Omega_s, X \cup \{y\}, A\rangle$ is a* Variable Finite Automaton (VFA)*, where X is a finite set of variables;* Act *is a finite set of actions; and Ω is a possibly infinite alphabet with $\Omega_s \subseteq \Omega$ finite subset, $\Omega \cap X = \emptyset$. $A = \langle \Gamma, Q, q_0, F, \delta\rangle$ is a NFA with alphabet $\Gamma = \mathsf{Act} \times (\Omega_s \cup X \cup \{y\})$ and $y \notin (\Omega \cup X)$ is a distinguished* placeholder.

Given a function $m : \Omega \to (\Omega_s \cup X \cup \{y\})$, let $m(\alpha(a)) = \alpha(m(a))$. When unambiguous, we will write $m(\eta)$ for m homomorphically applied to η.

Definition 1.10 (Language of VFAs). *A string* $\eta \in (\mathsf{Act} \times \Omega)^*$ *is a* legal instance *of* $w \in \Gamma^*$ *and w is a* witnessing pattern *of* η*, if there exists a function* $m : \Omega \to (\Omega_s \cup X \cup \{y\})$ *such that* $m(\eta) = w$ *and m is a* correspondence*, i.e.*

1. $\forall a \in \Omega_s . m(a) = a$
2. $\forall x \in X$*. if* $(\exists a, b \in \Omega . m(a) = x$ *and* $m(b) = x)$ *then* $a = b$ *and* $a, b \notin \Omega_s$

A string $\eta \in L(\mathcal{A})$ *iff there exists* $w \in L(A)$ *such that* η *is a legal instance of w.*

Note that here we explicitly present the correspondence between strings and witnessing patterns as a function. Our definition is equivalent to that of [16] when actions are ignored.

Of course, we are interested in the behaviour of VFAs with infinite alphabets, typically when $\Omega = Res, \Omega_s = Res_s, \Omega \setminus \Omega_s = Res_d$.

2 Properties of `Usages` and UA

This section studies `Usages` and UAs from a formal languages point of view.

`Usages`: `Usages` are a process algebra whose semantics is given in term of set of traces. Hence, they can be also regarded as suitable grammar defining a language. Although we found in the literature no widely accepted notion of regular/context-free nominal language, we argue that the recursion operator μ of `Usages` makes the generated languages context-free and non-regular. This is justified by the fact that the language generated by $\nu n.\mu h.(\alpha(n) \cdot h \cdot \alpha(n) + hh + \varepsilon)$ (miming balanced parenthesis) is not recognised by any of the following regular nominal automata: HD-Automata [19], Fresh Register Automata [24], Register Automata [18], VFAs and UAs. To the best of our knowledge, context-free nominal models have been studied only in [10], that introduces the class of *quasi context-free languages* (we refer the reader to Def. 1 of the original paper).

It turns out that the class of languages defined by `Usages` and the class of quasi context-free languages have a non empty intersection, and that neither includes the other, as stated in Property 2.1 below.

Our comparison takes care of the fact that quasi context-free languages are not defined on data words, i.e. they have no actions on resources, while UAs do. The strings generated by UAs belong to $(\mathsf{Act} \times Res)^*$, with the additional constraint that an action over a dynamic resource r must be proceeded by $\mathsf{new}(r)$. Instead, quasi context-free languages have no actions on resources. Therefore we will ignore actions, and only consider the resources accessed when talking of `Usages`, so fixing Res to be the alphabet of both.

Property 2.1. There exists

1. a language generated by a usage U that is not quasi context-free;
2. a quasi context-free language that can be generated by no usage U.

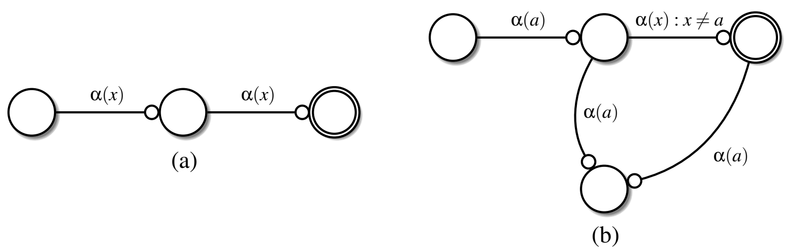

Fig. 2. On the left/right a UA recognising a language the complement/Kleene star of which can not be recognised by any UA.

For showing statement (1) above, consider the usage $U = \mu h.(\nu n.\alpha(n)) \cdot h$. As a matter of fact, there is no bound on the number of fresh resources that can occur in a string generated by U, while in a quasi context-free language the bound is given by the number of the registers, that are a characterizating feature of the models for generating/recognising these languages (for details, see [10]). The second statement holds because there is no usage U such that $[\![U]\!] = Res^*$, that is a quasi context-free language. We conjecture that this lack of expressivity of `Usages` comes from the absence of an explicit mechanism for disposing and reusing resources. In contrast, quasi context-free models can overwrite the content of a register, so forgetting that the resource previous contained therein already appear previously in the generated string.

Usage Automata: Logical connectives between policies have an equivalent counterpart as language operators. The complement of a language recognised by a UA φ is the set of traces that violates the negation of policy expressed by φ. The union/intersection of the languages recognised by two UAs φ, ψ is the set of traces violating the conjunction/disjunction of the two policies expressed by φ and ψ.

Hence, closure properties are not only interesting from a theoretical viewpoint, but also deeply connected with the intended applications of UAs.

Theorem 2.1. *The set of languages accepted by UAs is closed under union and intersection. but it is not closed under complement and Kleene star.*

Below, we give an intuition on the proofs of non closedness, as they give some additional insight on the expressivity of UAs (see [12]). To see that UAs are not closed by complement, consider the UA in Fig. 2(a). It recognises those strings containing at least two occurrences of $\alpha(r)$ for some $r \in Res$. The complement of this language contains those strings where all mentioned resources are pairwise distinct. This is a non-regular property, because an automaton should store all previously used resources.

The UA in Fig. 2(b) shows that UAs are not closed under Kleene star. It recognises those strings containing one $\alpha(a)$ followed by $\alpha(r)$ for some $r \in Res, r \neq a$. Now consider the Kleene star of the language $L(\varphi)$ and, by contradiction, let ψ be the UA recognising it. We can take arbitrary long strings $s \in L(\psi)$ of the form $s = \alpha(a)\alpha(d_1)\ldots\alpha(a)\alpha(d_n)$. Since variables in ψ are finite not all d_i can be bound to a variable, this implies that some self-loop labelled $\alpha(d_k)$ obtained by completion is traversed in recognising s. Then also

$s' = \alpha(a)\alpha(d_1)\ldots\alpha(a)\ldots\alpha(a)\alpha(d_n)$ obtained from s removing $\alpha(d_k)$ is recognised by φ, this is a contradiction because $s' \notin L(\psi)^*$.

The last fact does not however reduce the power of UAs in expressing security policies. Consider a language of the form $L(\varphi)^*$ and let η be a trace in the semantics $[\![U]\!]$ of a usage U. For any $\eta \in L(\varphi)^*$ there exists also a prefix $\eta' \in L(\varphi)$ with $\eta' \in [\![U]\!]$ by Definition 1.2. This means that checking U against $L(\varphi)^*$ is the same as checking it against $L(\varphi)$. However, there exists a safety property that is not expressible by a UAs. Informally this safety policy requires to take a token $\alpha(a)$ before performing any other kind of action, while if two tokens are taken, any sequence of actions is then allowed. The above sketched safety property is expressed by the VFA in Fig. 3. In general, the following theorem holds.

Theorem 2.2. *UAs are less expressive than VFAs.*

It would also be interesting to formally compare the expressive power of some variants of UAs and of VFAs. We have a couple of preliminary results. First, consider the restriction of VFAs obtained by only permitting the distinguished placeholder y to occur in self-loops. We conjecture that this variant of VFAs has the same expressive power of UAs. Now, consider the extension of UAs with a wild-card, introduced in [3]. A wild-card can stand for any resource, and so it plays the role of the placeholder y in a VFA. Not surprisingly UAs extended in this way are just as expressive as VFAs.

3 Model checking

As mentioned above, UAs have been introduced to specify and enforce security policies of systems, the behaviour of which is abstractly represented by the language of a usage U. The security policies considered are actually safety properties, expressing that nothing bad will occur during a computation, abstractly represented by the string η [17]. The approach taken in [5] follows the default-accept paradigm, i.e. only the unwanted behaviour is explicitly mentioned — this assumption justifies the way UAs are instantiated, and in particular the completion step made therein. Consequently, the language of φ is the set of *unwanted traces*, and an accepting state is considered offending. Then U respects the property φ, in symbols $U \models \varphi$, if and only if $\eta \in [\![U]\!] \Rightarrow \eta \notin L(\varphi)$.

To show that a usage U respects a policy φ, the authors of [5] resorted to model-checking, in spite of the possible infinite resources a usage can generate. This is done by carefully collapsing the verification to a well-known problem: the emptiness of the

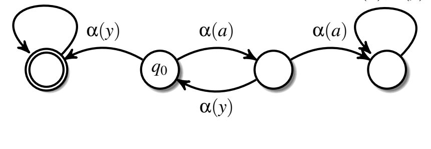

Fig. 3. A VFA the language of which is not accepted by any UA. The alphabet is $\Omega = Res_d \cup \{a\}$ with $\mathsf{Act} = \{\alpha\}$.

intersection between a pushdown and a finite state automata, that is decidable [13]. The first, i.e. the model, comes from U, the second, i.e. the property, from the UA φ. Indeed, this is the classical automata based model checking technique by Vardi and Volper [25] with φ expressing unwanted traces, instead.

Since VFAs are more expressive than UA, the question arises whether the same checking technique of [5] can be used to model check a usage against more expressive policies, expressed by VFAs. The answer is positive and one can model check a usage U against a VFAs, defining policy compliance in the obvious way: $U \models \mathcal{A}$ if and only if $\eta \in [\![U]\!] \Rightarrow \eta \notin L(\mathcal{A})$.

To do this we introduce *symbolic* VFAs. Following [5], we let their alphabet be the finite set of witnesses $\mathsf{W} \subset \{\#_i\}_{i\in\mathbb{N}}$, where $\{\#_i\}_{i\in\mathbb{N}} \cap Res = \emptyset$. We also need a distinguished symbol $\_ \notin Res \cup \{\#_i\}_{i\in\mathbb{N}}$.

We recall from [5] the crucial notion of *collapsing* mapping, that is the link between `Usages`, VFAs and their symbolic counterparts. As a matter of fact, this is the technical machinery that deals with urelements, and it permits to abstract from their actual identity.

Definition 3.1 (Collapsing). *Given a finite set of witnesses* W*, a collapsing mapping* $\kappa : Res \to Res_s \cup \mathsf{W} \cup \{\_\}$ *of* $R \subset Res_d$ *onto* W *is a function such that:*

1. $\kappa(r \in Res_s) = r$
2. $\kappa(R) = \mathsf{W}$ *and it is injective*
3. $\kappa(Res_d \setminus R) = \{\_\}$

We write $\kappa(\alpha(a))$ *for* $\alpha(\kappa(a))$ *and* $\kappa(\eta)$ *for the homomorphic extension of* κ *to* η.

The following property is very technical, and simplifies the procedure for model checking `Usages` against UAs and VFAs. Roughly, it states that well-formedness of traces can be checked by the so-called *unique-witness* automaton.

Property 3.1 (Unique-witness). Given a finite set of witnesses W and an initial usage U, there exists a *unique-witness FSA* N_{W} such that:

- $\eta \notin N_{\mathsf{W}} \Longrightarrow \forall \#_i \in \mathsf{W}.$ there is a single $\mathsf{new}(\#_i)$ in η
- $\eta \in [\![U]\!] \Longrightarrow \eta \notin L(N_{\mathsf{W}})$

By exploiting the construction given in [5], we can now associate with a usage U a symbolic pushdown automaton $\mathsf{B}_{\mathsf{W}}(U)$, the language of which is denoted by $L(\mathsf{B}_{\mathsf{W}}(U))$. The following theorem puts together some results proved in [5].

Theorem 3.1. *Given an initial usage U, there exist a finite set* W *of witnesses and a pushdown automaton* $\mathsf{B}_{\mathsf{W}}(U)$ *on the finite alphabet* $\mathsf{Act} \times (Res_s \cup \mathsf{W} \cup \{\_\})$ *such that:*

- *Given a collapsing* κ *such that* $\kappa(Res_d) \subseteq \mathsf{W} \cup \{\_\}$ *then:*

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

- *Given a collapsing* κ *such that* $\kappa(Res_d) \supseteq \mathsf{W}$*, then:*

$$\forall \eta'.\ (\eta' \in L(\mathsf{B}_{\mathsf{W}}(U)) \wedge \eta' \notin N_{\mathsf{W}}) \Rightarrow (\exists \eta.\ \eta \in [\![U]\!] \wedge \eta' = \kappa(\eta))$$

Definition 3.2 (Symbolic VFAs). *Let $\mathcal{A} = \langle \mathsf{Act}, Res, Res_s, X \cup \{y\}, A\rangle$ be a VFA. Given a finite set of witnesses* W*, let* $Res_{\mathsf{W}} = Res_s \cup \mathsf{W} \cup \{\_\}$.
The symbolic VFA *on* W *is* $\mathcal{A}_{\mathsf{W}} = \langle \mathsf{Act}, Res_{\mathsf{W}}, Res_s, X \cup \{y\}, A\rangle$. *Language recognition for symbolic VFAs additionally requires the correspondence m to be such that* $m(\_) = y$.

The following theorem makes clear the links between the language of a VFA and that of its symbolic automaton.

Theorem 3.2. *Let $\mathcal{A} = \langle \mathsf{Act}, Res, Res_s, X \cup \{y\}, A\rangle$ be a VFA, and let* W *be a set of witnesses such that* $|\mathsf{W}| = |X|$, $\mathcal{A}_{\mathsf{W}}$ *as in Definition 3.2 and let K be the set of the collapsing* κ *such that* $\kappa(Res_d) = \mathsf{W} \cup \{\_\}$*, then:*

- $\forall \eta.(\eta \in \mathcal{A} \Rightarrow \exists \kappa \in K. \kappa(\eta) \in \mathcal{A}_{\mathsf{W}})$
- $\forall \kappa \in K, \eta.(\kappa(\eta) \in \mathcal{A}_{\mathsf{W}} \Rightarrow \eta \in \mathcal{A})$

We carry over VFAs the notions of substitution and instantiation, which transforms a VFA into a *Finite* State Automaton. The language recognised by any VFA can then be represented under collapsing by a finite class of its instantiations.

Definition 3.3 (Instantiation of VFAs). *Let $\mathcal{A}_{\mathsf{W}} = \langle \mathsf{Act}, Res_{\mathsf{W}}, Res_s, X \cup \{y\}, A\rangle$ be a symbolic VFA with $A = \langle \Gamma, Q, q_0, F, \delta\rangle$, $\Gamma = \mathsf{Act} \times (Res_s \cup X \cup \{y\})$.*
Given a function $\overline{m} : X \cup Res_s \to Res_s \cup \mathsf{W}$ it is a substitution *for $\mathcal{A}$ if it is the identity on Res_s and it is injective on X.*
Given a substitution $\overline{m}$ the instantiation of $\mathcal{A}$ is $\mathcal{A}^{\overline{m}} = \langle Res_{\mathsf{W}}, Q, q_0, F, \delta^*\rangle$, where

$$\begin{aligned}\delta^* = &\{(q, \alpha(\overline{m}(v)), q') \mid (q, \alpha(v), q') \in \delta, v \neq y\} \cup \\ &\{(q, \alpha(d), q') \mid (q, \alpha(y), q') \in \delta, d \in (Res_{\mathsf{W}} \setminus (Res_s \cup Image(\overline{m}))\}\end{aligned}$$

We remark that, by the finiteness of W, $\mathcal{A}^{\overline{m}}$ is a standard FSA on a finite alphabet.

Theorem 3.3. *Let $\mathcal{A} = \langle \mathsf{Act}, Res_{\mathsf{W}}, Res_s, X \cup \{y\}, A\rangle$ be a symbolic VFAs. Then:*

$$\eta \in L(\mathcal{A}) \Leftrightarrow \exists \text{ substitution } \overline{m}. \eta \in L(\mathcal{A}^{\overline{m}})$$

To simplify the technical development, we find convenient to resort to the well-known *weak-until* operator $A \mathfrak{W} B$ between automata, meaning that A holds until B holds or B always holds. We refer to a standard book on model checking, e.g. [1], or to [5] for more details.

Theorem 3.4 (Model checking). *Let U be an initial usage on the resources $Res = Res_d \cup Res_s$; let $\mathcal{A} = \langle \mathsf{Act}, Res, Res_s, X \cup \{y\}, A\rangle$ be a VFA; and let* W *be a set of witnesses such that* $|\mathsf{W}| = |X|$*. Then $U \vDash \mathcal{A}$ if and only if:*
$\forall$ *substitution* $\overline{m} : X \cup Res_s \to Res_s \cup \mathsf{W}.\ L(\mathsf{B}_{\mathsf{W}}(\mathsf{U})) \cap L(\mathcal{A}_{\mathsf{W}}^{\overline{m}} \mathfrak{W} N_{\mathsf{W}}) = \emptyset$

This theorem gives us the means for an efficient model checking procedure. Given a substitution $\overline{m}$, it is indeed decidable to check whether $L(\mathsf{B}_{\mathsf{W}}) \cap L(\mathcal{A}_{\mathsf{W}}^{\overline{m}} \mathfrak{W} N_{\mathsf{W}}) = \emptyset$ and there are finitely many substitutions $\overline{m}$, because Res_s, X and W are finite.

We can then re-use the model checker LocUsT [2] for verifying `Usages` against VFAs. As for complexity issues, we can restate the theorem established in [5] for VFAs. The proof is mostly the same with only minor changes regarding the number of instantiations of VFAs.

Theorem 3.5. *The worst-case asymptotic behaviour of model-checking an usage U against an automaton φ with n variables is* $O(|U|^{|n|+1})$.

Conclusions

We have first studied two classes of nominal automata, namely Usage Automata (UAs) [5] and Variable Finite Automata (VFAs) [16], aiming at using them to express resource control policies. We analysed closure properties of the languages recognized by UAs, and showed that the expressive power of UAs is weaker that the one of VFAs. Then, we considered `Usages` [5], a nominal process calculus for modelling the (abstract) behaviour of programs with dynamic creation of resources. The class of languages defined by `Usages` is neither included nor contains the class of quasi context-free languages [10].

We slightly extended the symbolic technique of [5], that is based on collapsing and that reduces the two nominal automata mentioned above to standard Finite State Automata. Also the execution traces of a nominal calculus can be collapsed to traces of standard pushdown automata. This enables us to model check the compliance of execution traces against a property expressed in terms of a VFA. Indeed the collapsing above brings back us to the classical problem of verifying the emptiness of the intersection between a pushdown and a finite state automaton. Our results guarantee the correctness and the completeness of our proposal.

We plan to study whether the symbolic technique used here can be extended and applied to other classes of nominal automata (e.g. Finite Memory Automata) and to more expressive nominal process calculi to specify systems (e.g. with resources garbaging). It would be also interesting to further develop the taxonomy about nominal automata by placing UAs and VFAs into the expressivity hierarchy of [22]. A co-algebraic presentation of these automata could help, especially for investigating their relation with functors on nominal sets possibly with fusion of names. The operational approach to express properties based on nominal automata is deeply connected with the logical approach. It would then be important to exactly relate the expressive power of different kinds of nominal automata with that of various logics, e.g. EMSO [8, 7] or LTL [1].

Acknowledgments. We would like to thank the anonymous referees for their comments that greatly helped us to improve the quality of our paper.

References

[1] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[2] M. Bartoletti and R. Zunino. LocUsT: a tool for checking usage policies. Technical Report TR08-07, University of Pisa, 2008.

[3] Massimo Bartoletti, Gabriele Costa, Pierpaolo Degano, Fabio Martinelli, and Roberto Zunino. Securing Java with local policies. *Journal of Object Technology*, 8(4):5–32, 2009.

[4] Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. Planning and verifying service composition. *Journal of Computer Security*, 17(5):799–837, 2009.

[5] Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari, and Roberto Zunino. Model checking usage policies. In Christos Kaklamanis and Flemming Nielson, editors, *TGC*, volume 5474 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2008. Extended version to appear in *Math. Stuct. Comp. Sci.*

[6] Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari, and Roberto Zunino. Local policies for resource usage analysis. *ACM Trans. Program. Lang. Syst.*, 31(6), 2009.

[7] M. Benedikt, C. Ley, and G. Puppis. Automata vs. logics on data words. In *Computer Science Logic*, pages 110–124. Springer, 2010.

[8] Benedikt Bollig. An automaton over data words that captures EMSO logic. In Joost-Pieter Katoen and Barbara König, editors, *CONCUR*, volume 6901 of *Lecture Notes in Computer Science*, pages 171–186. Springer, 2011.

[9] P. Bouyer. A logical characterization of data languages. *Information Processing Letters*, 84(2):75–85, 2002.

[10] Edward Y. C. Cheng and Michael Kaminski. Context-free languages over infinite alphabets. *Acta Inf.*, 35(3):245–267, 1998.

[11] V. Ciancia and E. Tuosto. A novel class of automata for languages on infinite alphabets. Technical report, CS-09-003, University of Leicester, UK, 2009.

[12] Pierpaolo Degano, Gianluca Mezzetti, and Gian-Luigi Ferrari. Nominal models and resource usage control. Technical Report TR-11-09, Dipartimento di Informatica, Università di Pisa, 2011.

[13] Javier Esparza. On the decidability of model checking for several μ-calculi and Petri nets. In *Proc. 19th Int. Colloquium on Trees in Algebra and Programming*, volume 787 of *Lecture Notes in Computer Science*. Springer, 1994.

[14] M.J. Gabbay and A.M. Pitts. A new approach to abstract syntax with variable binding. *Formal aspects of computing*, 13(3):341–363, 2002.

[15] A. Gordon. Notes on nominal calculi for security and mobility. *Foundations of Security Analysis and Design*, pages 262–330, 2001.

[16] O. Grumberg, O. Kupferman, and S. Sheinvald. Variable automata over infinite alphabets. *Language and Automata Theory and Applications*, pages 561–572, 2010.

[17] Kevin W. Hamlen, J. Gregory Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. on Programming Languages and Systems*, 28(1):175–205, 2006.

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

[19] U. Montanari and M. Pistore. π-calculus, structured coalgebras, and minimal hd-automata. *Mathematical Foundations of Computer Science 2000*, pages 569–578, 2000.

[20] F. Neven, T. Schwentick, and V. Vianu. Towards regular languages over infinite alphabets. *Mathematical Foundations of Computer Science 2001*, pages 560–572, 2001.

[21] Davide Sangiorgi and David Walker. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press, 2001.

[22] L. Segoufin. Automata and logics for words and trees over an infinite alphabet. In *Computer Science Logic*, pages 41–57. Springer, 2006.

[23] Christian Skalka, Scott Smith, and David Van Horn. Types and trace effects of higher order programs. *Journal of Functional Programming*, 18(2):179–249, 2008.

[24] N. Tzevelekos. Fresh-register automata. *ACM SIGPLAN Notices*, 46(1):295–306, 2011.

[25] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS*, pages 332–344. IEEE Computer Society, 1986.