Università degli Studi di Pisa

Dipartimento di Informatica
Dottorato di Ricerca in Informatica
Settore Scientifico Disciplinare: INF/01

Ph.D. Thesis

Nominal Context-Free Behaviour

Gianluca Mezzetti

Contents

Abstract

This thesis investigates and proposes models for programming and verifying adaptive software at different abstraction levels. First, we design the kernel of a programming language, endowed with primitives for programming the adaptation to different working environments. We provide the language with a type and effect system that allows us to statically prove properties of the behaviour of the program when plugged in different execution environments. Then we extend our language to program the use of the resources currently available in the environment. In this case, the identity and the number of resources is unknown a-priori. The previous analysis technique needs to be extended to capture the behaviour of these programs. We exploit nominal techniques in the literature to propose novel automata models that represent the behaviour and the properties of programs that use an unbounded number of unknown resources as (regular and context-free) set of traces. The theoretical properties of these automata are investigated and related with static program verification. We prove that we are able to check regular properties of the usage patterns of the resources when resource reuse is inhibited.

Acknowledgements

While, as required by law, the results in this thesis are product of my own work, none of them would be possible without the help of several people.

I would like to thank my PhD advisors, Pierpaolo Degano and Gian-Luigi Ferrari for their enthusiastic support during these past four years. Their research contribution to this thesis is witnessed by each Definition, Theorem and $\square$, but there is another history to tell.

Pierpaolo is the kind of person that is meticulous in following its code of conduct in all the aspects of the work, still being open to diplomacy and taking great delight to be an insolent anarchic, sometimes, cum grano salis. He tried to teach me how to balance these things in my character, fighting with me for four long years, and, among the other things, I sincerely thank him (and his garbage can) for this.

Giangi is the kind of person that follows everything, there no council, board or condominium meeting where he is not requested, there is no research field which does not interest him, he always have some paper for you to read. I thank him for having transmitted me this energy, and for having helped me linking different research fields. Even though I was not able to look forward as him, following him blinded has always been rewarding.

Ringrazio inoltre la mia famiglia per avermi sempre supportato in questo lavoro, ciecamente e senza poter comprendere le mie soddifazioni. Ringrazio Matteo, Lillo, Davide, Luca, Manu, Damiano, Giulio, Francesco, Michele, Andrea e Marta con i quali ho condiviso bellissimi momenti di relax.

Eventually, I would like to thank Alexander Kurz and Nikos Tzevelekos, that read with such care my thesis, returning insightful observations, that I would enjoy to investigate for another PhD.

However there is only one Computer Science PhD in one men's life, and now this is unfortunately over.

Gianluca

Introduction

Devices and the software therein are connected at any time and anywhere. Internet is *de facto* the infrastructure providing us with access points for our digitally instrumented life. A great variety of activities and tasks performed by individuals are mediated, supported and affected by different heterogeneous digital systems that in turn cooperate each other without human intervention. These digital entities can be any combination of hardware devices and software pieces, and their activities can change the physical and the virtual environment where they are plugged in. Consequently, the digital environment is an open-ended communication infrastructure, through which its entities can interact each other in a loosely coupled manner, and they can access resources of different kinds, e.g., local or remote, private or shared, data or programs, devices or services.

Some illustrative, yet largely incomplete, cases of computational models and technologies that realise the scenario above are: *Service Oriented Computing, Internet of Things and Cloud Computing.*

In the Service Oriented Computing paradigm, applications are built by composing software units called services, which are published, linked, and invoked on-demand by other services using standard internet-based protocols. Applications can dynamically reconfigure themselves, by re-placing unreachable services with new ones.

The Internet of Things extends the inter-connected digital environment to every-day objects, e.g. webTV, cars, smartphones, tablets, ebook readers, etc. These objects often have a limited computational power, but they are capable of connecting to the Internet and of interacting each other. Because of the portability of these object, the abstract digital environment they inhabit is open-ended and highly dynamic.

Cloud computing makes available huge computational facilities and dynamic resources, such as networks, servers, storage, applications and services are made available in the cloud. The remote customers, depending on the kind of service offered, can plug in their applications, data, platforms or network infrastructures into the cloud data centres. A key point here is that these resources are “virtualised” so that they appear to the software as fully dedicated to them, and potentially unlimited.

These new settings require that the software is programmed to dynamically exploit the resources available in the environment, by reconfiguring itself from time to time as new ones show up and old ones disappear. Then we assisted to a shift of programming paradigm. In the past, programs were targeted to close, specific and controlled environments, where all the resources were known a-priori. Now, the programs are developed to work with little assumptions on the resources available, still being able to reconfigure themselves and use them whenever they appear.

Because of the continuous interactions with external software and resources, security issues are growingly important. The constraints often concern usage patterns of the a-priori unknown resources, so that security requirement themselves need to be adaptive.

The main challenge for engineering systems in the scenarios sketched above is to provide the developers with the suitable techniques to program adaptive software systems, including all the tools and methodologies for proving or guaranteeing functional and non-functional properties of the code.

This thesis advocates formal methods as an useful machinery to tackle these challenges.

We will propose models for programming adaptive components and for abstracting their behaviour, with a particular focus on resource usage. Since our goal is to investigate the key aspects of adaptivity at different levels of abstractions, we will adopt a multi-tier approach ranging from the programming language tier to the one of abstract mathematical models. At the programming language level, we will propose a core programming language, with explicit primitives for programming adaptation. Several illustrative examples of actual program codes will guide us to isolate the behavioural aspects that need to be modelled at our high abstraction level. Our abstract model will be specifically targeted to investigate those aspects in isolation. In the abstraction step, we will try to maintain a minimal number of assumptions on the structure of the resources, so preserving in the model their peculiarity of being dynamic and a-priori unknown.

Our path starts in Chapter 2, where we define a core programming language, called ContextML, with adaptivity and security primitives. The design of ContextML is based on the Context-Oriented Programming (COP) paradigm introduced in [23]. In COP, the developer is provided with primitive constructs to modify the environment (called context) and to specify behaviour that are activated depending on the actual state of the environment. We give a formal semantics of ContextML as a trace language. A trace is a sequence of actions representing the various adaptations to the environment and the resource usage.

To specify which adaptation and resource usage patterns are correct, ContextML is also provided with primitives to specify security policies, i.e. properties that deems the illegal sequences of actions. The program is stopped before an offending trace is generated.

To guarantee at compile-time that a program never gets stuck, we use a type and effect system [47], that constructs an effect in a suitable process calculus over-approximating all the traces a program may generate. This abstraction is checked to guarantee that the program will behave correctly in all the contexts it will be plugged in and that all the security policies are satisfied.

One of the limitations of the COP approach is that the programmer must envisage all the different situations the program may run into and write the associated behaviour. However, being the environment mutable and open-ended, an unbounded number of behaviour needs to be specified. Section 2.5 shows an extension of ContextML where adaptation is specified in a parametric way, i.e. by symbolically abstracting the unknown resources that may occur in the environment.

This extension highlights some of the characteristics that make behavioural analysis hard in this case and constitutes the basis for better shaping our challenges, through examples. The analysis framework shown in Chapter 2 will constitute our reference setting, suggesting an automata-based model-checking approach [60] to check security properties of programs.

It turns out that the ContextML programs operate on unknown a-priori resources the number of which is unbounded. In the literature, nominal techniques [50] have been exploited to tackle the unboundedness, by abstracting the infinite symbols as opaque entities that can be tested for equality only. Nominal automata [9] and process calculi [6, 32] have been developed to recognise languages on infinite alphabet.

In Chapter 3 we focus on three nominal models: Usages process calculus [6], Usage Automata (UA) [6] and Variable Finite Automata (VFA) [33].

In [6], Usages are shown effective to abstract the behaviour of programs with an unbounded number of resources. The UA are proposed to express properties of such programs. A machinery is developed to model-check Usages behaviour against UA properties.

Motivated by the fact that Usages and UA can respectively represent the behaviour and the properties of portion of ContextML, in Chapter 3, we investigate some language theoretic properties. These results allow us to compare them with other models in the literature. In particular, it turns out that UA are less expressive than *Variable Finite Automata* [33].

In Section 3.2 we carry over Variable Finite Automata [33] the technique used for UA [6], so allowing to model-check Usages against the wider class of languages represented by VFA.

In Section 3.3 we show, through an example, the limitation of Usages in capturing the behaviour of ContextML programs when resources need to be released and reused. Also, it turns out that Usages are incomparable with the well-known *quasi context-free languages* [19], so leaving open the quest for a model that is able to go beyond the expressiveness of both models.

In Chapter 4 we propose a new class of context-free automata, *pushdown nominal automata*, that are more expressive than Usages and quasi context-free languages and capture the behaviour of ContextML programs. We study some language theoretic properties of our model and we compare it against other automata the literature. We also investigate the decidability of the emptiness problem, that is directly connected with automata-based model-checking techniques.

The results in this thesis have been presented at international workshop and conferences, in particular: ContextML has been presented at different design stages at the

- *Programming Language Approaches to Concurrency and Communication-cEntric Software* workshop [25] in Tallinn, Estonia
- *6th International Conference on Coordination Models and Languages* [24] in Stockholm, Sweden
- *11th Int. Conf. on Information Systems and Industrial Management* in Venice, Italy [10].

The results in Chapter 3 have been presented at the *17th International Conference on Implementation and Application of Automata* in Porto, Portugal [26].

Our new model, the pushdown nominal automata in Chapter 4 have been presented at the *18th International Conference on Implementation and Application of Automata* in Halifax, Canada [27].

The next chapter fixes the notation and reviews some notions that are necessary in the rest of the thesis.

Chapter 1

Preliminaries

This chapter recalls some notions in the literature about Context Oriented Programming, resource usage security policies and nominal automata.

Table 1.1 recalls less common mathematical notation.

1.1 Context Oriented Programming

A major concern of current software engineering is the development of adaptive software components, capable of dynamically modifying their behaviour depending on changes in their execution environment and in response to the interactions with other components. The problem of developing adaptive components has been investigated from different perspectives (control theory, artificial intelligence, programming languages) and various solutions have been proposed. We refer to [18, 17, 52] for a more comprehensive discussion.

We recall now a prominent approach, called Context-Oriented Programming (COP) and discussed in [35], that allows to describe fine-grain adaptability mechanisms at programming language level. Following this paradigm, standard programming languages are extended with suitable constructs to express context-dependent behaviour in a modular fashion. The seminal paper [23] introduces ContextL, an extension of Common Lisp with COP features.

The claimed benefits of COP are justified by Software Engineering purposes. Using COP, the developer gains additional expressiveness, being able to deal with adaptation using language constructs. Moreover, [35] argues that COP enhance the separation of the *crosscutting concern* arising from adaptation. In the programming languages jargon, a crosscutting concern is some functional or non-functional behaviour that does not fit the dominant modularisation/logic of

$\mathbb{N}, \emptyset$	the set of natural numbers and the empty set
$\underline{\mathtt{r}} = \{i \mid 1 \leq i \leq r\}$	set of numbers in $\mathbb{N}$
$X \mathrel{\not\lesseqgtr} X'$	incomparable sets: $X \not\subseteq X'$ and $X \not\supseteq X'$
$\wp(X), \wp_f(X)$	set of subsets of X, set of finite subsets of X
$\|X\|$	cardinality of the set X
$f : X \to X'$	function f with domain X and codomain X'
$f : X \rightharpoonup X'$	partial function f with domain X and codomain X'
$Img(f)$	image of the function f
$f\{^b/_a\}$	function such that $f\{^b/_a\}(x) = f(x)$ when $x \neq a$, $f\{^b/_a\}(a) = b$ otherwise
$f \restriction_X$	restriction of f to X

Table 1.1: Mathematical notation

the program. The separation of crosscutting concerns enhances the readability and the maintainability of the programs, and hence their development.

The concept of *behavioural variations* is central to COP paradigm: variations express a chunk of behaviour that can substitute or modify a portion of the basic behaviour of the application. As highlighted in [35] the whole COP paradigm is built on the following key concepts:

- The layers are groups of context-dependent behavioural variations made visible by programmers. They can be activated and deactivated at runtime into the program flow and can be passed and manipulated, because they are first class entities. Behavioural variation of the program are defined using layers as labels.
- The actual behaviour of the program depends on the combination of all active variations, which constitutes the *context*. The *dispatching* mechanism decides which variation has to be executed.

In the line of research started by ContextL, the context is simply a stack of layers. The activation of a layer L is triggered by the **with**(L) construct, that pushes L on the context. Layered definitions are available at a method-abstraction level and object-abstraction level.

After ContextL various COP languages have been put forward. Among them we recall ContextJ [2], ContextFJ* [21] and ContextFJ [34]. Only a few of the ones we mentioned investigate semantics issues and provide a sound strong type system.

To illustrate the application of the key COP concepts, consider the example in Figure 1.1, written in ContextJ. The code defines the class Person with two redefinitions of the method toString. The two definitions are behavioural variations, the first one is for the Address layer, the other for the Employment layer. When the method toString is called on an instance of Person, the implementation of which method to execute is chosen depending on the layers currently active in the context. The **with** statement activates a layer within the scope of its block. As for the dispatching procedure, the method to be invoked is chosen searching a matching active layer in reverse activation order. The **proceed** statement is much like the **super** of Java and it executes the method implementation in the next active layer. The output of the program in Figure 1.1 is:

```
Output: Name: Pascal Costanza; Address: 1000 Brussel;
[Employer] Name: VUB; Address: 1050 Brussel
```

1.2 Resource usage analysis

The discipline named *Program analysis* [47] concerns static techniques for computing reliable approximations about the dynamic behaviour of programs. These approximations are used for improving various aspects of the generated code, for verifying functional or non-functional properties of the program.

In the adaptive setting sketched in the introduction, security emerges as a relevant non-functional program requisite. To specify these requisites, one needs to describe the executions that are unacceptable, through the definition of a set of *security policies* [54].

According to [54], a more formal definition follows.

Definition 1.2.1. A program history η is a finite or infinite ordered sequence of atomic program events. A *security policy* ϕ is specified by giving a predicate P on sets of program histories. Let H be the set of the program histories η that a program e may generate, e satisfies the security policy ϕ if and only if H satisfies P. Whenever this is the case we write $H \models \phi$.

Intuitively a *safety policy* expresses that nothing bad will occur during a computation. The formal definition follows.

Definition 1.2.2. A *safety policy* is a security policy such that

- the satisfaction of the policy can be decided by checking each execution in isolation, i.e. a set of executions H satisfies a policy ϕ if and only it is expressed by a predicate P such that for all histories $\eta \in H$ the singleton $\{\eta\}$ satisfies P.

```
class Person {
  private String name, address;
  private Employer employer;
  Person(String newName, String newAddress,Employer newEmployer){
    this.name = newName;
    this.employer = newEmployer;
    this.address = newAddress;
  }
  String toString() {return "Name: " + name;}
  layer Address {
    String toString() {
    return proceed() + "; Address: " + address;}
    }
  layer Employment {
    String toString() {
    return proceed() + "; [Employer] " + employer;}
  }
}

class Employer {
  private String name, address;
  Employer(String newName, String newAddress) {
    this.name = newName;
    this.employer = newAddress;
  }
  String toString() {return "Name: " + name;}
  layer Address {
    String toString() {
    return proceed() + "; Address: " + address;}
  }
}
...
Employer vub = new Employer("VUB", "1050 Brussel");
Person somePerson = new Person("Pascal Costanza", "1000 Brussel", vub);
with (Address) {
  with (Employment) {
    System.out.println(somePerson);
  }
}
```

Figure 1.1: ContextJ code example

- it is *prefix-closed*: if a history $\eta \in H$ is safe (i.e. $\{\eta\}$ satisfies P) than each prefix of η is safe.
- such that checking an infinite execution can be done by checking all its finite prefixes, i.e. $H \models \phi$ if and only if for all $\eta \in H$ and for all the finite prefix η' of η it holds that $\{\eta'\}$ satisfies P.

In this thesis, we will focus on safety usage policies of resources, i.e. safety policies enforcing the correct usage pattern of the resources in the execution environment. From now on, we will focus on sets H of finite program histories only.

We briefly review here the analysis approach of Skalka et al.[56], extended in [5] with security primitives. The events of interest in the execution history are all the accesses to resource, we denote by $\alpha(a)$ the action α performed on the resource a.

During the evaluation of a program e, a history η is built starting from the empty sequence ε by attaching the occurred events.

$$\varepsilon, e \to \alpha(a), e' \to \alpha(a)\beta(b), e'' \to \cdots \to \eta, *$$

All the executions constitute a language on an alphabet made of actions α and resources a.

To enforce execution safety, [5] introduce a *policy framing primitive.* A policy framing is a primitive of the form $\phi[e]$ that make possible to guard a program fragment e against a policy. The policy ϕ is enforced at each execution step by verifying that the actual history respects ϕ (written $\eta \models \phi$):

$$\varepsilon, \phi[e] \xrightarrow{\text{if } \varepsilon\models\phi} \alpha(a), \phi[e'] \xrightarrow{\text{if } \alpha(a)\models\phi} \alpha(a)\beta(b), \phi[e''] \xrightarrow{\text{if } \alpha(a)\beta(b)\models\phi} \cdots \to \eta, *$$

As soon as a policy is not satisfied, the program gets stuck. Framings can be nested.

In [6] regular policies of the histories are investigated, in the form of the *Usage Automata* of Section 1.3.1. An analysis framework is developed to guarantee at compile-time that a program never gets stuck because it attempts to violate a security policy. The framework uses a type and effect system ([47]) to assign a type and an effect to each valid program. The effect is an expression in a suitable process algebra (the Usages in Section 1.3.4) representing all the histories that the program can generate.

To check that a program is safe, one can use the effect U to verify that all active policies are satisfied. The authors of [6] resorted to automata-based model-checking [29]. This is done by carefully reducing the verification to a well-known problem: the emptiness of the intersection between a pushdown and a finite state automaton, that is decidable.

1.3 Nominal Trace Models

The previous section highlighted the relevance of describing the languages of the histories arising from adaptive program executions. Programs are often plugged in dynamic environments, offering a multiplicity of dynamic resources, that is unbounded a-priori. Automata on finite alphabets seem insufficient to accurately describe histories where an unbounded number of resources may appear.

Nominal techniques [9, 50] have been fruitful exploited to develop automata on infinite 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 (schemata), Web-based objects (URLs), security protocols (e.g. nonces and time-stamps), Cloud systems (virtualised resources), etc [55, 15, 11]. When the language expresses the sequence of actions made on the resources, it is called a *data-words* language [16, 14, 9].

The problem of handling unboundedly many fresh (or restricted) names has been tackled also under the perspective of foundational calculi for concurrent and distributed systems, under the term *nominal* calculi [51, 13, 32, 40, 45].

This section surveys some nominal models in the literature. We have chosen to recall here only the ones that appear in detail in the thesis.

Although there is no universally recognised notion of nominal regular and context-free languages we will often informally speak of regular and context-free automata. This is because often the authors introducing the model themselves classify their automata as the nominal equivalent of the finite state automata. As for context-free automata, in the classical automata theory settings, they are characterised by the Dyck-like language $L_R = \{ww^R\}$ of words w followed by their reverse. However, this notion becomes fuzzy in the nominal setting because we can read L_R as a set of words with a fixed number of different symbols ($L_R = \{ww^R \mid \text{ the number of symbols in } w \text{ is } r\}$) or not.

We now fix some notation. Assume an infinite alphabet Σ, partitioned in a finite set of *static* symbols Σ_s and an infinite set of *dynamic* symbols Σ_d. The first is intended to represent the resources known before program execution starts, while Σ_d contains the resources that may be acquired or generated at run-time. The Kleene star applied to Σ is denoted Σ^*.

A string w is a sequence of symbols on an alphabet. The reverse word of w is denoted w^R, the empty one by ε. The length of w is written $|w|$.

When dealing with data-words, we assume Act to be the set of atomic actions α, β that can be performed on the symbols in Σ. The pair (α, a) (written $\alpha(a)$) denotes that the action α is performed on the symbol a. A data word w is a string $\alpha(a)\beta(b)\cdots \in (\mathsf{Act} \times \Sigma)^*$.

We write $\|X\|$, overloaded to any object X, to denote the set of symbols used in X. For example, $\|w\|$ denotes the set of symbols in the string w.

Given an automaton A, we denote by $L(A)$ the language recognised by A and by $\mathcal{L}(M)$ the set of all languages recognised by the automaton model M passed as argument, e.g. $\mathcal{L}$(Finite State Automata) is the set of regular languages.

1.3.1 Usage Automata

Usage Automata (UA) [6] are regular nominal automata recognising data words. They have been used in [6] to express safety properties of resource usage. Consider the UA in the example of Figure 1.2. The automaton controls opening, reading and writing operations on the files. We say that the automaton describes the *usage policy* for the files resources. Essentially, it demands 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' brings to the offending state q_2, provided that f' has not been previously opened (the sink q_3 capture this case). For instance, the string $\mathsf{open(foo.txt)}\,\mathsf{read(bar.txt)}$ is offending, while $\mathsf{open(foo.txt)}\,\mathsf{read(foo.txt)}$ is legal. We assume that a UA remains in the same state by recognising an action that matches no labels of the outgoing edges. E.g. The self-loops in the state q_1 of Figure 1.2 might then be safely omitted, and are there drawn for clarity.

It is convenient to give some auxiliary definitions. Assume a countable set of variables Var; from now onwards, let $V \subset \mathsf{Var}$.

Definition 1.3.1 (Substitution). A *substitution for* V is a function $\sigma : V \to R, R \subseteq \Sigma$.

For technical reasons, the domain and the codomain of the substitution functions are always specified explicitly, so that, whenever a substitution is given, also these two additional sets are known (e.g. see Definition 1.3.4).

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

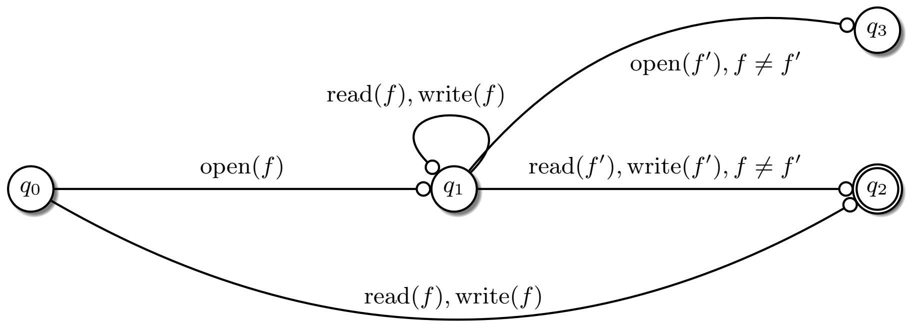


Figure 1.2: An example of UA that demands a file f to be opened before being used.

Below, we recall the syntax and the semantics of *guards*, that will label the edges of UA.

Definition 1.3.2 (Guards)**.** Given a set V of variables we inductively define the set G of guards on $\Sigma_s \cup V$, ranged over by ζ, ζ', as follows:

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

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 $\sigma \models g''$).

Definition 1.3.3 (Usage Automata)**.** A *Usage Automaton (UA) B* is a tuple $\langle S, Q, q_0, F, E\rangle$. The finite set $S \subseteq \mathsf{Act} \times (\Sigma_s \cup \mathsf{Var})$ 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 B, we will refer to the variables occurring in S with $Var(B)$.

Definition 1.3.4 (Instantiation of UA)**.** Let $B = \langle S, Q, q_0, F, E\rangle$ be a UA and $\sigma : Var(B) \to R$ be a substitution. The *instantiation* of B under σ is the automaton $B_\sigma = \langle R, Q, q_0, F, \delta_\sigma\rangle$, where $\delta_\sigma = X_\sigma \cup \mathrm{Comp}_\sigma(X_\sigma)$ with

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

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

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

Definition 1.3.5 (Language of UA)**.** The string $\eta \in L(B)$ iff there exists a substitution $\sigma : Var(B) \to R$ for some $R \subseteq \Sigma$ such that $\eta \in L(B_\sigma)$.

1.3.2 Variable Finite Automata on Data Words

We recall now the Variable Finite Automata [33], extended on data words in [26].

Definition 1.3.6 (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 Ω_s finite subset of Ω, $\Omega \cap X = \emptyset$. $A = \langle \Gamma, Q, q_0, F, \delta\rangle$ is a non deterministic finite state automaton (NFA, for short) 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 $m : \Omega \to (\Omega_s \cup X \cup \{y\})$, m is extended to pairs $\alpha(a) \in \mathsf{Act} \times \Omega$ such that $m(\alpha(a)) = \alpha(m(a))$. When unambiguous, we will write $m(\eta)$ for m homomorphically applied to η.

Definition 1.3.7 (Language of VFA)**.** 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. It is the identity on $\Omega_s : \forall a \in \Omega_s . m(a) = a$
2. It is injective on the contraimage of $X : \forall x \in X, a, b \in \Omega.$ if $m(a) = x$ and $m(b) = x$ then $a = b$

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 rephrase differently the definition in [33], by explicitly presenting the correspondence between strings and witnessing patterns as a function. Our definition is provably equivalent to the original one when actions are ignored.

Of course, we are interested in the behaviour of VFA with infinite alphabets, typically when $\Omega = \Sigma, \Omega_s = \Sigma_s, \Omega \setminus \Omega_s = \Sigma_d$.

1.3.3 Finite Memory Automata

We recall now the definition of *Finite Memory Automata*, given in [38]. This model does not distinguish between static and dynamic symbols, rather it uses a flat countable infinite alphabet that here we denote Ω.

Definition 1.3.8. Let As be the set of the assignments $\sigma : \underline{\mathtt{r}} \to \Omega$, where $\underline{\mathtt{r}} = \{i \mid 1 \leq i \leq r\}$, that are injective. A *Finite Memory Automaton* (FMA) with r registers on the alphabet Ω is a tuple $A = \langle Q, q_0, \sigma_0, \rho, \mu\rangle$ where: Q is the set of states; q_0 is the initial state; $\sigma_0 : \underline{\mathtt{r}} \to \Omega \in \mathsf{As}$ is the initial assignment of the registers; $\rho : Q \rightharpoonup \underline{\mathtt{r}}$ is the partial reassignment function; δ is the transition relation, $\delta \subseteq Q \times \underline{\mathtt{r}} \times Q$.

A configuration is a tuple $C \in (Q \times \mathsf{As} \times \Omega^*)$, where: the first component records the current state, the second is the assignment of the registers, and the last one is the portion of the input to be scanned.

A step between configurations $(q, \sigma, ww') \to (q', \sigma', w'), w \in \Omega$ occurs iff:

- If $w = \sigma(k)$ for some k then $\sigma' = \sigma$ and $(q, k, q') \in \delta$
- If for all $k \in \underline{\mathtt{r}}.w \neq \sigma(k)$ then $\rho(q)$ is defined and $\sigma'(\rho(q)) = w$, for each $k \neq \rho(q).\sigma'(k) = \sigma(k)$ and $(q, \rho(q), q') \in \delta$.

Let $\to^*$ denote the reflexive and transitive closure of $\to$, the language recognised by a FMA is the set of strings w such that there exists a run $C_0 = (q_0, \sigma_0, w) \to^* C_n = (q_n, \sigma_n, \varepsilon)$ such that $q_n \in F$.

Extensions

The History-Register Automata [59] extends Finite Memory Automata [38] and Fresh Register Automata [58].

The automata are equipped with a fixed number m of *histories* and n of *registers*. A history is a finite set of symbols in Ω, while registers are singletons subsets of Ω. We call *place*, without distinction histories and registers.

The set of assignment HAs, associating each place with its symbols is defined as follows:

$$\mathsf{HAs} = \{\sigma : \underline{\mathtt{m} + \mathtt{n}} \to \wp_f(\Omega) \mid \forall i > m. |\sigma(i)| \leq 1\}$$

Let $X \subseteq \underline{\mathtt{m}}$ then define $\sigma@X$ to be the set $\cap_{i\in X}\sigma(i) \setminus \cup_{i\notin X}\sigma(i)$, assuming $\cap_\emptyset = \Omega$. The function $\sigma[X \mapsto S]$ is the set of pairs $\{(i, \sigma(i)) \mid i \notin X\} \cup \{(i, S) \mid i \in X\}$. Assume the function $\sigma[a\ in\ X]$ is such that

$$\sigma[a\ in\ X](i) = \begin{cases} \sigma(i) \setminus \{a\} & \text{if } i \notin X \\ \sigma(i) \cup \{a\} & \text{if } i \in X \cap \underline{\mathtt{m}} \\ \{a\} & \text{if } i \in X \setminus \underline{\mathtt{m}} \end{cases}$$

We are now ready to define the History-Register Automata.

Definition 1.3.9. A *History-Register Automaton* (HRA) of type (m, n) (an (m,n)-HRA) is a tuple $\langle Q, q_0, \sigma_0, \delta, F\rangle$ where: Q is the set of states; q_0 is the initial state; $F \subseteq Q$ is the set of final states; $\sigma_0 \in \mathsf{HAs}$ is the initial assignment and $\delta \subseteq Q \times \mathsf{Lab} \times Q$ is the transition relation, where $\mathsf{Lab} = \wp(\underline{\mathtt{m}+\mathtt{n}})^2 \cup \wp(\underline{\mathtt{m}+\mathtt{n}})$.

We denote transitions $q \xrightarrow{X,X'} q'$ or $q \xrightarrow{X''} q'$, the latter being called *reset* transitions. A configuration C is a a pair $C = (q, \sigma) \in Q \times \mathsf{HAs}$.

A step between configuration $(q, \sigma) \xrightarrow{x} (q', \sigma')$ occurs iff

- $x = a \in \Omega$ and there exists $q \xrightarrow{X,X'} q' \in \delta$ such that $a \in \sigma@X$ and $\sigma' = \sigma[a\ in\ X']$
- $x = \varepsilon$ and there exists $q \xrightarrow{X} q' \in \delta$ such that $\sigma' = \sigma[X \mapsto \emptyset]$

The language accepted by A is $L(A) = \{a_1 \ldots a_n \in \Omega^* \mid (q_0, \sigma_0) = C_1 \xrightarrow{a_1} C_2 \cdots \xrightarrow{a_{n-1}} C_n = (q_n, \sigma_n), q_n \in F\}$

Fresh Register Automaton are a restriction of HRA with only one history.

Definition 1.3.10. A Fresh Register Automaton (FRA) is a $(1, n)$-HRA such that:

- $\sigma_0(1) = \bigcup_{i \in \underline{\mathtt{1}+\mathtt{n}}} \sigma_0(i)$
- for all $(q, l, q') \in \delta$ it holds $l = (X, X')$ and $1 \in X'$
- for all $(q, (\{1\}, X'), q') \in \delta$ there exists $(q, (\emptyset, X'), q') \in \delta$

1.3.4 Usages

Usages (or History Expressions) [57, 6] are a simple process algebra conceived to abstract the set of histories that may be generated by a program that uses unboundedly many resources.

In this thesis we will consider Usages also as pure languages generators. In this case it is useful to change some aspects of the original definition, to make it more essential.

This section reviews both definitions of Usages and shows some of their properties.

The syntax follows, where we assume Nam to be a countable set of names ($\mathsf{Nam} \cap \Sigma = \emptyset$) and Act to be a set of actions containing the action $\mathtt{new}$, and the actions $[_\phi,]_\phi$ for each policy ϕ.

$$\text{(epsilon)}\frac{}{\varepsilon \cdot U, \mathcal{R} \xrightarrow{\varepsilon} U, \mathcal{R}} \qquad \text{(act)}\frac{}{\alpha(a), \mathcal{R} \xrightarrow{\alpha(a)} \varepsilon, \mathcal{R}}$$

$$\text{(rec)}\frac{}{\mu h.U, \mathcal{R} \xrightarrow{\varepsilon} U\{\mu h.U/h\}, \mathcal{R}} \qquad \text{(dot)}\frac{U, \mathcal{R} \xrightarrow{\alpha(a)} U', \mathcal{R}'}{U \cdot V, \mathcal{R} \xrightarrow{\alpha(a)} U' \cdot V, \mathcal{R}'}$$

$$\text{(plus}_1\text{)}\frac{U, \mathcal{R} \xrightarrow{\alpha(a)} U', \mathcal{R}'}{U + V, \mathcal{R} \xrightarrow{\alpha(a)} U', \mathcal{R}'} \qquad \text{(plus}_2\text{)}\frac{V, \mathcal{R} \xrightarrow{\alpha(a)} V', \mathcal{R}'}{U + V, \mathcal{R} \xrightarrow{\alpha(a)} V', \mathcal{R}'}$$

$$\text{(nu)}\frac{}{\nu n.U, \mathcal{R} \xrightarrow{\texttt{new}(a)} U\{a/n\}, \mathcal{R} \cup \{a\}} \text{if } a \in \Sigma_d \setminus \mathcal{R}$$

Table 1.2: Operational semantics of the Usages.

Definition 1.3.11 (Usages)**.** Usages are inductively defined as follows:

$U, V ::=$	ε	empty
	h	recursion variable
	$\alpha(a)$	$\alpha(a) \in \mathsf{Act} \times (\Sigma \cup \mathsf{Nam}), \alpha \neq \texttt{new}$
	$U \cdot V$	sequence
	$U + V$	choice
	$\mu h.U$	recursion
	$\nu n.U$	resource creation, $n \in \mathsf{Nam}$
	$\phi[U]$	safety framing

The operators of the calculus are similar to those of the π-calculus, but Usages have full sequentialization (for brevity, we will write UV in place of $U \cdot V$), general recursion and no parallel operator; μh and νn are binders, the first one on recursion variables, the second on names. The usages are endowed with a safety framing operator $\phi[U]$ (see Section 1.2) to record the security policies to be enforced on the traces. The safety framing $\phi[U]$ is an abbreviation for the usage $[_\phi \cdot U \cdot]_\phi$, for simplicity we assume that a dummy static resource is omitted in $[_\phi,]_\phi$, hence being $[_\phi,]_\phi \in \mathsf{Act} \times \Sigma_s$. We assume given a fixed a notion of validity $\eta \models \varphi$, telling whether a history satisfies a policy or not.

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 $a \in \Sigma_d$ appears as parameter of an action.

The semantics of Usages is specified by the labelled transition system in Table 1.2. 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 $\mathcal{R} \subseteq \Sigma_d$ is the set of dynamic resources generated so far.

Definition 1.3.12 (Semantics of Usages)**.** Given a closed usage U let $[\![U]\!]$ be the set of traces (or *histories*) $\eta = v_1 \dots v_n (v_i \in (\mathsf{Act} \times \Sigma) \cup \{\varepsilon\}, 1 \leq i \leq n)$ such that:

$$\exists U', \mathcal{R}'.\ U, \emptyset \xrightarrow{v_1} \cdots \xrightarrow{v_n} U', \mathcal{R}'$$

Usages as language generators

When considering the Usages purely as language generators, we will not use the safety framing operator and we will consider the variation of rule (nu) below, where `new` action is hidden.

$$\text{(nu)}\,\frac{}{\nu n.U, \mathcal{R} \xrightarrow{\varepsilon} U\{a/n\}, \mathcal{R} \cup \{a\}}\,\text{if } a \in \Sigma_d \setminus \mathcal{R}$$

The recursion operator μ makes Usages a context-free model (in our terms), as informally justified by the following example.

Example 1.3.1. *The expression $\nu n.\mu h.(\alpha(n) \cdot h \cdot \alpha(n) + \varepsilon)$ generates a set of traces of the form $\{ww^R \mid$ all the symbols in w are pairwise distinct$\}$.*

Properties for program analysis

Validity of Usages Given a history η we denote with $\eta^{-[]}$ the trace purged of all framings actions $[_\phi,]_\phi$. For details and examples, see [7].

The multiset $ap(\eta)$ of the *active policies* of a history η is defined as follows:

$$\begin{array}{lll} ap(\varepsilon) = \{\,\} & & ap(\eta\,[_\phi) = ap(\eta) \cup \{\phi\} \\ ap(\eta\,\gamma) = ap(\eta) & \gamma \in (\mathsf{Act} \times \Sigma) \setminus \{[_\phi,]_\phi\} & ap(\eta\,]_\phi) = ap(\eta) \setminus \{\phi\} \end{array}$$

The validity of a trace η ($\models \eta$ in symbols), w.r.t. all the active policies appearing therein, is inductively defined as follows, assuming a notion of policy compliance $\eta \vDash \phi$.

$$\begin{array}{llll} \vDash \varepsilon & & & \\ \vDash \eta' v & v \in (\mathsf{Act} \times \Sigma) & \text{if } \vDash \eta' \text{ and } (\eta' v)^{-[]} \vDash \phi & \text{for all } \phi \in ap(\eta' v) \end{array} \tag{1.1}$$

A usage U is *valid* when $\models \eta$ for all $\eta \in [\![U]\!]$.

The following lemma states that validity is a prefix-closed property.

Property 1.3.2. *If a history η is valid, then each prefix of its is valid.*

The following definition is technical:

Definition 1.3.13 (Well formed traces)**.** A trace η is *well-formed* if it is never the case that:

1. $\eta = \eta'\texttt{new}(a)\eta''$ for some η', η'' with $a \in \Sigma_s$ or
2. $\eta = \eta'\texttt{new}(a)\eta''\texttt{new}(a)\eta'''$ for some $\eta', \eta'', \eta''', a$ or
3. $\eta = \eta'\alpha(a)\eta''\texttt{new}(a)\eta'''$ for some $\eta', \eta'', \eta''', a, \alpha \neq \texttt{new}$

The first condition assures that no static resource is the target of a `new`, the second guarantees that no resource is the target of a `new` twice, the third checks that no resource is used before it has been target of a `new`.

We recall from [6] the crucial notion of *collapsing* mapping. Let W be a finite set of witnesses such that $\mathsf{W} \subset \{\#_i\}_{i\in\mathbb{N}}$, where $\{\#_i\}_{i\in\mathbb{N}} \cap \Sigma = \emptyset$. We also need a distinguished symbol $\_ \notin \Sigma \cup \{\#_i\}_{i\in\mathbb{N}}$.

Definition 1.3.14 (Collapsing)**.** Given a finite set of witnesses W, a collapsing mapping $\kappa : \Sigma \to \Sigma_s \cup \mathsf{W} \cup \{\_\}$ of $R \subset \Sigma_d$ onto W is a function such that:

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

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

The well-formedness of collapsed traces can be checked by the so-called *unique-witness* automaton, that we recall here from [6].

Definition 1.3.15 (Unique Witness Automaton)**.** Let W be a finite set of witness, the unique-witness automaton N_{W} is defined as the union of the automaton $N_{\#_i}$ for all $\#_i \in \mathsf{W}$, where each automaton $N_{\#_i} = \langle \mathsf{Act} \times (\Sigma_s \cup \mathsf{W} \cup \{\_\}), \{q_0^i, q_1^i, q_2^i\}, q_0^i, \{q_2^i\}, \delta_{\#_i}\rangle$ is defined as follows:

$$\begin{aligned}\delta_{\#_i} = \{&q_0^i \xrightarrow{\texttt{new}(\#_i)} q_1^i, q_1^i \xrightarrow{\texttt{new}(\#_i)} q_2^i\}\\ &\cup \{q_0 \xrightarrow{v} q_0, q_1 \xrightarrow{v} q_1 \mid v \neq \texttt{new}(\#_1)\}\\ &\cup \{q_2 \xrightarrow{v} q_2 \mid v \in \mathsf{Act} \times (\Sigma_s \cup \mathsf{W} \cup \{\_\})\}\end{aligned}$$

The unique-witness automaton can guarantee that each witness is created at most once, as stated below.

Property 1.3.3 (Unique-witness)**.** *Given a finite set of witnesses* W *and an initial usage U, the unique-witness FSA* N_{W} *is such that:*

- $\forall \eta \in (\mathsf{Act} \times (\Sigma_s \cup \mathsf{W} \cup \{\_\}))^*$ *it holds*
  $\eta \notin L(N_\mathsf{W}) \Longrightarrow \forall \#_i \in \mathsf{W}.$ *there is at most one* $\texttt{new}(\#_i)$ *in* η
- $\forall \eta \in (\mathsf{Act} \times \Sigma)^*$ *it holds*
  $\eta \in [\![U]\!] \Longrightarrow \kappa(\eta) \notin L(N_\mathsf{W})$

By exploiting the construction given in [6], 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 alphabet of $\mathsf{B}_\mathsf{W}(U)$ is a finite set of witness W that represents in a symbolic manner the relative equalities and dis-equalities of a history η. It turns out that these relations uniquely characterise the language of U, the following theorem makes this characterization precise.

Theorem 1.3.4. *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 $\mathsf{Act} \times (\Sigma_s \cup \mathsf{W} \cup \{\_\})$ such that:*

- *Given a collapsing κ such that $\kappa(\Sigma_d) \subseteq \mathsf{W} \cup \{\_\}$ then:*
  $\forall \eta.\ \eta \in [\![U]\!] \Rightarrow \kappa(\eta) \in L(\mathsf{B}_\mathsf{W}(U))$
- *Given a collapsing κ such that $\kappa(\Sigma_d) \supseteq \mathsf{W}$, then:*
  $\forall \eta'.\ (\eta' \in L(\mathsf{B}_\mathsf{W}(U)) \wedge \eta' \notin N_\mathsf{W}) \Rightarrow (\exists \eta.\ \eta \in [\![U]\!] \wedge \eta' = \kappa(\eta))$

1.3.5 Quasi Context-Free Languages

In [19] an automata model and a grammar model is presented, provably recognising the same class of languages, called Quasi Context-Free Languages (QCFL). We recall here the definition of the automata model, the *Infinite Alphabet Pushdown Automata.*

Definition 1.3.16. Let the set As be as in Definition 1.3.8. An *Infinite Alphabet Pushdown Automaton* (IAPA) with r registers is a tuple $A = \langle Q, q_0, \sigma_0, \rho, \mu \rangle$ where: Q is the set of states; q_0 is the initial state; $\sigma_0 : \underline{\mathtt{r}} \to \Sigma \in \mathsf{As}$ is the initial assignment of the registers; $\rho : Q \rightharpoonup \underline{\mathtt{r}}$ is the partial reassignment function; δ is the transition function, mapping elements from $Q \times (\underline{\mathtt{r}} \cup \{\varepsilon\}) \times \underline{\mathtt{r}}$ to finite subsets of $Q \times \underline{\mathtt{r}}^*$

A configuration is a tuple $C \in (Q \times \mathsf{As} \times \Sigma^* \times \Sigma^*)$, where: the first component records the current state, the second is the assignment of the registers, the third is the portion of the input yet to be read, and the last one is the content of the stack, read top-down.

A step between configurations $(q, \sigma, ww', aS) \to (q', \sigma', w', w_pS), w \in \Sigma \cup \{\varepsilon\}, a \in \Sigma$ occurs iff:

- If ρ is undefined then $\sigma' = \sigma$, otherwise $\sigma'(\rho(q)) \neq \sigma(\rho(q))$ and for $k \neq \rho(q)$ holds $\sigma'(k) = \sigma(k)$.
- If $w = \varepsilon$ then for some i holds $\sigma'(i) = a$ and there exists $(q', x_1 \ldots x_n) \in \delta(q, \varepsilon, i)$ such that $w_p = \sigma'(x_1) \ldots \sigma'(x_n)$.
- If $w \neq \varepsilon$ then for some k, i holds $\sigma'(k) = w, \sigma'(i) = a$ and there exists $(q', x_1 \ldots x_n) \in \delta(q, k, i)$ such that $w_p = \sigma'(x_1) \ldots \sigma'(x_n)$.

The language recognised by a IAPA is the set of strings w such that there exists a run $C_0 = (q_0, w, \sigma_0, \sigma_0(r)) \to^* C_n = (q_n, \varepsilon, \sigma_n, \varepsilon)$. Note that the initial stack assignment is made only by the symbol on the last register and that acceptance is *by empty stack*.

Chapter 2

A Semantics for Context-awareness

In this chapter we propose the kernel of a programming language suitable for the development of complex adaptive software. We will focus on three key characteristics of adaptive components: (*i*) the mechanisms to manipulate the context, (*ii*) the security policies governing the behaviour and the resource usage, (*iii*) the interactions with other components.

As for our model for the interactions with other components, we do not wire a component to a specific communication infrastructure. Our communication model is based on a bus, through which messages are exchanged. We assume given an abstract, declarative representation of the operational environment.

We suitably extend and integrate together techniques from Context Oriented Programming (Section 1.1), type theory and model-checking. In particular, we develop a static technique ensuring that a component (*i*) adequately reacts to context changes, (*ii*) accesses resources in accordance with security policies, (*iii*) exchanges messages on the bus, complying with a specific communication protocol provided by the operating environment.

Our proposal requires several stages.

I First, in Section 2.2, we design a core programming language for programming adaptive components, called ContextML. ContextML embeds constructs for resource manipulation and mechanisms to declare and enforce security policies by adopting the local sandbox approach of [7]. The language also features message passing constructs for communicating with external parties.

II Next, we provide ContextML with a type and effect system (Section 2.3). We exploit it for ensuring that programs adequately react to context changes and for computing as effect an abstract representation of the overall behaviour.

This representation, in the form of a subset of the *Usages* (Section 1.3.4), describes the sequences of resource manipulations and communication with external parties in a succinct form.

III Finally, we model-check effects to verify that the component behaviour is correct, i.e. that the behavioural variations can always take place, that resources are manipulated in accordance with the given security policies and that the communication protocol is respected. The model-checking is performed in two phases. The first determines whether security policies are obeyed, the second one verifies compliance with the protocol (Section 2.4).

In the next section we introduce a motivating example, that is also instrumental in displaying our methodology at a glance.

2.1 A motivating example: an e-library application

Consider a simple scenario consisting of a smartphone application that uses some service supplied by a cloud infrastructure. The cloud offers a repository to store and synchronize a library of ebooks and computational resources to execute customised applications (among which full-text search).

A user buys ebooks online and reads them locally through the application. The purchased ebooks are stored into the remote user library and some books are kept locally in the smartphone. The two libraries may not be synchronized. The synchronization is triggered on demand and depends on several factors: the actual bandwidth available for connection, the free space on the device, etc. We specify below the fragment of the application that implements the full-text search over the user's library.

Consider now the context dependent behaviour emerging because of the different energy profiles of the smartphone. We assume that there are two: one is active when the device is plugged in, the other is active when it is using its battery. These profiles are represented by two *layers*: `ACMode` and `BatMode`. The function `getBatteryProfile` returns the layer describing the current active profile depending on the value of the sensor (`plugged`):

$$\text{fun } \texttt{getBatteryProfile}\ x = \textbf{if}\ (\texttt{plugged})\ \textbf{then}\ \texttt{ACMode}\ \textbf{else}\ \texttt{BatMode}$$

Layers can be activated, so modifying the context. The expression

$$\textbf{with}(\texttt{getBatteryProfile}())\ \textbf{in}\ exp_1 \tag{2.1}$$

activates the layer obtained by calling `getBatteryProfile`. The scope of this activation is the expression exp_1 in Figure 2.1(a). In lines 2-10, there is the following *layered expression*:

$$\texttt{ACMode}.\ \langle \textsc{do search} \rangle,$$
$$\texttt{BatMode}.\ \langle \textsc{do something else} \rangle$$

This is the way context-dependent *behavioural variations* are declared. Roughly, a layered expression is an expression defined by cases. The cases are the different layers that may be active in the context, here `BatMode` and `ACMode`. Each layer has an associated expression. A *dispatching mechanism* inspects at runtime the context and selects an expression to be reduced. If the device is plugged in, then the search is performed, abstracted by ⟨DO SEARCH⟩. Otherwise, something else gets done, abstracted by ⟨DO SOMETHING ELSE⟩. Note that if the programmer neglects a case, then the program throws a runtime error being unable to adapt to the actual context.

In the code of exp_1 (Figure 2.1(a)), the function g consists of nested layered expressions describing the behavioural variations matching the different configurations of the execution environment. The code exploits context dependency to take into account also the actual location of the execution engine (remote in the cloud at line -3- or local on the device -4-), the synchronization state of the library -5,6- and the active energy profile -2,10-. The smartphone communicates with the cloud system over the bus through message passing primitives -7-9-.

The search is performed locally only if the library is fully synchronized and the smartphone is plugged in. If the device is plugged in but the library is not fully synchronized, then the code of function g is sent to the cloud and executed remotely by a suitable server.

In Figure 2.1(b) we show a fragment of the environment provided by the cloud infrastructure. The service considered is offering generic computational resources to the devices connected on the bus by continuously running function f. The function f listens to the bus for incoming code (a function) and an incoming layer. Then, it executes the received function in a context extended with the received layer.

In the code of the cloud it appears a security policy ϕ to be enforced before running the received function. This is expressed by the security framing $\phi[\ldots]$ that causes a sandboxing of the enclosed expression, to be executed under the strict monitoring of ϕ. Take ϕ to be a policy expressing that writing on the library `write(library)` is forbidden (so only reading is allowed). The framing guarantees that the execution of foreign code does not alter the remote library.

```
fun g x =
  ACMode.
    IsCloud. search(),
    IsLocal.
      LibrarySynced. search(y),
      LibraryUnsynced.
        send_ty[τ](ACMode);
        send_ty[τ'](g);
        receive_ty[τ''];
  BatMode.⟨DO SOMETHING ELSE⟩
g()
```

(a) The definition of exp_1

```
fun f x =
  let lyr = receive_ty[τ] in
  let g = receive_ty[τ'] in
    φ[with(lyr) in
      let res = g() in
        send_ty[τ''](res)
    ]; f()
f()
```

(b) The code for a service

Figure 2.1: Fragments of an App and of a service in the cloud

In this example, we simply state that ϕ only concerns actions on resources, e.g. the library. Our approach also allows us to enforce security policies governing behaviour adaptation and communication.

The cloud system constraints communications on the bus by also declaring a protocol P, prescribing the viable interactions. Additionally, the cloud infrastructure will make sure that the protocol P is indeed an abstraction of the behaviour of the various services of it involved in the interactions. We do not address here how protocols are defined by the environment and we only check whether a user respect the given protocol.

The actual protocol guaranteed by the environment is

$$P = (send_{ty}[\tau]send_{ty}[\tau']receive_{ty}[\tau''])^*$$

It expresses that the client must send a value of type τ then a value of type τ' and then must receive back a value of type τ''. These actions can be repeated a certain number of times. We will discuss later on the actual types τ, τ', τ''.

Function `getBatteryProfile` returns a value of type $ly_{\{\texttt{ACMode},\texttt{BatMode}\}}$. This type means that the returned layer is one between `ACMode` and `BatMode`.

The type of function g is $\tau' = \texttt{unit} \xrightarrow{\mathbb{P}|U} \tau''$, assuming that the value returned by the `search` function has type τ''. The type τ' is annotated by a set of preconditions $\mathbb{P}$ (see below) and a latent effect U (discussed later on).

$$\mathbb{P} = \{\{\texttt{ACMode}, \texttt{IsLocal}, \texttt{LibrarySynced}\}, \{\texttt{ACMode}, \texttt{IsCloud}\}, \dots\}$$

Each precondition in $\mathbb{P}$ is a set of layers. To apply g, the context of the application must contains all the layers in υ, for a precondition $\upsilon \in \mathbb{P}$.

As we will see later on, our type system guarantees that the dispatching mechanism always succeeds at runtime. In our example, the expression (2.1) will be well-typed whenever the context in which it will be evaluated contains `IsLocal` or `IsCloud` and `LibraryUnsynced` or `LibrarySynced`. The requirements about `ACMode` and `BatMode` coming from exp_1 are ensured in (2.1). This is because the type of `getBatteryProfile` guarantees that one among them will be activated in the context by the **with**.

An effect U (a restriction of a usage in Section 1.3.4) represents (an over-approximation of) the sequences of events, i.e. of resource manipulation or layer activations or communication actions. The effect U in τ' is the latent effect of g, over-approximating the set of histories, i.e. the sequences of events, possibly generated by running g.

Effects are then used to check whether a client complies with the policy and the interaction protocol provided by the environment. Verifying that the code of g obeys the policy ϕ is done by standard model-checking the effect of g against the policy ϕ. Obviously, the app never writes, so the policy ϕ is satisfied, assuming that the code for the `BatMode` case has empty effect.

To check compliance with the protocol, we only consider communications. Thus, the effect of exp_1 becomes:

$$U_{sr} = send_{ty} \cdot send_{ty}[\tau'] \cdot receive_{ty}[\tau'']$$

Verifying whether the program correctly interacts with the cloud system consists of checking that the histories generated by U_{sr} are a subset of those allowed by the protocol P. In our scenario this is indeed the case.

2.2 ContextML: a context-oriented ML core

ContextML is a fragment of ML extended to deal with adaptation, providing us with mechanisms to change the context and to define behavioural variations in a functional style. We extend it by introducing resources manipulation, enforcement of security properties and communication.

Resources available in the system are represented by identifiers and can be manipulated by a fixed set of actions.

We enforce security properties by protecting an expression e through the *policy framing* $\phi[e]$ seen in Section 1.2. Roughly, it means that during the evaluation of e the computation must respect ϕ. Our policies turn out to be regular properties of computation histories; we delay the actual definition of policy compliance to Section 2.4.

The communication model is based on a bus which allows programs to interact with the environment by message passing. The operations of writing and reading values over this bus can be seen as a simple form of asynchronous I/O. We will not specify this bus in detail, but we will consider it as an abstract entity representing the whole external environment and its interactions with programs. Therefore, ContextML programs operate in an open-ended environment.

The syntax and the structural operational semantics of ContextML follow.

Syntax Let $\mathbb{N}$ be the naturals, Ide a set of identifiers, $\mathsf{LayerNames}$ a finite set of layer names, $\mathsf{Policies}$ a set of security policies, Σ_s a finite set of resources identifiers and Act a finite set of actions for manipulating resources. Then, the syntax of ContextML is:

$$n \in \mathbb{N} \qquad x, f \in \mathsf{Ide} \qquad L \in \mathsf{LayerNames}$$
$$\phi \in \mathsf{Policies} \qquad r \in \Sigma_s \qquad \alpha, \beta \in \mathsf{Act}$$

$$\begin{aligned}
v, v_1, v' &::= n \mid L \mid () \mid \lambda_f\, x \ \Rightarrow e \\
e, e_1, e' &::= \phi[e] \mid v \mid x \mid e_1 e_2 \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid e_1\ \mathbf{op}\ e_2 \mid \\
&\qquad \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \mid \mathbf{with}(e_1)\ \mathbf{in}\ e_2 \mid \mathbf{unwith}(e_1)\ \mathbf{in}\ e_2 \mid lexp \\
&\qquad \mathbf{send}_{\text{ty}}(e) \mid \mathbf{receive}_{\text{ty}} \mid \alpha(r) \mid aux \\
lexp &::= L.e \mid L.e, lexp \\
aux &::= \overline{\phi}[e] \mid \mathbf{with}(\overline{L})\ \mathbf{in}\ e_2 \mid \mathbf{unwith}(\overline{L})\ \mathbf{in}\ e_2
\end{aligned}$$

Additionally, we assume the syntactic sugar $e_1; e_2 \triangleq (\lambda_f x \Rightarrow e_2)\, e_1$ where x and f are not free in e_2.

The novelties of ContextML with respect to ML are primitives for handling resources, policy framing and communication and some features borrowed from COP languages (for their description we refer the reader to Section 1.1). Usually, COP paradigm have layers as expressible values; the (**unwith**) **with** construct for manipulating the context by (de)activating layers; layered expressions ($lexp$), defined by cases, each specifying a context-dependent behaviour. The expression $\alpha(r)$ indicates that we access the resource r through the action α, possibly causing side effects. The security properties are enforced by policy framing $\phi[e]$ guaranteeing that the computation satisfies the policy ϕ. Of course, policy framings can be nested. The communication is performed by $\mathbf{send}_{\text{ty}}$ and $\mathbf{receive}_{\text{ty}}$. They allow us to interact with the external environment by writing/reading values of type τ (see Section 2.3) to/from the bus. The auxiliary expressions (aux) are not intended to be used directly by the programmer, but they are used by our static and dynamic semantics only.

Dynamic Semantics We endow ContextML with a small-step operational semantics, only defined for closed expressions as usual. Note that, since ContextML programs can read values from the bus, a closed expression can be open with respect to the external environment. For example, **let** $x =$ **receive**$_{\text{ty}}$ **in** $x + 1$ is closed but it reads an unknown value v from the bus. To give meaning to such programs, we have an early input rule similar to that of the π-calculus [53].

Our semantics records the events occurring during program execution by cumulating a *history*, i.e. the sequence of such events. Events ev indicate (de)activation layers, selection of behavioural variations and program actions, be they resource accesses, entering/exiting policy framing and communication. Our semantics will be history dependent, in that the histories effect the semantics of a program when a policy framing is encountered, possibly halting the execution.

The syntax of events ev and programs histories η is the following:

$$ev ::= (\!|_L \mid |\!)_L \mid \{_L \mid \}_L \mid \mathsf{Disp}(L) \mid \alpha(r) \mid \mathit{send}_{ty} \mid \mathit{receive}_{ty} \mid [_\phi| \,]_\phi \tag{2.2}$$

$$\eta ::= \varepsilon \mid ev \mid \eta\,\eta \tag{2.3}$$

The event $(\!|_L$ ($|\!)_L$ respectively) marks that we begin (end), the evaluation of a **with** body in a context where the layer L is activated (deactivated). Symmetrically, the event $\{_L$ ($\}_L$, respectively) signals that we begin (end) the evaluation of a **unwith** body in a context where the layer L is masked (unmasked); the event $\mathsf{Disp}(L)$ signals that layer L has been selected by the dispatch mechanism; the event $\alpha(r)$ marks that the action α has been performed over the resource r; the event $\mathit{send}_{ty}/\mathit{receive}_{ty}$ indicates that we have sent/read a value of type τ over/from the bus; the event ($]_\phi$) $[_\phi$ marks that we begin (end, respectively) the enforcement of the policy ϕ.

A context C is a stack of layers, each layer appearing at most once. We denote by $|C|$ the set of active layer in the context C, i.e. the set of layers appearing on the stack. Whenever a layer L is in $|C|$, we will say that L is *active* in C.

We define now two operations on contexts: the first $C - L$ removes a layer L from the context C if present, the second $L :: C$ pushes L over $C - L$. Formally:

Definition 2.2.1. We denote the empty context by $[\,]$ and a context with n elements with top L_1 by $[L_1, \dots, L_n]$.
Let $C = [L_1, \dots, L_{i-1}, L_i, L_{i+1}, \dots, L_n], 1 \leq i \leq n$ then

$$C - L = \begin{cases} [L_1, \dots, L_{i-1}, L_{i+1}, \dots L_n] & \text{if } L = L_i \\ C & \text{otherwise} \end{cases}$$

Also, let $L :: C = [L, L_1, \dots, L_n]$ where $[L_1, \dots, L_n] = C - L$.

The transitions have the form $C \vdash \eta, e \to \eta', e'$, meaning that in the context C, starting from a program history η, the expression e may evolve to e' and the history η to η' in one evaluation step.

The semantic rules are shown in Figure 2.2, most of them are inherited from ML. We briefly comment on them.

The rules for $\mathbf{with}(e_1)$ **in** e_2 ($\mathbf{unwith}(e_1)$ **in** e_2, respectively) evaluate e_2 in a context where the layer obtained evaluating e_1 is activated (deactivated). Additionally, we store in the history the events $\llparenthesis_L$ and $\rrparenthesis_L$ ($\{_L$ and $\}_L$) marking the beginning and the end of the evaluation of e_2 (note that being within the scope of layer L activation is recorded by the auxiliary expressions $\mathbf{with}(\overline{L})$).

When a layered expression $e = L_1.e_1, \ldots, L_n.e_n$ has to be evaluated (rule lexp), the current context is inspected top-down to select the expression e_i to which e reduces. This dispatching mechanism is implemented by the partial function Dsp, defined as

$$Dsp([L'_0, L'_1, \ldots, L'_m], A) = \begin{cases} L'_0 & \text{if } L'_0 \in A \\ Dsp([L'_1, \ldots, L'_m], A) & \text{otherwise} \end{cases}$$

that returns the first layer in the context $[L'_0, L'_1, \ldots, L'_m]$ which matches one of the layers in the set A. If no layer matches, then the computation gets stuck.

The rule (action) establishes that performing an action α over a resource r yields the unit value () and extends η with $\alpha(r)$.

The rules governing communications reflect our notion of protocol, that abstractly represents the behaviour of the environment, showing the sequence of direction/type of messages. Accordingly, our primitives carry types as tags, rather than dynamically checking the exchanged values. In particular, there is no check that the type of the received value matches the annotation of the receive primitive. Our static analysis will guarantee the correctness of this operation.

In detail, $\mathbf{send}_{\mathrm{ty}}(e)$ evaluates e and sends the obtained value over the bus. Additionally, the history is extended with the event $send_{ty}$. A $\mathbf{receive}_{\mathrm{ty}}$ reduces to the value v read from the bus and appends the corresponding event to the current history. This rule is similar to that used in the early semantics of the π-calculus, where we guess a name transmitted over the channel [53].

The rules for framing say that an expression $\phi[e]$ can reduce to $\phi[e']$, provided that the resulting history η' obeys the policy ϕ, in symbols $\eta'^{-[]} \models \phi$ (see Section 1.3.4 and Section 2.4 for a precise definition). Also here, placing a bar over ϕ records that the policy is active. If η' does not obey ϕ, then the computation gets stuck. Of course, we store in the history through $[_\phi/]_\phi$ the point where we begin/end the enforcement of ϕ.

$$\text{if}_1\,\frac{C \vdash \eta, e_0 \to \eta', e_0'}{C \vdash \eta, \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 \to \eta', \textbf{if } e_0' \textbf{ then } e_1 \textbf{ else } e_2}$$

$$\text{if}_2\,\frac{}{C \vdash \eta, \textbf{if } 0 \textbf{ then } e_1 \textbf{ else } e_2 \to \eta, e_2} \qquad \text{if}_3\,\frac{v \neq 0}{C \vdash \eta, \textbf{if } v \textbf{ then } e_1 \textbf{ else } e_2 \to \eta, e_1}$$

$$\text{let}_1\,\frac{C \vdash \eta, e_1 \to \eta', e_1'}{C \vdash \eta, \textbf{let } x = e_1 \textbf{ in } e_2 \to \eta', \textbf{let } x = e_1' \textbf{ in } e_2}$$

$$\text{let}_2\,\frac{}{C \vdash \eta, \textbf{let } x = v \textbf{ in } e_2 \to \eta, e_2\{v/x\}} \qquad \text{op}_1\,\frac{C \vdash \eta, e_1 \to \eta', e_1'}{C \vdash \eta, e_1 \textbf{ op } e_2 \to \eta', e_1' \textbf{ op } e_2}$$

$$\text{op}_2\,\frac{C \vdash \eta, e_2 \to \eta', e_2'}{C \vdash \eta, v \textbf{ op } e_2 \to \eta', v \textbf{ op } e_2'} \qquad \text{op}_3\,\frac{v = v_1 \mathsf{\ op\ } v_2}{C \vdash \eta, v_1 \textbf{ op } v_2 \to \eta, v}$$

$$\text{app}_1\,\frac{C \vdash \eta, e_2 \to \eta', e_2'}{C \vdash \eta, e_1\, e_2 \to \eta', e_1\, e_2'} \qquad \text{app}_2\,\frac{C \vdash \eta, e_1 \to \eta', e_1'}{C \vdash \eta, e_1\, v \to \eta', e_1'\, v}$$

$$\text{app}_3\,\frac{}{C \vdash \eta, (\lambda_f\, x \Rightarrow e)v \to \eta, e\{\lambda_f\, x \Rightarrow e/f, v/x\}}$$

$$\text{with}_1\,\frac{C \vdash \eta, e_1 \to \eta', e_1'}{C \vdash \eta, \textbf{with}(e_1) \textbf{ in } e_2 \to \eta', \textbf{with}(e_1') \textbf{ in } e_2}$$

$$\text{with}_2\,\frac{}{C \vdash \eta, \textbf{with}(L) \textbf{ in } e \to \eta \llparenthesis_L, \textbf{with}(\bar{L}) \textbf{ in } e}$$

$$\text{with}_3\,\frac{L :: C \vdash \eta, e \to \eta', e'}{C \vdash \eta, \textbf{with}(\bar{L}) \textbf{ in } e \to \eta', \textbf{with}(\bar{L}) \textbf{ in } e'} \qquad \text{with}_4\,\frac{}{C \vdash \eta, \textbf{with}(\bar{L}) \textbf{ in } v \to \eta \rrparenthesis_L, v}$$

$$\text{unwith}_1\,\frac{C \vdash \eta, e_1 \to \eta', e_1'}{C \vdash \eta, \textbf{unwith}(e_1) \textbf{ in } e_2 \to \eta', \textbf{unwith}(e_1') \textbf{ in } e_2}$$

$$\text{unwith}_2\,\frac{}{C \vdash \eta, \textbf{unwith}(L) \textbf{ in } e \to \eta \{_L, \textbf{unwith}(\bar{L}) \textbf{ in } e}$$

$$\text{unwith}_3\,\frac{C - L \vdash \eta, e \to \eta', e'}{C \vdash \eta, \textbf{unwith}(\bar{L}) \textbf{ in } e \to \eta', \textbf{unwith}(\bar{L}) \textbf{ in } e'}$$

$$\text{unwith}_4\,\frac{}{C \vdash \eta, \textbf{unwith}(\bar{L}) \textbf{ in } v \to \eta \}_L, v} \qquad \text{lexp}\,\frac{L_i = Dsp(C, \{L_1, \ldots, L_n\})}{C \vdash \eta, L_1.e_1, \ldots, L_n.e_n \to \eta \mathsf{Disp}(L_i), e_i}$$

$$\text{action}\,\frac{}{C \vdash \eta, \alpha(r) \to \eta\, \alpha(r), ()} \qquad \text{send}_1\,\frac{C \vdash \eta, e \to \eta', e'}{C \vdash \eta, \textbf{send}_{\text{ty}}(e) \to \eta', \textbf{send}_{\text{ty}}(e')}$$

$$\text{send}_2\,\frac{}{C \vdash \eta, \textbf{send}_{\text{ty}}(v) \to \eta\; send_\tau, ()} \qquad \text{receive}\,\frac{}{C \vdash \eta, \textbf{receive}_{\text{ty}} \to \eta\; receive_\tau, v}$$

$$\text{framing}_1\,\frac{\eta^{-[]} \vDash \phi}{C \vdash \eta, \phi[e] \to \eta[_\phi, \overline{\phi}[e]} \qquad \text{framing}_2\,\frac{C \vdash \eta, e \to \eta', e' \qquad \eta'^{-[]} \vDash \phi}{C \vdash \eta, \overline{\phi}[e] \to \eta', \overline{\phi}[e']}$$

$$\text{framing}_3\,\frac{}{C \vdash \eta, \overline{\phi}[v] \to \eta]_\phi, v}$$

Figure 2.2: Semantic rules for new constructs

To better explain how does the evaluation of ContextML programs works, consider the following example of execution, referencing the code in Figure 2.1.

Example 2.2.1. *Assume that the layer IsCloud is active in the context and that* `getBatteryProfile()` *returns the layer ACMode, then we have the following transition:*

$$[IsCloud] \vdash \eta, \textbf{\textit{with}}(\texttt{getBatteryProfile}())\ \textbf{\textit{in}}\ exp_1 \to \eta', \textbf{\textit{with}}(ACMode)\ \textbf{\textit{in}}\ exp_1$$

The expression $\textbf{\textit{with}}(ACMode)\ \textbf{\textit{in}}\ exp_1$ *will now behave in the context* $[IsCloud]$ *as the expression* $search()$ *on line* 3 *of* exp_1.

Indeed, the layered expression semantics justifies the following transition

$$[ACMode, IsCloud] \vdash \begin{array}{l} \eta'', exp_2 = \\ \quad ACMode. \\ \qquad IsCloud.\ \ search(), \\ \qquad \vdots \\ \quad BatMode.\ \cdots \end{array} \longrightarrow \begin{array}{l} \eta''', exp_3 = \\ \quad ACMode. \\ \qquad IsCloud.\ \ e', \\ \qquad \vdots \\ \quad BatMode.\ \cdots \end{array}$$

where e' *is obtained by evaluating the expression* $search()$.

2.3 ContextML types

We provide here ContextML with a type and effect system. We use it for over-approximating the behaviour of a program and for ensuring that the dispatch mechanism always succeeds at runtime. The associated effect is a variant of the Usages in Definition 1.3.11, where the νn primitive is not used and the actions $\alpha(r)$ are the events ev in Equation (2.2).

Our type and effect system does not use the νn primitive of Usages because the number of symbols involved in all the computations of a program is finite: the resources r are finite in Σ_s, the layers appearing in a program are finite. In Section 2.5 we will show an extension of ContextML that will be able to use the (possibly unbounded) resources available in the various contexts, the abstraction of which behaviour needs the νn primitive.

Recall from [7] the partial ordering $U \sqsubseteq U'$ on the usages U, U' defined over the quotient induced by the (semantic preserving) equational theory in Figure 2.3. It is the least relation such that $U \sqsubseteq U$ and $U \sqsubseteq U + U'$. It holds that $U \sqsubseteq U'$ implies $[\![U]\!] \subseteq [\![U']\!]$. Intuitively $U \sqsubseteq U'$ means that U is a more precise approximation than U'. Note that the theory is not complete, i.e. $[\![U]\!] \subseteq [\![U']\!]$ does not implies $U \sqsubseteq U'$. Moreover, recall from Section 1.3.4 that the semantics of a usage is a prefix closed set of histories.

$$H + H \equiv H \equiv \varepsilon \cdot H \equiv H \cdot \varepsilon \qquad H_1 + H_2 \equiv H_2 + H_1$$
$$H_1 \cdot (H_2 \cdot H_3) \equiv (H_1 \cdot H_2) \cdot H_3 \qquad H_1 + (H_2 + H_3) \equiv (H_1 + H_2) + H_3$$
$$H_1 \cdot (H_2 + H_3) \equiv (H_1 \cdot H_2) + (H_1 \cdot H_3) \qquad (H_1 + H_2) \cdot H_3 \equiv (H_1 \cdot H_3) + (H_2 \cdot H_3)$$

Figure 2.3: Equational theory on the Usages

Back to the example in Section 2.1, assume that U is a usage over-approximating the behaviour of function g. Then, the usage U' of the fragment of the cloud service in Figure 2.1(b) is

$$U' = \mu h.receive_{ty}[\tau] \cdot receive_{ty}[\tau'] \cdot \phi\left[({}_{\texttt{ACMode}} \cdot U \cdot send_{ty}[\tau''])_{\texttt{ACMode}}\right] \cdot h$$

assuming $\tau = ly_{\texttt{ACMode}}$.

The semantics of ContextML (in particular the rules for framing) ensures that, once fixed policy compliance $\eta \vDash \phi$ (provided in Definition 2.4.1), the histories generated at runtime are all valid, in the sense of Equation (1.1).

Property 2.3.1. *If $C \vdash \varepsilon, e \rightarrow \eta', e'$, then η' is valid.*

We give now the logical presentation of our type and effect system. We are confident that an inference algorithm can be developed, along the lines of [57]. Our typing judgements have the form $\langle \Gamma; C \rangle \vdash e : \tau \triangleright U$. This means that in "in the type environment Γ and in the context C the expression e has type τ and effect U".

Our type system has types for representing integers, unit, layers and functions:

$$\sigma \in \wp(\mathsf{LayerNames}) \qquad \mathbb{P} \in \wp\left(\wp(\mathsf{LayerNames})\right)$$
$$\tau, \tau_1, \tau' ::= \texttt{int} \mid \texttt{unit} \mid ly_\sigma \mid \tau_1 \xrightarrow{\mathbb{P}|U} \tau_2$$

We annotate types with sets of layer names σ for analysis reason. In ly_σ, σ over-approximates the set of layers that an expression can be reduced to at runtime. In $\tau_1 \xrightarrow{\mathbb{P}|U} \tau_2$, $\mathbb{P}$ is a set of *preconditions* v. Each $v \in \mathbb{P}$ over-approximates the set of layers that must occur in the context to apply the function. The usage U is the latent effect, i.e. the sequence of events generated while evaluating the function.

Figure 2.4 introduces the rules for subtyping ($\tau_1 \leq \tau_2$). The rule (Sref) states that the subtyping relation is reflexive. The rule (Sly) says that a layer type

ly_σ is a subtype of $ly_{\sigma'}$ whenever the annotation σ is a subset of σ'. The rule (Sfun) defines subtyping for functional types. As usual, it is contravariant in τ_1 but covariant in $\mathbb{P}, \tau_2$ and U. The notion of subeffecting ($U \sqsubseteq U'$) is the one introduced at the beginning of this section. The ordering on the set of preconditions is defined as follows $\mathbb{P} \sqsubseteq \mathbb{P}'$ iff $\forall \upsilon \in \mathbb{P} \,.\, \exists \upsilon' \in \mathbb{P}' \,.\, \upsilon' \subseteq \upsilon$. By the (Tsub) rule, we can always enlarge types and effects.

$$\text{(Sref)}\frac{}{\tau \leq \tau} \qquad \text{(Sfun)}\frac{\tau_1' \leq \tau_1 \quad \tau_2 \leq \tau_2' \quad \mathbb{P} \sqsubseteq \mathbb{P}' \quad U \sqsubseteq U'}{\tau_1 \xrightarrow{\mathbb{P}|U} \tau_2 \leq \tau_1' \xrightarrow{\mathbb{P}'|U'} \tau_2'}$$

$$\text{(Sly)}\frac{\sigma \subseteq \sigma'}{ly_\sigma \leq ly_{\sigma'}} \qquad \text{(Tsub)}\frac{\langle\Gamma; C\rangle \vdash e : \tau' \triangleright U' \quad \tau' \leq \tau \quad U' \sqsubseteq U}{\langle\Gamma; C\rangle \vdash e : \tau \triangleright U}$$

Figure 2.4: Subtyping rules

Figure 2.5 shows the rules of our type and effect system. We only comment in detail on the rules for the non standard constructs.

The rule (Talpha) gives expression $\alpha(r)$ type `unit` and effect $\alpha(r)$. The rule (Tly) asserts that the type of a layer L is ly annotated with the singleton set $\{L\}$ and its effect is empty. In the rule (Tfun) we guess a set of preconditions $\mathbb{P}$, a type for the bound variable x and for the function f . For all precondition $\upsilon \in \mathbb{P}$ we also guess a context C' such that it does not contain any additional layer except those specified by the precondition υ. We determine the type of the body e under these additional assumptions. Implicitly, we require that the guessed type for f, as well as its latent effect U, match that of the resulting function. Additionally, we require that the resulting type is annotated with $\mathbb{P}$.

The rule (Tapp) is almost standard and reveals the mechanism of function precondition. The application gets a type if there exists a precondition $\upsilon \in \mathbb{P}$ such that it is satisfied in the current context C. A context satisfies the precondition υ whenever it contains all the layers in υ, in symbols $|C'| \supseteq \upsilon$, where $|C'|$ denotes the set of layers active in the context C'. The effect is obtained by concatenating the ones of e_2 and e_1 and the latent effect U. To better explain how preconditions work, consider the technical example in Figure 2.6. There, the function $\lambda_f\ x \Rightarrow L_1.0$ is shown having type $int \xrightarrow{\{L_1\}} int$ (for the sake of simplicity we ignore the effects). This means that L_1 must be in the context in order to apply the function.

The rule (Twith) establishes that the expression **with**(e_1) **in** e_2 has type τ, provided that the type for e_1 is ly_σ (recall that σ is a set of layers) and e_2 has type τ in the context C extended by the layers in σ. The effect is the union of the possible effects resulting from evaluating the body. This evaluation is carried on the different contexts obtained by extending C with one of the layers

$$(\text{TVar})\,\frac{\Gamma(x)=\tau}{\langle\Gamma;C\rangle\vdash x:\tau\rhd\varepsilon}\qquad(\text{Tint})\,\frac{}{\langle\Gamma;C\rangle\vdash n:\texttt{int}\rhd\varepsilon}$$

$$(\text{Tunit})\,\frac{}{\langle\Gamma;C\rangle\vdash ():\texttt{unit}\rhd\varepsilon}\qquad(\text{Tly})\,\frac{}{\langle\Gamma;C\rangle\vdash L:ly_{\{L\}}\rhd\varepsilon}$$

$$(\text{Talpha})\,\frac{}{\langle\Gamma;C\rangle\vdash\alpha(a):\texttt{unit}\rhd\alpha(a)}$$

$$(\text{Tfun})\,\frac{\forall v\in\mathbb{P}.\quad\langle\Gamma,x:\tau_1,f:\tau_1\xrightarrow{\mathbb{P}|U}\tau_2;C'\rangle\vdash e:\tau_2\rhd U\qquad v\supseteq|C'|}{\langle\Gamma;C\rangle\vdash\lambda_f\;x\Rightarrow e:\tau_1\xrightarrow{\mathbb{P}|U}\tau_2\rhd\varepsilon}$$

$$(\text{Tlet})\,\frac{\langle\Gamma;C\rangle\vdash e_1:\tau_1\rhd U\qquad\langle\Gamma,x:\tau_1,C\rangle\vdash e_2:\tau_2\rhd U'}{\langle\Gamma;C\rangle\vdash\textbf{let }x=e_1\textbf{ in }e_2:\tau_2\rhd U\cdot U'}$$

$$(\text{Tif})\,\frac{\langle\Gamma;C\rangle\vdash e_0:int\rhd U\qquad\langle\Gamma;C\rangle\vdash e_1:\tau\rhd U_1\qquad\langle\Gamma;C\rangle\vdash e_2:\tau\rhd U_2}{\langle\Gamma;C\rangle\vdash\textbf{if}\;\;e_0\;\;\textbf{then }e_1\textbf{ else }e_2:\tau\rhd U\cdot(U_1+U_2)}$$

$$(\text{Twith})\,\frac{\langle\Gamma;C\rangle\vdash e_1:ly_{\{L_1,\dots,L_n\}}\rhd U'\qquad\forall L_i\in\{L_1,\dots,L_n\}.\langle\Gamma;L_i::C\rangle\vdash e_2:\tau\rhd U_i}{\langle\Gamma;C\rangle\vdash\textbf{with}(e_1)\textbf{ in }e_2:\tau\rhd U'\cdot\sum_{L_i}(\!|_{L_i}\cdot U_i\cdot|\!)_{L_i}}$$

$$(\text{Tunwith})\,\frac{\langle\Gamma;C\rangle\vdash e_1:ly_{\{L_1,\dots,L_n\}}\rhd U'\qquad\forall L_i\in\{L_1,\dots,L_n\}.\langle\Gamma;C-L_i\rangle\vdash e_2:\tau\rhd U_i}{\langle\Gamma;C\rangle\vdash\textbf{unwith}(e_1)\textbf{ in }e_2:\tau\rhd U'\cdot\sum_{L_i}\{_{L_i}\cdot U_i\cdot\}_{L_i}}$$

$$(\text{Tlexp})\,\frac{\forall i.\langle\Gamma;C\rangle\vdash e_i:\tau\rhd U_i\qquad L_1\in|C|\vee\cdots\vee L_n\in|C|}{\langle\Gamma;C\rangle\vdash L_1.e_1,\dots,L_n.e_n:\tau\rhd\displaystyle\sum_{L_i\in\{L_1,\dots,L_n\}}\mathsf{Disp}(L_i)\cdot U_i}$$

$$(\text{Tapp})\,\frac{\langle\Gamma;C\rangle\vdash e_1:\tau_1\xrightarrow{\mathbb{P}|U}\tau_2\rhd U_1\qquad\langle\Gamma;C\rangle\vdash e_2:\tau_1\rhd U_2\qquad\exists v\in\mathbb{P}.v\subseteq|C|}{\langle\Gamma;C\rangle\vdash e_1e_2:\tau_2\rhd U_2\cdot U_1\cdot U}$$

$$(\text{Top})\,\frac{\langle\Gamma;C\rangle\vdash e_1:\texttt{int}\rhd U_1\qquad\langle\Gamma;C\rangle\vdash e_2:\texttt{int}\rhd U_2}{\langle\Gamma;C\rangle\vdash e_1\textbf{ op }e_2:\texttt{int}\rhd U_1\cdot U_2}$$

$$(\text{Tphi})\,\frac{\langle\Gamma;C\rangle\vdash e:\tau\rhd U}{\langle\Gamma;C\rangle\vdash\phi[e]:\tau\rhd[_\phi\cdot U\cdot]_\phi}\qquad(\text{Trec})\,\frac{}{\langle\Gamma;C\rangle\vdash\textbf{receive}_{\text{ty}}:\tau\rhd receive_{ty}}$$

$$(\text{Tsend})\,\frac{\langle\Gamma;C\rangle\vdash e:\tau\rhd U\qquad U'=U\cdot send_{ty}}{\langle\Gamma;C\rangle\vdash\textbf{send}_{\text{ty}}(e):\texttt{unit}\rhd U'}$$

Figure 2.5: Typing rules for ContextML

$$\dfrac{\dfrac{\dfrac{\langle \Gamma, x:\tau, f:\tau \xrightarrow{\{|C'|\}} \tau; C'\rangle \vdash 0:\tau \qquad L_1 \in C'}{\langle \Gamma, x:\tau, f:\tau \xrightarrow{\{|C'|\}} \tau; C'\rangle \vdash L_1.0:\tau}}{\langle \Gamma; C\rangle \vdash \lambda_f\, x \Rightarrow L_1.0 : \tau \xrightarrow{\{|C'|\}} \tau} \qquad \dfrac{\begin{array}{c}\langle \Gamma, g:\tau \xrightarrow{\{|C'|\}} \tau; C\rangle \vdash g:\tau \to \tau \\ \langle \Gamma, g:\tau \xrightarrow{\{|C'|\}} \tau; C\rangle \vdash 3:\tau \qquad |C'| \subseteq |C|\end{array}}{\langle \Gamma, g:\tau \xrightarrow{\{|C'|\}} \tau; C\rangle \vdash g\,3:\tau}}{\langle \Gamma; C\rangle \vdash \mathbf{let}\ g = \lambda_f\, x \Rightarrow L_1.0\ \mathbf{in}\ g\,3 : \tau}$$

Figure 2.6: Derivation of a function with precondition. We assume that $C' = [L_1]$, L_1 is active in C, $\mathsf{LayerNames} = \{L_1\}$ and, for typesetting convenience, we also denote $\tau = int$ and we ignore effects.

in σ. The special events $\llparenthesis_L$ and $\rrparenthesis_L$ express the scope of this layer activation. The rule (Tunwith) is similar to (Twith), but instead removes the layers in σ and use $\{_L$ and $\}_L$ to delimit layer hiding.

By (Tlexp) the type of a layered expression is τ, provided that each sub-expression e_i has type τ and that at least one among the layers $L_1, \ldots L_n$ occurs in C. When evaluating a layered expression one of the mentioned layers will be active in the current context so guaranteeing that layered expressions will correctly evaluate. The whole effect is the sum of sub-expressions effects U_i preceded by $Disp(L_i)$.

The expression $\mathbf{send}_{\mathrm{ty}}(e)$ has type `unit` and its effect is that of e extended with event $send_{ty}$. The expression $\mathbf{receive}_{\mathrm{ty}}$ has type τ and its effect is the event $receive_{ty}$. Note that the rules establish the correspondence between the type declared in the syntax and the checked type of the value sent/received. An additional check is however needed and will be carried on also taking care of the interaction protocol (Section 2.4).

For technical reasons, we need the following rules dealing with the auxiliary syntactic constructs.

$$(\text{Tbphi})\dfrac{\langle \Gamma; C\rangle \vdash e:\tau \triangleright U}{\langle \Gamma; C\rangle \vdash \overline{\phi}[e]:\tau \triangleright U\cdot]_\phi} \qquad (\text{Tbwith})\dfrac{\langle \Gamma; L :: C\rangle \vdash e_2:\tau \triangleright U}{\langle \Gamma; C\rangle \vdash \mathbf{with}(\overline{L})\ \mathbf{in}\ e_2:\tau \triangleright U\cdot\rrparenthesis_L}$$

$$(\text{Tbunwith})\dfrac{\langle \Gamma; C - L\rangle \vdash e_2:\tau \triangleright U}{\langle \Gamma; C\rangle \vdash \mathbf{unwith}(\overline{L})\ \mathbf{in}\ e_2:\tau \triangleright U\cdot\}_L}$$

Our type system enjoys the following soundness results, the proofs of which are delayed to the next sub-section. Given a history η and a usage U, by abuse of notation, ηU will denote the usage obtained by the sequentialization of η and U, where η stands for the usage obtained by sequentialization of all the events in the history η.

Theorem 2.3.2 (Subject reduction)**.** *Let e be a closed expression, if $\langle\Gamma; C\rangle \vdash e : \tau \triangleright U$ and $C \vdash \eta, e \to \eta', e'$, then $\langle\Gamma; C\rangle \vdash e' : \tau \triangleright U'$ with $\eta U \sqsupseteq \eta' U'$*

As a corollary we get that the usage obtained as effect of an expression e over-approximates the set of histories that may actually be generated during the execution of e.

Corollary 2.3.3 (Over-approximation)**.** *If $\langle\Gamma; C\rangle \vdash e : \tau \triangleright U$ and $C \vdash \varepsilon, e \to^* \eta, e'$, then $\eta \in [\![U]\!]$.*

We also have the following result, where $C \vdash \eta, e \not\to$ means that e is stuck.

Theorem 2.3.4 (Progress)**.** *Let e be a closed expression such that $\langle\Gamma; C\rangle \vdash e : \tau \triangleright U$. If $C \vdash \eta, e \not\to$ and ηU is valid, then e is a value.*

Subject reduction and progress prove the soundness of our type system.

Corollary 2.3.5. *If $\langle\emptyset; C\rangle \vdash e : \tau \triangleright U$ and U is valid, then e never gets stuck, i.e. $C \vdash \varepsilon, e \to^* \eta', e'$ with e' not a value implies $C \vdash \eta', e' \to \eta'', e''$.*

This corollary guarantees that a well-typed expression e will never get stuck because of policy violations, provided that its effect U is valid and that e complies with the communication protocol.

2.3.1 Proofs

In this section we give some additional definitions and we prove progress and subject reduction theorems.

Definition 2.3.1 (Substitution)**.** Given the expressions e,e' and the identifier x we define $e\{e'/x\}$ in Table 2.1.

Lemma 2.3.6 (Substitution Lemma)**.** *Let $\langle\Gamma, x : \tau'; C\rangle \vdash e : \tau \triangleright U$ and $\langle\Gamma; C\rangle \vdash v : \tau' \triangleright \varepsilon$ then $\langle\Gamma; C\rangle \vdash e\{v/x\} : \tau \triangleright U$*

Proof. We prove the lemma by structural induction over the expressions.

- *Case $e = \phi[e_1]$.*
  By the precondition of the rule (Tphi) $\langle\Gamma, x : t'; C\rangle \vdash e_1 : \tau \triangleright U'$ holds where $U = [_\phi U']_\phi$. By Definition 2.3.1 we know that $\phi[e_1]\{v/x\} = \phi[e_1\{v/x\}]$, hence by the applying inductive hypothesis $\langle\Gamma; C\rangle \vdash e_1\{v/x\} : \tau \triangleright U'$. So by the rule (Tphi) and by the Definition 2.3.1 we can conclude that $\langle\Gamma; C\rangle \vdash \phi[e_1]\{v/x\} : \tau \triangleright U$ holds.

$$n\{e'/x\} = n, \qquad L\{e'/x\} = L, \qquad ()\{e'/x\} = ()$$

$$(\lambda_f x' \Rightarrow e)\{e'/x\} = \lambda_f x \Rightarrow e\{e'/x\} \quad \text{if } f \neq x \wedge x \neq x' \wedge f, x' \notin fv(e')$$

$$x\{e'/x\} = e', \qquad x'\{e'/x\} = x' \quad \text{if } x' \neq x$$

$$(e_1\, e_2)\{e'/x\} = e_1\{e'/x\}e_2\{e'/x\}$$

$$(e_1\, op\, e_2)\{e'/x\} = e_1\{e'/x\}\, op\, e_2\{e'/x\}$$

$$(\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2)\{e'/x\} = \textbf{if } e_0\{e'/x\} \textbf{ then } e_1\{e'/x\} \textbf{ else } e_2\{e'/x\}$$

$$(\textbf{with}(e_1) \textbf{ in } e_2)\{e'/x\} = \textbf{with}(e_1\{e'/x\}) \textbf{ in } e_2\{e'/x\}$$

$$(\textbf{unwith}(e_1) \textbf{ in } e_2)\{e'/x\} = \textbf{unwith}(e_1\{e'/x\}) \textbf{ in } e_2\{e'/x\}$$

$$(\phi[e])\{e'/x\} = \phi[e\{e'/x\}] \qquad \alpha(r)\{e'/x\} = \alpha(r)$$

$$(\textbf{send}_{\text{ty}}(e))\{e'/x\} = \textbf{send}_{\text{ty}}(e\{e'/x\}) \qquad (\textbf{receive}_{\text{ty}})\{e'/x\} = \textbf{receive}_{\text{ty}}$$

$$(L_1.e_1, \ldots, L_n.e_n)\{e'/x\} = L_1.e_1\{e'/x\}, \ldots, L_n.e_n\{e'/x\}$$

Table 2.1: Definition of substitution for ContextML expressions

- *Case* $e = \textbf{\textit{with}}(e_1)$ ***in*** e_2.
  By the precondition of the rule (Twith) we know that $\langle \Gamma, x : \tau'; C\rangle \vdash e_1 : ly_{\{L_1,\ldots,L_n\}} \triangleright U'$ and $\forall L_i \in \{L_1, \ldots, L_n\}.\langle \Gamma, x : \tau'; L_i :: C\rangle \vdash e_2 : \tau \triangleright U_i$ hold where $U = U' \cdot \sum_{L_i} (\!|_{L_i} \cdot U_i \cdot |\!)_{L_i}$. By the Definition 2.3.1 $(\textbf{with}(e_1) \textbf{ in } e_2)\{v/x\} = \textbf{with}(e_1\{v/x\}) \textbf{ in } e_2\{v/x\}$. By the inductive hypothesis $\langle \Gamma, x : \tau'; C\rangle \vdash e_1\{v/x\} : ly_{\{L_1,\ldots,L_n\}} \triangleright U'$ and $\forall L_i \in \{L_1, \ldots, L_n\}.\langle \Gamma, x : \tau'; L_i :: C\rangle \vdash e_2\{v/x\} : \tau \triangleright U_i$ hold. So by the rule (Twith) and by Definition 2.3.1 we can conclude that $\langle \Gamma; C\rangle \vdash (\textbf{with}(e_1) \textbf{ in } e_2)\{v/x\} : \tau \triangleright U$.

- *Case* $e = \textbf{\textit{unwith}}(e_1)$ ***in*** e_2.
  Similar to the case for $e = \textbf{with}(e_1) \textbf{ in } e_2$.

- *Case* $e = L_1.e_1, \ldots, L_n.e_n$.
  By the rule (Tlexp) we know that $\forall i \in \{L_1, \ldots, L_n\} \quad \langle \Gamma, x : \tau'; C\rangle \vdash e_i : \tau \triangleright U_i$, $L_1 \in |C| \vee \cdots \vee L_n \in |C|$ and $U = \sum_{L_i \in \{L_1,\ldots,L_n\}} \mathsf{Disp}(L_i) \cdot U_i$. By Definition 2.3.1 we know that $(L_1.e_1, \ldots, L_n.e_n)\{v/x\}$. By the inductive hypothesis we can state that $\forall i \in \{L_1, \ldots, L_n\}$ $\langle \Gamma; C\rangle \vdash e_i\{v/x\} : \tau \triangleright U_i$.

Since $L_1 \in |C| \vee \cdots \vee L_n \in |C|$ by the rule (Tlexp) and by Definition 2.3.1 we can conclude that $\langle \Gamma; C\rangle \vdash (L_1.e_1, \dots, L_n.e_n)\{v/x\} : \tau \triangleright U$ hold.

- *Case* $e = \boldsymbol{send}_{ty}(e_1)$.
  By the rule (Tsend) we have that $\langle \Gamma, x : \tau'; C\rangle \vdash \mathbf{send}_{\mathrm{ty}}(e_1) : \tau \triangleright U'$ holds with $U = U' \cdot send_{ty}$. By the Definition 2.3.1 $\mathbf{send}_{\mathrm{ty}}(e_1)\{v/x\} = \mathbf{send}_{\mathrm{ty}}(e_1\{v/x\})$ and by inductive hypothesis that $\langle \Gamma; C\rangle \vdash e_1\{v/x\} : \tau \triangleright U'$. So by the Definition 2.3.1 and by the rule (Tsend) we can conclude that $\langle \Gamma; C\rangle \vdash \mathbf{send}_{\mathrm{ty}}(e_1)\{v/x\} : \tau \triangleright U$.

- *Case* $e = n$, $e = L$, $e = ()$, $e = \alpha(r)$, $e = \mathrm{receive}_{\mathrm{ty}}$.
  Since by the Definition 2.3.1 holds for these cases that $e\{v/x\} = e$ the property holds.

- The other cases are standard and we do not show them.

□

Property 2.3.7. *If* $\langle \Gamma; C\rangle \vdash e : \tau \triangleright U$, *then* $\forall C'$ *s.t.* $|C'| \supseteq |C|$ *it holds* $\langle \Gamma; C'\rangle \vdash e : \tau \triangleright U$

Proof. By straightforward induction on the shape of the typing derivation. □

Lemma 2.3.8 (Decomposition). $\langle \Gamma; C\rangle \vdash \lambda_f x \Rightarrow e : \tau \xrightarrow{\mathbb{P}|U} \tau' \triangleright U'$ *and* $\exists v \in \mathbb{P}.\, |C| \supseteq v$ *implies* $\langle \Gamma, x : \tau, f : \tau \xrightarrow{\mathbb{P}|U} \tau'; C\rangle \vdash e : \tau' \triangleright U$.

Proof. Sketch: By the rule (Tfun) we know that for all $v \in \mathbb{P}$ there exists a guessed context C' such that $|C'| \subseteq v$ and $\langle \Gamma, x : \tau, f : \tau \xrightarrow{\mathbb{P}|U} \tau'; C'\rangle \vdash e : \tau' \triangleright U$. Since $|C'| \subseteq |C|$, thesis follows by Property 2.3.7.

□

We prove the following additional property on the partial ordering of Usages.

Property 2.3.9. *Given the usages* U_1, U_2, U'_1 *and* U'_2 *such that* $U_1 \sqsubseteq U'_1$ *and* $U_2 \sqsubseteq U'_2$ *then* $U_1 \cdot U_2 \sqsubseteq U'_1 \cdot U'_2$

Proof. By the definition of $\sqsubseteq$ over Usages we know that $U'_1 = U_1 + U_3$ for some U_3 and $U'_2 = U_2 + U_4$ for some U_4. By exploiting the equational properties of Usages in Figure 2.3

$$\begin{aligned} U'_1 \cdot U'_2 &= (U_1 + U_3) \cdot (U_2 + U_4) \\ &= (U_1 \cdot (U_2 + U_4)) + (U_3 \cdot (U_2 + U_4)) \\ &= U_1 \cdot U_2 + U_1 \cdot U_4 + U_3 \cdot U_2 + U_3 \cdot U_4 \end{aligned}$$

So we can conclude that $U_1 \cdot U_2 \sqsubseteq U_1' \cdot U_2'$. □

Lemma 2.3.10. *If $\langle \Gamma; C \rangle \vdash v : \tau \triangleright U$ then $\forall C', \Gamma' \supseteq \Gamma$ we have that $\langle \Gamma'; C' \rangle \vdash v : \tau \triangleright U$.*

Proof. By straightforward induction on the shape of the typing derivation. □

Lemma 2.3.11 (Canonical form)**.** *1. If v is a value such that $\langle \Gamma; C \rangle \vdash v : ly_{\{L_1,\ldots,L_n\}} \triangleright U$, then $v \in \{L_1, \ldots, L_n\}$*

2. If v is a value such that $\langle \Gamma; C \rangle \vdash v : \tau_1 \xrightarrow{\mathbb{P}|U} \tau_2 \triangleright U'$, then it is of the form $v = \lambda_f x \Rightarrow e$.

Proof.

1. The type layer with annotation $\{L_1, \ldots, L_n\}$ can only be applying the subtyping rule (Tsub) only starting from a layer type annotated with a singleton $\{L\}$ for some $L \in \{L_1, \ldots, L_n\}$. This can be obtained by (Tly) rule only, hence $v = L$.

2. Standard.

□

We now recall and prove Theorem 2.3.2.

Theorem 2.3.2 (Subject reduction)**.** *Let e be a closed expression, if $\langle \Gamma; C \rangle \vdash e : \tau \triangleright U$ and $C \vdash \eta, e \to \eta', e'$, then $\langle \Gamma; C \rangle \vdash e' : \tau \triangleright U'$ with $\eta U \sqsupseteq \eta' U'$*

Proof. By induction on the depth of the typing derivation, and then by cases on the last rule applied. We only work out the relevant cases, the proof of the others being trivial.

- Case (TApp):

$$\text{(Tapp)}\frac{\langle \Gamma; C \rangle \vdash e_1 : \tau_1 \xrightarrow{\mathbb{P}|U} \tau_2 \triangleright U_1 \qquad \langle \Gamma; C \rangle \vdash e_2 : \tau_1 \triangleright U_2 \qquad \exists v \in \mathbb{P}.v \subseteq |C|}{\langle \Gamma; C \rangle \vdash e_1 e_2 : \tau_2 \triangleright U_2 \cdot U_1 \cdot U}$$

  – Case e_1 not value, $e_2 = v$.
    The transition $C \vdash \eta, e_1 e_2 \to \eta', e_1' e_2$ has been deduced with the (app$_2$) rule, the premise of which is $C \vdash \eta, e_1 \to \eta', e_1'$. We prove that $\langle \Gamma; C \rangle \vdash e_1' e_2 : \tau_2 \triangleright U'$ with $\eta U_2 \cdot U_1 \cdot U \sqsupseteq \eta' U'$. By the inductive hypothesis $\langle \Gamma; C \rangle \vdash e_1' : \tau_1 \xrightarrow{\mathbb{P}|U} \tau_2 \triangleright U_1'$ with $\eta U_1 \sqsupseteq \eta' U_1'$. This implies the thesis because then $\langle \Gamma; C \rangle \vdash e_1' e_2 : \tau_2 \triangleright U_2 \cdot U_1' \cdot U$ and by Property 2.3.9 $\eta U_2 \cdot U_1 \cdot U \sqsupseteq \eta' U_2 \cdot U_1' \cdot U$

– Case e_1, e_2 not value: similar to the proof above.

– Case $e_1 = \lambda_f x \Rightarrow e$ and $e_2 = v$.
By the typing rules $U_1 = \varepsilon$ and $U_2 = \varepsilon$, hence by the equational theory on the Usages $U_2 \cdot U_1 \cdot U = U$. The transition $C \vdash \eta, e_1 e_2 \rightarrow \eta, e\{v/x, e_1/f\}$ has been deduced with (app$_3$) rule. We have to prove that $\langle \Gamma; C\rangle \vdash e\{v/x, e_1/f\} : \tau_2 \triangleright U'$ with $\eta U_2 \cdot U_1 \cdot U = \eta U \sqsupseteq \eta U'$. By the typing rule and Lemma 2.3.8 we have that $\langle \Gamma, x : \tau_1, f : \tau_1 \xrightarrow{\mathbb{P}|U} \tau_2; C\rangle \vdash e : \tau_2 \triangleright U$. The thesis follows because we have that $\langle \Gamma; C\rangle \vdash e\{v/x, e_1/f\} : \tau_2 \triangleright U$ by Lemma 2.3.6, showing $U' = U$.

- Case (Tlexp):

$$\text{(Tlexp)}\frac{\forall i.\langle \Gamma; C\rangle \vdash e_i : \tau \triangleright U_i \qquad L_1 \in |C| \vee \cdots \vee L_n \in |C|}{\langle \Gamma; C\rangle \vdash L_1.e_1, \ldots, L_n.e_n : \tau \triangleright \sum_{L_i \in \{L_1,\ldots,L_n\}} \mathsf{Disp}(L_i) \cdot U_i}$$

The transition $C \vdash \eta, L_1.e_1, \ldots, L_n.e_n \rightarrow \eta\,\mathsf{Disp}(L_i), e_i$ has been deduced, the premise of which is $\langle \Gamma; C\rangle \vdash e_i : \tau : U_i$. The thesis follows because $\eta \sum_{L_i \in \{L_1,\ldots,L_n\}} \mathsf{Disp}(L_i) \cdot U_i \sqsupseteq \eta \mathsf{Disp}(L_i) U_1$ by Property 2.3.9.

- Case (Twith):

$$\text{(Twith)}\frac{\langle \Gamma; C\rangle \vdash e_1 : ly_{\{L_1,\ldots,L_n\}} \triangleright U' \qquad \forall L_i \in \{L_1, \ldots, L_n\}.\langle \Gamma; L_i :: C\rangle \vdash e_2 : \tau \triangleright U_i}{\langle \Gamma; C\rangle \vdash \mathbf{with}(e_1)\ \mathbf{in}\ e_2 : \tau \triangleright U' \cdot \sum_{L_i} (\!|_{L_i} \cdot U_i \cdot |\!)_{L_i}}$$

– Case e_1 not a value.
The transition $C \vdash \eta, \mathbf{with}(e_1)\ \mathbf{in}\ e_2 \rightarrow \eta', \mathbf{with}(e_1')\ \mathbf{in}\ e_2$ has been deduced with the rule (with$_1$), the premises of which is $C \vdash \eta, e_1 \rightarrow \eta', e_1'$. By the inductive hypothesis $\langle \Gamma; C\rangle \vdash e_1 : ly_{\{L_1,\ldots,L_n\}} \triangleright U''$ with $\eta U' \sqsupseteq \eta' U''$. Thesis follows because $\langle \Gamma; C\rangle \vdash \mathbf{with}(e_1')\ \mathbf{in}\ e_2 : \tau \triangleright U'' \cdot \sum_{L_i} (\!|_{L_i} \cdot U_i \cdot |\!)_{L_i}$ with $\eta U' \cdot \sum_{L_i} (\!|_{L_i} \cdot U_i \cdot |\!)_{L_i} \sqsupseteq \eta' U'' \cdot \sum_{L_i} (\!|_{L_i} \cdot U_i \cdot |\!)_{L_i}$ by Property 2.3.9.

– Case $e_1 = L$ (i.e. is a value), e_2 is not.
By Lemma 2.3.11 it is only the case that $L = L_i$ for some $L_i \in \{L_1, \ldots, L_n\}$ and by the typing rule (Tly) we have that $U' = \varepsilon$. The transition $C \vdash \eta, \mathbf{with}(L_i)\ \mathbf{in}\ e_2 \rightarrow \eta\ (\!|_{L_i}, \mathbf{with}(\overline{L_i})\ \mathbf{in}\ e_2$ has been deduced with the rule (with$_2$). The thesis follows because $\langle \Gamma; C\rangle \vdash \mathbf{with}(\overline{L_i})\ \mathbf{in}\ e_2 : \tau \triangleright \cdot U_i \cdot |\!)_{L_i}$ and $\eta U' \cdot \sum_{L_i} (\!|_{L_i} \cdot U_i \cdot |\!)_{L_i} = \eta \sum_{L_i} (\!|_{L_i} \cdot U_i \cdot |\!)_{L_i} \sqsupseteq \eta (\!|_{L_i} \cdot U_i \cdot |\!)_{L_i}$.

- Case (Tbwith)

$$\text{(Tbwith)}\frac{\langle \Gamma; L :: C\rangle \vdash e_2 : \tau \triangleright U}{\langle \Gamma; C\rangle \vdash \mathbf{with}(\overline{L})\ \mathbf{in}\ e_2 : \tau \triangleright U \cdot |\!)_L}$$

- Case $e_1 = \overline{L}$, e_2 not a value.
  The transition $C \vdash \eta, \mathbf{with}(\bar{L})\ \mathbf{in}\ e_2 \rightarrow \eta', \mathbf{with}(\bar{L})\ \mathbf{in}\ e_2'$ has been deduced by the (with$_3$) rule, the premise of which is $L :: C \vdash \eta, e_2 \rightarrow \eta', e_2'$. By inductive hypothesis we have that $\langle\Gamma; L :: C\rangle \vdash e_2 : \tau \triangleright U'$ with $U \sqsupseteq U'$. The thesis follows because $\langle\Gamma; C\rangle \vdash \mathbf{with}(\overline{L})\ \mathbf{in}\ e_2' : \tau \triangleright U'$ and $\eta U\rrbracket \sqsupseteq \eta' U'\rrbracket$ by Property 2.3.9.
- Case $e_1 = \overline{L}, e_2 = v$
  Immediate by Lemma 2.3.10.

- Case (Tif):

$$(\text{Tif})\dfrac{\langle\Gamma; C\rangle \vdash e_0 : int \triangleright U \qquad \langle\Gamma; C\rangle \vdash e_1 : \tau \triangleright U_1 \qquad \langle\Gamma; C\rangle \vdash e_2 : \tau \triangleright U_2}{\langle\Gamma; C\rangle \vdash \mathbf{if}\ \ e_0\ \ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau \triangleright U \cdot (U_1 + U_2)}$$

  - Case e_0 similar to the cases above.
  - Case $e_0 = v$:
    By the typing rules $U = \varepsilon$. The transitions lead to either e_1 or e_2 without modifying history η. By the hypothesis of the typing rule, the thesis follows because $U \cdot (U_1 + U_2) = (U_1 + U_2) \sqsupseteq U_1$ and $(U_1 + U_2) \sqsupseteq U_2$.

- Case (Tsend):

$$(\text{Tsend})\dfrac{\langle\Gamma; C\rangle \vdash e : \tau \triangleright U}{\langle\Gamma; C\rangle \vdash \mathbf{send}_{\text{ty}}(e) : \texttt{unit} \triangleright U \cdot send_{ty}}$$

  - Case e not a value: similar to the cases above.
  - Case $e = v$:
    The transition $\eta, \mathbf{send}_{\text{ty}}(v) \rightarrow \eta send_{ty}, ()$ has been deduced with rule (send$_2$) and by the typing rules $U = \varepsilon$. The thesis follows because $\langle\Gamma; C\rangle \vdash () \triangleright \varepsilon$ and $\eta U \cdot send_{ty} \sqsupseteq \eta send_{ty} \cdot \varepsilon$.

- Case (Talpha):

$$(\text{Talpha})\dfrac{}{\langle\Gamma; C\rangle \vdash \alpha(a) : \texttt{unit} \triangleright \alpha(a)}$$

  The transition $C \vdash \eta, \alpha(r) \rightarrow \eta\alpha(r), ()$ has been deduced by (action) rule. Since $\langle\Gamma; C\rangle \vdash () : \texttt{unit} \triangleright \varepsilon$, we can conclude that $\eta\alpha(r) \sqsupseteq \eta\alpha(r) \cdot \varepsilon$.

- Case (Tphi):

$$(\text{Tphi})\dfrac{\langle\Gamma; C\rangle \vdash e : \tau \triangleright U}{\langle\Gamma; C\rangle \vdash \phi[e] : \tau \triangleright [_\phi{\cdot}U{\cdot}]_\phi}$$

  The transition $C \vdash \eta, \phi[e] \rightarrow \eta[_\phi, \overline{\phi}[e]$ has been deduced by rule (framing$_0$). By the premises of the typing rule we get that $\langle\Gamma; C\rangle \vdash \overline{\phi}[e] : \tau \triangleright U{\cdot}]_\phi$. The thesis is directly obtained because $\eta[_\phi{\cdot}U{\cdot}]_\phi \sqsupseteq \eta[_\phi{\cdot}U{\cdot}]_\phi$.

- Case (Tbphi):

  $$(\text{Tbphi})\frac{\langle\Gamma; C\rangle \vdash e : \tau \triangleright U}{\langle\Gamma; C\rangle \vdash \overline{\phi}[e] : \tau \triangleright U\cdot]_\phi}$$

  - case $e = \overline{\phi}[v]$
    The transition $C \vdash \eta, \overline{\phi}[v] \to \eta]_\phi, v$ has been deduced by the rule (framing$_2$), the premise of which is $\langle\Gamma; C\rangle \vdash v : \tau \triangleright \varepsilon$, hence $U = \varepsilon$. The thesis is immediate because $\eta U]_\phi = \eta]_\phi \sqsupseteq \eta]_\phi \cdot \varepsilon$.
  - case $e = \overline{\phi}[e_1]$
    The transition $C \vdash \eta, \overline{\phi}[e_1] \to \overline{\phi}[e_1']$ has been deduced with the rule (framing$_1$). By the premises of the typing rule we have that $\langle\Gamma; C\rangle \vdash e_1 : \tau \triangleright U$. Since by the premises of (framing$_1$) rule we have that $C \vdash \eta, e_1 \to \eta', e_1'$, we can use the inductive hypothesis so that we have $\langle\Gamma; C\rangle \vdash e_1' : \tau \triangleright U'$ such that $\eta U \sqsupseteq \eta' U'$. By Property 2.3.9 we have that $\eta U]_\phi \sqsupseteq \eta' U']_\phi$.

- Case (Tsub)

  $$(\text{Tsub})\frac{\langle\Gamma; C\rangle \vdash e : \tau' \triangleright U' \qquad \tau' \leq \tau \qquad U' \sqsubseteq U}{\langle\Gamma; C\rangle \vdash e : \tau \triangleright U}$$

  If $C \vdash \eta, e \to \eta', e'$, then by inductive hypothesis and by the premises of the typing rule we have that $\langle\Gamma; C\rangle \vdash e' : \tau' \triangleright U''$ with $\eta U' \sqsupseteq \eta' U''$. Thesis follows because $U \sqsupseteq U'$ implies $\eta U \sqsupseteq \eta U'$. Then $\langle\Gamma; C\rangle \vdash e' : \tau \triangleright \eta' U''$ and $\eta U \sqsupseteq \eta' U''$.

□

Corollary 2.3.3 is restated and proved below.

Corollary 2.3.3 (Over-approximation)**.** *If $\langle\Gamma; C\rangle \vdash e : \tau \triangleright U$ and $C \vdash \varepsilon, e \to^* \eta, e'$, then $\eta \in [\![U]\!]$.*

Proof. By induction on the length i of the computation, by repeatedly applying the subject reduction theorem we can prove that $\langle\Gamma; C\rangle \vdash e' : \tau \triangleright U'$ with $U \sqsupseteq \eta U'$. Since by definition $\eta \in [\![\eta U']\!]$ and since $U_0 \sqsubseteq U_1 \Rightarrow [\![U_0]\!] \sqsubseteq [\![U_1]\!]$, we have that $\eta \in [\![U]\!]$. □

We restate and prove Theorem 2.3.4.

Theorem 2.3.4 (Progress)**.** *Let e be a closed expression such that $\langle\Gamma; C\rangle \vdash e : \tau \triangleright U$. If $C \vdash \eta, e \not\to$ and ηU is valid, then e is a value.*

Proof. By induction on the depth of the typing derivation, and then by cases on the last rule applied. We only work out the relevant cases, the proof of the other being trivial.

- Case (Tlexp):

  $$(\text{Tlexp})\frac{\forall i. \langle \Gamma; C\rangle \vdash e_i : \tau \rhd U_i \qquad L_1 \in |C| \vee \cdots \vee L_n \in |C|}{\langle \Gamma; C\rangle \vdash L_1.e_1, \ldots, L_n.e_n : \tau \rhd \sum_{L_i \in \{L_1,\ldots,L_n\}} \mathsf{Disp}(L_i) \cdot U_i}$$

  If $L_1.e_1, \ldots, L_n.e_n$ is stuck, then it is the case that $Disp(|C|, \{L_1, \ldots, L_n\})$ is not defined. However this cannot be the case because one of the $L_i, 1 \leq i \leq n$ is in C.

- Case (TApp):

  $$(\text{Tapp})\frac{\langle \Gamma; C\rangle \vdash e_1 : \tau_1 \xrightarrow{\mathbb{P}|U} \tau_2 \rhd U_1 \qquad \langle \Gamma; C\rangle \vdash e_2 : \tau_1 \rhd U_2 \qquad \exists v \in \mathbb{P}. v \subseteq |C|}{\langle \Gamma; C\rangle \vdash e_1 e_2 : \tau_2 \rhd U_2 \cdot U_1 \cdot U}$$

  If $e_1 e_2$ is stuck, then it is only the case that both e_1 and e_2 are values. In this case by Lemma 2.3.11 $e_1 = \lambda_f x \Rightarrow e$, hence rule (app3) applies, contradiction.

- Case (Tphi):

  $$(\text{Tphi})\frac{\langle \Gamma; C\rangle \vdash e : \tau \rhd U}{\langle \Gamma; C\rangle \vdash \phi[e] : \tau \rhd [_\phi \cdot U \cdot]_\phi}$$

  If $\phi[e]$ is stuck, then it is only the case that not $\eta^{-[]} \models \phi$. Since $\eta[_\phi \cdot U \cdot]_\phi$ valid, then $\eta[_\phi \in [\![\eta[_\phi \cdot U \cdot]_\phi]\!]$, hence $\eta[_\phi$ is valid and hence $\eta^{-[]} \models \phi$, contradiction.

- Case (Tbphi):

  $$(\text{Tbphi})\frac{\langle \Gamma; C\rangle \vdash e : \tau \rhd U}{\langle \Gamma; C\rangle \vdash \overline{\phi}[e] : \tau \rhd U \cdot]_\phi}$$

  - Case e is a value: the proof is similar to the one above.
  - If e is not a value, then, by inductive hypothesis, it is only the case that
    $C \vdash \eta, e \to \eta', e'$ but not $\eta'^{-[]} \models \phi$. However, by subject reduction $\langle \Gamma; C\rangle \vdash e' : \tau \rhd U'$ with $\eta U \sqsupseteq \eta' U'$. Since $\eta U \cdot]_\phi \sqsupseteq \eta' U' \cdot]_\phi$ and $\eta U \cdot]_\phi$ is valid then $\eta' U' \cdot]_\phi$ is valid. In particular, there is a history $\eta\eta' \in [\![\eta' U']_\phi]\!]$ with such an unmatched $[_\phi$. Since $\eta'\eta''$ is valid then $(\eta'\eta'')^{-[]} \models \phi$, hence $\eta'^{-[]} \models \phi$ by Property 1.3.2, contradiction.

□

We restate and prove Corollary 2.3.5.

Corollary 2.3.5. *If $\langle \emptyset; C \rangle \vdash e : \tau \triangleright U$ and U is valid, then e never gets stuck, i.e. $C \vdash \varepsilon, e \to^* \eta', e'$ with e' not a value implies $C \vdash \eta', e' \to \eta'', e''$.*

Proof. Let $C \vdash \varepsilon, e \to^* \eta', e'$ and, by contradiction, let e' be a non-value that is stuck. By induction on the length i of the computation, by repeatedly applying the subject reduction theorem, we have that $\langle \Gamma; C \rangle \vdash e'.\tau \triangleright U'$ and $\eta' U' \sqsubseteq U$. Since U is valid also $\eta' U'$ is valid. Then the progress theorem suffices to show that either that e' is a value or that e' is not stuck. □

2.4 Model-Checking Policies and Protocols

In this section we introduce a model-checking machinery for verifying whether a usage is compliant with respect to a policy ϕ and a protocol P. The idea is that the environment specifies P, and only accepts a user to join that follows P during the communication.

Policy checking A policy ϕ is a safety property (Definition 1.2.2) specified by a standard Finite State Automaton (FSA). We take a default-accept paradigm, i.e. only the unwanted behaviour is explicitly mentioned. Consequently, the language of ϕ is the set of *unwanted traces*, hence an accepting state is considered as offending.

We depict in the left part of Figure 2.7 a simple policy ϕ_2 that prevents the occurrence of two consecutive actions α on the resource r at the beginning of the computation.

We now define the meaning of $\eta \vDash \phi$, completing the definition of validity presented in Section 1.3.4.

Definition 2.4.1 (Policy compliance)**.** Let η be a history without framing events, then $\eta \vDash \phi$ iff $\eta \notin L(\phi)$.

The semantics of a usage may contain histories with redundant framings, i.e. nesting of the same policy framing. For instance, $\mu h. (\phi[\alpha(r)h] + \varepsilon)$ generates $[_\phi \alpha(r)[_\phi \alpha(r)]]$. Formally, a history η has *redundant framing* whenever the active policies $ap(\eta')$ contain a duplicate ϕ for some prefix η' of η.

Redundant framing can be eliminated without affecting validity of a history [7]. This is because the expressions monitored by the inner-framings are already under the scope of the outermost one and the definition of validity in Section 1.3.4 uses $\eta^{-[]}$. Actually, given U there is a *regularisation* algorithm returning his regularized version $U{\downarrow}$ such that *(i)* each history in $[\![U{\downarrow}]\!]$ has no

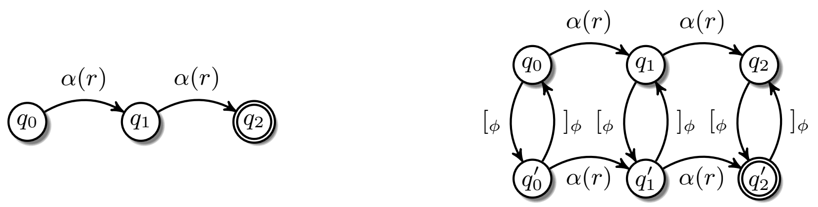


Figure 2.7: On the left: a policy ϕ_2 expressing that two consecutive actions α on r at the beginning of the computation are forbidden. On the right: the framed automaton obtained from ϕ_2.

redundant framing, *(ii)* $U\downarrow$ is valid if and only if U is valid [7]. Hence, checking validity of a usage U can be reduced to the problem of checking validity of a usage $U\downarrow$ without redundant framings.

Our approach fits into the standard *automata based* model-checking [60]. Indeed, there is an efficient and fully automata based method for checking the $\models$ relation for a regularised usage U.

Let $\{\phi_i\}$ be the set of all policies ϕ_i occurring in U. From each ϕ_i it is possible to obtain a *framed automaton* $\phi_i^{[]}$ such that η is valid iff $\eta \notin L(\bigcup \phi_i^{[]})$. The detailed construction of the framed automaton is in [7]. Roughly the framed automaton for the policy ϕ has two copies of ϕ. The first copy has no offending states, the second has the same offending states of ϕ. Intuitively, one uses the first copy when the actions are made while the policy is not active. The second copy is reached when the policy is activated by a framing event. Indeed, there are edges labelled with $[_\phi$ from the first copy to the second and $]_\phi$ in the opposite direction. So when a framing gets activated we can also reach an offending state. Figure 2.7 shows the framed automaton used to model-check the policy ϕ_2.

Validating a regularised usage U amounts to verify $[\![U]\!] \cap \bigcup L(\phi_i^{[]})$ is empty. Using the fact that for any usage U there exists a pushdown automaton $B(U)$ that recognises the semantics of U (see Theorem 1.3.4 for the most general result), we can state the following:

Theorem 2.4.1 (Model-Checking Policies). *A given usage U is valid if and only if* $L(B(U\downarrow)) \cap \bigcup L(\phi_i^{[]}) = \emptyset$.

Since regular languages are closed by union, context-free languages are closed by intersection with a regular language and the emptiness of context-free languages is decidable [36] the above theorem is decidable.

Protocol compliance We are now ready to check whether a program will well-behave when interacting with other parties through the bus. We take a protocol P to be sequence S of send_{ty} and $\mathit{receive}_{ty}$ actions designating the coordination interactions, possibly repeated (in symbols S^*), as defined below:

$$P ::= S \mid S^* \qquad\qquad S ::= \varepsilon \mid \mathit{send}_{ty}.S \mid \mathit{receive}_{ty}.S$$

A protocol P specifies the regular set of allowed interaction histories. We require a program to interact with the bus following the protocol, but we do not force the program to do the whole interaction specified. For this motivation the language $L(P)$ of P is a prefix closed set of histories, obtained by considering all the prefixes of the sequences defined by P. Then we only require that all the histories generated by a program (projected so that only send_{ty} and $\mathit{receive}_{ty}$ appear) belong to $L(P)$.

Let U^{sr} be a projected usage where all non send_{ty},$\mathit{receive}_{ty}$ events have been removed. Then we define compliance to be:

Definition 2.4.2 (Protocol compliance)**.** Let e be an expression such that $\langle\Gamma, C\rangle \vdash e : \tau \triangleright U$, then e is compliant with P if $[\![U^{sr}]\!] \subseteq L(P)$.

The theorem below gives us a decidable model-checking procedure to establish protocol compliance. In its statement, we write $\overline{L(P)}$ to denote the complement of $L(P)$. Note that the types annotating $\mathit{send}_{ty}/\mathit{receive}_{ty}$ can be kept finite in both $L(P)$ and $\overline{L(P)}$, because we only take the types occurring in the effect U under checking.

Theorem 2.4.2 (Model-Checking Protocols)**.** *Let e be an expression such that $\langle\Gamma, C\rangle \vdash e : \tau \triangleright U$ and let P be a protocol. Then e is compliant with P iff*

$$L(B(U^{sr})) \cap \overline{L(P)} = \emptyset$$

The theorem follows directly by exploiting the property of $B(U)$ used for justifying Theorem 2.4.1.

2.5 Parametrized behavioural variation

ContextML only supports a finite number of behavioural variation, i.e. the ones specified by the labels appearing in all the layered expressions. This is a drawback shared by most of the COP language proposals. However, the applications are often required to adapt to scenarios where an unbounded number of resources appear and disappear. This section introduces an extension of ContextML that overcomes the problem discussed above, by introducing behavioural variations with parameters.

A *parametrised behavioural variation* is a chunk of behaviour that depends on the value of a parameter, representing a resource actually occurring in the execution environment. We assume that the actual values p of the parameters are taken from a countable set of symbols Σ_d. The context is modified accordingly, becoming a stack of layers with actual parameters. A layered expression depends on a formal parameter, that is bound with the actual parameter during the dispatching phase.

The ContextML extended syntax is given below:

$$n \in \mathbb{N} \qquad x, f \in \mathsf{Ide} \qquad L \in \mathsf{LayerNames}$$
$$\phi \in \mathsf{Policies} \qquad r \in \Sigma_s \qquad p \in \Sigma_d \qquad \alpha, \beta \in \mathsf{Act}$$

$$\begin{aligned}
v, v_1, v' &::= n \mid L(p) \mid p \mid () \mid \lambda_f\, x \;\Rightarrow e \\
e, e_1, e' &::= \phi[e] \mid v \mid x \mid e_1 e_2 \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid e_1\ \mathbf{op}\ e_2 \mid \\
&\qquad \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \mid \mathbf{with}(e_1)\ \mathbf{in}\ e_2 \mid \mathbf{unwith}(e_1)\ \mathbf{in}\ e_2 \mid \\
&\qquad e_1 =_P e_2 \mid \nu \mid \urcorner e \mid lexp \\
&\qquad \mathbf{send}_{\mathrm{ty}}(e) \mid \mathbf{receive}_{\mathrm{ty}} \mid \alpha(r) \mid \alpha(e) \\
lexp &::= L(x).e \mid L(x).e, lexp
\end{aligned}$$

Parameters are created by the operator ν, that picks up a fresh symbol (i.e. one never used before) p from Σ_d, representing a resource. A parameter can be reused if it is disposed by $\urcorner$ operator. The values are extended with parameters; layers are enriched with a parameter and the boolean operator $=_P$ is introduced to check the equality of two parameters. The labels in the layered expressions are provided with a formal parameter x to be bound with the actual one in the context. The actions can be performed on parameters, to do this we allow to execute the action α on the parameter to which e evaluates, written $\alpha(e)$.

The context (in Definition 2.2.1) is extended with parameters as follows:

Definition 2.5.1 (Parametrised context)**.** Let [] be the empty context, a parametrised context, is denoted by $[L_1(p_1), \ldots, L_n(p_n)]$.

The dispatching procedure below is extended to also retrieve the actual parameter.

$$Dsp([L'_0(p_0), L'_1(p_1), \ldots, L'_m(p_m)], A) =$$
$$\begin{cases} L'_0(p_0) & \text{if } L'_0(x) \in A \text{ for some } x \\ Dsp([L'_1(p_1), \ldots, L'_m(p_m)], A) & \text{otherwise} \end{cases}$$

The extended semantics rules are in Figure 2.8. The rule (fresh),(reuse) reduces ν to fresh resource p, i.e. one that is either never appeared in the history η generated so far (written $p \notin \|\eta\|$ in (nu$_1$)) or it has been recently disposed. The history records the resource creation by the event `new`(p). The dual operator $\daleth e$, in the rule (daleth), reduces e to a resource p and marks the disposal in the history, so allowing p to be possibly picked up as being fresh later on. Note that $\daleth$ only modifies the freshness condition of the resource p, still allowing it to remain in the context. The rules $(=^1_P)$,$(=^2_P)$ reduce the left and right member of the equality operator $=_P$. When two parameters are obtained, their equality is checked $((=^3_p t)$,$(=^3_p f))$. The rules (with$_i$) additionally push the parameter on the context. The rule (lexp) bounds as well the formal parameter with the actual parameter obtained by the dispatching procedure. The actions in (actionP) can be performed on parameter p in Σ_d, the actions are stored in the history together with p.

Consider now the following examples exploiting parametrised behavioural variation, and implementing a bridging component in the spirit of software defined networks [22].

Example 2.5.1. *The code below implements a portion of a network component that can be plugged in different networking scenarios. Whenever the component senses two network connections (layers* `Connection1Found`, `Connection2Found` *are present in the context) it sets up a bridging between the two (abstracted by the actions* `BridgeEndPoint1`, `BridgeEndPoint2`*). However the bridging can be performed only if the two networks are different. Hence we assume that* `Connection1Found`, `Connection2Found` *in the context carry the network identifiers x, y as parameters, and the bridging is executed only if* $\mathtt{x}! =_\mathtt{p} \mathtt{y}$.

```
fun bridge =
  Connection1Found(x).
    Connection2Found(y).
      if(x !=_p y) then
        BridgeEndPoint1(x);
        BridgeEndPoint2(y);
      else
        skip(x);
```

Abstracting the behaviour of the `bridge` function requires more expressive abstractions than the ones used in Section 2.3.
The actions `BridgeEndPoint1(x)`, `BridgeEndPoint2(y)` are recorded in the history together with their parameters. Hence the histories have an a-priori unbounded number of symbols, representing the a-priori unknown resources

$$\text{fresh}\;\frac{p \in \Sigma_d \qquad p \notin \|\eta\|}{C \vdash \eta, \nu \;\to \eta\; \mathtt{new}(p), p}$$

$$\text{reuse}\;\frac{p \in \Sigma_d \qquad \eta = \eta_1 \mathtt{del}(p) \eta_2 \qquad \eta_2 \neq \eta_3 \mathtt{new}(p) \eta_4}{C \vdash \eta, \nu \;\to \eta\; \mathtt{new}(p), p}$$

$$\text{daleth}\;\frac{p \in \Sigma_d}{C \vdash \eta, \daleth p \;\to \eta \mathtt{del}(p)\;, ()} \qquad \text{parml}\;\frac{C \vdash \eta, e \to \eta', e'}{C \vdash \eta, L(e) \;\to \eta, L(e')}$$

$$=^1_P\;\frac{C \vdash \eta, e_1 \to \eta', e_1'}{C \vdash \eta, e_1 \;=_P\; e_2 \to \eta', e_1' \;=_P\; e_2}$$

$$=^2_P\;\frac{C \vdash \eta, e_2 \to \eta', e_2'}{C \vdash \eta, p \;=_P\; e_2 \to \eta', p \;=_P\; e_2'} \qquad =^{3t}_P\;\frac{p \in \Sigma_d}{C \vdash \eta, p \;=_P\; p \to \eta, 1}$$

$$=^{3f}_P\;\frac{\Sigma_d \ni p_1 \neq\; p_2 \in \Sigma_d}{C \vdash \eta, p_1 \;=_P\; p_2 \to \eta, 0}$$

$$\text{with}_1\;\frac{C \vdash \eta, e_1 \to \eta', e_1'}{C \vdash \eta, \mathbf{with}(e_1)\;\mathbf{in}\; e_2 \to \eta', \mathbf{with}(e_1')\;\mathbf{in}\; e_2}$$

$$\text{with}_2\;\frac{}{C \vdash \eta, \mathbf{with}(L(p))\;\mathbf{in}\; e \to \eta\; (\!|_{L(p)}, \mathbf{with}(\overline{L(p)})\;\mathbf{in}\; e}$$

$$\text{with}_3\;\frac{L(p) :: C \vdash \eta, e \to \eta', e'}{C \vdash \eta, \mathbf{with}(\overline{L(p)})\;\mathbf{in}\; e \to \eta', \mathbf{with}(\overline{L(p)})\;\mathbf{in}\; e'}$$

$$\text{with}_4\;\frac{}{C \vdash \eta, \mathbf{with}(\overline{L(p)})\;\mathbf{in}\; v \to \eta\; |\!)_{L(p)}, v}$$

$$\text{lexp}\;\frac{L_i(p_i) = Dsp(C, \{L_1(x_1), \ldots, L_n(x_n)\})}{C \vdash \eta, L_1(x_1).e_1, \ldots, L_n(x_n).e_n \to \eta\; \mathsf{Disp}(L_i), e_i\{{}^{p_i}/_{x_1}\}}$$

$$\text{actionP}\;\frac{p \in \Sigma_d}{C \vdash \eta, \alpha(p) \to \eta\; \alpha(p), ()}$$

Figure 2.8: Semantics for ContextML extended with parametrised behavioural variation

in the environment. Actually histories have the shape of data-words seen in Section 1.3. By exploiting Usages, introduced in Section 1.3.4, we can represent the abstract behaviour of a run of `bridge` function by the following usage:

$$(\nu x.\ \nu y.\ \texttt{BridgeEndPoint1}(x)\texttt{BridgeEndPoint1}(y)) + (\nu x.\texttt{skip}(x))$$

where `Disp` actions are omitted. The expression above represents the fact that the bridging only takes place when the two parameters x, y are different.

Consider now the following example, inspired by a service oriented scenario [30], focusing on the behaviour arising from parameters creation.

Example 2.5.2. *The recursive function* `worker1` *below implements a sort of load balancer, that activates several services in the context, deploying them to different locations. We do not detail the code for activating the service or for the condition* `COND`*.*

```
fun worker1 =
    if COND then
       let loc = ν in
          ...      //activating a new service in location loc
         with(ServiceActiveIn(loc)) in
            worker1();
    else
       skip;
```

The set of traces generated by a run of `worker1` function above is over-approximated by the usage

$$\mu h.\nu p.\alpha(p)\ h + \varepsilon$$

Note that there is no bound on the number of parameters (the locations) that this program can generate and they are pairwise distinct. We call the latter phenomenon *unbound freshness.*

Note that also security policies ϕ need to be rethink to capture the behaviour of parametrised behavioural variations. Indeed, they need to take into account the unbounded number of parameters that may occur in the history. Examples of safety policies ϕ are "No network is part of a bridge twice", "Bridging always happen between two different networks", etc.

Discussions

We investigated language-based methods for the development of complex adaptive systems following COP paradigm. We introduce static techniques for ensuring that a component developed in ContextML language (*i*) adequately reacts to context changes, (*ii*) securely manipulates its resources and (*iii*) correctly interacts with other parties.

Several COP programming languages have been proposed (see e.g. *ContextL* [23] and *ContextJ* [2]). Usually COP features are introduced within the object oriented paradigm so providing behavioural variations at object level.

Most of the research efforts have mainly tackled implementation issues. To the best of our knowledge only few papers provide a precise semantic description.

In [34] an extension of *Featherweight Java* [37] has been proposed. In this calculus layers are not expressible values. Furthermore, a static type system ensures that there exists a binding for each dispatched method call. This fact is based on the strong assumption that layers do not introduce new methods but only refine existent ones. Our type system relaxes this assumption.

Our approach is much similar to the one of Clarke et al. [21] and the main difference is that we consider a functional language while [21] considers Featherweight Java object oriented language.

Our parametrised extension is similar to the context-dependent state in [61], where dynamic variables (in Python programming language) are added and deleted depending on the current context.

The next chapters, motivated by Examples 2.5.1 and 2.5.2, investigate and propose nominal trace models for abstracting both the behaviour and the properties of programs involving dynamic resources. The reference framework is the one in this chapter, where model-checking is reduced to verifying properties of suitably constructed automata.

Chapter 3

Language Theory for Usage Automata and Usages

This chapter studies how classical language theoretic properties contribute to assess and better evaluate the expressiveness and the exploitation of nominal models for program analysis.

Examples 2.5.1 and 2.5.2 above show that nominal models can be considered to abstract the behaviour and the properties of adaptive programs (in particular the one arising from parametrised behavioural variations of ContextML). We focus here on Usages in Section 1.3.4 as abstraction of program behaviour, and on Usage Automata (UA) in Section 1.3.1 as a formalism to express security policies ϕ.

We establish some closure properties of the UA, by studying them as nominal automata from a language theoretic viewpoint. Closure properties are not only interesting from a theoretical viewpoint, but they are also deeply connected with applications of UA to software verification. Indeed, logical connectives between policies have a counterpart as language operators. Let B, B' represent the policies ϕ, ϕ', respectively. The complement of the language of B represents the negation of ϕ. The union/intersection of the languages recognised by B, B' represents the conjunction/disjunction of ϕ, ϕ'.

The investigation of language-theoretic properties of UA are also instrumental for comparing the expressiveness of the UA with that of other automata in the literature. In particular we compare UA against the VFA (Section 1.3.2). We prove the latter more expressive.

We set up the machinery to check Usages against security policies expressed through VFA, so enlarging the class of security policies that fits the verification framework of [6] depicted in Section 1.2.

To conclude we show some limitations of Usages, which make them unsuitable to capture all the facets of adaptive programs.

3.1 Usage Automata

We introduce two helpful sub-classes of UA: *saturated* and *frozen* UA. Using these classes we will eventually give an alternative characterisation of the languages recognised by UA. The introduction of these two models is mostly justified by technical reasons, as they provide the machinery for proving properties in Sections 3.1.4 and 3.1.5 and to establish the correspondence between UA and VFA.

For simplicity, in the proofs of this chapter we will assume that Act is a singleton and we will write v for $\alpha(v)$. All the proofs straightforwardly extend when there are more actions.

Additionally, in this section we will always consider an alphabet S containing all the static resources Σ_s and variables V, i.e. $S = \mathsf{Act} \times (\Sigma_s \cup V)$. This is not a restriction since we can always extend the alphabet of a UA to such an S while preserving the accepted language. Indeed, it suffices to add a sink and appropriate edges, labelled with the new symbols.

3.1.1 Saturated UA

Definition 3.1.1 (Saturated UA). Let $B = \langle S, Q, q_0, F, E\rangle$ be a UA and let $\sigma : Var(B) \to R$ be a substitution. Then B is σ-saturated iff given $(q, \alpha(r), q) \in \mathrm{Comp}(X_\sigma)$ in $B_\sigma = \langle R, Q, q_0, F, \delta_\sigma\rangle$ we have that $r \in \Sigma_d$ and $\forall x \in Var(B).\sigma(x) \neq r$.
The automaton B is *saturated* iff for all substitutions σ, B is σ-saturated.

Example 3.1.1. *The automaton in Figure 3.1(a) is not saturated. Under the substitution σ, with $\sigma(x) = d \in \Sigma_d$, $\mathrm{Comp}_\sigma(X_\sigma)$ contains both an edge $(q_0, \alpha(b), q_0)$ and $(q_0, \alpha(d), q_0)$. This violates both the requirements. We can easily obtain a saturated automaton recognising the same language of (a) by adding appropriate self loops to it, see Figure 3.1(b).*

The following property holds.

Property 3.1.2. *Let B be a saturated UA and let $\sigma : Var(B) \to R$ be a substitution, then:*

$$\forall q_i, q_i', q_j, q_j' \in Q, v \in \mathsf{Act}\times R.(q_i, v, q_i') \in \mathrm{Comp}_\sigma(X_\sigma) \Leftrightarrow (q_j, v, q_j') \in \mathrm{Comp}_\sigma(X_\sigma)$$

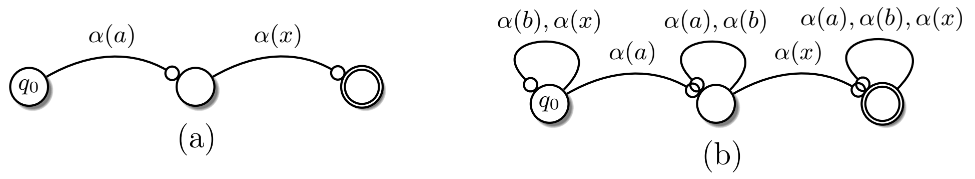


Figure 3.1: A UA in (a) and its saturated version in (b). The alphabet is $S = \mathsf{Act} \times (\{a, b\} \cup \{x\}), \mathsf{Act} = \{\alpha\}$

Indeed, since the set of the outgoing edges in a saturated automaton is the same for every state (and for every instantiation), the set of the edges added in the completion $\mathrm{Comp}_\sigma(X_\sigma)$ is the same for every state.

Saturation

Definition 3.1.2 (Saturation of a UA)**.** Let $S = \mathsf{Act} \times (\Sigma_s \cup V)$, $V \subset Var$ and let $B = \langle S, Q, q_0, F, E\rangle$ be a usage automaton. Let $\mathcal{C}$ be

$$\mathcal{C} = \left\{ (q, \alpha(v), g_v, q) \;\middle|\; \begin{array}{l} v \in Var(B) \cup \Sigma_s \\ \alpha \in \mathsf{Act} \\ g_v = \bigwedge\limits_{(q,\alpha(v'),g',q') \in E} g' \to (v \neq v') \end{array} \right\}$$

The *saturation* of B is the usage automaton $B_S = \langle S, Q, q_0, F, E \cup \mathcal{C}\rangle$.

Lemma 3.1.3. *The automaton B_S is saturated.*

Proof. Let $(B_S)_\sigma$ be an instantiation of B_S with $R \subseteq \Sigma, \sigma : Var(B_S) \to R$. Let $X_S \cup \mathrm{Comp}(X_S)$ be the edges of such instantiation.

By contradiction, let $(q, r, q) \in \mathrm{Comp}(X_S)$ and $r \in \Sigma_s$. Since $\mathrm{Comp}(X_S) \cap X_S = \emptyset$ $(q, r, q) \notin X_S$. By definition of $\mathcal{C}$ we have $(q, r, g_r, q) \in \mathcal{C}$, but since $(q, r, q) \notin X_S$ clearly $\sigma \nvDash g_r$. Hence we have a false statement in g_r of the form $g' \to v' \neq r$ with an associated edge $(q, v', g', q') \in E$ such that $\sigma \vDash g' \wedge v' = r$, this implies $(q, r, q') \in X_S$, in contradiction with the hypothesis.

By contradiction, let $(q, r, q) \in \mathrm{Comp}(X_S)$ and $\sigma(x) = r$ for some x. By definition of $\mathcal{C}$ we have $(q, x, g_x, q) \in \mathcal{C}$, but since $(q, r, q) \notin X_S$ clearly $\sigma \nvDash g_r$. The proof goes on as above. □

Lemma 3.1.4. $L(B) = L(B_S)$

Proof. Given $R \subseteq \Sigma$ and a substitution $\sigma : Var(B) \to R$, let $X \cup \mathrm{Comp}(X)$ be the edges of the instantiation B_σ of B and let $X_S \cup \mathrm{Comp}(X_S)$ be the ones of $(B_S)_\sigma$.

By definition we note the following fact:

Fact 3.1.5.

$$(q, r, q) \in X \cup \mathrm{Comp}(X) \Leftrightarrow$$
$$\exists g, v.(q, v, g, q) \in E \wedge \sigma(v) = r \wedge \sigma \models g \quad \vee \qquad (3.1)$$
$$\forall v, g, q'.((q, v, g, q') \in E \Rightarrow \sigma(v) \neq r \vee \sigma \not\models g) \qquad (3.2)$$

To obtain the thesis we only need to prove that given a substitution σ the set of the edges of the two instantiation of the automata are the same:

$$(q, r, q') \in X \cup \mathrm{Comp}(X) \Leftrightarrow (q, r, q') \in X_S \cup \mathrm{Comp}(X_S)$$

For $q' \neq q$ the statement is easy.
For $q' = q$ the statement become

$$(q, r, q) \in X \cup \mathrm{Comp}(X) \Leftrightarrow (q, r, q) \in X_S \cup \mathrm{Comp}(X_S)$$

$(\Rightarrow)$
$(q, r, q) \in X \cup \mathrm{Comp}(X) \Rightarrow (q, r, q) \in X_S \cup \mathrm{Comp}(X_S)$

1. If $(q, r, q) \in X$ then there exists an edge $(q, v, g, q') \in E$ such that $\sigma \models g \wedge \sigma(v) = r$, from this $(q, r, q) \in X_S$ follows easily.

2. If $(q, r, q) \in \mathrm{Comp}(X)$ then we are in case 3.2 of fact 3.1.5.

   a) If $r \notin \Sigma_s$ and does not exists $x \in V.\sigma(x) = r$ then does not exists any edge $(q, v, g, q') \in E$ such that $\sigma(v) = r$. Hence we obtain that $(q, r, q) \in \mathrm{Comp}(X_S)$.

   b) If $r \in \Sigma_s \vee \exists x \in V.\sigma(x) = r$ then let

   $$A = \{(q, v, g, q) \in E \cup \mathcal{C} \mid \sigma(v) = r\}$$

   If $r \in \Sigma_s$ then by saturation we have $(q, r, g', q) \in A$ for some guard g'. If $\sigma(x) = r$, for some x, then by saturation we have that $(q, r, g', q) \in A$ for some guard g'. Hence A is not empty and for each $r \in \Sigma$ such that $r \in \Sigma_s$ or $\sigma^{-1}(r) \neq \emptyset$ the set A always contains an edge $(q, r, g', q) \in \mathcal{C} \cap A$.

   It is not possible that the guards of the edges in A are all false.

i. By indirect reasoning, if all guards are false we obtain a contradiction. Let $(q, \bar{v}, \bar{g}, q) \in A \cap \mathcal{C}$, by the hypothesis $\sigma \nvDash \bar{g}$ and this implies $\sigma \vDash \neg\bar{g}$. Since $(q, \bar{v}, \bar{g}, q) \in \mathcal{C}$ the guard $\bar{g}$ is of the form $\wedge_{i \in I} g_i \to \bar{v} \neq v_i$. and I is not empty since $\forall \sigma'.\sigma' \vDash true$. Since $\sigma \vDash \neg\bar{g}$, we have $\sigma \vDash \neg(g_j \to \bar{v} \neq v_j)$ and hence $\sigma \vDash (g_j \wedge v = v_j)$ for some $j \in I$. Associated with $g_i \to \bar{v} \neq v_j$ we have an associated edge $(q, v_j, g_j, q_i) \in E$. We obtain a contradiction since:

A. If $q_j = q$ then, since $\sigma \vDash q_j$ and $\sigma(\bar{v}) = \sigma(v_j) = r$, there exists an edge $(q, v_j, g_j, q) \in A$, in contradiction with the hypothesis that all guards in A are false.

B. If $q_j \neq q$ then we have found $(q, v_j, g_j, q_j) \in E$ such that $\sigma \vDash g_j$ and $\sigma(\bar{v}) = \sigma(v_j) = r$, in contradiction with the hypothesis that $(q, r, q) \in \mathrm{Comp}(X)$.

Hence, there exists an edge $(q, v, g, q) \in (E \cup \mathcal{C}) \cap A$ such that $\sigma \vDash g$ and $\sigma(v) = r$. Then we obtain the thesis $(q, r, q) \in X_S$.

$(\Leftarrow)$

1. If $(q, r, q) \in \mathrm{Comp}(X_S)$ then, since $X_S \supseteq X$, by definition, $\mathrm{Comp}(X_S) \subseteq \mathrm{Comp}(X)$, obtaining the thesis $(q, r, q) \in \mathrm{Comp}(X)$.

2. If $(q, r, q) \in \mathrm{Comp}(X)$ then there exists an edge $(q, v, g, q) \in E \cup \mathcal{C}, \sigma \vDash g, \sigma(v) = r$

   a) If $(q, v, g, q) \in E, \sigma \vDash g, \sigma(v) = r$ then we obtain the thesis $(q, r, q) \in X$.

   b) If $(q, v, g, q) \in \mathcal{C}, \sigma \vDash g, \sigma(v) = r$ but, by indirect reasoning, $(q, r, q) \notin X \cup \mathrm{Comp}(X)$. Since $(q, r, q) \notin X$ then, by completion, $(q, r, q) \in \mathrm{Comp}(X)$, contradiction.

□

Then we have the following theorem:

Theorem 3.1.6. *Every UA can be saturated preserving the recognized language.*

Proof. It follows directly from Definition 3.1.2, Lemma 3.1.4 and Lemma 3.1.3 □

3.1.2 Frozen UA

The second class consists of *frozen* UA. To define them, note that substitutions and guards melt together the values of some variables. A substitution σ does this over the whole automaton, and a guard g under σ drives the instantiation removing or not the edges where it occurs.

Definition 3.1.3 (Freezing Substitution)**.** Given a UA B, a substitution $\sigma : Var(B) \to R$ is *freezing* if it is injective and $\forall x \in Var(B)$ it holds $\sigma(x) \in \Sigma_d$.

Under a freezing substitution, an automaton has all variables distinct. Theorem 3.1.13 below shows that the needed fusions can be handled anyway through additional edges and states.

Definition 3.1.4 (Frozen UA)**.** Let $B = \langle S, Q, q_0, F, E \rangle$ be a UA. The automaton B is *frozen* iff both conditions hold:

- there exists a sink state $\star \in Q$ such that $\star \notin F$, and
- for all non freezing substitution σ if $(q, \alpha(r), q') \in \delta_\sigma$ in B_σ with $q' \neq \star$ then
  $(r \notin Img(\sigma) \wedge r \in \Sigma_d)$.

A *ubiquitous guard* for B is any guard g such that for all σ if $\sigma \models g$ then σ is freezing.

The three Usage Automata in Figure 3.2 are frozen.

Property 3.1.7. *Let B be a frozen UA:*

1. *let σ be a non-freezing substitution on $Var(B)$ and let B_σ be the instantiation of B, then:*
   $\nexists\, (q, \alpha(r), q') \in \delta_\sigma.\ q' \neq q, \star$ *and* $\forall\, (q, \alpha(v), g, q') \in E, q' \neq \star.\ \sigma \not\models g$
2. *for all substitutions σ on $Var(B)$ if $(\exists (q, \alpha(v), g, q') \in E, q' \neq \star, q.\ \sigma \models g)$ then σ is freezing.*

Proof. 1. For the first claim of the conjunction: Let us suppose that such an edge $(q, r, q'), q' \neq q, \star$ exists. Since $q' \neq q$ then $(q, r, q') \in X_\sigma$, this implies that there exists an edge $(q, v, g, q') \in E, \sigma \models g, \sigma(v) = r$, in contradiction with $\forall x.\sigma(x) \neq r \wedge r \notin \Sigma_s$ whenever $q' \neq \star$. For the second claim of the conjunction: By contradiction, let us suppose that there exists $(q, v, g, q') \in E, q' \neq \star.\sigma \models g$, then we have an edge $(q, r, q') \in X_\sigma$ such that $r = v \in \Sigma_s \vee \sigma(v) = r$, in contradiction with $\forall x.\sigma(x) \neq r \wedge r \notin \Sigma_s$ whenever $q' \neq \star$.

2. Consequence of the contrapositive of 1.

□

Guards of the edges not leading to the sink play a marginal role in a frozen UA. They are ubiquitous, hence they are satisfiable by freezing substitutions, only. Additionally, under a freezing substitution guards are either tautologies or unsatisfiable.

Property 3.1.8. *Let B be a frozen UA. If there exists a freezing substitution $\sigma : Var(B) \to R$ such that $\sigma \models g$, then for all freezing substitutions $\sigma' : Var(B) \to R$ it holds $\sigma' \models g$.*

Proof. By contradiction, if $\sigma \models g$ but $\sigma' \not\models g$ for some freezing σ' then there is an atomic formula g' in g such that $\sigma \models g'$ and $\sigma' \not\models g'$ or viceversa. Such a g' does not exists. We have that $g' \neq true$ since $true \models \sigma$ and $true \models \sigma'$ for all σ' Also $g' \neq (x = v)$ and $g' \neq (x \neq v)$ with $x \in Var(B), v \in Var(B) \cup \Sigma_s$ since the first case implies that σ or σ' is not freezing, the second case is impossible since $x \neq v$ is satisfied by any freezing substitution. □

We present now some auxiliary definitions, that will be used in Definition 3.1.11. Roughly, we wish to decompose the language of a UA as the union of a finite class C of frozen UA. Each frozen UA represents the portion of the language recognised by using a class of substitutions. Each of these classes induces and is represented by a *respectful equivalence relation* (Definition 3.1.5) that project labels (giving *quotients* below) and guards (Definition 3.1.6) of the original UA to obtain its frozen instantiation in C.

We assume to be always able to choose a canonical representative over the finite subsets of $Var \cup \Sigma_s$.

Definition 3.1.5 (Respectful Equivalence Relations)**.** An equivalence relation $R \subseteq (Var(B) \cup \Sigma_s) \times (Var(B) \cup \Sigma_s)$ respects the identity of the static resources if $(a, b) \in R \wedge a, b \in \Sigma_s$ implies $a = b$. The set of respectful equivalence relations $\equiv_i$ over $\Sigma_s \cup Var(B)$ is $\mathcal{R}_B$.

Quotients

Given an equivalence relation $\equiv_i \in \mathcal{R}_B$, let $\bar{n} \in (Var(B) \cup \Sigma_s)/_{\equiv_i}$ be the equivalence class of a generic element $n \in (Var(B) \cup \Sigma_s)$. Then, we denote with $[n]_i$ the canonical representative of $\{v \mid v \in \bar{n}\}$ and with $M_i = \{[n]_i \mid n \in (Var(B) \cup \Sigma_s)\}$.
By definition it follows:

Property 3.1.9.

1. $M_i \subseteq (Var(B) \cup \Sigma_s)$
2. $M_i = Var_i \cup \Sigma_s, Var_i \subseteq Var(B)$.

Definition 3.1.6. Given a relation $\equiv_i \in \mathcal{R}_B$, and a guard g over $(\Sigma_s \cup Var(B))$ we inductively define the following rewriting system over guards:

$$\begin{aligned}
[n = m]_i &\Rightarrow [n]_i = [m]_i \\
[true]_i &\Rightarrow true \\
[\neg G]_i &\Rightarrow \neg [G]_i \\
[G \wedge G']_i &\Rightarrow [G]_i \wedge [G']_i
\end{aligned}$$

Note that the system admits a normal form for each guard. The guards modified in this way are still over $(\Sigma_s \cup Var(B))$.

Definition 3.1.7. We define

$$g^* = \bigwedge_{\substack{u,v \in \Sigma_s \cup Var(B) \\ u \neq v}} \neg(u = v)$$

as the *ubiquitous* guard of our frozen automaton.

It is easy to show that g^* respects the properties requested by Definition 3.1.4.

Substitutions

Definition 3.1.8 (Closure of a function)**.** Given a function $f : A \to B$, the reflexive, symmetric, transitive closure is obtained by looking at it as a relation $f \subseteq (A \cup B) \times (A \cup B)$. Hence:

$$\begin{aligned}
\text{Closure}_{rst}(f) = f \cup \{(u,u) &\mid u \in A \cup B\} \cup \\
\cup \{(u,v) &\mid (v,u) \in \text{Closure}_{rst}(f)\} \cup \\
\cup \{(u,z) &\mid \exists v.(u,v) \in \text{Closure}_{rst}(f) \wedge (v,z) \in \text{Closure}_{rst}(f)\}
\end{aligned}$$

Definition 3.1.9 (Compatibility)**.** A substitution $\sigma : Var(B) \to R, R \subseteq \Sigma$ is *compatible* with $\equiv_i \subseteq \mathcal{R}_B$ iff $\equiv_i \subseteq Closure_{rst}(\sigma)$. A compatible relation $\equiv_i \in \mathcal{R}_B$ is maximal with σ, in symbols $\sigma \between \equiv_i$, whenever does not exists any relation $\equiv_j \in \mathcal{R}_B$ compatible with σ such that $\equiv_i \subset \equiv_j$.

Property 3.1.10. *Given $\sigma : Var(B) \to R$, if $\equiv_i \between \sigma$ then $\equiv_i$ is unique.*

Proof. $Closure_{rts}(\sigma) \cap ((Var(B) \cup \Sigma_s) \times (Var(B) \cup \Sigma_s))$ is a respectful equivalence relation maximal compatible with σ. □

Definition 3.1.10 (Freezing of a substitution)**.** Given a substitution $\sigma : Var(B) \to R$ and a relation $\equiv_i \in \mathcal{R_B}$ such that $\sigma \between \equiv_i$ we define the substitution $[\sigma]_i$, the freezing of σ with respect to $\equiv_i$, as follows

$\forall x \in Var_i.$ $[\sigma]_i(x) = \sigma(x)$

$\forall x \in Var(B) \setminus Var_i.$ $[\sigma]_i(x) = d$ with d dynamic resource not in the image of $[\sigma]_i$

It is easy to verify that $[\sigma]_i$ is a freezing substitution.

Property 3.1.11.

1. *$\forall v \in Var(B).[\sigma]_i([v]_i) = \sigma(v)$.*

2. *Let $\sigma : Var(B) \to R$ be a freezing substitution and $\equiv_i \in \mathcal{R}_B$ a relation then there exists a unique substitution $[\sigma]_i^{-1} : Var(B) \to R$ such that $[\sigma]_i^{-1} \between \equiv_i$ and $\sigma \restriction_{Var_i} = [\sigma]_i^{-1} \restriction_{Var_i}$.*

3. *Let $\sigma : Var(B) \to R$ be a freezing substitution, $\equiv_i \in \mathcal{R}_B$ and g a guard then*
$$[\sigma]_i^{-1} \vDash g \Leftrightarrow \sigma \vDash [g]_i$$

4. *Given $\sigma : Var(B) \to R$ and $\equiv_i \in \mathcal{R}_B$, if $\sigma \between \equiv_i$ then*
$$\sigma \vDash g \Leftrightarrow [\sigma]_i \vDash [g]_i$$

Proof. 1. By compatibility, since $[v]_i \equiv_i v$, then $\sigma([v]_i) = \sigma(v)$. Since $[v]_i \in Var_i$, by definition, $[\sigma]_i([v]_i) = \sigma([v]_i)$ obtaining the thesis.

2. Let $[\sigma]_i^{-1}$ be:
$$x \in Var_i \Rightarrow [\sigma]_i^{-1}(x) = \sigma(x)$$
$$x \notin Var_i \Rightarrow [\sigma]_i^{-1}(x) = \sigma([x]_i)$$
it is easy to show that $[\sigma]_i^{-1}$ is compatible with $\equiv_i$ and that $\equiv_i$ is maximal compatible.

3. $(\Rightarrow)$
   By contradiction, if $[\sigma]_i^{-1} \models g$ and $\sigma \not\models [g]_i$ then there exists an atomic formula g' in g such that: $[\sigma]_i^{-1} \models g'$ and $\sigma \not\models [g']_i$ or vice versa. We examine the atomic formulas one by one concluding that such a g does not exists.

   - We have that $g' \neq \mathit{true}$ since then $[\sigma]_i^{-1} \models g'$ and $\sigma \models [g']_i$.
   - If $g' = (x = v), x \in Var(B), v \in Var(B) \cup \Sigma_s$ then, since σ is freezing, if $\sigma \models [g']_i$ the only case is that $x \equiv_i v$, but then, since $[\sigma]_i^{-1}$ is maximal compatible with $\equiv_i$ also $[\sigma]_i^{-1} \models g'$. Hence the only case is that $\sigma \not\models [g']_i$ and $[\sigma]_i^{-1} \models g'$. By $\sigma \not\models [g']_i$ we get that $x \not\equiv_i v$, since $[\sigma]_i^{-1} \models g'$ implies $[\sigma]_i^{-1}(x) = [\sigma]_i^{-1}(v)$ we obtain a contradiction with the fact that $\equiv_i$ is maximal compatible with $[\sigma]_i^{-1} \models g'$.
   - If $g' = (x \neq v), x \in Var(B), v \in Var(B) \cup \Sigma_s$ we repeat the reasoning above using the fact that if $\sigma \models [g']_i$ then $x \not\equiv_i v$ and if $\sigma \not\models [g']_i$ then $x \equiv_i v$.

   $(\Leftarrow)$
   By contradiction, if $[\sigma]_i^{-1} \not\models g$ and $\sigma \models [g]_i$ then there exists an atomic formula g' in g such that: $[\sigma]_i^{-1} \not\models g'$ and $\sigma \models [g']_i$ or vice versa. The proof is then the same as above.

4. $(\Rightarrow)$
   By contradiction, let $\sigma \models g$ and $[\sigma]_i \not\models [g]_i$. Then there exists an atomic formula g' in g such that $\sigma \models g'$ and $[\sigma]_i \not\models [g']_i$ or vice versa. We examine the atomic formulas one by one concluding that such a g' does not exists. The result is obtained as in the proof above using the fact that $\sigma \between \equiv_i$.

   $(\Leftarrow)$
   By contradiction, let $\sigma \not\models g$ and $[\sigma]_i \models [g]_i$. Then there exists an atomic formula g' in g such that $\sigma \models g'$ and $[\sigma]_i \not\models [g']_i$ or vice versa. To prove that this fact is contrary to the hypothesis we use the reasoning of the previous case.

$\square$

3.1.3 Frozen instantiation

We use now saturated and frozen automata to give an alternative characterization of the languages accepted by UA.

Definition 3.1.11 (Frozen instantiation)**.** Given a relation $\equiv_i \in \mathcal{R}_B$ and a UA B, we define its frozen instantiation $[B]_i = \langle S, Q \cup \{\star\}, q_0, F, [E]_i \rangle$.

The set $[E]_i$ is the smallest set satisfying:

$$\forall u \in (\Sigma_s \cup Var(B)).\, ((q, u, g, q') \in E \Rightarrow (q, [u]_i, [g]_i \wedge g_i^*, q') \in [E]_i)$$
$$\forall u \in (\Sigma_s \cup Var(B)), q \in Q.(q, u, \neg g^*, \star) \in [E]_i$$

Theorem 3.1.12. *$[B]_i$ is frozen.*

Theorem 3.1.13. *Let B be a UA, there exists then a finite set $\{B_i\}_{i \in I}$, B_i frozen and σ-saturated automaton (for all freezing substitutions σ) such that $L(B) = \bigcup_{i \in I} L(B_i)$*

Proof. We prove that $L_i = L(B_i)$ where B_i are frozen and σ-saturated (for any freezing substitutions σ) automata from Definition 3.1.11. Let

$$L^{\cup} = \bigcup \{L([B]_i) \mid \equiv_i \in \mathcal{R}_B\}$$

we prove that

$$L(B) = L^{\cup}$$

We rewrite the claim

$$\eta \in B \Leftrightarrow \exists i.\eta \in B_i$$

that is:

$$\exists \sigma : Var(B) \to H, H' \subseteq \Sigma.\eta \in B_\sigma \Leftrightarrow \exists i.\exists \sigma' : Var(B) \to H', H' \subseteq \Sigma.\eta \in ([B]_i)_{\sigma'}$$

$\Rightarrow$

We distinguish two cases by looking whether σ is freezing or not.

1. If σ is freezing:

   The thesis is obvious since we take the empty equivalence relation $\equiv_j$. This implies that $[v]_j = v$ for all v and $\sigma \vDash g^*$. Hence the two automata B and $[B]_j$ have the same labels on edges, the guards differs only by g^*, that is satisfied. The additional edges to the sink are never active, since their guards are unsatisfiable.

2. If σ is not freezing:

   We take j such that $\equiv_j \between \sigma$, $H = H'$ and $\sigma' = [\sigma]_j$. We show that $\eta \in B_\sigma \Rightarrow \eta \in ([B]_j)_{[\sigma]_j}$ by proving that the two automata are the same. The only thing to check is the equality of edges.

Let X be the edges of B_σ and let $[X]_j$ be the ones of $([B]_j)_{[\sigma]_j}$. The sets X and $[X]_j$ are defined:

$$\begin{aligned} X &= \{(q, \sigma(v), q') && \mid (q, v, g, q') \in E \wedge \sigma \vDash g\} \\ [X]_j &= \{(q, [\sigma]_j([v]_j), q') && \mid (q, [v]_j, [g]_j \wedge g_j^*, q') \in [E]_j \wedge [\sigma]_j \vDash [g]_j \wedge g_j^*\} \end{aligned}$$

We note that $[\sigma]_j$ is freezing by the maximality of $\equiv_j$ and hence $[\sigma]_j \vDash g^*$. The edges to the sink are never replicated in $[X]_j$ since their guards are not satisfied. The equality of $X, [X]_j$ follows from the fact that $\sigma(v) = [\sigma]_j([v]_j)$ by definition and by $\sigma \between \equiv_j$ By the latter, $\sigma \vDash g \Leftrightarrow [\sigma]_j \vDash [g]_j \wedge g_j^*$ by Property 3.1.11 and $(q, v, g, q') \in E \Rightarrow (q, [v]_j, [g]_j) \in [E]_j$ and $(q, v, g \wedge g_j^*) \in [E]_j \Rightarrow \exists v', [v']_j = v. \exists g', [g']_j = g.(q, v', g', q') \in E$ by construction. This proves:

$$(q, v, g, q') \in E \wedge \sigma \vDash g \Leftrightarrow (q, [v]_j, [g]_j \wedge g^*, q') \in [E]_j \wedge [\sigma]_j \vDash [g]_j \wedge g_j^*$$

$\Rightarrow$

1. If $\sigma : Var(B) \to R$ is freezing:

   Let $i = j$, then by Property 3.1.11 there exists a unique $[\sigma]_j^{-1}$ such that $[\sigma]_j^{-1} \between \equiv_j$ and $\sigma \restriction_{Var_j} = [\sigma]_j^{-1} \restriction_{Var_i}$.

   Let

   $$\begin{aligned} [X]_j &= \{(q, \sigma(v), q') \mid (q, v, g \wedge g^*, q') \in [E]_j \wedge \sigma \vDash g\} \\ X &= \{q, [\sigma]_j^{-1}(v'), q') \mid (q, v', g', q') \in E \wedge [\sigma]_j^{-1} \vDash g'\} \end{aligned}$$

   be respectively the edges of $([B]_j)_{([\sigma]_j^{-1})}$ and B_σ.

   The two sets are equals by proving $(q, \sigma(v), q') \in [X]_j \overbrace{\Leftrightarrow}^{1} (q, v, g \wedge g^*, q') \in [E]_j \wedge \sigma \vDash g \wedge g^* \overbrace{\Leftrightarrow} 2 (q, v', g', q') \in E \wedge [\sigma]_j^{-1} \vDash g' \overbrace{\Leftrightarrow}^{3} (q, [\sigma]_j^{-1}(v'), q') \in X$ with $(q, \sigma(v), q') = (q, [\sigma]_j^{-1}(v'), q')$.

   $\overbrace{\Leftrightarrow}^{1}$ by definition.

   $\overbrace{\Leftrightarrow}^{2}$ by definition.

a) $\Rightarrow$:
If $(q, v, g \wedge g^*, q') \in [E]_j$ then, by construction, there exists an edge in $(q, v', g', q') \in E$ such that $[v']_j = v$ and $[g']_j = g$. This implies that $v' \equiv_j v$ and since $[\sigma]_j^{-1} \between \equiv_j$ we have $[\sigma]_j^{-1}(v) = [\sigma]_j^{-1}(v')$. By properties of $[\sigma]_j^{-1}$ we have $[\sigma]_j^{-1}(v) = \sigma(v)$. By Property 3.1.11 $[\sigma]_j^{-1} \models [g']_j \Leftrightarrow \sigma \models g'$.

b) $\Leftarrow$
By construction of $[E]_j$ and by the equalities showed above.

$\overbrace{\Leftrightarrow}^{3}$ by definition.

2. If $\sigma : Var(B) \to R$ is not freezing:

Let $i = j$, and let $[X]_j \cup \text{Comp}([X]_j)$ be the edges of $([B]_j)_\sigma$. Since $\sigma \not\models g^*$ we have that $\forall r \in \Sigma_s \cup Img(\sigma), q \in Q.(q, r, \star) \in [X]_j$ and nothing else is in.

If η is recognized by the instantiation $([B]_j)_\sigma$ then it is the only case that $\eta = a_1 \ldots a_n$ and for all $1 \leq k \leq n.(q_0, a_k, q_0) \in \text{Comp}([X]_j)$, with q_0 final, hence $a_k \notin \Sigma_s \cup Img(\sigma)$.

This implies that these edges are also edges in $\text{Comp}(X)$ of B_σ.

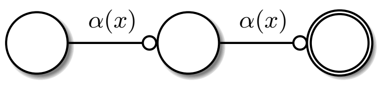
□

Example 3.1.14. *Figure 3.2 shows three frozen automata. The union of their languages gives the language recognized by the automaton in Figure 3.1. The state $\star$ is the sink, g^* is the ubiquitous guard. Clearly if $\sigma \models g^*$ (i.e. σ is freezing) the automata are σ-saturated.*

3.1.4 Closure properties

Theorem 3.1.15 (Complement of UA)**.** *Usage automata are not closed under complement.*

Proof. Consider the following automaton B:



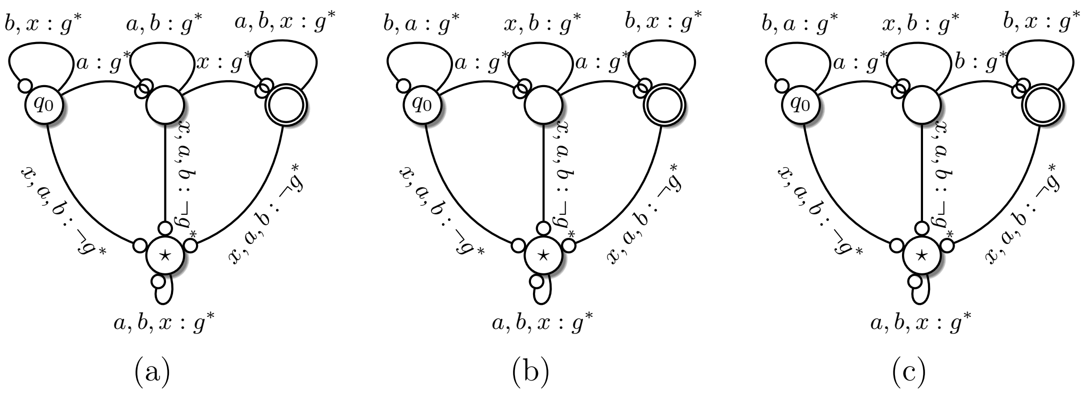


Figure 3.2: The automata (a),(b),(c) are frozen. Their alphabet is $S = \mathsf{Act} \times (\{a, b\} \cup \{x\})$ with $\mathsf{Act} = \{\alpha\}, g^* = x \neq a \wedge x \neq b$. We write here a, b, x for $\alpha(a), \alpha(b), \alpha(x)$, respectively.

This automaton recognizes the language containing at least twice the same symbol, hence its complement recognizes the words whose resources are all pairwise different. This property can not be expressed by a UA. Let $\overline{B}$ be such a UA and let $\eta = a_1 \dots a_n$ be a string with $a_i \neq a_j, i \neq j$ and $n = \left|\overline{B}\right| + 2$. $\eta \in L(\bar{B})$. Recall from Section 1.3.4 the notion of collapsing. Every collapsing κ such that $|\kappa(\Sigma)| = \left|\overline{B}\right| + 1$ will cause $\kappa(a_u) = \kappa(a_v)$ for some a_u, a_v in η. Hence we obtain that $\eta \in L(\bar{B})$ but $\eta\kappa \notin L(\bar{B})$ obtaining an absurd by Theorem 4.7 in [6]. □

Theorem 3.1.16 (Closure under Union). *Usage Automata are closed under union.*

Proof. We assume w.l.o.g. the union of two saturated automata (since the saturation is always feasible) on an alphabet $S = \Sigma_s \cup V, V \subseteq Var$. Indeed, we note that extending both variables and static resources in S, the language recognized by the automaton does not change.

Then, let $B = \langle S, Q, q_0, F, E\rangle, C = \langle S, Q', q_0', F', E'\rangle$ be two saturated automata, S $= \Sigma_s \cup V, V \subseteq Var$. We define

$$
\begin{aligned}
B \cup C = \langle S, & \\
& \wp(Q \cup Q'), \\
& \{q_0, q_0'\}, \\
& \{P \in \wp(Q \cup Q') \mid \exists q \in P \cap (F \cup F')\}, \\
& E_\cup \rangle
\end{aligned}
$$

with

$$E_{\cup} = \big\{(P, v, g, P') \;\big|\; P' = \{q' \mid \exists q \in P.(q, v, g, q') \in E \cup E' \quad \big\}$$

We now show that $L(B \cup C) = L(B) \cup L(C)$. To do this we will prove that given a substitution $\sigma : V \to R, R \subseteq \Sigma \setminus \{\_\}$ the set $E_{\cup}$ correctly mimes E and E'.

In symbols, let $X_B, X_C, X_{B\cup C}$ be the edges of the instantiation respectively of $B, C, B \cup C$.

We consider w.l.o.g B and we show that if (q, r, q') is an edge of B_σ then for all $P \in \wp(Q \cup Q')$ if $q \in P$ then there exists P' such that $q' \in P'$ and $(P, r, P') \in (B \cup C)_\sigma$.

1. If $(q, r, q') \in X_B$ then there exists an edge $(q, v, g, q') \in E, \sigma \vDash g, \sigma(v) = r$. Hence, by taking $P' = \{q' \mid \exists q \in P.(q, v, g, q') \in E \cup E'\}$, we have that $q' \in P'$. Since $\sigma \vDash g$ we obtain that $(P, r, P) \in X_{B\cup C}$.

2. If $(q, r, q) \in \mathrm{Comp}(X_B)$ then, by saturation $r \in \Sigma_d \setminus Img(\sigma)$. Since the alphabets of $B, C, B \cup C$ are the same, clearly $(P, r, P) \in \mathrm{Comp}(X_{B\cup C})$

Viceversa, if (P, r, P') is an edge of $(B \cup C)_\sigma$ then for every $q' \in P' \cap Q$ there exists $q \in P \cap Q$ such that $(q, r, q') \in B_\sigma$ and $q' \in P' \cap Q'$ there exists $q \in P \cap Q'$ such that $(q, r, q') \in C_\sigma$.

1. If $(P, r, P') \in X_{B\cup C}$ then there exists an edge $(P, v, g, P') \in E_{\cup}, \sigma \vDash g, \sigma(v) = r$. This implies that for every $q' \in P' \cap Q$ there exists $q \in P \cap Q$ such that $(q, v, g, q') \in E$ and for every $q' \in P' \cap Q'$ there exists $q \in P \cap Q'$ such that $(q, v, g, q') \in E'$. The thesis follows by $\sigma \vDash g$ and $\sigma(v) = r$.

2. if $(P, r, P') \in \mathrm{Comp}(X_{B\cup C})$ the thesis follows by noting that the saturation condition
   If $(P, r, P) \in \mathrm{Comp}(X_\sigma)$ implies $r \in \Sigma_d$ and $\forall x \in V.\sigma(x) \neq r$
   holds in $E_{\cup}$ for non-empty P.

The proof of the equality of the languages follows by induction: Given any path in B_σ it can be replicated stepwise from the beginning. Viceversa, given any path $(P \xrightarrow{v,g} P' \ldots \xrightarrow{v^f, g^f} P'^f)$ in $(B \cup C)_\sigma$ with $f \in P^f \cap Q$(w.l.o.g) final state, starting from f we can recreate a path from $q_0 \in Q$ to P^f. □

Theorem 3.1.17 (Closure under concatenation)**.** *Usage Automata are closed under concatenation*

Proof. Sketch: To build the concatenation automaton of the automaton B, B', we need to fuse the final states of the first automaton with the initial one of the second. To do that we need to take as many copies of A' as the number of final states of A. □

A corollary of the previous theorem is that we can concatenate a UA a finite number of times with itself. However, the next theorem shows that this mechanisms does not scale up to the Kleene star.

Theorem 3.1.18 (Closure under intersection)**.** *Usage Automata are closed under intersection.*

Proof. Let $B = \langle \Sigma_s \cup V, Q, q_0, F, E\rangle, C = \langle \Sigma_s \cup V', Q', q'_0, F', E'\rangle$ be two UA with $V \cap V' = \emptyset$

Let $\equiv_i \subseteq \Sigma_s \cup V \times \Sigma_s \cup V'$ be a respectful equivalence relation. We will denote with $[a]_i$ the equivalence class of a under $\equiv_i$. Clearly, there exists a finite number of such relations $\equiv_i$.

Given $\equiv_i$, we construct a UA $(B \cap C)_i = \langle \Sigma_s \cup V \cup V', Q \times Q', (q_0, q'_0), F \times F', E_i\rangle$ with

$$E_i = \{((p, q), \alpha([r']), g \wedge g', (p', q')) \mid (p, \alpha(r), g, p') \in E \text{ and } (q, \alpha(r'), g', q') \in E' \text{ and } r \equiv_i r'\}$$

We then consider the union of such finite class of automata.

It remains to prove that $\eta \in L(B) \cap L(C)$ iff $\eta \in L(B \cap C)$. We follow these deductions

$$\begin{array}{ll}
\eta \in L(B) \cap L(C) & \Leftrightarrow \text{ by definition} \\
\eta \in L(B) \wedge \eta \in L(C) & \Leftrightarrow \text{ by definition} \\
\exists \sigma . \eta \in L(B_\sigma) \wedge \exists \sigma' . \eta \in L(C'_\sigma) & \Leftrightarrow \text{ by definition} \\
\exists \sigma, \sigma' . \eta \in L(B_\sigma) \cap L(C'_\sigma) & \Leftrightarrow \text{ by the fact below (*)} \\
\exists \equiv_i . \exists \sigma^* . \eta \in L(((B \cap C)_i)_{\sigma^*}) & \Leftrightarrow \text{ by definition} \\
\exists \equiv_i . \eta \in L((B \cap C)_i) \Leftrightarrow \eta \in L(B \cap C). &
\end{array}$$

(*) We establish the following relation between $\sigma, \sigma', \sigma^*$ and $\equiv_i$.

$$\sigma(x) = \sigma'(y) = a \Leftrightarrow x \equiv_i y \text{ and } \sigma^*([x]_i) = a$$

We show that we can move from a state in one automaton then we can do the same in the other one. We will use a bold style for states of the automaton

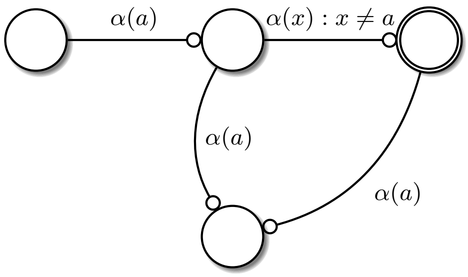
$B_\sigma \cap C'_\sigma$. It is easy to verify the following deductions:

$$\begin{aligned}
&(\mathbf{p},\mathbf{q}) \xrightarrow{\sigma(x)} (\mathbf{p}',\mathbf{q}') && \Leftrightarrow \\
&p \xrightarrow{\sigma(x)} p' \text{ and } q \xrightarrow{\sigma(y)} q' \text{ and } \sigma(x)=\sigma(y) && \Leftrightarrow \\
&p \xrightarrow{x} p' \text{ and } q \xrightarrow{y} q' && \Leftrightarrow \\
&(p,q) \xrightarrow{[x]} (p',q') && \Leftrightarrow \\
&(p,q) \xrightarrow{\sigma^*([x])} (p',q')
\end{aligned}$$

□

Theorem 3.1.19 (Kleene Star). *The UA are not closed under Kleene star.*

Proof. Consider the language recognized by the following automaton B, the recognized strings must contain exactly one $\alpha(a)$



By contradiction let $C = \langle S, Q, q_0, F, E\rangle$ be the the automaton recognizing $L^*(B)$. Consider the string $\eta = \alpha(a)\alpha(d_1)\alpha(a)\alpha(d_2)\ldots\alpha(a)\alpha(d_n)$ with $a \in \Sigma_s, d_i \in \Sigma_d, i \neq j \Rightarrow d_i \neq d_j$ and $n \geq |E|+1$. The string $\eta \in L(C)$, hence there exists a substitution σ such that $\eta \in L(C_\sigma)$. Because $|X_\sigma| = |E| < n\ d_i$, in the path recognizing η there must be an edge $(q, \alpha(d_j), q)$ obtained by completion. Hence also $\eta' = \eta\{\varepsilon/\alpha(d_j)\} \in L(C_\sigma)$ (obtained by removing $\alpha(d_j)$) and this is a contradiction. Indeed $\eta' \notin L(C)$ because $\alpha(a)\alpha(a)$ is a substring of η'.

□

However, Theorem 3.1.19 does not reduce the power of UA in expressing safety policies. Consider a language of the form L^* and let η be a trace in the semantics $[\![U]\!]$ of a usage U. For any $\eta \in L^*$ there exists also a prefix $\eta' \in L^i$, with $\eta' \in [\![U]\!]$ by Definition 1.3.12. This means that checking U against L^* is the same as checking it against L.

To sum up, we have the following closure properties for UA

	$\cup$	$\cap$	$\overline{\cdot}$	$\bullet$	$*$
$\mathcal{L}$(UA)	✓	✓	×	✓	×

3.1.5 Expressiveness

We compare now the expressive power of the UA with the one of the Variable Finite Automata (VFA) recalled in Section 1.3.2.

From UA to VFA

The first step of the construction is to prove the following simple lemma.

Lemma 3.1.20. *For each frozen UA there exists a frozen and saturated (for all freezing substitutions) UA recognising the same language .*

Proof. Given a frozen UA automaton B we can saturate it with the construction in Section 3.1.1. To make it frozen again we add to every edge in the saturation the ubiquitous guard. This does not change the language recognised by the automaton since the only state that we can reach in the instantiation of a non freezing σ is q_0 and q_0 is already saturated. $\square$

Every language recognised by a saturated and frozen UA B on $\Sigma = \Sigma_s \cup \Sigma_d$ can be recognised by a VFA $\mathcal{A} = \langle Act, \Sigma, \Sigma_s, Var(B) \cup \{y\}, A\rangle$. Below we intuitively describe the steps needed to construct the underlying automaton A.

- Remove all the edges with an unsatisfiable guard: they would never be present in any instantiation.
- Keep the edges with satisfiable guard g and remove g: this step is correct since if g is satisfiable then any substitution σ such that $\sigma \vDash g$ is freezing, and then all freezing substitutions will satisfy g, by Property 3.1.8.
- Remove the sink $\star$ and all the edges involving it.
- Add to every node a self-loop with label $\alpha(y)$ for every action $\alpha \in \mathsf{Act}$, so accounting for completion of edges. The special symbol y can be put in correspondence with any resource that is different from any value associated with a variable and any static resource, that turns out to be guaranteed by saturation.

The formal definition of this construction is in Definition 3.1.12 below.

Definition 3.1.12. Let $B = \langle S, Q, q_0, F, E\rangle$ with $S = \mathsf{Act} \times (\Sigma_s \cup V), V \subseteq Var$ be a UA frozen and saturated for freezing substitutions. We consider the

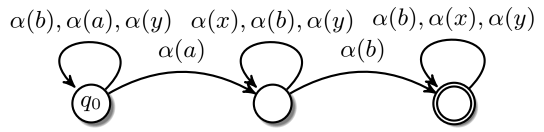

$\alpha(b), \alpha(a), \alpha(y)$
$\alpha(x), \alpha(b), \alpha(y)$
$\alpha(b), \alpha(x), \alpha(y)$
$\alpha(a)$
$\alpha(b)$
q_0

Figure 3.3: a VFA recognizing the same language of the UA in Figure 3.2(c).

VFA $\mathcal{A} = \langle \mathsf{Act}, \Sigma, \Sigma_s, Var(B) \cup \{y\}, A\rangle$ with $A = \langle \Gamma, Q \setminus \{\star\}, q_0, \delta, F\rangle$ with $\Gamma = \Sigma_s \cup Var(B) \cup \{y\}$. The function δ is obtained from E:

$$\begin{aligned}\delta = &\{(q, \alpha(v), q') \mid \exists g.(q, \alpha(v), g, q') \in E \wedge q, q' \neq \star \wedge g \text{ is satisfiable}\}\\ &\cup \{(q, \alpha(y), q) \mid q \in Q\}\end{aligned}$$

The result of the transformation above, applied to Figure 3.2(c), is in Figure 3.3.

We now state two lemmata showing a correspondence between freezing substitutions of UA and legal instances of VFA.

Lemma 3.1.21. *Let $\sigma : Var(B) \to R$ be a freezing substitution and σ^{-1} its inverse. Let c be $\sigma^{-1} \cup \{(d, y) \mid d \in \Sigma_d \setminus Img(\sigma)\}$. Then, if $c(\eta) = s$ then η is a legal instance of s in $\mathcal{A}$.*

Lemma 3.1.22. *Given B and $\mathcal{A}$ from the definition above, if η is a legal instance of s in $\mathcal{A}$ then any $\sigma = \{(s_i, \eta_i) \mid s_i \neq y \wedge s_i \notin \Sigma_s\}$ can be extended to a freezing substitution for B.*

Proof. Clearly if $(u, v) \in \sigma$ then $u \in X = V = Var(B)$. The relation σ is a function and it is freezing by the property 2 of the legal instances. The extension to the domain $Var(B)$ can be obtained by assigning a fresh dynamic resource to each $x \in Var(B)$ that is not mentioned in σ. □

We can now prove the correctness of the construction in Definition 3.1.12.

Lemma 3.1.23. *Given a UA B, let $\mathcal{A}$ be the VFA obtained from the construction in Definition 3.1.12, then $L(\mathcal{A}) = L(B)$.*

Proof. Preliminaries: Let η be a legal instance of s, and let σ be the associated freezing substitution (obtained as in Lemma 3.1.22). Then:

1. $(q, v, q') \in \delta, s \neq y \Rightarrow (q, \sigma(v), q') \in X_\sigma \cup \mathrm{Comp}(X_\sigma)$ in B_σ
2. $(q, y, q') \in \delta \Rightarrow \forall a \in \Sigma_d \setminus Img(\sigma).(q, a, q) \in X_\sigma \cup \mathrm{Comp}(X_\sigma)$ in B_σ

Now, let $\eta \in L(B_\sigma)$ with σ freezing substitution, and let c be the associated correspondence between η and $c(\eta)$ (obtained as in Lemma 3.1.21).

1. $(q, a, q') \in X_\sigma \Rightarrow (q, c(a), q') \in \delta$
2. $(q, a, q') \in \mathrm{Comp}(X_\sigma) \Rightarrow (q, y, q) \in \delta \wedge c(a) = y$

$\eta \in L(\mathcal{A}) \Rightarrow \eta \in L(B)$
The proof can be easily completed by induction on the length of η.
$\eta \in L(B) \Rightarrow \eta \in L(\mathcal{A})$
The proof can be easily completed by induction on the length of η. The properties in the preliminaries can be used since the thesis is trivial for a non freezing substitution σ. Indeed, the characters in η would be in $\Sigma_d \setminus Img(\sigma)$ and $yyyy\ldots$ would be the witnessing pattern in (A). □

The following theorem holds.

Theorem 3.1.24. *For each UA there exists a VFA recognising the same language.*

Proof. Every UA can be decomposed (preserving the language) into the union of a finite set of frozen automata:

$$B^{\cup} = \bigcup \{[B]_i \mid \equiv_i \in \mathcal{R}_B\}$$

Each $[B]_i$ can be transformed into an equivalent VFA $[A]_i$ by saturating and applying the construction in Definition 3.1.12. Since VFA are closed by union, by taking

$$A^{\cup} = \bigcup \{[A]_i\}$$

we obtain a VFA whose language is equivalent to $B_\cup$ and hence to B.

□

Actually, UA are less expressive than VFA, as stated by the following theorem.

Theorem 3.1.25. *No UA accepts the language recognised by the VFA in Figure 3.4.*

Proof. By contradiction, let $B = \langle S, Q, q_0, F, E \rangle$ be the automaton recognizing the language of the VFA $\mathcal{A}$ in Figure 3.4. Let $\eta = \alpha(a)\alpha(d_1)\ldots\alpha(a)\alpha(d_n)\alpha(d_n)$ with $d_i \in \Sigma_d, d_i \neq d_j, a \in \Sigma_s$ and $n \geq |E| + 1$. Since $\eta \in L(\mathcal{A})$ then $\eta \in L(B_\sigma)$ for some σ. Since $|X_\sigma| = |E|$, by the pigeonhole principle, in the path recognizing η there is an edge $(q, d_j, q) \in \mathrm{Comp}_\sigma(X_\sigma)$. Hence, also $\eta' = \alpha(a)\alpha(d_1)\ldots\alpha(d_{j-1})\alpha(a)\alpha(a)\alpha(d_{j+1})\ldots\alpha(a)\alpha(d_n)\alpha(d_n) \in L(B_\sigma)$ and this implies $\eta' \in L(B)$ but $\eta' \notin L(\mathcal{A})$ □

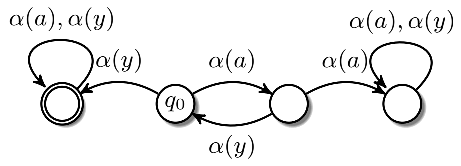


Figure 3.4: The language recognized by this VFA is not accepted by any UA. The alphabet is $\Omega = \Sigma_d \cup \{a\}$ with $\mathsf{Act} = \{\alpha\}$. Informally the safety policy expressed by this automaton requires to take a token $\alpha(a)$ before performing any other kind of action, if two tokens are taken, any sequence of actions is then allowed.

A corollary of the previous theorem is that VFA are more expressive than UA to express security properties of programs.Indeed, the language in Figure 3.4 is actually a safety property that can not be expressed by any UA.

Consider now a variant VFA, the following fact can be easily proved.

Property 3.1.26. *The restriction of VFA, obtained by only permitting the distinguished placeholder y to occur in self-loops, has the same expressive power of UA.*

3.2 Model-Checking Usages against VFA

We carry over to VFA the symbolic technique for model-checking Usages against UA (developed in [6]). Theorems 3.1.24 and 3.1.25 prove that there are safety policies that a VFA can express and a UA can not. Hence using VFA one can verify a larger class of safety properties of programs.

We start by re-defining policy compliance, similarly to Definition 2.4.1, but with VFA in place of finite state automata.

Definition 3.2.1 (Policy compliance)**.** Let $\mathcal{A}$ be a VFA, then $U \vDash \mathcal{A}$ if and only if $\eta \in [\![U]\!] \Rightarrow \eta \notin L(\mathcal{A})$.

We introduce now *symbolic* VFA. Following [6], we let their alphabet be the finite set of witnesses $\mathsf{W} \subset \{\#_i\}_{i\in\mathbb{N}}$, where $\{\#_i\}_{i\in\mathbb{N}} \cap \Sigma = \emptyset$. We also need a distinguished symbol $\_ \notin \Sigma \cup \{\#_i\}_{i\in\mathbb{N}}$.

Definition 3.2.2 (Symbolic VFA)**.** Let $\mathcal{A} = \langle \mathsf{Act}, \Sigma, \Sigma_s, X \cup \{y\}, A\rangle$ be a VFA. Given a finite set of witnesses W, let $\Sigma_\mathsf{W} = \Sigma_s \cup \mathsf{W} \cup \{\_\}$.
The *symbolic VFA* on W is $\mathcal{A}_\mathsf{W} = \langle \mathsf{Act}, \Sigma_\mathsf{W}, \Sigma_s, X \cup \{y\}, A\rangle$. Language recognition for symbolic VFA additionally requires the correspondence m to be such that $m(\_) = y$.

The collapsing seen in Definition 1.3.14 is the technical machinery that links VFA with their symbolic automaton, as shown in the following theorem.

Theorem 3.2.1. *Let $\mathcal{A} = \langle \mathsf{Act}, \Sigma, \Sigma_s, X \cup \{y\}, A\rangle$ be a VFA, and let W be a set of witnesses such that $|\mathsf{W}| = |X|$, $\mathcal{A}_{\mathsf{W}}$ as in Definition 3.2.2 and let K be the set of the collapsing $\kappa : \Sigma \to \Sigma_s \cup \mathsf{W} \cup \{\_\}$ such that $\kappa(\Sigma_d) = \mathsf{W} \cup \{\_\}$, then:*

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

Proof.

- If $\eta \in L(\mathcal{A})$ then there exists a correspondence $m : \Sigma \to (\Sigma_s \cup X \cup \{y\})$ and $s \in L(A)$ such that $m(\eta) = s$. Since $|X| = |\mathsf{W}|$, then there exists an isomorphism $\iota : X \to \mathsf{W}$. We define a collapsing $\hat{\kappa}$ in the following way:

$$\hat{\kappa}(a) = \begin{cases} \iota(x) & \text{if } m(a) = x \\ \_ & \text{if } m(a) = y \\ a & \text{if } m(a) = a, a \in \Sigma_s \end{cases}$$

  It is easy to verify that the function $\hat{m} = \iota^{-1} \cup (\_, y) \cup \{(a, a)\}_{a \in \Sigma_s}$ is a correspondence such that $\hat{m}(\hat{\kappa}(\eta)) = s, s \in L(A)$. Hence $\hat{\kappa}(\eta) \in L(\mathcal{A}_{\mathsf{W}})$.

- Let $C \subset \Sigma$ be the subset of the symbols of η. Since $\kappa(\eta) \in L(\mathcal{A}_{\mathsf{W}})$ then there exists a correspondence m and a witnessing pattern $s \in L(A)$ s.t. $m(\kappa(\eta)) = s$ with $m(\_) = y$. We now consider the function

$$m'(a) = \begin{cases} (\kappa; m)(a) & a \in C \\ y & \text{if } a = y \\ g(a) & \text{otherwise} \end{cases}$$

  with

$$\begin{array}{l} g \text{ any function such that} \\ \quad g(\Sigma \setminus C) = (X \setminus Img(\kappa; m)) \cup \{y\}, \text{ injective on } X \setminus Img(\kappa; m) \end{array}$$

  Then, the correspondence $m' : \Sigma \to \Sigma_s \cup X \cup \{y\}$ makes η a legal instance of s. Hence $\eta \in L(\mathcal{A})$.

$\square$

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

Definition 3.2.3 (Instantiation of VFA)**.** Let $\mathcal{A}_{\mathsf{W}} = \langle \mathsf{Act}, \Sigma_{\mathsf{W}}, \Sigma_s, X \cup \{y\}, A\rangle$ be a symbolic VFA with $A = \langle \Gamma, Q, q_0, F, \delta\rangle$, $\Gamma = \mathsf{Act} \times (\Sigma_s \cup X \cup \{y\})$.
Given a function $\overline{m} : X \cup \Sigma_s \to \Sigma_s \cup \mathsf{W}$ it is a *substitution* for $\mathcal{A}_{\mathsf{W}}$ if it is the identity on Σ_s and it is injective on X.
Given a substitution $\overline{m}$ the instantiation of $\mathcal{A}_{\mathsf{W}}$ is $\mathcal{A}_{\mathsf{W}}^{\overline{m}} = \langle \Sigma_{\mathsf{W}}, Q, q_0, F, \delta^*\rangle$, where

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

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

Theorem 3.2.2. *Let $\mathcal{A}_{\mathsf{W}} = \langle \mathsf{Act}, \Sigma_{\mathsf{W}}, \Sigma_s, X \cup \{y\}, A\rangle$ be a symbolic VFA:*

$$\eta \in L(\mathcal{A}_{\mathsf{W}}) \Leftrightarrow \exists \textit{ substitution } \overline{m}.\eta \in L(\mathcal{A}_{\mathsf{W}}^{\overline{m}})$$

Proof. $\Rightarrow$
If $\eta \in L(\mathcal{A}_{\mathsf{W}})$ then there exists a correspondence $m : \Sigma_{\mathsf{W}} \to (\Sigma_s \cup X \cup \{y\})$ with $m(\_) = y$ such that $m(\eta) = s$ for some $s \in L(A)$. By definition m is injective on $\Sigma_s \cup X$, then we take the inverse $\overline{m} = (m \restriction_{(\Sigma_s \cup X)})^{-1}$. We note that:
If $m(\alpha(a)) = \alpha(v), v \neq y$ and $(q, \alpha(v), q') \in \delta$ then $(q, \alpha(a), q') \in \delta^*$ by construction.
If $m(\alpha(a)) = \alpha(y)$ and $(q, \alpha(y), q') \in \delta$ then by construction and by the fact that m is a correspondence we have $(q, \alpha(a), q') \in \delta^*$.
Hence if $\eta \in L(\mathcal{A}_{\mathsf{W}})$ then there exists a path in A for s leading to a final state. This path can be reproduced in $L(\mathcal{A}_{\mathsf{W}}^{\overline{m}})$.
$\Leftarrow$
The proof is the same as above, using an extension of $\overline{m}^{-1}$ as correspondence. □

We recall now the well-known *weak-until* operator $A \mathfrak{W} B$ between automata [3, 6], meaning that A holds until B holds or B always holds.

Definition 3.2.4 (Weak Until Operator)**.** Let $A = \langle S, Q_A, q_0^A, F_A, \delta_A\rangle, B = \langle S, Q_B, q_0^B, F_B, \delta_B\rangle$ be two FSA on the alphabet S such that for each state q and label v there exists a transition from q that is labelled v. The *weak until automaton* $A \mathfrak{W} B = \langle S, Q, q_0, F, \delta\rangle$ is defined as follow:

- $Q = Q_A \times Q_B$
- $F = F_A \times (Q_B \setminus F_B)$
- $q_0 = (q_0^A, q_0^B)$

- $$\begin{aligned}\delta =&\{(q_A, q_B) \xrightarrow{v} (q'_A, q'_B) \mid q_A \xrightarrow{v} q'_A \in \delta_A, q_B \xrightarrow{v} q'_B \in \delta_B, q_B \in Q_B \setminus F_B\}\\ \cup&\{(q_A, q_B) \xrightarrow{v} (q_A, q_B) \mid q_B \in F_B\}\end{aligned}$$

To simplify the technical development, the next theorem will use the unique-witness automaton (N_{W}) seen in Definition 1.3.15 and the *weak-until* operator.

Theorem 3.2.3 (Model-checking). *Let U be an initial usage on the resources $\Sigma = \Sigma_d \cup \Sigma_s$; let $\mathcal{A} = \langle \mathsf{Act}, \Sigma, \Sigma_s, X \cup \{y\}, A\rangle$ 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:*

$$\forall \textit{ substitution } \overline{m} : X \cup \Sigma_s \to \Sigma_s \cup \mathsf{W}.\ L(\mathsf{B}_{\mathsf{W}}(\mathsf{U})) \cap L(\mathcal{A}^{\overline{m}}_{\mathsf{W}}\ \mathfrak{W}\ N_{\mathsf{W}}) = \emptyset$$

Proof.
- (correctness)$\Leftarrow$
  We prove the contrapositive:
  Let us assume $\eta' \in [\![U]\!]$ with $\eta \in L(\mathcal{A})$

  – By the properties of VFA:
    By Theorem 3.2.1 there exists a collapsing κ with $\kappa(\Sigma_d) = \mathsf{W} \cup \{\_\}$ such that $\kappa(\eta) \in L(\mathcal{A}_{\mathsf{W}})$. By Theorem 3.2.2 there exists a substitution $\overline{m}$ such that $\kappa(\eta) \in L(\mathcal{A}^{\overline{m}}_{\mathsf{W}})$.

  – By the properties of Usages: By Theorem 1.3.4, given the collapsing κ above, $\kappa(\eta) \in L(\mathsf{B}_{\mathsf{W}}(U))$

  Since η is well-formed $\kappa(\eta) \notin L(N_{\mathsf{W}})$.
  Thesis follows since then $\kappa(\eta) \in L(\mathsf{B}_{\mathsf{W}}(U)) \cap L(\mathcal{A}^{\overline{m}}_{\mathsf{W}}\mathfrak{W}N_{\mathsf{W}})$.

- (completeness)$\Rightarrow$
  We prove the contrapositive:
  The only case is that $\eta \in L(\mathsf{B}_{\mathsf{W}}(U))$ and $\eta \in L(\mathcal{A}^{\overline{m}}_{\mathsf{W}})$ and $\eta \notin L(N_{\mathsf{W}})$. Indeed, by definition of weak-until automaton, $\eta \in L(N_{\mathsf{W}})$ implies $\eta \notin L(\mathcal{A}^{\overline{m}}_{\mathsf{W}}\ \mathfrak{W}\ N_{\mathsf{W}})$

  – By the properties of Usages:
    Let κ be an injective collapsing such that $\kappa(\Sigma_d) = \mathsf{W} \cup \{\_\}$. By Theorem 1.3.4 there exists η' such that $\eta = \kappa(\eta')$ and $\eta' \in [\![U]\!]$.

  – By the properties of VFA:
    By Theorem 3.2.2 $\eta \in L(\mathcal{A}_{\mathsf{W}})$. By Theorem 3.2.1, since the collapsing above is such that $\kappa(\Sigma_d) = \mathsf{W} \cup \{\_\}$, we obtain that $\eta' \in L(\mathcal{A})$

Hence we have proven that $U \not\models \mathcal{A}$ since $\eta' \in [\![U]\!] \cap L(\mathcal{A})$.

□

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

We can then re-use the model-checker LocUsT [4] for verifying Usages against VFA. As for complexity issues, we can restate the theorem established in [6] for VFA. The proof is mostly the same with only minor changes regarding the number of instantiations of VFA.

Theorem 3.2.4. *The worst-case asymptotic behaviour of model-checking an usage U against an automaton B with n variables is $\mathcal{O}(|U|^{|n|+1})$.*

The model-checking framework presented in this chapter requires to reduce the problem of verifying a Usages against a property expressed by a VFA to standard automata-based model checking. This paradigm relies on the fact that VFA are able to distinguish only a finite number of symbols, so allowing to consider only a finite number of resources in Usages (Theorem 1.3.4).

3.3 Usages

We start our investigation of the Usages by proving the theorem below.

Theorem 3.3.1. *The language $(\{\alpha\} \times \Sigma)^*$ is not generated by any Usages.*

Proof. The proof has the following structure: for simplicity we consider the Usages as language recognisers seen in Section 1.3.4 and without actions, i.e. $U, V ::= a \in \Sigma | \dots$ A few lemmata are proved to provide support for the final argument.

In the following we will use the word *redex* to identify the source of a transition deduced by only applying an (instance of an) axiom.

Lemma 3.3.2. *If $U_0, \mathcal{R} \xrightarrow{\varepsilon} U_1, \mathcal{R}'$ and $\mu h.U$ is its redex, then $U_0\{^{\mu h.U}/_h\} = U_1$.*

Proof. (By induction on the depth of the proof)
There are two cases:

- Base case: $U_0 = \mu h.U$ and the thesis follows easily.

- Inductive case: $U_0 \xrightarrow{\varepsilon} U_1$ has been proved by rule (seq) as last step, as below:

$$\frac{U_0' \xrightarrow{\varepsilon} U_1'}{U_0' \cdot V \xrightarrow{\varepsilon} U_1' \cdot V}$$

By the inductive hypothesis we have that $U_1' = U_0'\{{}^{\mu h.U}/_{h}\}$ and the thesis follows easily.

□

Lemma 3.3.3. *If $U_0, \mathcal{R}_0 \xrightarrow{a_0} U_1, \mathcal{R}_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{n-1}} U_n, \mathcal{R}_n$ and k is the least index such that $\mu h.U$ is the redex of $U_k \xrightarrow{a_k} U_{k+1}$ and rule (rec) is never used in reducing $U_i, i < k$ then $U_i = C_i[\mu h.U]$ for some C_i and $a_k = \varepsilon$.*

Lemma 3.3.4. *Let $U_0, \mathcal{R}_0 \xrightarrow{a_0} U_1, \mathcal{R}_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{n-1}} U_n, \mathcal{R}_n$ be a computation and let k be the least index such that $\mu h.U$ is the redex of $U_k \xrightarrow{a_k} U_{k+1}$ and rule (rec) is never used in reducing $U_i, i < k$ then $\exists U_i' \equiv U_i, i < k$ such that*
$U_0', \mathcal{R}_0 \xrightarrow{a_0} U_1', \mathcal{R}_1 \xrightarrow{a_1} \cdots U_k' \xrightarrow{a_{k+1}} U_{k+2}, \mathcal{R}_{k+2} \xrightarrow{a_{k+2}} \cdots \xrightarrow{a_{n-1}} U_n, \mathcal{R}_n$ and (rec) is never used reducing $U_i, 0 \leq i \leq k$.

Proof. By Lemma 3.3.3 $U_i = C_i[\mu h.U], i \leq k$, also $\mu h.U$ is never the redex of $U_i, i < k$, hence also $U_i' = C_i[U\{{}^{\mu h.U}/_{h}\}], \mathcal{R}_i \xrightarrow{a_i} U_{i+1}' = C_{i+1}[U\{{}^{\mu h.U}/_{h}\}], \mathcal{R}_{i+1}$. By Lemma 3.3.2 $U_k' = U_k[U\{{}^{\mu h.U}/_{h}\}] = U_{k+1}$. □

Lemma 3.3.5. *Let $U_0, \mathcal{R}_0 \xrightarrow{a_0} U_1, \mathcal{R}_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{n-1}} U_n$ then $\exists U_0' \equiv U_0$ such that $U_0', \mathcal{R}_0 \xrightarrow{a_0 a_1 \ldots a_{n-1}} U_m', \mathcal{R}_m$ for some $U_m', \mathcal{R}_m$ such that $\forall i. U_i', \mathcal{R}_i \xrightarrow{a_i} U_{i+1}', \mathcal{R}_{i+1}$ is deduced using (rec) rule.*

Proof. By repeated application of Lemma 3.3.4. □

Property 3.3.6 (of capture avoiding substitutions)**.** *Given $\mu h.U$, if h occurs in the scope of νn in U, then $U\{{}^{\mu h.U}/_{h}\}$ contains a term $\nu n'.U', n' \neq n$ and n does not occur in U'.*

Proof. Follows because unfolding a recursion is capture-avoiding and all the bound names are different. □

For simplicity , we write a_n for the dynamic resource replacing a name n in U when the rule (new) is applied.

Let U with k occurrences of νn_i be such that $[\![U]\!] = \Sigma^*$ and let
$s = a_{n_1} a_{n_2} \ldots a_{n_{k+1}} a_{n_1} a_{n_2} \ldots a_{n_{k+1}}$. By Lemma 3.3.5, there exists $U' \equiv U$ such that $U', \emptyset \xrightarrow{s} \overline{U}, \{a_{n_1}, \ldots, a_{n_{k+1}}\} \cup \mathcal{R}$ with no transition deduced using rule (rec).

Since $\nu.n_{k+1}$ does not occur in U, then there exists a subterm of U of the form $\mu h.\overline{U}$, with $\nu n_i.U''$, $(0 < i \leq k)$ in $\overline{U}$. Therefore, $U' = U\{\mu h.\overline{U}/h\}$ and the replacing term contains $\nu n_{k+1}.U''\{n_{k+1}/n_i\}, n_{k+1} \neq n_i$ for some i because our assumption of keeping bound names apart.

Therefore $\nu n_{k+1}.U''\{n_{k+1}/n_i\}$ occurs in U' for some U'' (by Lemma 3.3.5) and n_i does not occur in $U''\{n_{k+1}/n_i\}$. Also n_{k+1} must occur at least twice in $U''\{n_{k+1}/n_i\}$ and nowhere else. Since $U''\{n_{k+1}/n_i\}$ cannot generate a_{n_i}, it cannot generate $a_{n_{k+1}}a_{n_1}\dots a_{n_k}a_{n_{k+1}}$.

□

We now show a consequence of Theorem 3.3.1 above. Let us consider again the load balancer example in Example 2.5.2. Note that there, the locations `loc` where the services are deployed, are never released. This feature inhibits the reuse of locations, that is useul in many cases.

To allow locations reuse we can use the ˥ operator, as follows.

Example 3.3.7.

```
fun worker2 =
    if COND then
        let loc = ν in
          ...      //activating a new service in location loc
          with(ServiceActiveIn(loc)) in
            ˥ loc;
            worker2();
    else
        skip;
```

The abstract behaviour of a computation of the `worker2` function is a trace of the form:

$$\texttt{new}(a_1)\texttt{del}(a_1)\texttt{new}(a_2)\texttt{del}(a_2)\dots\texttt{new}(a_n)\texttt{del}(a_n)$$

where $a_1, a_2, \dots, a_n$ are the locations obtained by invoking the operator ν and they are not necessarily distinct. By Theorem 3.3.1 there is no usage recognising such traces.

We compare now Usages with *quasi context-free languages*(of Section 1.3.5) 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 Usages do. The strings generated by a usage belong to $(\mathsf{Act} \times \Sigma)^*$. Our choice here is to consider Act as a singleton, then to ignore the only actions in strings, e.g. the string $\alpha(a_1)\alpha(a_2)\alpha(a_3)$ is equivalent to $a_1a_2a_3$.

Property 3.3.8. *There exists*

1. *a language generated by a usage U that is not quasi context-free;*
2. *a quasi context-free language that can not be generated by any usage U.*

Proof. For showing statement (1), 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. The statement (2) holds because there is no usage U such that $[\![U]\!] = \Sigma^*$. $\square$

Discussions

We have first studied some language theoretic properties of UA. We showed that the expressive power of UA is weaker than the one of VFA.

We slightly extended the symbolic technique of [6] to model-check the compliance of the traces generated by a Usages against a property expressed in terms of a VFA.

We investigated the expressiveness of Usages, showing that they are not able to express the behaviour of classes of ContextML programs and they are not able to recognise Σ^*

The lack of expressiveness of Usages in both cases comes from the absence of an explicit mechanism for disposing and reusing resources.

This observations foster in the next chapter the development of more expressive automata, capturing the expressiveness of both Usages and quasi context-free languages and allowing resource disposal and reuse.

Chapter 4

Automata-Based Models

In this chapter we propose two novel automata models for representing the behaviour of programs that uses fresh resources. Our automata will overcome some of the limitations of Usages shown in Theorem 3.3.1, and will be able to represent the behaviour arising from adaptive programs highlighted by the ContextML examples below.

We introduce a new variant of our running example in Examples 2.5.2 and 3.3.7, where the ⌉ operator has been moved after `worker3` invocation.

Example 4.0.9.

```
fun worker3 =
   if COND then
      let loc = ν in
         ...       //activating a new service in location loc
         with(ServiceActiveIn(loc)) in
            worker3();
            ⌉ loc;
   else
      skip;
```

Intuitively, the abstract behaviour of a computation of the `worker3` function is a trace of the form:

$$\mathtt{new(a_1)new(a_2)}\ldots\mathtt{new(a_n)}\ \mathtt{del(a_n)}\ldots\mathtt{del(a_2)del(a_1)}$$

where $a_1, a_2, \ldots, a_n$ are the distinct locations obtained by ν. The locations appear in a pattern that have the same structure of the words in the context-free language $\{ww^R\}$, where w^R stands for the reverse of w.

We can add a further ingredient to our running examples by extending Example 3.3.7.

Example 4.0.10.

```
fun worker4 =
    if COND then
      let loc = ν in
        ...     //activating a new service in location loc
        with(ServiceActiveIn(loc)) in
          ⌉ loc;
          worker4();
    else
      clients();
fun clients =
     if COND2
      ServiceActiveIn(x). InvokeServiceIn(x);
      unwith(ServiceActiveIn) in clients()
     else
      skip;
```

After all services are deployed, function `worker4` serves the clients by calling `clients`. The `clients` function invokes and removes the services active in the context one by one.

The traces generated by the program above are of the form

$\mathtt{new}(a_1)\mathtt{del}(a_1)\dots\mathtt{new}(a_n)\mathtt{del}(a_n)\mathtt{InvokeServiceIn}(a_n)\dots\mathtt{InvokeServiceIn}(a_1)$

where $a_1, \dots, a_n \in \Sigma_d$.

The main point here is that the locations $a_1, \dots, a_n$ are referred in the invocation of `InvokeServiceIn` *after* their disposal. This example emphasises that the interaction between freshness and disposal adhere to a different mechanisms than standard variable scoping. We refer to this phenomenon as *late usage* of a resource.

We address the various aspects highlighted above in an abstract model that

(i) expresses the context-free behaviour highlighted in Example 4.0.9

(ii) handles unboundedly many resources (as required in Example 2.5.1),

(iii) with unbounded freshness, as seen in Example 2.5.2

(iv) with a resource disposal mechanism, as seen in Example 3.3.7

(v) grants the late use of resources of Example 4.0.10.

We express context-free resource usage traces through *Pushdown Nominal Automata* (PSNA), that extend classical pushdown automata, so satisfying requirement (i). The alphabet of PSNA is infinite, so we undertake item (ii) above in the style of nominal techniques [31].

To guarantee freshness of resources (i.e. of the symbols of the alphabet), PSNA exploit (finitely many) additional structures, called *m-registers*, that store resources. A resource is fresh if no m-register contains it. Since m-registers have unbounded capacity, we guarantee unbounded freshness (item (iii)). As expected, when a resource is released, it is removed from the m-register that stores it, and so it can be re-used as fresh later on, so addressing item (iv). We grant a late usage of resources allowing them to be mentioned after they have been disposed, as required by item (v). The stack of the PSNA is the key to obtain such a behaviour: a resource in use, i.e. occurring in a m-register, say N, can be mentioned in the stack, and appear on its top also after it has been removed by N. In the over-simplified example above, the picked locations $a$will be recorded in a m-register N, and pushed on the stack. After $\urcorner a$, one can still access a because it is still recorded on the stack.

To keep low the complexity of the definitions and proofs, our formal development will not consider actions, but resources only.

Here we also investigate the problem of handling unboundedly many resources and of unbounded freshness for regular languages. To do that, we introduce the new model of *Finite State Nominal Automata* (FSNA), that are finite state automata enriched with a finite number of m-registers.

We then prove that the languages recognized by both FSNA and PSNA are closed under union, and the first ones are also closed under intersection, provided that symbols are not released. The intersection of a language accepted by a FSNA with that of a PSNA is recognised by a PSNA, provided that neither automata release resources. Neither the languages of FSNA and PSNA, instead, are closed under complement and concatenation.

We also establish the decidability of the emptiness problem for FSNA and for the subclass of PSNA in which symbols are not released. Consequently, it is feasible to model-check a property expressed as a (resticted) FSNA against a model expressed as a (restricted) PSNA, by verifying the emptiness of their intersection, in the style of [60].

We also compare the expressiveness and some properties of our models with other proposal in the literature. In particular, we considered the regular languages over infinite alphabet and their recognizers investigated in [33, 38, 6, 58, 59, 14, 44, 20, 40], as well as the context-free languages over infinite alphabet of [19, 6, 13, 15, 48, 46, 49].

We start by introducing the *mindful registers* (*m-registers* for short).

4.1 m-registers

An m-register N is actually a stack S of symbols in Σ_d and an *activation state* ($x \in \{1, 0\}$). An empty stack makes the m-register *empty*, as well, and we denote it by _. When the tag x is 1 then the m-register is *active*, otherwise the m-register is *inactive*. The operations on an m-register N are built on the standard `push`, `top` and `pop` operations as follows:

$$\begin{aligned}
\text{s-push}(a, \langle x, S\rangle) &= \langle 1, \texttt{push}(a, S)\rangle \\
\text{s-top}(\langle 1, S\rangle) &= \texttt{top}(S) \\
\text{s-pop}(\langle x, S\rangle) &= \begin{cases} \langle 0, \texttt{pop}(S)\rangle \text{ if } S \neq \_ \\ \langle 0, S\rangle \text{ if } S = \_ \end{cases}
\end{aligned}$$

An s-push operation makes an m-register active, regardless of its activation state. The operation s-top yields a value only if the m-register is active, otherwise it is undefined, as well as when the m-register is empty. Finally, after a s-pop, the m-register N becomes inactive. Note that s-popping an empty m-register results in a no-operation, so making it impossible to discern an inactive m-register from an empty one.

A symbol is *fresh* with respect to an m-register when it does not appear in its stack.

4.2 Finite State Nominal Automata

Before formally defining *Finite State Nominal Automata* (FSNA for short) we intuitively illustrate their recognising mechanisms through the automaton R_0 in Figure 4.1. A run on R_0 recognising a word w is a sequence of configurations leading from its initial state q_0 to its final state q_2. We assume that R_0 has two m-registers that will be part of configurations. In the initial configuration the m-registers are empty and we render them as $\left[\_, \_\right]$.

The leftmost edge is $q_0 \xrightarrow[2+]{\varepsilon} q_1$, and following it the automaton reads no symbol (recorded by ε) and goes from the configuration $\langle q_0, \left[\_, \_\right]\rangle$ to a configuration of the form $\langle q_1, \left[\_, \boxed{a}\right]\rangle$ where a symbol $a \in \Sigma_d$ is s-pushed in the m-register number 2, as dictated by the label 2+, provided it is fresh w.r.t. both the

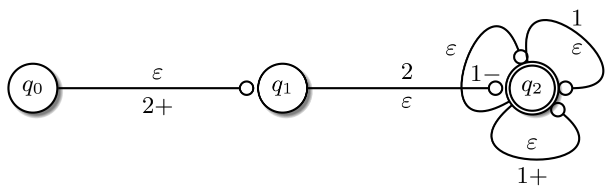


Figure 4.1: The FSNA R_0 recognising the language $\{aw \mid a \in \Sigma_d, w \in \Sigma_d^*, a \notin |w|\}$

m-registers of R_0. By using the edge $q_1 \xrightarrow[\varepsilon]{2} q_2$ the automaton reaches the configuration $\langle q_2, [\_, \boxed{\text{a}}] \rangle$ and reads a, i.e. the s-top symbol of the m-register number 2, while nothing is done on the m-registers because of the label ε.

There are three edges looping in state q_2. The edge $q_2 \xrightarrow[1+]{\varepsilon} q_2$ s-pushes a *fresh* symbol in the m-register number 1 and reads no symbol; $q_2 \xrightarrow[\varepsilon]{1} q_2$ recognises the s-top symbol of m-register number 1 and leaves the m-registers untouched. Slightly differently, the edge labelled $q_2 \xrightarrow[1-]{\varepsilon} q_2$ s-pops a symbol from the m-register number 1 (because the label is $1-$) and recognises no symbol. After following it the m-register number 1 becomes inactive and the edge $q_2 \xrightarrow[\varepsilon]{1} q_2$ can not be followed.

A run on R_0 is $\langle q_0, [\_, \_] \rangle \xrightarrow{\varepsilon} \langle q_1, [\_, \boxed{\text{a}}] \rangle \xrightarrow{a} \langle q_2, [\_, \boxed{\text{a}}] \rangle \xrightarrow{\varepsilon} \langle q_2, [\boxed{\text{b}}, \boxed{\text{a}}] \rangle \xrightarrow{b} \langle q_2, [\boxed{\text{b}}, \boxed{\text{a}}] \rangle \xrightarrow{\varepsilon} \langle q_2, [\_, \boxed{\text{a}}] \rangle \xrightarrow{\varepsilon} \langle q_2, [\boxed{\text{c}}, \boxed{\text{a}}] \rangle \xrightarrow{c} \langle q_2, \varepsilon, [\boxed{\text{c}}, \boxed{\text{a}}] \rangle$.

The reader may convince himself that the language recognised by R_0 is $\{aw \mid a \in \Sigma_d, w \in \Sigma_d^*, a \notin |w|\}$.

In the formal definition and hereafter we use some notation and abbreviation introduced in Chapter 1. We denote the set of the natural numbers by $\mathbb{N}$, $\underline{\mathtt{r}}$ is the segment of the natural numbers $\{i \mid 1 \leq i \leq r\}$, w is a word in Σ^* with length $|w|$, the set $\|w\|$ denotes the symbols used in w, ε is the empty word.

Definition 4.2.1 (Finite State Nominal Automata).
A *finite state nominal automaton* (FSNA) is $R = \langle Q, q_0, \Sigma, \delta, r, F \rangle$ where:

- Q is a finite set of states, $q, q_1, q', \dots \in Q$
- $q_0 \in Q$
- $\Sigma = \Sigma_s \cup \Sigma_d$ is the infinite alphabet (Σ_s is finite, Σ_d denumerable, $\Sigma_s \cap \Sigma_d = \emptyset$)

- $r \in \mathbb{N}$ is the number *m-registers*
- δ is the transition relation between pairs (q, σ) and (q', Δ), $\sigma \in \Sigma_s \cup \underline{\mathtt{r}} \cup \{\varepsilon\}, \Delta \in \{i+, i- \mid i \in \underline{\mathtt{r}}\} \cup \{\varepsilon\}$
  We call a transition *new* when $\Delta = i+$; *delete* when $\Delta = i-$; *update* when $\Delta \neq \varepsilon$.
  For brevity, we write $q \xrightarrow[\Delta]{\sigma} q' \in \delta$ whenever $(q, \sigma, q', \Delta) \in \delta$
- $F \subseteq Q$ is the set of final states

A *configuration* is a tuple $C = \langle q, w, [N_1, \ldots, N_r] \rangle$ where q is the current state, $w \in \Sigma^*$ is the word to be recognised and $[N_1, \ldots, N_r]$ is an array of r *m-registers* with symbols in Σ_d. The configuration $\langle q_f \in F, \varepsilon, [N_1, \ldots, N_r] \rangle$ is *final.*

The application of a transition is detailed in the following definition:

Definition 4.2.2 (Recognizing Step)**.** Given an FSNA R, a step $\langle q, w, [N_1, \ldots, N_r] \rangle \rightarrow \langle q', w', [N'_1, \ldots, N'_r] \rangle$ occurs if and only if there exists a transition $q \xrightarrow[\Delta]{\sigma} q' \in \delta$ such that both conditions hold:

1. $$\begin{cases} \sigma = \varepsilon & \Rightarrow w = w' \text{ and} \\ \sigma = i & \Rightarrow w = \text{s-top}(N_i)w' \text{ and} \\ \sigma \in \Sigma_s & \Rightarrow w = \sigma w' \end{cases}$$

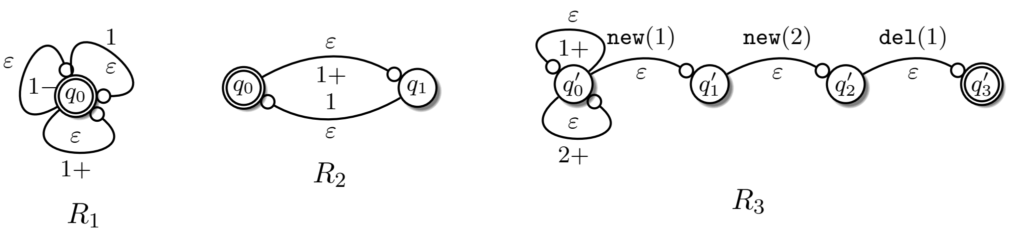
2. $$\begin{cases} \Delta = i+ & \Rightarrow N'_i = \text{s-push}(b, N_i) \wedge \forall j.b \notin \|N_j\| \wedge \forall j \ (j \neq i).N_j = N'_j \text{ and} \\ \Delta = i- & \Rightarrow N'_i = \text{s-pop}(N_i) \wedge \forall j \ (j \neq i).N_j = N'_j \text{ and} \\ \Delta = \varepsilon & \Rightarrow \forall j.N_j = N'_j \end{cases}$$

Finally, the (nominal) language accepted by R is
$L(R) = \{w \in \Sigma^* \mid \langle C_1 = \langle q_0, w, [\_, \ldots, \_] \rangle \rightarrow^* C_k, \text{with } C_k \text{ final}\}$ and we call it *regular.*

A couple of examples follow.

Example 4.2.1. *The FSNA R_1 in Figure 4.2 recognises Σ^*. The run ρ_1 recognises the word aab. Note that any symbol can be chosen in place of b, even a itself, because the m-registers are empty when a fresh symbol is required by the edge labelled* 1+.

By removing the edge $q_0 \xrightarrow[1-]{\varepsilon} q_0$ from R_1 we obtain the automaton R_2 in Figure 4.2. Without that deletion edge, there is no way to forget a symbol from



$\rho_1 = \langle q_0, aax, [\_], \_\rangle \xrightarrow{\varepsilon} \langle q_0, aax, [\boxed{\text{a}}]\rangle \xrightarrow{a} \langle q_0, ax[\boxed{\text{a}}]\rangle \xrightarrow{a} \langle q_0, x, [\_]\rangle \xrightarrow{\varepsilon} \langle q_0, x, [\boxed{\text{x}}]\rangle \xrightarrow{x} \langle q_0, \varepsilon, [\boxed{\text{x}}]\rangle$

$\rho_2 = \langle q_0, abc, [\_]\rangle \xrightarrow{\varepsilon} \langle q_1, abc, [\boxed{\text{a}}]\rangle \xrightarrow{a} \langle q_0, bc, [\boxed{\text{a}}]\rangle \xrightarrow{\varepsilon} \langle q_1, bc, [\boxed{\begin{array}{c}\text{b}\\\text{a}\end{array}}]\rangle \xrightarrow{b} \langle q_0, c, [\boxed{\begin{array}{c}\text{b}\\\text{a}\end{array}}]\rangle \xrightarrow{\varepsilon}$
$\langle q_1, c, [\boxed{\begin{array}{c}\text{c}\\\text{b}\\\text{a}\end{array}}]\rangle \xrightarrow{c} \langle q_0, \varepsilon, [\boxed{\begin{array}{c}\text{c}\\\text{b}\\\text{a}\end{array}}]\rangle$

$\rho_3 = \langle q_0', \mathtt{new}(a)\mathtt{new}(b)\mathtt{del}(a), [\_, \_]\rangle \xrightarrow{\varepsilon} \langle q_0', \mathtt{new}(a)\mathtt{new}(b)\mathtt{del}(a), [\boxed{\mathtt{new(a)}}, \_]\rangle \xrightarrow{\varepsilon}$
$\langle q_0', \mathtt{new}(a)\mathtt{new}(b)\mathtt{del}(a), [\boxed{\mathtt{new(a)}}, \boxed{\mathtt{new(b)}}]\rangle \xrightarrow{\mathtt{new}(a)} \langle q_1', \mathtt{new}(b)\mathtt{del}(a), [\boxed{\mathtt{new(a)}}, \boxed{\mathtt{new(b)}}]\rangle \xrightarrow{\mathtt{new}(b)}$
$\langle q_2', \mathtt{del}(a), [\boxed{\mathtt{new(a)}}, \boxed{\mathtt{new(b)}}]\rangle \xrightarrow{\mathtt{del}(a)} \langle q_3', \varepsilon, [\boxed{\mathtt{new(a)}}, \boxed{\mathtt{new(b)}}]\rangle$

Figure 4.2: Three examples of FSNA R_i and of their runs ρ_i. The automaton R_1 accepts Σ^*; note that the dynamic symbol x can be any symbol in Σ_d, even a, because the m-register is empty when x is s-pushed and there is no restriction on its freshness. The automaton R_2 accepts L_0 in Example 4.2.1; and R_3 accepts strings $\mathtt{new}(a)\mathtt{new}(b)\mathtt{new}(a)$ $(a \neq b)$.

the m-registers. Hence all the issued symbols are recorded in the m-registers stack, and when a new symbol is s-pushed it must be fresh with respect to all of them. The language accepted by the FSNA R_2 is $L_0 = \{w \in \Sigma_d^* \mid \forall i, j.\, w[i] \neq w[j]\}$. The run ρ_2 recognises the string abc.

The next example considers traces of data-words, the symbols of which consist of an action applied to a resource. As shown below, only minor variations of our automata are required to handle a finite number of actions acting on both static and dynamic resources.

Example 4.2.2. *Consider again the* `worker3` *function in the Example 4.0.9. We consider traces in the form of data-words, where the actions are* `new` *and* `del`*, standing for picking and disposing resources. The FSNA R_3 in Figure 4.2 accepts the unwanted traces where a second resource is picked (*$\mathtt{new}(2)$*) before having released the last one picked (*$\mathtt{del}(1)$*). Note that the symbols σ assume the form* $\mathtt{new}(u), \mathtt{del}(u), u \in \underline{2} \cup \Sigma_s$.

We introduce now a sub-class of FSNA where no delete transitions are allowed, as for the automaton R_2 in Figure 4.2. It will come out that this restriction reduces the expressiveness of FSNA, e.g. none of this restricted automata can accept Σ^*.

Definition 4.2.3 (FSNA$_+$)**.** An FSNA$_+$ is a FSNA with no delete transition $q \xrightarrow[i-]{\sigma} q'$.

In the statement below $\mathcal{L}(\text{FSNA}_+), \mathcal{L}(\text{FSNA})$ denote the classes of language of FSNA$_+$, FSNA, respectively.

Property 4.2.3. *$\mathcal{L}(FSNA_+) \subset \mathcal{L}(FSNA)$*

Proof. By contradiction, assume there exists a FSNA$_+$ with r m-registers that accepts Σ^*. Consider a string ww with $|w| = |\|w\|| = r + 1$ and let $b \in \|w\|$ be such that $b \neq$ s-top(N_i) after w has been scanned; note that such a b always exists. However, ww cannot be accepted because b still occurs in one of the r m-registers and thus cannot be s-pushed again. □

We enable now our FSNA$_+$s to update more than one m-register in a single transition. This variation will be useful in the proof of Theorem 4.3.5, as expected, this parallelization does not extend the expressiveness of the FSNA. For simplicity, we permit below to update two m-registers only, the extension to any finite number being straightforward.

Definition 4.2.4 (Finite Nominal Automata 2)**.** A *finite state nominal automaton 2* (FSNA2) is a tuple $R = \langle Q, q_0, \Sigma, \delta 2, r, F\rangle$ where:

- Q, q_0, Σ, r and F are as in Definition 4.2.1
- $\delta 2$ is a relation between triples (q, σ) and $(q', (\Delta_1, \Delta_2))$ such that $\Delta_1 = \Delta_2$ only if they are both ε and $\forall i \in \underline{\mathrm{r}}. \Delta_1, \Delta_2 \neq i-$

Definition 4.2.5 (Computation step)**.**
A step $\langle q, w, [N_1, \dots, N_r]\rangle \to \langle q', w', [N'_1, \dots, N'_r]\rangle$ occurs iff there exists $q \xrightarrow[(\Delta_1,\Delta_2)]{\sigma} q' \in \delta 2$ such that

1. As in Definition 4.2.2
2. $\begin{cases} \forall j \in \underline{\mathrm{r}}\ (j \neq \Delta_1, \Delta_2).\, N_j = N'_j \text{ and} \\ \Delta_1 \neq \varepsilon \Rightarrow N'_{\Delta_1} = \text{s-push}(b_1, N_{\Delta_1}), \forall j \in \underline{\mathrm{r}}.\, b_1 \notin \|N_j\|, b_1 \neq \text{s-top}(N'_{\Delta_2}) \text{ and} \\ \Delta_2 \neq \varepsilon \Rightarrow N'_{\Delta_2} = \text{s-push}(b_2, N_{\Delta_2}), \forall j \in \underline{\mathrm{r}}.\, b_2 \notin \|N_j\|, b_2 \neq \text{s-top}(N'_{\Delta_1}) \text{ and} \end{cases}$

As anticipated, the FSNA2 have the same expressive power of FSNA$_+$.

Theorem 4.2.4. *Given a FSNA2 A there exists an FSNA$_+$ A' accepting the same language.*

Proof. Let $R = \langle Q, q_0, \Sigma, \delta 2, r, F\rangle$ and define $R' = \langle Q \times \{1\} \cup Q \times \{2\}, (q_0, 1), \Sigma, \delta, r, F \times \{1\}\rangle$ where $(q, 1) \xrightarrow[\Delta_1]{\sigma} (q, 2)$, $(q, 2) \xrightarrow[\Delta_2]{\varepsilon} (q', 1) \in \delta$ iff $q \xrightarrow[(\Delta_1,\Delta_2)]{\sigma} q' \in \delta 2$. Now it is immediate proving the equality of the accepted languages. □

4.2.1 Some Properties

This section studies some language theoretical properties of the two classes of automata FSNA and FSNA$_+$ introduced so far.

The introduction of m-registers is not sufficient for breaking the barrier between regular and context-free languages, beacause m-registers are not full-fledged stacks: they become inactive after an s-pop. This is shown by the following “classical” example. Although it is immediate to see that increasing the number of m-registers increases the expressive power of FSNA, we can only have finitely many m-registers, and so the following Dyck-like language is not regular.

Example 4.2.5. *Let $L_r = \{ww^R \in \Sigma_d^* \mid |w| = r$ and $\forall i, j.\, w[i] \neq w[j]\}$ then no FSNA R with less that r states and r m-registers accepts L_r. Indeed, a standard argument on FSA proves that r states are required. Assume now that R has less than r registers N_i and accepts ww^R. By the pigeonhole principle, there is at least a symbol of w, say a, such that $\forall i.a \neq$ s-top(N_i) when w has been read. Since $a \in \|w\|$, a needs to be s-pushed while traversing w^R, but it is fresh so it can be replaced by any other (fresh) different symbol, which makes R to accept also ww'^R, where $w' \neq w^R$: contradiction.*

We establish now a few closure properties w.r.t. standard language operations: union ($\cup$), intersection ($\cap$), complementation ($\overline{\ \cdot\ }$), concatenation ($\cdot$) and Kleene star ($*$).

To simplify and structure the proofs of these properties we need some auxiliary definitions.

Definition 4.2.6 (Merge function)**.** Let $m : \{1, 2\} \times \underline{\mathtt{r}} \to \underline{2\mathtt{r}}$ be a function. Stipulating $m_1(x) = m(1, x), m_2(x) = m(2, x)$, m is a merge iff m_1 and m_2 are injective.

The registers i, j are *merged* by m, in symbols $i \overset{\mathtt{m}}{\leftrightarrow} j$, when $m_1(i) = m_2(j)$, we write $i \overset{\mathtt{m}}{\not\leftrightarrow} j$ when they are not merged and whenever $i = \varepsilon$ or $j = \varepsilon$.

We stipulate that m extends to a relation between m-registers such that $m[N_1^1, \ldots, N_r^1, N_1^2, \ldots, N_r^2] = [M_1, \ldots, M_{2r}]$ iff $\forall i \in \underline{\mathtt{r}}, j \in \underline{\mathtt{2r}}$. N_i^1, N_i^2, M_j are active and $\forall i, j \in \underline{\mathtt{r}}$

1. s-top(N_i^1) = s-top$(M_{m_1(i)})$ and s-top(N_i^2) = s-top$(M_{m_2(i)})$
2. s-top(N_i^1) = s-top(N_j^2) then $i \overset{\mathtt{m}}{\longleftrightarrow} j$
3. $\bigcup_{i\in\underline{\mathtt{r}}} \|N_i^1\| \cup \bigcup_{i\in\underline{\mathtt{r}}} \|N_i^2\| = \bigcup_{i\in\underline{\mathtt{2r}}} \|M_i\|$

Definition 4.2.7 (Effective Update)**.** Given two merge functions m, m', the *effective update* $\overset{m}{m'}(\Delta_1, \Delta_2)$ of $\Delta_1, \Delta_2 \in \underline{\mathtt{r}} \cup \{\varepsilon\}$ is the pair $(\overline{\Delta_1}, \overline{\Delta_2})$ where:
if $\Delta_1 \overset{\mathtt{m}'}{\longleftrightarrow} \Delta_2$ then $\overline{\Delta_1} = m_1'(\Delta_1)$ and $\overline{\Delta_2} = \varepsilon$;
if $\Delta_1 \overset{\mathtt{m}'}{\not\longleftrightarrow} \Delta_2$ then

- $\overline{\Delta_1}$ is such that:
  - if $\Delta_1 = \varepsilon$ then $\overline{\Delta_1} = \varepsilon$
    else if $m_1(\Delta_1) \neq m_1'(\Delta_1)$ then $\overline{\Delta_1} = \varepsilon$ else $\overline{\Delta_1} = m_1'(\Delta_1)$
- $\overline{\Delta_2}$ is such that:
  - if $\Delta_2 = \varepsilon$ then $\overline{\Delta_2} = \varepsilon$
    else if $m_2(\Delta_2) \neq m_2'(\Delta_2)$ then $\overline{\Delta_2} = \varepsilon$ else $\overline{\Delta_2} = m_2'(\Delta_2)$

Definition 4.2.8 (Evolution)**.** Given a merge m we say that the merge m' is the *evolution* of m with respect to Δ_1, Δ_2, in symbols $m \overset{\Delta_1,\Delta_2}{\curvearrowright} m'$, iff

1. $\forall i \in \{1, 2\}, j \in \underline{\mathtt{r}}$ $(j \neq \Delta_1, \Delta_2)$. $m_i(j) = m_i'(j)$
2. if $\Delta_1 \overset{\mathtt{m}}{\longleftrightarrow} j, j \neq \Delta_2$ then $\Delta_1 \overset{\mathtt{m}'}{\not\longleftrightarrow} j$
3. if $i \overset{\mathtt{m}}{\longleftrightarrow} \Delta_2, i \neq \Delta_1$ then $i \overset{\mathtt{m}'}{\not\longleftrightarrow} \Delta_2$

To intuitively illustrate the definitions above, in Figure 4.3 we show a portion of the automaton $R = R_1 \cap R_2$ recognising the intersection of the two FSNA$_+$ R_1 and R_2.

The intersection automaton R is obtained by the standard construction that builds the new states as the product of the old ones. Additionally, each pair $\langle q^1, q^2\rangle$ $(q^1, q^2 \in R_1, R_2$ resp.) is enriched with a *merge* function m. The m describes how the m-registers of the two automata are mapped into those of R

The idea underlying m is to guarantee the following invariant $\mathcal{I}$ along the runs: if R_1 and R_2 are in configurations $\langle q_0^1, w, [\![\boxed{\mathrm{y}}]\!]\rangle$ and $\langle q_0^2, w, [\![\boxed{\mathrm{x}}]\!]\rangle$ then R

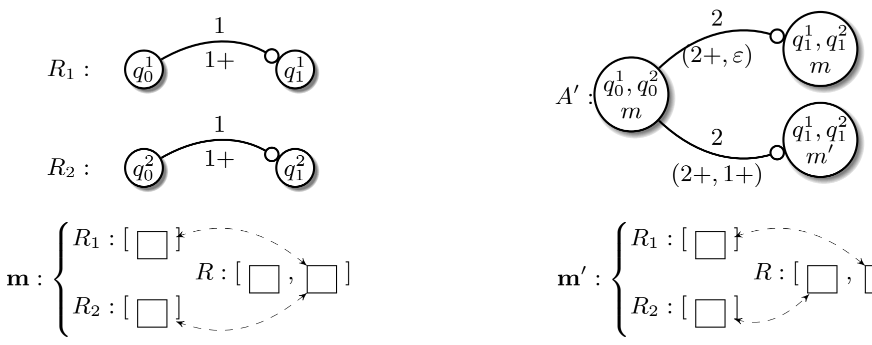


Figure 4.3: R is a portion of the FSNA2 automaton recognizing the intersection of the languages of R_1 and R_2, the diagrams at the bottom represent the merges m, m', where $m_1(1) = 2, m_2(1) = 2, m'_1(1) = 2, m'_2(1) = 1$.

will be in configuration $\langle\langle q_0, q'_0, m\rangle, w, [\boxed{h}, \boxed{z}]\rangle$ and if two m-registers have the same s-tops then they are merged by m (and vice versa). This is illustrated in the left-most configurations of Figure 4.4: if $x = y = a$ then m maps the two registers to one register of R (here the second one), and $z = a$. The edges of the automaton are also defined in the standard way. However, the m-register mentioned in σ of R is the one merged by m, provided that R_1 and R_2 agree on σ. Also the updates $(\overline{\Delta}, \overline{\Delta'})$ in R are determined by the updates Δ_1 of R_1 and Δ_2 of R_2 under the merge m, and form an effective update (see Definition 4.2.7).

Consider again Figure 4.3. The transition $t : \langle\langle q_0^1, q_0^2\rangle, m\rangle \xrightarrow[2+]{2} \langle\langle q_1^1, q_1^2\rangle, m\rangle$ is present because there are $q_0^1 \xrightarrow[1+]{1} q_1^1$ and $q_0^2 \xrightarrow[1+]{1} q_1^2$ and m maps the first m-register of R_1 and that of R_2 to the second of R. Instead, the state $\langle\langle q_0^1, q_0^2\rangle, m'\rangle$ (omitted in the figure) has no outgoing edges, because the symbols read by R_1 and R_2 are kept apart by m'.

There are transitions that only differ for the merge function in their target state. Not all the possible merges respect however the invariant $\mathcal{I}$ mentioned above. Indeed, we only keep those that are *evolution* (in the sense of Definition 4.2.8) of the merge in the source state, according to the updates Δ_1, Δ_2 of R_1, R_2 respectively. For example, the transition $\langle\langle q_0^1, q_0^2\rangle, m\rangle \xrightarrow[(1+,2+)]{2} \langle\langle q_1^2, q_1^2\rangle, m'\rangle$ permits the recognizing step $C \xrightarrow{a} C'$, where the m-register of R_1 now has got a d, while that of R_2 has got c, and m' keeps them apart.

Instead, if both m-registers store the same dynamic symbol c, the merge is still m, and the transition t above enables the step $C \xrightarrow{a} C''$ and guarantees the invariant.

We are now ready to state and prove some closure properties of FSNA and

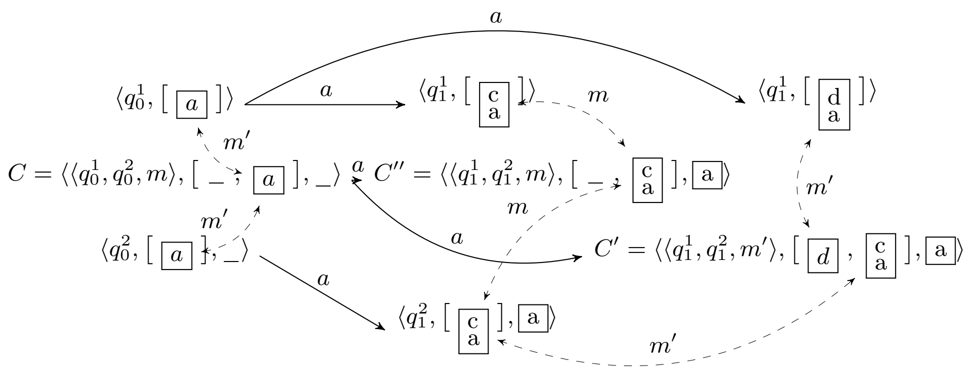

a
$\langle q_0^1, [\boxed{a}]\rangle$
a
$\langle q_1^1, [\boxed{\text{c a}}]\rangle$
m
$\langle q_1^1, [\boxed{\text{d a}}]\rangle$
m'
$C = \langle\langle q_0^1, q_0^2, m\rangle, [\_, \boxed{a}], \_\rangle$
a
$C'' = \langle\langle q_1^1, q_1^2, m\rangle, [\_, \boxed{\text{c a}}], \boxed{a}\rangle$
m'
m
m'
$\langle q_0^2, [\boxed{a}], \_\rangle$
a
$C' = \langle\langle q_1^1, q_1^2, m'\rangle, [\boxed{d}, \boxed{\text{c a}}], \boxed{a}\rangle$
a
$\langle q_1^2, [\boxed{\text{c a}}], \boxed{a}\rangle$
m'

Figure 4.4: Two recognizing steps of R (middle), built from steps of R_1 (top), and steps of R_2 (bottom). The step $C \xrightarrow{a} C'$ simultaneously updates two m-registers.

FSNA$_+$:

Theorem 4.2.6 (Closure properties)**.**

	$\cup$	$\cap$	$\overline{\cdot}$	$\bullet$	$*$
$\mathcal{L}(FSNA)$	✓	×	×	✓	✓
$\mathcal{L}(FSNA_+)$	✓	✓	×	×	×

Proof. Union: it suffices a new initial state with two outgoing ε-transition to the old initial states.

Concatenation (and Kleene star) for FSNA: a sequence of transitions from the final states of the first FSNA make inactive all the m-registers, leading to state q_c having loops that can empty them all. Then an ε-transition goes from q_c to the initial state of the second automaton (see Figure 4.6(a)).

Complement of FSNA: Consider $L = \{w \mid \exists a.w[i] = a$ and a appears $2n+1$ times in $w\}$ that is recognized by the FSNA in Figure 4.6(b). Assume that its complement $\overline{L} = \{w \mid \forall a.(\exists i.w[i] = a) \Rightarrow\ a$ appears $2n$ times$\}$ is recognized by an automaton R with r m-registers. This automaton also accepts ww, where $w = a_1, \dots, a_{r+1}, \forall i, j.a_i \neq a_j$. However, after recognizing w, there exists some a_i that is not s-top of any m-register. The word ww' where w' is obtained by w by replacing a_i with a fresh symbol b is accepted by R, as well: contradiction because $L \ni ww' \notin \overline{L}$.

Complement for FSNA$_+$: Property 4.2.3 (Σ^* is the complement of $\emptyset$) suffices.

Concatenation (and Kleene star): Consider $L = \{ww' \mid w \in L$ and $w' \in L'\}$, with L, L' languages of two FSNA$_+$ R, R'. If R accepts a string w such that

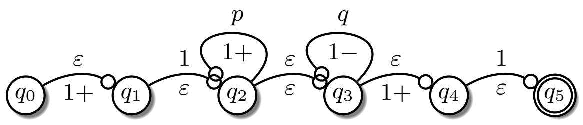


Figure 4.5: The language $L_2 = \{ap^nq^mb \mid a \in \Sigma_d, a = b \Rightarrow m > n\}$

$a \in w, a \in w'$ but $a \neq$ s-top(N_i) for all the m-registers in the final configuration of all accepting runs, then we obtain a contradiction because a cannot be s-pushed since not fresh.

Intersection of FSNA Let $L_1 = \{ap^nq^ma\}$, with $a \in \Sigma_d$ and p, q be chosen symbols in Σ_s. Clearly, L_1 is regular. Consider the language recognized by the automaton in Figure 4.5: $L_2 = \{ap^nq^mb \mid a \in \Sigma_d, a = b \Rightarrow m > n\}$. Now, the language $L_1 \cap L_2 = \{ap^nq^ma \mid m > n\}$ can not be recognized by any FSNA.

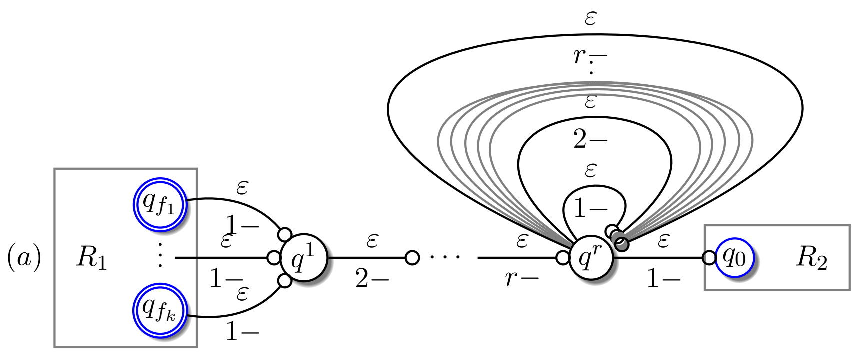
Intersection of FSNA$_+$ We formalise here the intuitive construction given in Figure 4.3, the proof uses Definition 4.2.9 and Lemmata 4.2.7 and 4.2.8 given below. Given two automata R_1 and R_2, we construct the intersection automaton $R_1 \cap R_2$ recognising $\mathcal{L}(R_1) \cap \mathcal{L}(R_2)$.
The proof of the equivalence $\mathcal{L}(R_1 \cap R_2) = \mathcal{L}(R_1) \cap \mathcal{L}(R_2)$ can be obtained by induction using Lemmata 4.2.7 and 4.2.8. Without loss of generality, we consider the simplifying assumption that all the registers are active in the initial configuration. Note that this constraint does not alter the expressiveness of FSNA$_+$.

Definition 4.2.9 (Intersection Automaton)**.** The intersection automaton of two FSNA$_+$s
$R_1 = \langle Q_1, q_0^1, \Sigma, \delta_1, r, F_1\rangle$ and $R_2 = \langle Q_2, q_0^2, \Sigma, \delta_2, r, F_2\rangle$ is the following FSNA2:

$$R_1 \cap R_2 = \langle \overline{Q}, \overline{q_0}, \Sigma, \overline{\delta}, 2r, \overline{F}\rangle, \text{ where}$$

- $\overline{Q} = Q_1 \times M \times Q_2$ where M is a set of merge functions
- $\overline{q_0} = \langle q_0^1, m^*, q_0^2\rangle$, with m^* s.t. m_1^*, m_2^* are the identity functions
- $\overline{F} = \{\langle q_1, m, q_2\rangle \mid q_1 \in F_1, q_2 \in F_2\}$
- $\langle q_1, m, q_2\rangle \xrightarrow[\Delta_1,\Delta_2]{\overline{\sigma}} \langle q_1', m', q_2'\rangle \in \overline{\delta}$ iff $m \overset{\Delta_1,\Delta_2}{\curvearrowright} m'$ and

  $q_1 \xrightarrow[\Delta_1]{\sigma_1} q_1' \in \delta_1$ and $q_2 \xrightarrow[\Delta_2]{\sigma_2} q_2' \in \delta_2$ and
  - if $\sigma_1, \sigma_2 \in \underline{\mathrm{r}}$ then $\overline{\sigma} = m_1(\sigma_1) = m_2(\sigma_2)$ and



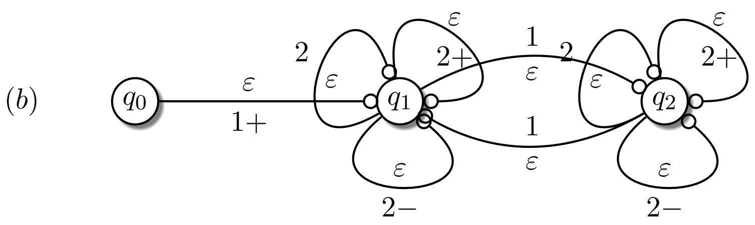


Figure 4.6: (a) The concatenation automaton of two automata R_1 and R_2 has the states and the transition of both of them, as initial state the one of R_1, as final states the ones of R_2. The final states of R_1 are connected to q^r, q^r is connected to the initial state of R_2. The self-loops in q^r are used to empty the r m-registers. (b) An automaton recognizing $L = \{w \mid \exists a.w[i] = a \text{ and } a \text{ appears } 2n+1 \text{ times in } w\}$

– if $\sigma_1, \sigma_2 \in \Sigma_s$ then $\overline{\sigma} = \sigma_1 = \sigma_2$ and

– $(\overline{\Delta_1}, \overline{\Delta_2}) = \overset{m}{m'}(\Delta_1, \Delta_2)$

or $q_1 \xrightarrow[\Delta_1]{\varepsilon} q_1' \in \delta_1$ and

– $\overline{\sigma} = \varepsilon$ and

– $q_2' = q_2$ and

– $(\overline{\Delta_1}, \overline{\Delta_2}) = \overset{m}{m'}(\Delta_1, \varepsilon)$

or $q_2 \xrightarrow[\Delta_2]{\varepsilon} q_2' \in \delta_2$
symmetric to the previous case.

Now we prove the two lemmata used before for proving that $R_1 \cap R_2$ accepts $L(R_1) \cap L(R_2)$. Intuitively, the first states that whenever two automata R_1, R_2 make a step with the same label, also the automaton $R_1 \cap R_2$ can perform the very same step.

Lemma 4.2.7. *Let R_1 and R_2 be two FSNA, let $a \neq \varepsilon$ and*
$step_1 : \langle q_1, aw, [N_1^1, \dots, N_r^1] \rangle \longrightarrow \langle q_1', w, [N_1'^1, \dots, N_r'^1] \rangle$ and
$step_2 : \langle q_2, aw, [N_1^2, \dots, N_r^2] \rangle \longrightarrow \langle q_2', w, [N_1'^2, \dots, N_r'^2] \rangle$
be steps of R_1 and R_2 respectively.
Then for any m and $[M_1 \dots, M_{2r}]$ m-registers of $R_1 \cap R_2$ such that

$$m([N_1^1, \dots, N_r^1, N_1^2, \dots, N_r^2]) = [M_1, \dots, M_{2r}]$$

there exists the step of $R_1 \cap R_2$
$\overline{step} : \langle \langle q_1, m, q_2 \rangle, aw, [M_1, \dots, M_{2r}] \rangle \longrightarrow \langle \langle q_1', m', q_2' \rangle, w, [M_1', \dots, M_{2r}'] \rangle$ with

$$m'([N_1'^1, \dots, N_r'^1, N_1'^2, \dots, N_r'^2]) = [M_1', \dots, M_{2r}']$$

Proof. Assume that $q_1 \xrightarrow[\Delta_1]{\sigma_1} q_1'$ and $q_2 \xrightarrow[\Delta_2]{\sigma_2} q_2'$ justify step$_1$ and step$_2$. Then we have the following cases, depending on the labels of these transition.

- $\sigma_1, \sigma_2 \in \underline{\mathtt{r}}$

  Define m' such that $\forall i, j \in \underline{\mathtt{r}}\, (i \neq \Delta_1, j \neq \Delta_2). m_1'(i) = m_1(i), m_2'(j) = m_2(j)$.

  If s-top$(N_{\Delta_1}'^1) =$ s-top$(N_{\Delta_2}'^2)$ $(\Delta_1, \Delta_2 \neq \varepsilon)$

  - if $\forall i. i \overset{\mathtt{m}}{\not\leftrightarrow} \Delta_2 \wedge \Delta_1 \overset{\mathtt{m}}{\not\leftrightarrow} i$ then let $m_1'(\Delta_1) = m_2(\Delta_2)$ and $m_2'(\Delta_2) = m_2(\Delta_2)$.
  - Otherwise if $k \overset{\mathtt{m}}{\leftrightarrow} \Delta_2$ or $\Delta_1 \overset{\mathtt{m}}{\leftrightarrow} k$ then let $m_1'(\Delta_1) = m_2'(\Delta_2) = h \notin$ Img(m).

  If s-top$(N_{\Delta_1}'^1) \neq$ s-top$(N_{\Delta_2}'^2)$ but

  - $m_1'(\Delta_1)$ (resp. $m_2'(\Delta_2)$) is such that s-top$(N_{\Delta_1}'^1) =$ s-top$(N_k'^2), \Delta_1 \neq \varepsilon$ for some $k \neq \Delta_2$ then let $m_1'(\Delta_1) = m_2(k)$.
  - Otherwise,
    - if $\Delta_1 \overset{\mathtt{m}}{\leftrightarrow} k$ for some $k \neq \Delta_2$, then let $m_1'(\Delta_1) = h \notin$ Img(m).
    - Otherwise, if $\Delta_1 \overset{\mathtt{m}}{\not\leftrightarrow} k$ for all $k \neq \Delta_2$ then let $m_1'(\Delta_1) \in \{h\} \cup m(\Delta_1)$, with $h \notin$ Img(m).

  Recall that m is a merge and that note that m m' may possibly differ in Δ_1, Δ_2. Now we show that also m' is a merge, by showing its projections injective. By contradiction, assume m' is not injective, then if $m'(\Delta_1) \neq m(\Delta_1)$ (resp. for Δ_2), by construction, it is only the case that $m_1'(\Delta_1) = m_2'(k)$ for some k. If $k = \Delta_2$ then $m_1'(\Delta_1) = m_2'(k) \notin$ Img(m), contradiction because m'_1, m'_2 are injective since there is no k s.t. $m'_1(k) = m'_1(\Delta_1)$ or $m'_2(k) = m'_2(\Delta_2)$. If $k \neq \Delta_2$ then we have that only m'_2 can be non-injective, but this requires $m'_2(\Delta_2) = m'_2(k), k \neq \Delta_2$ but this is not possible by construction.

We show that $m \overset{\Delta_1,\Delta_2}{\curvearrowright} m'$: condition (1) is trivially satisfied by construction, conditions (2-3) are taken explicitly into account in the construction.

Since step$_1$ and step$_2$ fulfil the hypothesis, by condition 1.2 of Definition 4.2.2, it turns out that both $N^1_{\sigma_1}, N^2_{\sigma_2}$ are active and $a = \text{s-top}(N^1_{\sigma_1}) = \text{s-top}(N^2_{\sigma_2})$. This last fact implies that $m_1(\sigma) = m_2(\sigma)$ since m is merge. By letting $\overline{\sigma} = m_1(\sigma)$, conditions (1) and (2) imply that $\text{s-top}(M_{\overline{\sigma}}) = a$.

By construction of $A_1 \cap A_2$ we then have the transition

$$\overline{t} : \langle q_1, m, q_2 \rangle \xrightarrow[\overline{\Delta_1},\overline{\Delta_2}]{\overline{\sigma}} \langle q'_1, m', q'_2 \rangle \in \overline{\delta}$$

where $(\overline{\Delta_1}, \overline{\Delta_2}) = \overset{m}{m'}(\Delta, \Delta')$.

Next we shall prove that m' is a merge of m-registers through which $[M'_1, \dots, M'_{2r}]$ can be obtained from $[M_1, \dots, M_{2r}]$ and $\overline{\text{step}}$ is justified by $\overline{t}$.

First, $\forall i \in \underline{\mathtt{r}}(i \neq \overline{\Delta_1}, \overline{\Delta_2}).M_i = M'_i$. If $\overline{\Delta_1} \neq \varepsilon \wedge m(\Delta_1) = m'(\Delta_1)$ then let $M'_{\overline{\Delta_1}} = \text{s-push}(b_1, M_{\overline{\Delta_1}})$ with $b_1 = \text{s-top}(N'^1_{\Delta_1})$ otherwise let $M'_{\overline{\Delta_1}} = M_{\overline{\Delta_1}}$. Symmetrically for $M'_{\overline{\Delta_2}}$ (note that $\overline{\Delta_1} \neq \overline{\Delta_2}$).

We prove now that $m'([N'^1_1, \dots, N'^1_r, N'^2_1, \dots, N'^2_r]) = [M'_1 \dots, M'_{2r}]$. The conditions (1) and (2) are satisfied because for all $i, j \in \underline{\mathtt{r}}$ s.t. $i \neq \Delta_1, j \neq \Delta_2$ we have $m'_1(i) = m_1(i), m'_2(j) = m_2(j)$, the involved m-register are left untouched and m is a merge of m-registers. By construction of $M'_{m'(\Delta_1)}$ (resp. $M'_{m'(\Delta_2)}$) we also have that $\text{s-top}(M'_{m'(\Delta_1)}) = \text{s-top}(N'_{\Delta_1})$. The condition (3) is implied by the construction of m', condition (4) holds because b_1, b_2 are in $\bigcup_{i \in \underline{2\mathtt{r}}} \|M'_i\|$ only if they are in $\bigcup_{i \in \underline{\mathtt{r}}} \|N'^1_i\|$ or $\bigcup_{i \in \underline{\mathtt{r}}} \|N'^2_i\|$ respectively.

We now show that the condition 2 of Definition 4.2.5 is satisfied. With the construction above $\bigcup_{i \in \underline{2\mathtt{r}}} \|M_i\| = \bigcup_{i \in \underline{\mathtt{r}}} \|M'_i\| \setminus \{b_1, b_2\}$. Since $\bigcup_{i \in \underline{\mathtt{r}}} \|N^1_i\| \cup \bigcup_{i \in \underline{\mathtt{r}}} \|N^2_i\| = \bigcup_{i \in \underline{2\mathtt{r}}} \|M_i\|$, if $b_1 \in \bigcup_{i \in \underline{\mathtt{r}}} \|N^2_i\|$ then $b_1 \in \bigcup_{i \in \underline{\mathtt{r}}} \|M_i\|$, otherwise, by condition 2 of Definition 4.2.2 for step$_1$, $b_1 \notin \bigcup_{i \in \underline{\mathtt{r}}} \|N^1_i\|$. This holds symmetrically for b_2. Then if b_1, b_2 are pushed on top of $M_{\overline{\Delta_1}}, M_{\overline{\Delta_2}}$ then they are fresh, i.e. $b_1, b_2 \notin \bigcup_{i \in \underline{\mathtt{r}}} \|M_i\|$. Also, when both are pushed on top of $M_{\overline{\Delta_1}}, M_{\overline{\Delta_2}}$ we have $b_1 \neq b_2$, so satisfying condition 2 of Definition 4.2.5.

Since a satisfies condition 1 of Definition 4.2.5 for $\bar{t}$, the following step exists:

$$\langle\langle q_1, aw, m, q_2\rangle, [M_1, \ldots, M_r]\rangle \xrightarrow{a} \langle\langle q'_1, w, m', q'_2\rangle, [M'_1, \ldots, M'_r]\rangle$$

- if $\sigma_1, \sigma_2 \in \Sigma_s$, the proof is analogous to that of the previous case: take m' as above, then by construction of $R_1 \cap R_2$ we have the following transition, where $\overline{\sigma} = \sigma_1 = \sigma_2$

$$\bar{t} : \langle q_1, m, q_2\rangle \xrightarrow[\overline{\Delta_1, \Delta_2}]{\overline{\sigma}} \langle q'_1, m', q'_2\rangle$$

With the same construction of $[M'_1, \ldots, M'_r]$ above, we obtain

$$\langle\langle q_1, aw, m, q_2\rangle, [M_1, \ldots, M_r]\rangle \xrightarrow{a} \langle\langle q'_1, w, m', q'_2\rangle, [M'_1, \ldots, M'_r]\rangle$$

- $\sigma_1 = \varepsilon$ or $\sigma_2 = \varepsilon$, trivial.

□

The following lemma states that whenever the automaton $R_1 \cap R_2$ makes a step, also the automata R_1 and R_2 can perform the very same step.

Lemma 4.2.8. *Let R_1 and R_2 be two FSNA, let $a \neq \varepsilon$ and let $\overline{step}$: $\langle\langle q_1, m, q_2\rangle, aw, [M_1, \ldots, M_{2r}]\rangle \longrightarrow \langle\langle q'_1, m', q'_2\rangle, w, [M'_1, \ldots, M'_{2r}]\rangle$ be a step of $R_1 \cap R_2$*
then for any $[N^1_1 \ldots, N^1_r], [N^2_1 \ldots, N^2_r]$ such that

$$m([N^1_1, \ldots, N^1_r, N^2_1, \ldots, N^2_r]) = [M_1, \ldots, M_{2r}]$$

there exist two steps of R_1 and R_2
$step_1 : \langle q_1, aw, [N^1_1, \ldots, N^1_r]\rangle \longrightarrow \langle q'_1, w, [N'^1_1, \ldots, N'^1_r]\rangle$ and
$step_2 : \langle q_2, aw, [N^2_1, \ldots, N^2_r]\rangle \longrightarrow \langle q'_2, w, [N'^2_1, \ldots, N'^2_r]\rangle$ and

$$m'([N'^1_1, \ldots, N'^1_r, N'^2_1, \ldots, N'^2_r]) = [M'_1, \ldots, M'_{2r}]$$

Proof. Assume that $\langle q_1, m, q_2\rangle \xrightarrow[\overline{\Delta_1, \Delta_2}]{\overline{\sigma}} \langle q'_1, m', q'_2\rangle$ justifies $\overline{\text{step}}$.

- if $\overline{\sigma} \neq \varepsilon$ then
  by construction of $R_1 \cap A_2$ we have that $t_1 : q_1 \xrightarrow[\Delta_1]{\sigma_1} q'_1$ and $t_2 : q_2 \xrightarrow[\Delta_2]{\sigma_2} q'_2$.

  We only prove the case $\overline{\sigma} \in \underline{\mathtt{r}}$; the others follow from a similar argument.

– if $\overline{\sigma} \in \underline{\mathtt{r}}$:
let $[N_1^{'1}, \dots, N_r^{'1}]$ (resp. $[N_1^{'2}, \dots, N_r^{'2}]$) be such that $\forall i \in \underline{\mathtt{r}}\, (m'_1(i) \neq \overline{\Delta_1} \wedge m'_1(i) = m_1(i)).N_i^{'1} = N_i^1$. If $\Delta_1 \neq \varepsilon$ then take

$$N_{\Delta_1}^{'1} = \text{s-push}(\text{s-top}(M'_{m'_1(\Delta_1)}), N_{\Delta_1}^1)$$

Condition 1 of Definition 4.2.2 for t_1 is satisfied because s-top$(M_{\overline{\sigma}})$ = s-top$(N_{\sigma_1}) = a$ since $\overline{\sigma} = m_1(\sigma_1)$ by construction and m is a merge of m-registers.

We prove now condition 2 of Definition 4.2.2. For $b =$ s-top$(M'_{m'_1(\Delta_1)})$ we consider the two cases: $\overline{\Delta_1} \neq \varepsilon$ and $\overline{\Delta_1} = \varepsilon$. In the first case the condition is guaranteed by the fact that $b \notin \sum_{j \in \underline{\mathtt{2r}}} \|M_j\|$, hence, since m is a merge of m-registers (condition 3) $b \notin \sum_{j \in \underline{\mathtt{r}}} \|N_j^1\|$. When $\overline{\Delta_1} = \varepsilon$ then we have $m'_1(\Delta_1) \neq m'_2(\Delta_2)$, in this case $m'_1(\Delta_1) \neq m_1(\Delta_1)$, by injectivity of m'_1, by the fact that m is a merge of m-registers and by the fact that m'_1 only differs from m_1 in Δ_1 we have that $\forall i \in \underline{\mathtt{r}}$.s-top$(M_{m'_1(\Delta_1)}) \neq$ s-top(N_i^1).

For $c =$ s-top$(M'_{m'_2(\Delta_2)})$. We consider two cases: $\overline{\Delta_2} \neq \varepsilon$ and $\overline{\Delta_2} = \varepsilon$. In the first case the condition is guaranteed by the fact that $c \notin \sum_{j \in \underline{\mathtt{2r}}} \|M_j\|$, hence, since m is a merge of m-registers (condition 3) $c \notin \sum_{j \in \underline{\mathtt{r}}} \|N_j^2\|$. When $\overline{\Delta_2} = \varepsilon$ then we have two cases: $m'_2(\Delta_2) = m'_1(\Delta_1) = \overline{\Delta_1}$ or $m'_2(\Delta_2) \neq m'_1(\Delta_1)$. In the first case the condition is satisfied by the same reasoning above because $c =$ s-top$(M'_{\overline{\Delta_1}})$, the second case is verified only when $m'_2(\Delta_2) \neq m_2(\Delta_2)$, in this case, by injectivity of m'_2, by the fact that m is a merge of m-registers and by the fact that m'_2 only differs from m_2 in Δ_2 we have that $\forall i \in \underline{\mathtt{r}}$.s-top$(M_{m'_2(\Delta_2)}) \neq$ s-top(N_i^2).

Hence all the conditions for t_1, t_2 are satisfied, so both step$_1$ and step$_2$ exist.

We are left to prove that $m'([N_1^{'1}, \dots, N_r^{'1}, N_1^{'2}, \dots, N_r^{'2}]) = [M'_1 \dots, M'_{2r}]$. The conditions (1) and (2) are satisfied because for all $i, j \in \underline{\mathtt{r}}$ s.t. $i \neq \Delta_1, j \neq \Delta_2$ we have $m'_1(i) = m_1(i), m'_2(j) = m_2(j)$, the involved m-register are left untouched and m is a merge of m-registers. By construction of N'_{Δ_1} (resp. N'_{Δ_2}) we also have that s-top$(M'_{m'(\Delta_1)})$ = s-top(N'_{Δ_1}). The condition (3) is implied by the construction of N'_{Δ_1} because (for some j) $m'_1(\Delta_1) = m'_2(j)$ implies s-top$(N_{\Delta_1}^{'1})$ = s-top$(M'_{m'(\Delta_1)})$ = s-top$(N_j^{'2})$, condition (4) holds because b, c are in $\bigcup_{i \in \underline{\mathtt{r}}} \|N_i^{'1}\|$ or $\bigcup_{i \in \underline{\mathtt{r}}} \|N_i^{'2}\|$ only if they are in $\bigcup_{i \in \underline{\mathtt{2r}}} \|M'_i\|$ respectively.

□

Having proved both Lemmata 4.2.7 and 4.2.8, we conclude the proof of the whole theorem.

□

Note that the same argument used in the proof of the intersection of two FSNAs suffices to establish the following property.

Property 4.2.9. *There exists languages L_1 and L_2, accepted by FSNA and FSNA$_+$, respectively such that $L_1 \cap L_2$ is not a regular nominal language.*

We now study the decidability of some typical problems, namely those of membership, universality and emptiness. Given an automaton R, the first and the second problems amount to check if a word w and Σ^* are accepted by R; the third if $\mathcal{L}(R) = \emptyset$.

Theorem 4.2.10.

1. *The membership problem for FSNA is decidable*
2. *The universality problem is undecidable for FSNA, while for FSNA$_+$ it is decidable, and the answer is always negative*
3. *The emptyness problem is decidable for FSNA*

Proof.

1. A trivial linear non-deterministic procedure suffices
2. Theorem 4.2.11 proved in the next sub-section guarantees that FSNA are more expressive than FMA. Now the proof follows because universality is undecidable for FMA [46].
   Since FSNA$_+$ cannot generate Σ^*, the second claim is proved.
3. The actual content of the m-registers is negligible when reasoning about emptiness, only their activation states are important because a step can be inhibited by an inactive m-register. So, we can abstract a configuration $\langle q, w, [N_1, \ldots, N_r] \rangle$ as a pair $\langle q, [x_1, \ldots, x_r] \rangle$, where x_i is the activation state of N_i. Suppose now that there exist an accepting run for w. We build another accepting run, possibly for a different word w', where no pair $\langle q, [x_1, \ldots, x_r] \rangle$ occurs twice, in a pumping lemma fashion. Since the abstract configurations are finite, we only need to check a finite number of runs.

	$\cup$	$\cap$	$\overline{\cdot}$	$\bullet$	$*$
$\mathcal{L}$(FSNA)	✓	×	×	✓	✓
$\mathcal{L}$(FSNA$_+$)	✓	✓	×	×	×
$\mathcal{L}$(VFA)	✓	✓	×	??	??
$\mathcal{L}$(FMA)	✓	✓	×	✓	✓
$\mathcal{L}$(UA)	✓	✓	×	✓	×
$\mathcal{L}$(FRA)	✓	✓	×	×	×
$\mathcal{L}$(HRA)	✓	✓	×	✓	✓
$\mathcal{L}$(CRA)	✓	✓	×	??	??
$\mathcal{L}$(NFMA)	✓	✓	×	✓	✓

Table 4.1: A comparison of the closure properties of FSNA, PSNA automata with the ones of other automata in the literature, namely VFA [33], FMA [38], UA [6], FRA [58], HRA [59], CRA [14] and NFMA [39]

□

Note in passing that Theorem 4.2.10.2 implies that there is no algorithm for minimizing an FSNA. Indeed if such a minimisation algorithm exists, then we could answer the universality problem by checking if the minimised automaton is isomorphic to the automaton in Figure 4.2, that is the minimum automaton recognising Σ^* (up to renaming of states).

4.2.2 Comparisons

In the literature there are many nominal languages, working on infinite alphabets or on data-words. We consider here only those that are intuitively *regular*, in that they cannot express Dyck-like languages, e.g. the language $\{ww^R\}$ when $|w|$ is not bounded (see Example 4.2.5). An incomplete list of the regular languages in the literature includes *variable finite automata* (VFA) [33], *finite memory automata* (FMA) [38] and their extension with non-deterministic reassignment (NFMA) [39], *Usage Automata* (UA) [6], *fresh-register automata* (FRA) [58] and their evolution *history register automata* (HRA) [59], *class register automata* (CRA) [14], *Data Walking Automata* (DWA) [44], the variant of *HD-automata* in [20] and `fp-automata` [40].

A notion similar to the one of the m-register can be found in HRA [59] and in the chronicles of the *chronicle deallocating automata* [42].

Table 4.1 recalls the closure properties of FSNA and FSNA$_+$ and of those models above for which the literature provides these results.

The next theorem investigates the relationship among our models and the (regular) ones in the literature in terms of expressiveness. When considering data words, we assume for simplicity that there is a single action α on resources, that will be omitted in words, i.e. we write a instead of $\alpha(a)$. We write $A \gtrless B$ when two sets A, B are incomparable, i.e. $A \not\subseteq B, B \not\subseteq A$.

Theorem 4.2.11 (Comparison)**.**

1. *$\mathcal{L}(FSNA) \supset \mathcal{L}(VFA) \supset \mathcal{L}(UA)$*
2. *$\mathcal{L}(FSNA) \supset \mathcal{L}(FMA)$*
3. *$\mathcal{L}(FSNA_+) \supset \mathcal{L}(UA)$*
4. *$\mathcal{L}(FSNA_+) \gtrless \mathcal{L}(VFA)$*
5. *$\mathcal{L}(FSNA_+) \gtrless \mathcal{L}(FMA)$*
6. *$\mathcal{L}(FSNA) \gtrless \mathcal{L}(HRA)$*
7. *$\mathcal{L}(FSNA_+) \gtrless \mathcal{L}(\texttt{fp-automata})$*
8. *$\mathcal{L}(FSNA) \supseteq \mathcal{L}(\texttt{fp-automata})$*

Proof. Sketch:

1. $\mathcal{L}$(FSNA) $\supset$ $\mathcal{L}$(VFA) $\supset$ $\mathcal{L}$(UA)
   The VFA have been proved more expressive than UA in [26]. A FSNA simulates a VFA by using m-registers associated to each variable (and never s-popping them), the y can be mapped to a m-register that is always s-popped after being used. The last condition matches the one of VFA requiring the symbols associated to each occurence of y in the witnessing pattern to be different from the other variables, but possibly equal to another symbol associated with y. The language L_0 in the Example 4.2.1 belongs to $\mathcal{L}$(FSNA) but not to $\mathcal{L}$(VFA).

2. $\mathcal{L}$(FSNA) $\supset$ $\mathcal{L}$(FMA)
   The main differences between the two models are the following. The registers of FMA have an initial assignment, while FSNA have static resources playing the same role (and initialization can anyway be done by initial ε-transitions that FMA have not). FMA associate the reassignment function ρ with states rather than with edges, and their effects are obtained by FSNA when all the edges starting from a state q have the same Δ. Additionally, ρ reassigns a register using the input symbol, while FSNA

update an m-register (through an ε-transition) and then recognizes the fresh symbol in it. In FMA, all the registers have to be different, and a reassignment may update a register with the same symbol it contains, and FSNA have two edges $\Delta \neq \varepsilon$ and $\Delta = \varepsilon$. So, $\mathcal{L}(\text{FSNA}) \supseteq \mathcal{L}(\text{VFA})$ and the language L_0 in Example 4.2.1 shows that inclusion is strict.

3. $\mathcal{L}(\text{FSNA}_+) \supset \mathcal{L}(\text{UA})$
   The expressiveness of UA is the same of VFA without the y (see [28]), so the construction in item 1 suffices (note that there will be no delete transitions).

4. $\mathcal{L}(\text{FSNA}_+) \gtreqless \mathcal{L}(\text{VFA})$ and 5. $\mathcal{L}(\text{FSNA}_+) \gtreqless \mathcal{L}(\text{FMA})$
   Consider $L_0 = \{w \in \Sigma_d^* \mid \forall i, j.\, w[i] \neq w[j]\}$ of Example 4.2.1. We have that $L_0 \in \mathcal{L}(\text{FSNA}_+)$ but $L_0 \notin \mathcal{L}(\text{VFA}) \cup \mathcal{L}(\text{FMA})$. Also $\Sigma^* \in \mathcal{L}(\text{FMA}) \cap \mathcal{L}(\text{VFA})$ but $\Sigma^* \notin \mathcal{L}(\text{FSNA}_+)$.

6. $\mathcal{L}(\text{FSNA}) \gtreqless \mathcal{L}(\text{HRA})$
   Consider the language $L = \{a_0 b_0 \dots a_n b_n \mid a_i \neq a_j, b_i \neq b_j\}$, L is not recognised by any FSNA but it is recognised by an HRA because of the capability of using multiple stories. On the other hand the language $L' = \{a_1 \dots a_n b_1 \dots b_n \mid i \neq j \Rightarrow a_i \neq a_j \wedge b_i \neq b_j, n - i \geq j \Rightarrow b_i \neq a_j\}$ is in $\mathcal{L}(\text{FSNA})$ but not in $\mathcal{L}(\text{HRA})$. This is because m-registers are stacks while places are sets.

7. $\mathcal{L}(\text{FSNA}_+) \gtreqless \mathcal{L}(\texttt{fp-automata})$ and $\mathcal{L}(\text{FSNA}) \supseteq \mathcal{L}(\texttt{fp-automata})$
   Both FSNA, FSNA$_+$ recognise the language $\{a_1 \dots a_n \mid i \neq j \Rightarrow a_i \neq a_j\}$, that instead is not by any `fp-automata`. However FSNA$_+$ cannot recognise Σ^*, a language in $\mathcal{L}(\texttt{fp-automata})$. However `fp-automata` expressiveness does not go beyond the one of PSNA, deallocation of `fp-automata` can be reproduced in PSNA by delete transitions; swapping the contents of two registers in `fp-automata` via a permutation (there are only a finite number of them) can be done by PSNA by suitably mentioning/updating/deleting the corresponding registers in the next states.

□

4.3 Pushdown Nominal Automata

In the beginning of the chapter we motivated our interest in developing a model that is able to address the five points (i-v). Below, we extend FSNA with a stack, so obtaining our version of nominal context-free automata. Of course, the nominal language $w\,w^R$ of Example 4.2.5 is accepted by one of these automata. We allow stacks to store elements of the infinite alphabet Σ, and to push on them strings of symbols in Σ, possibly retrieved through the indexes of m-registers. E.g., one may wish to push the string $a\,3\,b$ that actually pushes $a\,\text{s-top}(N_3)\,b$. A preliminary definition is in order to handle these cases.

Definition 4.3.1. Let $\zeta \in (\Sigma_s \cup \underline{\mathtt{r}})^*$ and let S be a stack. Then, $\mathit{Pushreg}(\zeta, S)$ extends the standard push operation as follows

$\mathit{Pushreg}(\varepsilon, S) = S$

$$\mathit{Pushreg}(z\,\zeta', S) = \mathit{Pushreg}(\zeta', \mathit{push}(\sigma, S)) \text{ where } \sigma = \begin{cases} z & \text{if } z \in \Sigma_s \\ \text{s-top}(N_z) & \text{if } z \in \underline{\mathtt{r}} \end{cases}$$

Definition 4.3.2 (Pushdown Nominal Automata)**.** A *Pushdown Nominal Automata* (PSNA) is $A = \langle Q, q_0, \Sigma, \delta, r, F \rangle$ where:

- Q, q_0, r, F are as in FSNA (Definition 4.2.1)
- δ is a relation between triples (q, σ, Z) and (q', Δ, ζ) where $\sigma \in \Sigma_s \cup \underline{\mathtt{r}} \cup \{\varepsilon, \top\}, Z \in \Sigma_s \cup \underline{\mathtt{r}} \cup \{\varepsilon, ?\}, \Delta \in \{i+, i- \mid i \in \underline{\mathtt{r}}\} \cup \{\varepsilon\}, \zeta \in (\Sigma_s \cup \underline{\mathtt{r}})^*$.
  For $(q, \sigma, Z, q', \Delta, \zeta) \in \delta$ we use the notation $q \xrightarrow[\Delta,\zeta]{\sigma,Z} q'$

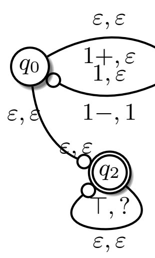
A configuration is a tuple $C = \langle q, w, [N_1, \ldots, N_r], S \rangle$ where $q, w, [N_1, \ldots, N_r]$ are as in FSNA and S is a stack with symbols in Σ.
A configuration $\langle q_f \in F, \varepsilon, [N_1, \ldots, N_r], \_ \rangle$ is *final.*

As defined below, PSNA may use in a richer way than standard pushdown automata the top of the stack, call it a. First, we can compare the current symbol in the input with a, if the symbol σ in the transition to be applied is $\top$. Also, if $Z = \varepsilon$ the string obtained from ζ is pushed on the stack, as explained above. Instead, if $Z = i$ the top a is popped from the stack, provided that the s-top of the i^{th} m-register is a. Finally, if $Z =?$ a pop occurs, with no further constraints.

Definition 4.3.3 (Recognizing Step)**.**
Given a PSNA A, the step $\langle q, w, [N_1, \ldots, N_r], S \rangle \rightarrow \langle q', w', [N_1', \ldots, N_r'], S' \rangle$ occurs iff $q \xrightarrow[\Delta,\zeta]{\sigma,Z} q' \in \delta$ and the following hold

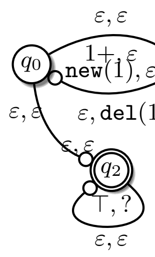
1. condition 1 of Definition 4.2.2 and $\sigma = \top \Rightarrow w = \mathit{top}(S)w'$ and
2. condition 2 of Definition 4.2.2 and

(a)


ε, ε
$1+, \varepsilon$
$1, \varepsilon$
q_0
q_1
$1-, 1$
ε, ε
ε, ε
q_2
$\top, ?$
ε, ε

$$\rho_1 = \langle q_0, [\_], \_\rangle \xrightarrow{\varepsilon} \langle q_1 [\boxed{\text{a}}], \_\rangle \xrightarrow{a} \langle q_0, [\_], \boxed{\text{a}}\rangle \xrightarrow{\varepsilon} \langle q_1, [\boxed{\text{a}}], \boxed{\text{a}}\rangle \xrightarrow{a}$$
$$\langle q_0, [\_], \boxed{\begin{array}{c}\text{a}\\ \text{a}\end{array}}\rangle \xrightarrow{\varepsilon} \langle q_1, [\boxed{\text{b}}], \boxed{\begin{array}{c}\text{a}\\ \text{a}\end{array}}\rangle \xrightarrow{b} \langle q_0, [\_], \boxed{\begin{array}{c}\text{b}\\ \text{a}\\ \text{a}\end{array}}\rangle \xrightarrow{\varepsilon}$$
$$\langle q_2, [\_], \boxed{\begin{array}{c}\text{b}\\ \text{a}\\ \text{a}\end{array}}\rangle \xrightarrow{b} \langle q_2, [\_], \boxed{\begin{array}{c}\text{a}\\ \text{a}\end{array}}\rangle \xrightarrow{a} \langle q_2, [\_], \boxed{\text{a}}\rangle \xrightarrow{a} \langle q_2, [\_], \_\rangle$$

(b)


ε, ε
$1+, \varepsilon$
new(1), ε
q_0
q_1
ε, del(1)
ε, ε
ε, ε
q_2
$\top, ?$
ε, ε

$$\rho_2 = \langle q_0, [\_], \_\rangle \xrightarrow{\varepsilon} \langle q_1 [\boxed{\text{a}}], \_\rangle \xrightarrow{\texttt{new}(a)} \langle q_0, [\boxed{\text{a}}], \boxed{\texttt{del}(\text{a})}\rangle \xrightarrow{\varepsilon}$$
$$\langle q_1, [\boxed{\begin{array}{c}\text{b}\\ \text{a}\end{array}}], \boxed{\texttt{del}(\text{a})}\rangle \xrightarrow{\texttt{new}(b)} \langle q_0, [\boxed{\begin{array}{c}\text{b}\\ \text{a}\end{array}}], \boxed{\begin{array}{c}\texttt{del}(\text{b})\\ \texttt{del}(\text{a})\end{array}}\rangle \xrightarrow{\varepsilon}$$
$$\langle q_2, [\boxed{\begin{array}{c}\text{b}\\ \text{a}\end{array}}], \boxed{\begin{array}{c}\texttt{del}(\text{b})\\ \texttt{del}(\text{a})\end{array}}\rangle \xrightarrow{\texttt{del}(b)} \langle q_2, [\boxed{\begin{array}{c}\text{b}\\ \text{a}\end{array}}], \boxed{\texttt{del}(\text{a})}\rangle \xrightarrow{\texttt{del}(a)} \langle q_2, [\boxed{\begin{array}{c}\text{b}\\ \text{a}\end{array}}], \_\rangle$$

Figure 4.7: (a) A PSNA accepting $\{ww^R \mid w \in \Sigma_d^*\}$, and a run on *aabbaa*. (b) A PSNA$_+$ for the data word language of the Example 3.3.7 and a run on `new`(a) `new`(b) `del`(b) `del`(a) (`n` and `r` stand for `new` and `release`). Strings are omitted in configurations.

3. $$\begin{cases} Z = \varepsilon \Rightarrow S' = Pushreg(\zeta, S) \text{ and} \\ Z = i \Rightarrow S' = Pushreg(\zeta, pop(S)) \wedge top(S) = \text{s-top}(N_i) \text{ and} \\ Z = ? \Rightarrow S' = Pushreg(\zeta, pop(S)) \end{cases}$$

Finally, the (nominal) language accepted by A is

$$L(A) = \{w \in \Sigma^* \mid \exists \rho : \langle C_1 = \langle q_0, w, [\_, \dots, \_], \_\rangle \to^* C_k, \text{with } C_k \text{ final}\}$$

and we call it *context-free.*

Example 4.3.1. *Figure 4.7(a) shows a PSNA accepting $L_p = \{ww^R \mid w \in \Sigma_d^*\}$, and a run accepting aabbaa (for brevity, we do not write the strings to be recognized in the configurations, as the current symbols label the steps). The automaton behaves just as a FSNA in the $1^{st}, 3^{rd}, 5^{th}$ and 7^{th} steps of ρ_1. Additionally, in this initial part of the run, the stack is involved in the $2^{nd}, 4^{th}$ and 6^{th} step. They all occur because of edge $q_1 \xrightarrow[1-,1]{1,\varepsilon} q_0$, that causes the symbol in the m-register 1 to be pushed on the stack. In steps $8^{th}, 9^{th}, 10^{th}$ the edge $q_2 \xrightarrow[\varepsilon,\varepsilon]{\top,?} q_2$ causes the top of the stack to be (succesfully) matched with the current symbol (as dictated by the label $\top$) and popped (because of ?).*

As done for FSNA we introduce the class of automata that update two m-registers at the same time, and the sub-class of PSNA without delete transitions.

Definition 4.3.4 (PSNA2 and PSNA$_+$).

- A PSNA2 is a PSNA with transitions of the form $q \xrightarrow[(\Delta_1,\Delta_2),\zeta]{\sigma,Z} q'$ (cf. Definition 4.2.4)
- A PSNA$_+$ is a PSNA with no edges $q \xrightarrow[i-,\zeta]{\sigma,Z} q'$.

Just as done for FSNA, we can prove that PSNA2 and PSNA have the same expressive power. As expected, the class of languages accepted by PSNA strictly includes that accepted by PSNA$_+$. Indeed, the same proof of Property 4.2.3 applies here. In spite of the reduced expressiveness, PSNA$_+$ can accept a wide class of (Dyck-like) context-free languages, as shown by the following example.

Example 4.3.2. *Consider again the* `new-release` *(abbreviated* `new`, `del`*) language on data words of the Example 3.3.7. The PSNA$_+$ accepting this language is in Figure 4.7(b). The labels of transitions, but Δ, contain* $\mathtt{new}(u), \mathtt{del}(u), u \in \underline{\mathtt{r}} \cup \Sigma_s$. *Figure 4.7(b) also shows the run for* $\mathtt{new}(a)\,\mathtt{new}(b)\,\mathtt{del}(b)\,\mathtt{del}(a)$*; also here we omit the strings in configurations and we only mention the symbols in the m-registers. Note that, only keeping the names of the resources, we get* $\cup_{r\in\mathbb{N}} L_r$, *for L_r of Example 4.2.5.*

4.3.1 Some Properties

Obviously, the class of pushdown nominal languages includes that of the regular ones.

Property 4.3.3. $\mathcal{L}(FSNA) \subset \mathcal{L}(PSNA)$

Proof. Inclusion is trivially proved: from a give FSNA obtain the equivalent PSNA by adding labels $\Delta = \varepsilon, \zeta = \varepsilon, Z = \varepsilon$ to each edge. Example 4.2.5 suffices to prove that the inclusion is strict. □

We now study under which operators the classes of languages accepted by PSNA and PSNA$_+$ are closed.

Theorem 4.3.4 (Closure properties).

	$\cup$	$\cap$	$\overline{\cdot}$	$\bullet$	$*$
$\mathcal{L}(PSNA)$	✓	×	×	✓	✓
$\mathcal{L}(PSNA_+)$	✓	×	×	×	×

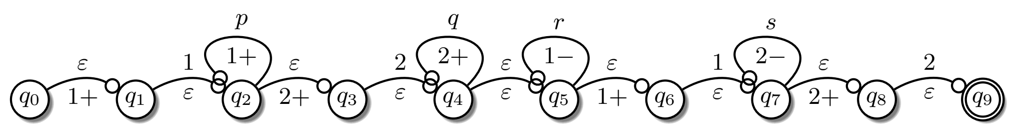


Figure 4.8: A FSNA recognising the language $L_1 = \{ap^nbq^mr^{n'}cs^{m'}d \mid a = c \Rightarrow n' > n, b = d \Rightarrow m' > m, a \neq b, c \neq d\}$

Proof. Union:
The construction is the same of Theorem 4.2.6 in both cases; of course the initial ε-transitions do not alter the stack ($\zeta = Z = \varepsilon$).
Intersection and complement:
Follows from the classical results on context-free languages.
Concatenation and Kleene star:
The proof in Theorem 4.2.6 applies here as well; note that the stack is empty in the initial and final configurations.

□

A classical result in automata theory is that the class of context-free languages is closed by intersection with the class of regular ones. We investigate the same property in the nominal case, and we find that only the intersection of FSNA$_+$ and PSNA$_+$ is a PSNA$_+$ (hence a PSNA).

Theorem 4.3.5 (Intersection)**.**

is a	*FSNA* ∩ *PSNA*	*FSNA* ∩ *PSNA*$_+$	*FSNA*$_+$ ∩ *PSNA*	*FSNA*$_+$ ∩ *PSNA*$_+$
PSNA	×	×	×	✓
PSNA$_+$	×	×	×	✓

Proof. Consider the FSNA language $L_1 = \{ap^nbq^mr^{n'}cs^{m'}d \mid a = c \Rightarrow n' > n, b = d \Rightarrow m' > m, a \neq b, c \neq d\}$ in Figure 4.8 and the PSNA$_+$ language $L_2 = a\{p\}^*b\{q\}^*\{r\}^*c\{s\}^*d$. The language $L_1 \cap L_2 = \{ap^nbq^mr^{n'}cs^{m'}d \mid n' > n, m' > m, a \neq b\}$, by classical reasoning on context-free languages, is not recognised by any PSNA$_+$ nor PSNA. Note that L_1 can be recognised by the PSNA obtained by adding $Z = \varepsilon, \zeta = \varepsilon$ to the edges in Figure 4.8 and L_2 is a nominal regular language recognised by both a FSNA$_+$ and a FSNA. Hence, also $L_1 \cap L_2$ is not a PSNA$_+$ nor a PSNA.
We prove now that PSNA$_+$ ∩ FSNA$_+$ is a PSNA$_+$:
The proof follows step by step that of Theorem 4.2.6, with additional care to

manage the stack, which however is only determined by how the PSNA handles it. The detailed construction follows.

Given the PSNA$_+$ $\langle Q_1, q_0^1, \Sigma, \delta_1, r, F_1\rangle$ and the FSNA$_+$ $\langle Q_2, q_0^2, \Sigma, \delta_2, r, F_2\rangle$, their *intersection automaton* (of type PSNA$_+$2) is $\langle \overline{Q}, \overline{q_0}, \Sigma, \overline{\delta}, 2r, \overline{F}\rangle$, where

- $\overline{Q} = Q_1 \times Q_2 \times M$, with M set of merge functions
- $\overline{q_0} = \langle q_0^1, q_0^2, \langle id_{\underline{\mathtt{r}}}, id_{\underline{\mathtt{r}}}\rangle\rangle$ – $\overline{F} = \{\langle q_1, q_2, m\rangle \mid q_1 \in F_1, q_2 \in F_2, m \in M\}$
- $\langle q_1, q_2, m\rangle \xrightarrow[\overline{\Delta_1},\overline{\Delta_2},\overline{\zeta}]{\overline{\sigma},\overline{Z}} \langle q_1', q_2', m'\rangle \in \overline{\delta}$ iff $m \overset{\Delta_1,\Delta_2}{\curvearrowright} m'$ and

  $q_1 \xrightarrow[\Delta_1,\zeta]{\sigma_1,Z} q_1' \in \delta_1$ and $q_2 \xrightarrow[\Delta_2]{\sigma_2} q_2' \in \delta_2$ and ($\sigma_1, \sigma_2 \in \underline{\mathtt{r}}$ or $\sigma_1, \sigma_2 \in \Sigma_s$) and

  – if $\sigma_1, \sigma_2 \in \underline{\mathtt{r}}$ then $\overline{\sigma} = m_1(\sigma_1) = m_2(\sigma_2)$ and

  – if $\sigma_1, \sigma_2 \in \Sigma_s$ then $\overline{\sigma} = \sigma_1 = \sigma_2$ and

  – $(\overline{\Delta_1}, \overline{\Delta_2}) = \overset{m}{m'}\big(\Delta_1, \Delta_2\big)$, $\overline{Z} = m_1(Z), \overline{\zeta} = m_1(\zeta)$

  or $q_1 \xrightarrow[\Delta_1,\zeta]{\top,Z} q_1' \in \delta_1$ and $q_2 \xrightarrow[\Delta_2]{\sigma_2} q_2' \in \delta_2$ and $\sigma_2 \in \underline{\mathtt{r}}, \overline{\sigma} = m_2(\sigma_2)$ and $\overline{\zeta} = m_1(\zeta)$ and either $Z = k \in \underline{\mathtt{r}}$ implies $k \overset{m}{\longleftrightarrow} \sigma_2$, $\overline{Z} = m_2(\sigma_2)$, $(\overline{\Delta_1}, \overline{\Delta_2}) = \overset{m}{m'}\big(\Delta_1, \Delta_2\big)$,
  or $Z =?$ implies $\overline{Z} = m_2(\sigma_2), (\overline{\Delta_1}, \overline{\Delta_2}) = \overset{m}{m'}\big(\Delta_1, \Delta_2\big)$

  or $q_1 \xrightarrow[\Delta_1\zeta]{\varepsilon,Z} q_1' \in \delta_1$ and $\overline{\sigma} = \varepsilon$, $(\overline{\Delta_1}, \overline{\Delta_2}) = \overset{m}{m'}\big(\Delta_1, \varepsilon\big)$, $\overline{Z} = m_1(Z), \overline{\zeta} = m_1(\zeta)$

  or $q_2 \xrightarrow[\Delta_2]{\varepsilon} q_2' \in \delta_2$ $\overline{\sigma} = \varepsilon$, $(\overline{\Delta_1}, \overline{\Delta_2}) = \overset{m}{m'}\big(\Delta_1, \varepsilon\big)$, $\overline{Z} = \varepsilon, \overline{\zeta} = \varepsilon$

□

We finally prove that the emptiness problem is decidable for PSNA$_+$. The proof relies on a variant of the classical pumping lemma. Roughly it says that given a language L recognised by a PSNA$_+$ there exists a constant n, such that any string $w \in L$, $|w| > n$ can be decomposed as $w = uvxyz$, such that also $w' = u'x'z'$ belongs to L with u', x', z' obtained from u, x, z by carefully substituting (distinguished) dynamic symbols and erasing v and y. Before proving it, we need some auxiliary definitions and lemmata.

Since we focus on the emptiness problem, we are interested in the existence of a word, rather than in its actual shape. Therefore, whenever immaterial, we feel free to omit from now onwards the word in configurations and the input symbol in transitions.[1]

Notation From now onwards, assume as given a PSNA$_+$ and let
$C = \langle q, [N_1, \dots, N_r], S\rangle$ be a configuration;
$\rho = C_1 \to^* C_k, C_i = \langle q_i, w_i, [N_1^i, \dots, N_r^i], S_i\rangle$ be a run;
$B = [B[1], \dots, B[n]]$ denote an array, and let $B[i, \dots, j], i \leq j$ denote the portion of the array between the i-th and j-th positions. We will also use the array notation for stacks, assuming that the leftmost item $S[1]$ is the bottom and $S[i]$ is the i-th element in it.
Also, call *swap* f an injective partial function $f : \Sigma_d \rightharpoonup \Sigma_d$, and its homomorphic extensions to strings, tuples, array and stacks.

What follows extends similar definitions and proofs of [1].

Definition 4.3.5 (C-rep)**.** Let E_S be the set of symbols occurring in the stack S of a configuration C such that $e \in E_S$ iff $\forall i.top(N_i) \neq e$. Let

$$first(e, S) = \begin{cases} 1 & \text{if } top(S) = e \\ first(e, pop(S)) + 1 & \text{otherwise} \end{cases}$$

Let $f_S : E_S \to \{1, \dots, |S|\}$ to be such that $f_S(e) = first(e, S)$ (note that f_S is injective).

The function C-rep(S) that returns a stack of symbols in $\Sigma_s \cup \{i \mid 1 \leq i \leq r\} \cup \{\underline{i} \mid 1 \leq i \leq |S|\}$ is defined by:

[1]Consequently, we have that $\langle q, [N_1, \dots, N_r], S\rangle \to \langle q', [N'_1, \dots, N'_r], S'\rangle$ iff there exists $q \xrightarrow[\Delta,\zeta]{\sigma,Z} q' \in \delta$ satisfying the conditions $(2-3)$ of Definition 4.3.3 and the following holds:

$$\textbf{(1m)} \begin{cases} \sigma = \top \Rightarrow top(S) \text{ is defined} \\ \sigma = i \Rightarrow \text{s-top}(N_i) \text{ is defined} \end{cases}$$

If $\langle q, [N_1, \dots, N_r], S\rangle \to \langle q', [N'_1, \dots, N'_r], S'\rangle$ because there exists $q \xrightarrow[\Delta,\zeta]{\sigma,Z} q' \in \delta$ satisfying the conditions $(1m, 2, 3)$ then by setting for any w' the word w such that

$$\begin{cases} \sigma & = \varepsilon \Rightarrow w = w' \\ \sigma & = \top \Rightarrow w = aw' \\ \sigma & = i \Rightarrow w = aw' \end{cases}$$

we have that also $\langle q, w, [N_1, \dots, N_r], S\rangle \to \langle q', w', [N'_1, \dots, N'_r], S'\rangle$. By induction, if $\langle q, [N_1, \dots, N_r], S\rangle \to^* \langle q', [N'_1, \dots, N'_r], S'\rangle$ then for any word w' there exists a word w such that $\langle q, w, [N_1, \dots, N_r], S\rangle \to^* \langle q', w', [N'_1, \dots, N'_r], S'\rangle$.

- C-rep$([]) = []$
- C-rep$(b :: S') = a :: C$-rep(S') iff
  - $b \in \Sigma_s, a = b$ or
  - $b \in \Sigma_d, \exists i.b = \mathit{top}(N_i), a = i$ or
  - $b \in \Sigma_d, \forall i.b \neq \mathit{top}(N_i), a = \underline{f_{E_S}(b)}$

Definition 4.3.6 (Activation state)**.** The activation state of a configuration C is an array $m = [m[1], \ldots, m[r]]$ where $m[i] = 1$ iff N_i is active, $m[i] = 0$ otherwise.

Definition 4.3.7 (Representative state)**.** The *representative state* of a configuration C is the triple (q, m, R) where m is the activation state of the mindful registers and $R =C$-rep(S), i.e. R represents S on C. We write $C \sim C'$ to indicate that C has the same representative state of C'.

Lemma 4.3.6. *Let $C_1 \to C_1'$ then for any configuration C_2 such that $C_2 \sim C_1$ there exists C_2' such that $C_2 \to C_2'$ and $C_2' \sim C_1'$.*

Proof. Let $t = (q', \Delta, \zeta) \in \delta(q, \sigma, Z)$ be used for justifying the transition $C_1 \to C_1'$. We first show that t justifies also $C_2 \to C_2'$ by constructing a suitable $C_2' = \langle q', [M_1', \ldots, M_r'], T' \rangle$. First of all, note that, being $C_1 \sim C_2$, the main stacks S and T have the same depth and the m-registers N_i and M_i have the same activation state. Therefore, since C_1 satisfies ($1m$), also C_2 does, i.e. :

$$\begin{cases} \sigma = \top & \Rightarrow \mathit{top}(T) \text{ is defined} \\ \sigma = i & \Rightarrow \text{s-top}(M_1) \text{ is defined} \end{cases}$$

For the same reason, in condition (3), the operations $\mathit{pop}(T), \mathit{top}(T)$ and s-top(M_i) are defined and so are the arguments of the operation Pushreg. Note that ζ may contain a reference to a register j and again we have that the required s-top(M_j) is defined, because the activation state of M_j is the same of N_j.

We are left to prove that condition (2) can be fulfilled by the M_j' and to prove that $C_1' \sim C_2'$. We proceed by cases on Δ.

Case $\Delta = i+$**)** Let $\forall j (j \neq i).M_j' = M_j$, $M_i' =$ s-push(c, M_i) with c to be $T[first(c, M_i)]$ if $b \in S$ to preserve the representative state, that requires to relate T' with S'. Otherwise we choose $c \notin M_j$, $\forall j \in \underline{r}$ and $c \notin T$. We are left to prove that $C'_2 \sim C'_1$. Trivially C'_2 and C'_1 have the same state q'. The activation state of the m-registers is also the same, because in both configurations only the i-th is affected (if active it is left such as well it becomes active because of the s-push), while the activation state of the others is the same in C'_1 and C'_2 because $C_1 \sim C_2$. Also $E_S = E_T$ and the same ζ is pushed on both stacks, so $E_{S'} = E_{T'}$. Now $f_{S'}(\text{s-top}(N_i)) = f_{T'}(\text{s-top}(M_i))$, that proves $C'_1 \sim C'_2$.

Case $\Delta = i-$) Let $M'_j = M_j, \forall j (j \neq i)$ and $M'_i = \text{s-pop}(M_i)$ which is defined because $C_1 \sim C_2$. The proof that the tuple $(q', \Delta, \zeta) \in \delta(q, \sigma, Z)$ justifies $C_2 \to C'_2$ is similar to the case above. Only the $i-th$ m-register is affected, if active it gets deactivated, it is left deactivated otherwise.

Case $\Delta = \varepsilon$) Letting $M'_j = M_j, \forall j \in \underline{r}$ suffices to fulfil condition (2). The proof that the tuple $(q', \Delta, \zeta) \in \delta(q, \sigma, Z)$ justifies $C_2 \to C'_2$ is similar to the case above. Only the $i-th$ m-register is affected, if active it gets deactivated, it is left deactivated otherwise.

□

Definition 4.3.8 (Level)**.** A level $G = (i, j, h)$ with height l on ρ is a triple (i, j, h) such that $1 \leq i < j < h \leq k$ and

- $|S_i| = |S_h|, |S_j| = |S_i| + l$
- $|S_i| \leq |S_u| \leq |S_j|$ for all $u.i \leq u \leq j$.
- $|S_h| \leq |S_u| \leq |S_j|$ for all $u.j \leq u \leq h$

Given a level on ρ, define two indices $l^G_\downarrow, f^G_\uparrow$, called respectively *last-push* and *first-pop* of G.

$$l^G_\downarrow = \max\{y \leq j \mid |S_y| = |S_i|\} \qquad f^G_\uparrow = \min\{y \geq j \mid |S_y| = |S_i|\}$$

Figure 4.9 shows an example of levels, $l_\downarrow$ and $f_\uparrow$.

Property 4.3.7. *Given a level (i, j, h) with height l on ρ, for each $k < l$ there exists a level (u, j, v) for some u, v with height k.*

Definition 4.3.9 (Full state)**.** Let G be a level on ρ and let $C_{l^G_\downarrow} = \langle q, [N_1, \ldots, N_r], a :: S\rangle, C_{f^G_\uparrow} = \langle q', [N'_1, \ldots, N'_r], S'\rangle$.

The *full state* of a level G on ρ is the tuple (c, q, m, q', m') such that:

- If $a \in \Sigma_s$ then $c = a$, if $a \in \Sigma_d$ and $\exists i.top(N_i) = c$ then $c = i$, if $a \in \Sigma_d$ and $\forall i.top(N_i) \neq c$ then $c = \star$.
- m, m' are the activation states of the m-registers in $C_{l^G_\downarrow}, C_{f^G_\uparrow}$, respectively.

Property 4.3.8. *Let $C_1 = \langle q_1, [N^1_1, \dots, N^1_r], S_1\rangle \rightarrow^n C_n = \langle q_n, [N^n_1, \dots, N^n_r], S_n\rangle$ and let $G = (1, j, n)$ be a level on the above run with height l, for some j. Then there exists a* cutoff run *$D_{l^G_\downarrow}, \dots, D_{f^G_\uparrow}$ with $D_i = \langle q_i, [N^i_1, \dots, N^i_r], S'_i\rangle$, $S'_i = T :: S_i[1, \dots, |S_i| - l]$ for any T with $top(T) = top(S_{l^G_\downarrow})$.*

Proof. Sketch: by definition of level the transitions does not depend on the content of the stack below the height of $S_{l^G_\downarrow}$ □

We now prove our restricted pumping lemma.

Lemma 4.3.9 (Pumping Lemma). *Let $A = \langle Q, q_0, \delta, r, \Sigma_s \cup \Sigma_d, F\rangle$ be a PSNA$_+$ and let $p' = 2^r |Q|^2 (|\Sigma_s| + r + 1)$ and $p = 2^r |Q| (|\Sigma_s| + r + p' + 1)^{p'} + 1$. For each word $w \in L(A)$ such that $|w| > p$ and ρ is an accepting run of minimum length, we can construct a word $w' \in L(A)$ with a* shorter *accepting run.*

Proof. Let $w \in L(A)$ such that $|w| > p$; take $\rho = C_1 \rightarrow \cdots \rightarrow C_k$, $C_i = \langle q_i, w_i, [N^i_1, \dots, N^i_r], S_i\rangle$ out of the set of the shortest accepting runs; and let l be the maximum height of the stack in ρ.

Case $l \leq p'$) Recall that $|w| > p$, hence ρ contains at least p configurations. There are at most $2^r |Q| (|\Sigma_s| + r + l + 1)^l < p$ different representative states of the configurations of ρ. Hence there are at least two configurations $C_x, C_y, x < y$ with the same representative state. By applying Lemma 4.3.6, from the run $C_y \rightarrow^* C_k$ we obtain that also $C_x \rightarrow^* F$ for some F with the same representative state of C_k. Therefore, also F is a final configuration. The thesis follows because the run $\rho' = C_1 \rightarrow^* C_x \rightarrow^* F$ is shorter than ρ.

Case $l > p'$) Note that there is a level on ρ with height l, say $G = (i, j, h)$. By Property 4.3.7, there exist at least l levels (u, j, v) with different heights that are levels on ρ.

There are only $p' < l$ different full states that can be associated with these levels, hence there exist two levels, say $U = (u_1, j, v_1)$ and $V = (u_2, j, v_2)$ with the same full state. Assume w.l.o.g. $n_U = |S_{u_1}| = |S_{v_1}| < |S_{u_2}| = |S_{v_2}| = n_V$. Let $C_{l^U_\downarrow}, C_{l^V_\downarrow}$ be the configuration with index last-push and let $C_{f^U_\uparrow}, C_{f^V_\uparrow}$ be the configuration with index first-pop of level U and V respectively (see Figure 4.9).

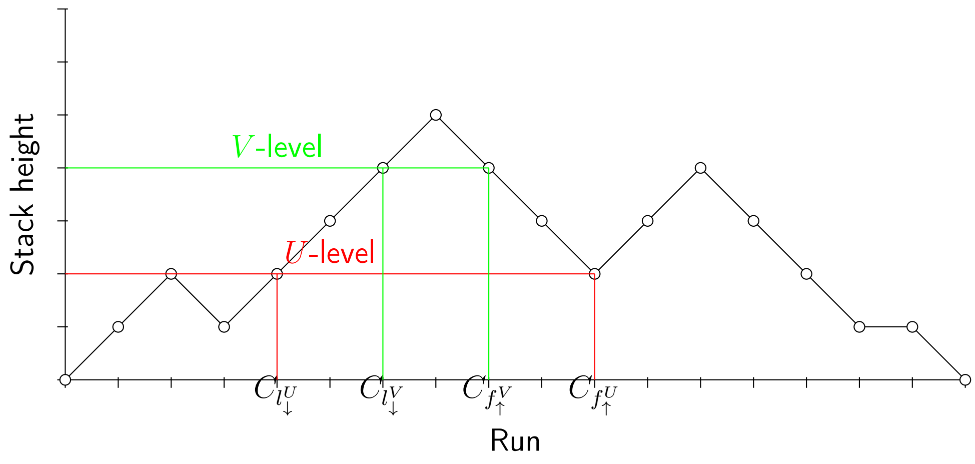


Figure 4.9: An example of V-level and G-level on a run.

By Property 4.3.8, from the run $C_{l^V_\downarrow} \to^* C_{f^V_\uparrow}$ it is possible to obtain a cut off run

$$D_{l^V_\downarrow} = \langle q_{l^V_\downarrow}, w_{l^V_\downarrow}, [N_1^{l^V_\downarrow}, \dots, N_r^{l^V_\downarrow}], T_{l^V_\downarrow}\rangle \to^*$$
$$D_{f^V_\uparrow} = \langle q_{f^V_\uparrow}, w_{f^V_\uparrow}, [N_1^{f^V_\uparrow}, \dots, N_r^{f^V_\uparrow}], T_{f^V_\uparrow}\rangle$$

where $T_i = S_{l^U_\downarrow} :: S_i[n_V, \dots, |S_i|]$.

Since $T_{l^V_\downarrow} = T_{f^V_\uparrow} = S_{l^U_\downarrow} = S_{f^U_\uparrow}$ and $C_{l^U_\downarrow}, C_{f^U_\uparrow}$ have the same full state of $C_{l^V_\downarrow}, C_{f^V_\uparrow}$, respectively, it follows that $D_{l^V_\downarrow} \sim C_{l^U_\downarrow}$ and $D_{f^V_\uparrow} \sim C_{f^U_\uparrow}$.

Consequently, by Lemma 4.3.6, from the run $D_{l^V_\downarrow} \to^* D_{f^V_\uparrow}$ we obtain the run $C_{l^U_\downarrow} \to^* H$ for some $H \sim D_{f^V_\uparrow} \sim C_{f^U_\uparrow}$. By the same lemma, from the run $C_{f^U_\uparrow} \to^* C_k$ we obtain a run $H \to^* F$, where F has the same representative state of C_k, hence it is final.

The thesis follows because the run $D_1 \to^* C_{l^U_\downarrow} \to^* H \to^* F$ is shorter than ρ.

□

We eventually prove the decidability of the emptyness problem for our pushdown nominal automata with no delete transitions; we conjecture that it is instead undecidable for PSNA.

Theorem 4.3.10. *Given a PSNA$_+$ A, it is decidable whether $L(A) = \emptyset$.*

Proof. (Sketch) By repeatedly applying the Pumping Lemma 4.3.9, $L(A)$ is non empty if it contains a word w', made of distinguished symbols, and such that $|w'| \leq n$. □

	$\cup$	$\cap$	$\overline{\cdot}$	$\bullet$	$*$
$\mathcal{L}$(PSNA)	✓	×	×	✓	✓
$\mathcal{L}$(PSNA$_+$)	✓	×	×	×	×
$\mathcal{L}$(QCFL)	✓	×	×	✓	✓
$\mathcal{L}$(Usages)	✓	×	×	×	×
$\mathcal{L}(NPA)$	✓	??	??	??	??
$\mathcal{L}(Pebble)$	✓	??	??	??	??

Table 4.2: A comparison of the closure properties of our automata with the ones of QCFL [19], Usages [6] and NPA [12].

	PSNA$_+$	PSNA	QCFL	Usages	*DMPA*	*Pebble*
Emptiness	✓	??	✓	✓	×	×

Table 4.3: Decidability of the emptiness problem for QCFL [19], Usages [6], DMPA [15] and Pebble [46] automata.

4.3.2 Comparisons

To the best of our knowledge, the literature has different notions of nominal context-free languages, i.e. able to express Dyck-like languages: *quasi context-free languages* (QCFL) [19], Usages introduced in [6], the context-free automata (that we call here NPA) in [13], DMPA [15], HOPAD [48] and *Pebble* automata [46].

Below, we compare the properties and the expressive power of PSNA with that of above models for different notions of nominal context-free languages.

Table 4.2 shows some closure properties of our automata with those of the other models, when presented in the literature. To the best of our knowledge, the closure of a context-free nominal language with a regular one has not been investigated explicitly for other nominal models, even if it is folklore that intersecting a QCFL with a FMA automaton can be done mimicking the construction of the intersection between FMA.

The decidability and the complexity of the emptiness problem has been investigated for QCFL, Usages, DMPA and *Pebble* automata, because of the relevance of this problem in verification. Table 4.3 summarises the decidability results of the emptiness problem for the above models and for ours.

We now compare the expressiveness of PSNA and PSNA$_+$ in comparison with that of other models in the literature. Also here, we assume that data words have a single action, not displayed in words (see Theorem 4.2.11).

Theorem 4.3.11 (Expressivness Comparison)**.**

– $\mathcal{L}(PSNA_+) \gneqq \mathcal{L}(QCFL)$

– $\mathcal{L}(PSNA) \supset \mathcal{L}(QCFL)$

– $\mathcal{L}(PSNA_+) = \mathcal{L}(Usages)$

– $\mathcal{L}(PSNA) \supset \mathcal{L}(Usages)$

– $\mathcal{L}(PSNA) \gneqq \mathcal{L}(DMPA)$

Proof.

- $\mathcal{L}$(PSNA) $\supset \mathcal{L}(QCFL)$
  We consider the *infinite alphabet pushdown automata* (IAPA) that recognize $\mathcal{L}(QCFL)$ [19]. The same argument in Theorem 4.2.11, item 2 ($\mathcal{L}$(FSNA) $\supset$ $\mathcal{L}(FMA)$) suffices for showing the inclusion, which is strict, because L_0 in the Example 4.2.1 is not recognised by any IAPA.

- $\mathcal{L}$(PSNA$_+$) $= \mathcal{L}$(Usages)
  Usages are built from (static and dynamic) symbols n (actually $\alpha(n)$, where α is an action on n) with operation of sequentialization $\cdot$, nondeterminism $+$, recursion and creation of a new dynamic symbol, through νn (see Table 1.2).
  When sequentialising two processes, the second cannot use any dynamic symbol used by the first one, just as it happens when two PSNA$_+$ are sequentialised by connecting the final states of the first with the initial one of the second. Since there is no deletion, the m-registers monotonically grow.
  Nondeterministic choice $+$ directly corresponds to the union of two automata.
  Recursion can be dealt with as done in [7] by transforming an expression in a BPA, that has an immediate counterpart as a PSNA$_+$.
  For each occurrence of a νn in the usage at hand we associate an m-register. Creation of a new symbol, i.e. reducing the νn, corresponds to updating the corresponding m-register. When a νn occurs within a recursive expression, a renaming occurs to guarantee freshness of the dynamic symbol to be generated. Note that only a finite number of m-registers is necessary, as the number of νn occurring in a usage is fixed and Property 3.3.6 holds.

- $\mathcal{L}(\text{PSNA}) \supset \mathcal{L}(\text{Usages})$
  $\Sigma^* \in \mathcal{L}(\text{PSNA})$, while $\Sigma^* \notin \mathcal{L}(\text{Usages})$ by Theorem 3.3.1.

- $\mathcal{L}(\text{PSNA}_+) \gtrless\!\!\!\!/\ \mathcal{L}(\text{QCFL})$
  The proof of Theorem 8, item 5 suffices ($\text{FSNA}_+ \supset \text{FMA}$).

- $\mathcal{L}(\text{PSNA}) \gtrless\!\!\!\!/\ \mathcal{L}(\text{DMPA})$ Using their multiple stacks, the DMPA can express the language (of patterns) $\{a^n b^n c^n\}$, that cannot be recognized by a PSNA. However, their notion of freshness also requires that a new symbol cannot occur in the stacks, which is not the case for PSNA.

$\square$

Discussions

Nominal languages are a suitable behavioural abstraction of many software systems dealing with an unbounded number resources, see e.g. [55, 15, 11]. Their properties often involve the usage patterns of these resources. In particular, Examples 2.5.1, 2.5.2, 3.3.7 and 4.0.9 illustrate these facts for ContextML programs.

We proposed novel automata for nominal languages, that we intuitively classify as context-free and regular (able and not, respectively, to recognise Dyck-like languages). The context-free and regular automata are called PSNA and FSNA, respectively, we also considered their sub-classes, PSNA_+ and FSNA_+, that inhibit disposal.

We studied some closure properties of our models: union, intersection, complementation, concatenation and Kleene star. We related their expressive power to that of analogous models in the literature and we investigated the decidability of the problems of emptiness and universality.

Decidability of the emptiness problem is important for verification, because, e.g. the standard *automata-based model-checking* procedure [60] requires to represent both the model and the properties as languages L and L' and to verify the emptiness of the intersection of L with the complement of L'.

To the best of our knowledge, no class of nominal languages proposed in the literature is unfortunately closed under complementation (except for limited forms e.g.[41]). However, this is not a problem when the properties of interest follow the so-called default-accept paradigm in which the *unwanted* behaviour is specified, because complementation is unnecessary at all. This is typically the case for security policies.

We have characterized the largest known classes for which the automata based model-checking is feasible for default-accept properties. We propose to take $PSNA_+$ to express the model and to intersect it with the $FSNA_+$ for the property. Our findings about intersection of these automata and about the decidability of emptiness problem for $PSNA_+$ guarantee that this verification technique is doable in the nominal setting. Further investigation is required to understand the impact of the disposal mechanism on the feasibility of model-checking.

A different approach to verification is proposed in [8]: compliance of a model with respect to a property is checked through a notion of simulation between automata. We can easily adapt this technique to our case, although at the moment decidability of simulation is still unresolved.

Conclusions

We studied and proposed models to abstract resource usage of adaptive systems. Our path departed from ContextML, an ML-like programming languages with primitives for adaptivity. We gave a formal semantics of ContextML and set up a machinery to obtain an abstraction of the behaviour of the programs, that can be model-checked against security properties.

We have shown an extension of ContextML that allows a program to be parametric with respect to the actual resources in the environment, so giving it the opportunity to fit better the actual environment. The only assumptions made on the resources are that they can be picked fresh from the environment, released and checked for equality.

In this case, the behaviour of ContextML programs depend on an unbound a-priori number of resources. The models used to model these behaviour need to deal with that unboundedness. We exploited nominal techniques to investigate the feasibility and the pragmatic utility these models.

In particular we investigated three nominal models: Usages [5], UA [6], VFA [33]. We proved VFA more expressive than UA. We showed a model-checking technique that verifies Usages models against VFA properties.

However, Usages turned out to be not fully adequate to capture the behaviour of programs involving the reuse of resources.

We made our own proposal, by defining two models of automata aimed at representing the properties and the behaviour of adaptive programs. Our most powerful model is intuitively context-free, in that it express Dyck-like language. We studied the language-theoretic properties of our automata, with a focus on the decidability issues of the properties that we felt useful for verification.

Our methodological contribution is the multi-tier approach we adopted. We distilled from the programming language the key primitives that we brought in the abstractions. The abstractions have been investigated from two different perspectives: process algebras and automata. In [6] it has been shown how to mechanically derive a process algebra as abstractions of a programming language. This derivation uses compositionality and structural congruences. Automata have been used to obtain verification procedures in the style of automata-based model-checking [60].

Future research is needed to fill in the gap between our foundational results and prototypical implementations of our abstract model-checking procedure. In this thesis we have shown the connections between the tiers mainly by examples. However, an interesting topic would be mechanically obtain a provable over-approximation of the behaviour of ContextML programs.

As for the specification of system properties, an interesting topic is the logical characterisation of PSNA languages, as [43] does for context-free languages. Such characterization can be useful to specify in a clear way and in abstract terms the behaviour of a software system in a top down approach.

Bibliography

[1] Antoine Amarilli and Marc Jeanmougin. A proof of the pumping lemma for context-free languages through pushdown automata. *CoRR*, abs/1207.2819, 2012.

[2] M. Appeltauer, R. Hirschfeld, M. Haupt, and H. Masuhara. ContextJ: Context-oriented programming with java. *Computer Software*, 28(1), 2011.

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

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

[5] Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari, and Roberto Zunino. Types and effects for resource usage analysis. In *Foundations of Software Science and Computational Structures*, pages 32–47. Springer, 2007.

[6] 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 *LNCS*, pages 19–35. Springer, 2008. Extended version to appear in Math. Stuct. Comp. Sci.

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

[8] Walid Belkhir, Yannick Chevalier, and Michael Rusinowitch. Guarded variable automata over infinite alphabets. *arXiv preprint arXiv:1304.6297*, 2013.

[9] Michael Benedikt, Clemens Ley, and Gabriele Puppis. Automata vs. logics on data words. In Anuj Dawar and Helmut Veith, editors, *CSL*, volume 6247 of *LNCS*, pages 110–124. Springer, 2010.

[10] Chiara Bodei, Pierpaolo Degano, Gian Luigi Ferrari, Letterio Galletta, and Gianluca Mezzetti. Formalising security in ubiquitous and cloud scenarios. In Agostino Cortesi, Nabendu Chaki, Khalid Saeed, and Slawomir T. Wierzchon, editors, *CISIM*, volume 7564 of *Lecture Notes in Computer Science*, pages 1–29. Springer, 2012.

[11] M. Bojanczyk. Data monoids. In Christoph Dürr and Thomas Wilke, editors, *STACS 2011*, volume 9, pages 105–116. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2011.

[12] Mikołaj Bojanczyk, Bartek Klin, and Sławomir Lasota. Automata theory in nominal sets.

[13] Mikolaj Bojanczyk, Bartek Klin, and Slawomir Lasota. Automata with group actions. LICS, pages 355–364, Washington, DC, USA, 2011. IEEE Computer Society.

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

[15] Benedikt Bollig, Aiswarya Cyriac, Paul Gastin, and K. Narayan Kumar. Model checking languages of data words. In Lars Birkedal, editor, *FOSSACS 2012*, volume 7213 of *LNCS*, pages 391–405. Springer, 2012.

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

[17] R. Bruni, A. Corradini, F. Gadducci, A. Lluch Lafuente, and A. Vandin. A conceptual framework for adaptation. In *FASE 2012*, volume to appear of *LNCS*. Springer, 2012.

[18] Betty H. C. Cheng et al. Software engineering for self-adaptive systems: A research roadmap. In *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2009.

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

[20] 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.

[21] Dave Clarke and Ilya Sergey. A semantics for context-oriented programming with layers. In *International Workshop on Context-Oriented Programming*, COP '09, pages 10:1–10:6, New York, NY, USA, 2009. ACM.

[22] ONF Market Education Committee et al. Software-defined networking: The new norm for networks. *ONF White Paper. Palo Alto, US: Open Networking Foundation*, 2012.

[23] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: an overview of contextl. In *Proceedings of the 2005 symposium on Dynamic languages*, pages 1–10. ACM, 2005.

[24] Pierpaolo Degano, Gian Luigi Ferrari, Letterio Galletta, and Gianluca Mezzetti. Types for coordinating secure behavioural variations. In Marjan Sirjani, editor, *COORDINATION*, volume 7274 of *Lecture Notes in Computer Science*, pages 261–276. Springer, 2012.

[25] Pierpaolo Degano, Gian-Luigi Ferrari, Letterio Galletta, and Gianluca Mezzetti. Typing context-dependent behavioural variations. In *PLACES 2012*, volume to appear in EPTCS, 2012.

[26] Pierpaolo Degano, Gian Luigi Ferrari, and Gianluca Mezzetti. Nominal automata for resource usage control. In Nelma Moreira and Rogério Reis, editors, *CIAA 2012*, volume 7381 of *LNCS*, pages 125–137. Springer, 2012.

[27] Pierpaolo Degano, Gian Luigi Ferrari, and Gianluca Mezzetti. Towards nominal context-free model-checking. In Stavros Konstantinidis, editor, *CIAA*, volume 7982 of *Lecture Notes in Computer Science*, pages 109–121. Springer, 2013.

[28] 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.

[29] 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.

[30] Christopher Ferris and Joel Farrell. What are web services? *Commun. ACM*, 46(6):31–, June 2003.

[31] 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.

[32] Andrew D. Gordon. Notes on nominal calculi for security and mobility. In Riccardo Focardi and Roberto Gorrieri, editors, *FOSAD 2000*, volume 2171 of *LNCS*, pages 262–330. Springer, 2001.

[33] Orna Grumberg, Orna Kupferman, and Sarai Sheinvald. Variable automata over infinite alphabets. In Adrian Horia Dediu, Henning Fernau, and Carlos Martín-Vide, editors, *LATA*, volume 6031 of *LNCS*, pages 561–572. Springer, 2010.

[34] R. Hirschfeld, A. Igarashi, and H. Masuhara. ContextFJ: a minimal core calculus for context-oriented programming. In *Proceedings of the 10th international workshop on Foundations of aspect-oriented languages*, pages 19–23. ACM, 2011.

[35] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology, March-April 2008, ETH Zurich*, 7(3):125–151, 2008.

[36] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to automata theory, languages, and computation*, volume 2. Addison-wesley Reading, MA, 1979.

[37] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23:396–450, May 2001.

[38] M. Kaminski and N. Francez. Finite-memory automata. *TCS*, 134(2):329–363, 1994.

[39] Michael Kaminski and Daniel Zeitlin. Finite-memory automata with non-deterministic reassignment. *Int. J. Found. Comput. Sci.*, 21(5):741–760, 2010.

[40] Alexander Kurz, Tomoyuki Suzuki, and Emilio Tuosto. A characterisation of languages on infinite alphabets with nominal regular expressions. In *Theoretical Computer Science*, pages 193–208. Springer, 2012.

[41] Alexander Kurz, Tomoyuki Suzuki, and Emilio Tuosto. On nominal regular languages with binders. In *Foundations of Software Science and Computational Structures*, pages 255–269. Springer, 2012.

[42] Alexander Kurz, Tomoyuki Suzuki, and Emilio Tuosto. Nominal regular expressions for languages over infinite alphabets. extended abstract. *CoRR*, abs/1310.7093, 2013.

[43] Clemens Lautemann, Thomas Schwentick, and Denis Thérien. Logics for context-free languages. In *Selected Papers from the 8th International Workshop on Computer Science Logic*, CSL '94, pages 205–216, London, UK, UK, 1995. Springer-Verlag.

[44] Amaldev Manuel, Anca Muscholl, and Gabriele Puppis. Walking on data words. In *Computer Science–Theory and Applications*, pages 64–75. Springer, 2013.

[45] Ugo Montanari and Marco Pistore. π-calculus, structured coalgebras, and minimal hd-automata. In Nielsen Mogens and Rovan Branislav, editors, *MFCS 2000*, volume 1893 of *LNCS*, pages 569–578. Springer, 2000.

[46] Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Transactions on Computational Logic (TOCL)*, 5(3):403–435, 2004.

[47] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Publishing Company, Incorporated, 1999.

[48] Pawel Parys. Higher-order pushdown systems with data. In Marco Faella and Aniello Murano, editors, *GandALF*, volume 96 of *EPTCS*, pages 210–223, 2012.

[49] D. Perrin and J.E. Pin. *Infinite words: automata, semigroups, logic and games*, volume 141 of *Pure and Applied Mathematics*. Elsevier, 2004.

[50] Andrew M. Pitts. Nominal sets names and symmetry in computer science: Names and symmetry in computer science. volume 57 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press.

[51] Andrew M. Pitts and Ian D. B. Stark. Observable properties of higher order functions that dynamically create local names, or what's new? In M. Borzyszkowski Andrzej and Sokolowski Stefan, editors, *MFCS 1993*, volume 711 of *LNCS*, pages 122–141. Springer, 1993.

[52] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *TAAS*, 4(2), 2009.

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

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

[55] Luc Segoufin. Automata and logics for words and trees over an infinite alphabet. In Zoltán Ésik, editor, *CSL 2006*, volume 4207 of *LNCS*, pages 41–57. Springer, 2006.

[56] Christian Skalka and Scott Smith. History effects and verification. In *Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Taipei, Taiwan, November 4-6, 2004. Proceedings*, volume 3302, page 107. Springer, 2004.

[57] 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.

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

[59] Nikos Tzevelekos and Radu Grigore. History-register automata. In *Foundations of Software Science and Computation Structures*, pages 17–33. Springer, 2013.

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

[61] Martin Von Löwis, Marcus Denker, and Oscar Nierstrasz. Context-oriented programming: beyond layers. In *Proceedings of the 2007 international conference on Dynamic languages: in conjunction with the 15th International Smalltalk Joint Conference 2007*, pages 143–156. ACM, 2007.