View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Archivio della Ricerca - Università di Pisa

Here you can't: context-aware security*

Chiara Bodei, Pierpaolo Degano, Letterio Galletta, and Francesco Salvatori

Dipartimento di Informatica, Università di Pisa
{chiara,degano,galletta}@di.unipi.it,francesco.salvatori@sns.it

Abstract. Adaptive systems improve their efficiency by modifying their behaviour to respond to changes of their operational environment. Also, security must adapt to these changes and policy enforcement becomes dependent on the dynamic contexts. We address some issues of context-aware security from a language-based perspective. More precisely, we extend a core adaptive functional language, recently introduced by some of the authors, with primitives to enforce security policies on the code execution. Then, we accordingly extend the existing static analysis in order to insert checks in a program. The introduced checks guarantee that no violation occurs of the required security policies.

1 Introduction

Context and Adaptivity Today's software systems are expected to operate *every time* and *everywhere*: they have therefore to cope with changing environments, without compromising the correct behaviour of applications and without breaking the guarantees on their non-functional requirements, e.g., security or quality of service. As a consequence, software needs effective mechanisms to sense the changes of the operational environment, namely the *context*, in which the application is plugged in, and to properly *adapt* to changes. At the same time, these mechanisms must maintain the functional and non-functional properties of applications after the adaptation steps.

The context is a key notion for adaptive software. It is usually a complex entity independent from the single applications. It includes different kinds of computationally accessible information coming both from outside (e.g., sensor values, available devices, code libraries offered by the environment), and from inside the application boundaries (e.g., its private resources, user profiles, etc.).

Context Oriented Programming (COP), introduced by Costanza [9], is a recent paradigm that explicitly deals with contexts and provides programming adaptation mechanisms to support dynamic changes of behaviour, in reaction to changes in the context. Also subsequent work [16,1,18,3] follow this approach to address the design and the implementation of concrete programming languages. The notion of context-dependent *behavioural variation* is central to this paradigm: it is a chunk of behaviour that can be activated depending on the current context hosting the application, so to dynamically modify the execution.

* This work has been partially supported by the MIUR-PRIN project Security Horizons.

Security and Contexts Security is one of the challenges arising in context-aware systems. The combination of security and context-awareness requires to address two distinct and interrelated aspects. On the one side, security requirements may reduce the adaptivity of software, by adding further constraints on its possible actions. On the other side, new highly dynamic security mechanisms are needed to scale up to adaptive software. Such a duality has already been put forward in the literature [26,6], that presents two ways of addressing it: *securing context-aware systems* and *context-aware security.*

Securing context-aware systems aims at rephrasing the standard notions of confidentiality, integrity and availability [24] and at developing techniques for guaranteeing them [26]. The challenge is to understand how to get secure and trusted context information. Contexts may contain indeed sensible data of the working environment (e.g., information about surrounding digital entities) that should be protected from unauthorised access and modification, in order to grant confidentiality and integrity. A trust model is needed, taking also care of the roles of entities that can vary from a context to another. Such a trust model is important also because contextual information can be inferred from the environmental one, provided by or extracted from digital entities therein. As a matter of fact, these entities may misbehave and forge deceptive data. Since information is distributed, denial-of-service can be even more effective, because it can prevent a whole group of digital entities to access relevant contextual information.

Context-aware security is dually concerned with the use of context information to drive security decisions. It has therefore to do with the definition and enforcement of high-level policies that talk about, are based, and depend on the notion of dynamic context. Consider, for instance, the usual no flash photography policy in museums. A standard security policy does *not* allow people to take pictures, using the flash. A context-aware security is more flexible: it instead forbids flashing *only* inside those rooms that exhibit delicate paintings.

Most of the work on securing context-aware systems and on context-aware security aims at implementing mechanisms at different levels of the infrastructure, e.g., in the middleware [25] or in the interaction protocols [15]. More foundational issues have instead been studied less within the programming languages approach we follow; while some work has been already carried on considering process algebras, e.g. the Ambient Calculus [7]. Moreover, the two dual aspects of context-aware security sketched above are often tackled separately, thus we still lack a unifying concept of security. Also, in the adaptive framework, the most relevant concern is controlling the accesses to resources and smart things. See e.g., [26,17,27], and [2,10] that show the relevance of role based access control policies in e-health applications.

Our proposal The kernel of our proposal is $\mathrm{ML}_{\mathrm{CoDa}}$, a core of ML extended with COP features. Its main novelty is to be a two-component language: a declarative constituent for programming the context and a functional one for computing (see [14] for details about its design).

The context in $\mathrm{ML}_{\mathrm{CoDa}}$ is a knowledge base implemented as a Datalog program [23,20]. To choose the right thing to do, adaptive programs can therefore

query the context by simply verifying whether a given property holds in it, in spite of the fact that this may involve possibly complex deductions.

Programming adaptation is specified through behavioural variations, that are activated depending on information picked up from the context, so to dynamically modify the execution. Differently from other proposals, a behavioural variation of ML_{CoDa} is a first class, higher-order construct: it can be referred to by identifiers, and passed as argument to, and returned by functions. This is a natural hook for programming dynamic and compositional adaptation patterns, as well as reusable and modular code. As a matter of fact ML_{CoDa}, as it is, offers the features needed for addressing context-aware security issues, in particular for defining access control policies and for enforcing them.

First, we can express *system-defined* policies in *stratified Datalog with negation*, which is one of the two components of ML_{CoDa}. This version of Datalog is sufficiently expressive for our policies. It is powerful enough to express all relational algebras [12]. In addition, it is fully decidable and guarantees polynomial response time. Furthermore, adopting a stratified-negation-model is common and many logical languages for defining control access policies compile in Stratified Datalog, e.g., [5,19,11].

Secondly, the *dispatching* mechanism of ML_{CoDa}, that selects behavioural variations, suffices for checking whether a specific policy holds, and for choosing the chunk of behaviour that does. Our language therefore requires no extensions to deal with security policies.

Actually, we can distinguish two classes of policies: those specified by the system to control the user's behaviour, and those expressed by the application. We are only interested in system policies. This is because the application developer has indeed full knowledge of his policies, and so he can specify them as behavioural variation constructs. Instead, the application has no *a priori* knowledge about the policies that contexts may require; there is then no warranty that the application was designed to comply with them.

In the world of "secure" adaptive software, a runtime error can occur because of two different reasons, besides the presence of usual bugs. An application can fail because it cannot adapt to the current context (*functional failure*) or because it violates a policy (*non-functional failure*). One would like to predict as earlier as possible if either case may occur. Note that some information is only available at runtime, e.g., the actual value of some elements in the running context is only known when the application is linked with it. Consequently, a fully static approach is not possible, and rather we have a two-phase verification: one at compile time and one at linking time.

This is the approach followed in [13], which proposes a two-phase static technique for verifying whether a program adequately reacts to all context changes, signalling possible functional failures. The first phase is based on a type and effect system that safely computes an approximation of the behaviour of the application at compile time. This approximation is then used at linking time to verify that the resources needed by the application to run will always be available in the actual context, and in its future modifications.

We extend here this technique to guarantee an application to never violate the required policies, so making useless any runtime monitor. However, this yes/no procedure may lead to reject too many applications. As a consequence, our extension is designed to also provide us with the means for instrumenting the code with suitable checks aimed at guarding the activities that can be considered risky. Actually, we have a sort of *runtime monitor* that is switched on and off at need, and, again, the dispatching mechanism of $\mathrm{ML}_{\mathrm{CoDa}}$ suffices for natively supporting it.

The implementation of our runtime monitor requires a preliminar step to detect the potentially unsafe operations the application may perform. Actually these are the operations that update the context (`tell` and `retract`) and may therefore lead to a violation of the policy Φ to be enforced.

Using the effects computed at compile time, we first build a graph $\mathcal{G}$ at linking time. By visiting $\mathcal{G}$, we safely predict which contexts the application will pass through while running. Before launching the execution, we detect the dangerous operations by checking the policy Φ on each node of the graph. While building $\mathcal{G}$, we also label its edges so to single out the risky `tell`/`retract` operations in the code. Our runtime monitor can then guard them, while it will be switched off on the remaining actions. Actually, we collect the labels of the risky operations and associate the value `on` with them, and `off` with all the others. Note in passing that this information becomes part of the context. Finally, each occurrence of a `tell`/`retract` will be replaced by a behavioural variation, that checks if Φ holds in the running context, when the value of its label is `on`.

The next section will introduce $\mathrm{ML}_{\mathrm{CoDa}}$ and our proposal with the help of a running example, along with an intuitive presentation of the various components of our compile time and linking time static analysis, as well as an informal presentation of how security is dynamically enforced. The formal definitions and the statements of the correctness of our proposal will follow in the remaining sections. The conclusion summarises our results and discusses some future work.

2 Running example

We illustrate our methodology by considering a multimedia guide to a museum implemented as a smartphone application, starting from the case study of [14]. Assume the museum has a wireless infrastructure exploiting different technologies, like WiFi, Bluetooth, Irda or RFID. When a smartphone is connected, the visitor can access the museum Intranet and its website, from which he can download information about the exhibit and further multimedia contents.

Each exhibit is equipped with a wireless adapter (Bluetooth, Irda, RFID) and a QR code. They are only used to provide the guide with the URL of the exhibit, which is retrieved by using one of the above technologies, depending on the smartphone capabilities. If the smartphone is equipped with a Bluetooth adapter, it can connect to the one of the exhibit and directly download the URL; otherwise, if the smartphone has a camera and a QR decoder, the guide can retrieve the URL by taking a picture of the code and by decoding it.

In $\mathrm{ML}_{\mathrm{CoDa}}$ the smartphone capabilities are stored in the context as Datalog clauses. Consider, for instance, the following clauses defining when the smartphone can either directly download the URL (the predicate `device(d)` holds whether the device $\mathtt{d} \in \{\mathtt{irda}, \mathtt{bluetooth}, \mathtt{rfid_reader}\}$ is available) or it can take the URL by decoding a picture (the parameter `x` in the predicate `use_qrcore` is a handle for using the decoder):

```
direct_comm() ← device(irda).
direct_comm() ← device(bluetooth).
direct_comm() ← device(rfid_reader).

use_qrcode(x) ← user_prefer(qr_code),
               qr_decoder(x),
               device(camera).
use_qrcode(x) ← qr_decoder(x),
               device(camera),
               ¬ device(irda),
               ¬ device(rfid_reader),
               ¬ device(bluetooth).
```

Contextual data, such as the above predicates `use_qrcode`(`decoder`) and `direct_comm`(), affect the download. To change the program flow in accordance to the current context, we exploit behavioural variations, offered by the functional part of $\mathrm{ML}_{\mathrm{CoDa}}$. Syntactically they are similar to pattern matching, where Datalog goals replace patterns and where parameters can additionally occur (see below). Behavioural variations are similar to functional abstractions, but their application triggers a *dispatching mechanism* that at runtime inspects the context and selects the first expression whose goal holds.

For example, in the following function `getExhibitData`, we declare a behavioural variation (called `url`) with an unused argument "_", that returns the URL of an exhibit. Retrieval of the URL depends on the smartphone capabilities, as explained above. If the smartphone can directly download the URL, then it does, through the channel returned by the function `getChannel`(), otherwise the smartphone takes a picture of the QR code and decodes it. Note that, in this second case, the variables `decoder` and `cam` will be assigned the handles of the decoder and the one of the camera deduced by the Datalog machinery. These handles are used by the functions `take_picture` and `decode_qr` to interact with the actual smartphone resources.

```
fun getExhibitData () =
  let url = (_){
    ← direct_comm().
         let c = getChannel () in
           receiveData c,
    ← use_qrcode(decoder), camera(cam).
         let p = take_picture cam in
           decode_qr decoder p }
  in
      getRemoteData #url
```

The behavioural variation (bound to) `url` is applied before invoking the function `getRemoteData` (for readability, here we use a slightly simplified syntax for behavioural variations application represented by #; for details see Section 3), that connects to the corresponding website and downloads the required information.

Formally, applying the function `getExhibitData` to unit, we have the following slightly simplified computation, where a transition $C, e \to C', e'$ says that the expression e is evaluated in the context C and reduces to e' changing the context C to C':

$$C, \texttt{getExhibitData}\,() \to^{\star} C, \texttt{getRemoteData}\,\#u \to^{\star}$$
$$C, \texttt{getRemoteData}(\texttt{receiveData}\,n) \qquad (*\ n \text{ is retuned by } \texttt{getChannel}\ *)$$

If the context C satisfies the goal $\leftarrow$ `direct_comm`(), moving from the second to the third configuration in the computation above, the dispatching mechanism selects the first expression of the behavioural variation u (the one bound to `url` in the body of the function `getExhibitData`).

To update the context at runtime, $\mathrm{ML}_{\mathrm{CoDa}}$ provides us with the constructs `tell` and `retract`, that add and remove Datalog facts, respectively. For instance, in our example, the context stores information about the room in which the user is, through the predicate `current_room`. If the user moves from the room *delicate paintings* to the one *sculptures*, the application updates the context by executing

```
retract current_room(delicate_paintings)
tell current_room(sculptures).
```

Assume now that one can take pictures in every room, but that in the rooms with delicate paintings it is forbidden to use the camera flash not to damage the exhibits. This policy is specified by the museum (the system) and it must be enforced during the user's tour. Since policies predicate on the context, they are easily expressed as Datalog goals. Let the fact `flash_on` hold when the flash is active and the fact `button_clicked` when the user presses the button of the camera. The above policy Φ is then expressed in Datalog as the goal

```
phi ← ¬ current_room(delicate_paintings)
phi ← ¬ button_clicked
phi ← ¬ flash_on
```

that, intuitively, is the result of compiling the following logical condition:

$$current_room(delicate_paintings) \Rightarrow (button_clicked \Rightarrow \neg flash_on)$$

Of course, the museum can specify other policies, and we assume that there is a unique global policy Φ (referred to in the code as `phi`), obtained by suitably combining all the required policies. The enforcement is obtained by a runtime monitor that checks the validity of Φ right before every context changes, i.e., before every `tell`/`retract`. We remark that the introduction of the runtime monitor requires no modification of the language, because our policies are Datalog goals and can be checked by simply invoking the dispatching mechanism.

An application fails to adapt to a context (functional failure), when the dispatching mechanism fails, i.e., when a behavioural variation gets stuck. Consider to evaluate `getExhibitData` on a smartphone without wireless technology and QR decoder. Of course, no context will ever satisfy the goals of the behavioural variation `url`. Therefore, when `url` is applied, no case can be selected.

Another kind of failure happens when an application violates a policy (non-functional failure). In our example, it happens when attempting to use the flash, if the context includes `current_room(delicate_paintings)`.

To avoid functional failure and to optimise the policy enforcement, we equip ML_{CoDa} with a two-phase static analysis: a type and effect system and a control-flow analysis. The analysis checks if an application will be able to adapt to its execution contexts, and detects which contexts can violate the required policies.

At *compile time*, we associate a type and an effect with an expression e. The type is (almost) standard, and the effect is an over-approximation of the actual runtime behaviour of e, called *history expression*. The effect abstractly represents the changes and the queries performed on the context during its evaluation.

To intuitively understand how this phase works, take the expression e_a:

```
eₐ = let x =
        if always_flash then
          let y = tell F₁¹ in tell F₂²
        else
          let y = tell F₁³ in tell F₃⁴
      in
        tell F₄⁵
```

For clarity, here (and in the syntax in Sect. 3), we show the labels of `tell`/`retract` in the code, inserted by the compiler during syntax analysis or type checking. The facts above are intended to be $F_1 \equiv$ `photocamera_started`; $F_2 \equiv$ `flash_on`; $F_3 \equiv$ `mode_museum_activated`; $F_4 \equiv$ `button_clicked`.

The type of e_a is `unit`, i.e. that of $\mathtt{tell\,F_4}$, and its history expression is

$$H_a = (((tell\,F_1^1 \cdot tell\,F_2^2)^3 + (tell\,F_1^4 \cdot tell\,F_3^5)^6)^7 \cdot tell\,F_4^8)^9$$

(in H_a $\cdot$ means sequential composition, + is for conditional expression). Depending on the value of `always_flash`, which records whether the user wants the flash to be always usable, the expression e_a can either perform the action $\mathtt{tell\,F_1}$ followed by $\mathtt{tell\,F_2}$, *or* the action $\mathtt{tell\,F_1}$ followed by $\mathtt{tell\,F_3}$. The context is informed that the flash is on or off, respectively. After that, e_a will perform $\mathtt{tell\,F_4}$, no matter what the previous choice was.

The labels of history expressions allow us to link the actions in histories to the corresponding points inside the code. For instance, the first $\mathtt{tell\,F_1}$ in H_a, which is labelled 1, corresponds to the first $\mathtt{tell\,F_1}$ in e_a, which is also labelled 1, while the $\mathtt{tell\,F_4}$, labelled 8 in H_a, corresponds to the label 5 in e_a. More precisely, the correspondences are $\{1 \mapsto 1, 2 \mapsto 2, 4 \mapsto 3, 5 \mapsto 4, 8 \mapsto 5\}$; instead, the abstract labels that do not annotate *tell*/*retract* have no corresponding labels.

The effects are exploited at *linking time* (*i*) to verify that the application can adapt to all contexts arising at runtime; and (*ii*) to identify which `tell`/`retract`

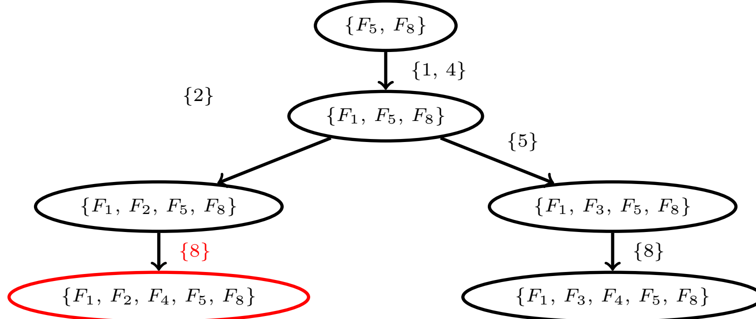

Fig. 1. The evolution graph for the context $C \supseteq \{F_5, F_8\}$ and the history expression $H_a = (((tell\, F_1^1.tell\, F_2^2)^3 + (tell\, F_1^4.tell\, F_3^5)^6)^7.tell\, F_4^8)^9$

are risky and need to be checked by the monitor. If our static analysis discovers that a `tell`/`retract` is possibly unsafe, i.e., it may lead to a violation, we can activate the monitor during its evaluation, otherwise the monitor keeps inactive. To do that, our control-flow analysis first builds a graph to trace how the initial context evolves during execution, and then it finds out in which contexts there might be a violation and which operation might cause it.

Back to our example, consider an initial context C that includes the facts $F_8 \equiv$ `current_room(delicate_paintings)` and F_5 (irrelevant here), but not the facts $\{F_1, F_2, F_3, F_4\}$. Starting from C (and from the history expression H_a computed above) our loading time analysis builds the graph described in Fig. 1 (we show only the relevant facts of C). Nodes represent contexts, possibly reachable at runtime, while edges represent transitions from one context to another. Each edge is annotated with the set of actions in H_a that may cause that transition. For instance, from the initial context it is possible to reach the context also including the fact F_1, because of the two *tell* operations labelled by 1 and by 4 in the history expression. Therefore, an edge can have more than one label (e.g., the one labelled $\{1, 4\}$). Note also that the same label may occur in more than one edge (e.g., the label 8).

As said, the labelling is done during the type checking and plays a key role in enforcing security policies. Here, e.g., by visiting the graph, we observe that the context corresponding to the node $\{F_1, F_2, F_4, F_5, F_8\}$ (in red in Fig. 1) violates our no-flash policy. This amounts to identifying a possible runtime violation. Since this node has a single incoming edge, labelled with 8 (in red in the Fig. 1), we can deduce that the possibly risky action is the corresponding dynamic `tell F_4`, labelled by 5 in the code. For preventing a violation, all we have to do is activating the runtime monitor right before executing this operation.

(DLET1)
$$\frac{\rho[(G.e_1, \rho(\tilde{x}))\,/\tilde{x}] \vdash C,\, e_2 \rightarrow C',\, e_2'}{\rho \vdash C,\, dlet\, \tilde{x} = e_1\, when\, G\, in\, e_2 \rightarrow C',\, dlet\, \tilde{x} = e_1\, when\, G\, in\, e_2'}$$

(DLET2)
$$\frac{}{\rho \vdash C,\, dlet\, \tilde{x} = e_1\, when\, G\, in\, v \rightarrow C,\, v}$$

(PAR)
$$\frac{\rho(\tilde{x}) = Va \qquad dsp(C,\, Va) = (e,\, \theta)}{\rho \vdash C,\, \tilde{x} \rightarrow C,\, e\,\theta}$$

(VAAPP3)
$$\frac{dsp(C,\, Va) = (e,\, \{\overrightarrow{c}/\overrightarrow{y}\})}{\rho \vdash C,\, \#((x)\{Va\},\, v) \rightarrow C,\, e\{v/x,\ \overrightarrow{c}/\overrightarrow{y}\}}$$

(TELL2)
$$\frac{dsp(C \cup \{F\},\, phi.()) = ((),\, \emptyset)}{\rho \vdash C,\, tell(F)^l \rightarrow C \cup \{F\},\, ()}$$

(RETRACT2)
$$\frac{dsp(C \smallsetminus \{F\},\, phi.()) = ((),\, \emptyset)}{\rho \vdash C,\, retract(F)^l \rightarrow C \smallsetminus \{F\},\, ()}$$

Fig. 2. The reduction rules for new constructs of $\mathrm{ML_{CoDa}}$

3 $\mathbf{ML_{CoDa}}$

We briefly survey the syntax and the operational semantics of $\mathrm{ML_{CoDa}}$; for more details, and for a longer, fully worked out example see [14,13].

Syntax $\mathrm{ML_{CoDa}}$ consists of two sub-languages: a Datalog with negation to describe the context, and a core ML extended with COP features.

The Datalog part is standard: a program is a set of facts and clauses. We assume that each program is safe [8]; to deal with negation, we adopt *Stratified Datalog* under the Closed World Assumption.

We enforce security properties by introducing policies Φ, expressed as Datalog goals, one of the components of $\mathrm{ML_{CoDa}}$. As a consequence, the language requires no extensions to deal with security policies. The mechanism for selecting behavioural variations is already there, and can be used for checking whether a specific policy holds, and for selecting the chunk of behaviour that does.

The functional part inherits most of the ML constructs. In addition to the usual ones, our values include Datalog facts F and behavioural variations. Moreover, we introduce the set $\tilde{x} \in DynVar$ of *parameters*, i.e., variables that assume values depending on the properties of the running context, while Var are standard identifiers, with the proviso that $Var \cap DynVar = \emptyset$. The syntax of $\mathrm{ML_{CoDa}}$ is below, where $C, C_p \in Context$ are contexts:

$$\begin{aligned}
Va &::= G.e \mid G.e, Va \\
v &::= c \mid \lambda_f x.e \mid (x)\{Va\} \mid F \\
e &::= v \mid x \mid \tilde{x} \mid e_1\, e_2 \mid let\, x\ =\ e_1\, in\, e_2 \mid if\, e_1\, then\, e_2\, else\, e_3 \mid \\
&\quad\ dlet\, \tilde{x}\ = e_1\, when\, G\, in\, e_2 \mid tell(e_1)^l \mid retract(e_1)^l \mid e_1 \cup e_2 \mid \#(e_1, e_2)
\end{aligned}$$

The facilitate our static analysis (see Section 5) we require that each `tell`/`retract` in the code is uniquely and mechanically associated with a label $l \in Lab_C$. As usual, labels do not affect the dynamic semantics of the calculus.

COP- oriented constructs include behavioural variations $(x)\{Va\}$, each consisting of a variation Va, i.e., a list of expressions $G_1.e_1, \ldots, G_n.e_n$ guarded by Datalog goals G_i. The variable x can freely occur in the expressions e_i. At run-time, the first goal G_i satisfied by the context determines the expression e_i to be selected (*dispatching*). Context-dependent binding is the mechanism to declare variables whose values depend on the context. The *dlet* construct implements the context-dependent binding of a parameter $\tilde{x}$ to a variation Va.

The *tell*/*retract* constructs update the context by asserting/retracting facts, provided that the resulting context satisfies the system policy Φ.

The append operator $e_1 \cup e_2$ concatenates behavioural variations, so allowing for dynamic compositions. The application of a behavioural variation $\#(e_1, e_2)$ applies e_1 to its argument e_2. To do so, the dispatching mechanism is triggered to query the context and to select from e_1 the expression to run, if any.

Semantics We now endow $\mathrm{ML}_{\mathrm{CoDa}}$ with a small-step operational semantics.

For the Datalog evaluation, we adopt the top-down standard semantics of stratified programs [8]. Given a context C and a goal G, $C \vDash G\, with\, \theta$ means that there exists a substitution θ, replacing constants for variables, such that the goal G is satisfied in the context C.

The semantics of $\mathrm{ML}_{\mathrm{CoDa}}$ is defined for expressions with no free variable, but possibly with free parameters, thus allowing for openendness. To this aim, we have in an environment ρ, i.e., a function mapping parameters to variations $DynVar \rightarrow Va$.

A transition $\rho \vdash C,\, e \rightarrow C',\, e'$ says that in the environment ρ, the expression e is evaluated in the context C and reduces to e' changing the context C to C'. We assume that the initial configuration is $\rho_0 \vdash C,\, e_p$ where ρ_0 contains the bindings for all system parameters, and C results from the linking of the system context and of the application context.

Most of the rules of the small-step operational semantics are inherited from ML. Fig. 2 shows the inductive definitions of those for our new constructs. For brevity, we omit the obvious congruence rules that reduce subexpressions, e.g., $\rho \vdash C,\, tell(e)^l \rightarrow C',\, tell(e')^l$ if $\rho \vdash C,\, e \rightarrow C',\, e'$.

We briefly comment below on the rules displayed. The rules (Dlet1) and (Dlet2) for the construct *dlet*, and the rule (Par) for parameters implement our context-dependent binding. For brevity, we assume here that e_1 contains no parameters. The rule (Dlet1) extends the environment ρ by appending $G.e_1$ in front of the existent binding for $\tilde{x}$. Then, e_2 is evaluated under the updated environment. Notice that the *dlet* does not evaluate e_1, but only records it in the environment in a sort of call-by-name style. The rule (Dlet2) is standard: the whole *dlet* reduces to the value which eventually e_2 reduces to.

The (Par) rule looks for the variation Va bound to $\tilde{x}$ in ρ. Then, the dispatching mechanism selects the expression to which $\tilde{x}$ reduces. The dispatching

mechanism is implemented by the partial function dsp, defined as

$$dsp(C,\ (G.e, Va)) = \begin{cases} (e,\ \theta) & \text{if } C \vDash G\, with\, \theta \\ dsp(C,\ Va) & \text{otherwise} \end{cases}$$

It inspects a variation from left to right to find the first goal G satisfied by C, under a substitution θ. If this search succeeds, the dispatching returns the corresponding expression e and θ. Then, $\tilde{x}$ reduces to $e\,\theta$, i.e., to e whose variables are bound by θ. Instead, if the dispatching fails because no goal holds, the computation gets stuck, because the program cannot adapt to the current context. Our static analysis is also designed to prevent this kind of runtime errors.

As an example of context-dependent binding, consider the simple conditional expression $\texttt{if}\, \tilde{x} = F_2\ \texttt{then}\ 42\ \texttt{else}\ 51$, in an environment ρ that binds the parameter $\tilde{x}$ to $e' = \texttt{G}_1.\texttt{F}_5, \texttt{G}_2.\,\texttt{F}_2$ and in a context C that satisfies the goal G_2 but not G_1:

$$\rho \vdash C,\ \texttt{if}\, \tilde{x} = F_2\ \texttt{then}\ 42\ \texttt{else}\ 51 \rightarrow C,\ \texttt{if}\, F_2 = F_2\ \texttt{then}\ 42\ \texttt{else}\ 51 \rightarrow C,\ 42$$

In the first step, we retrieve the binding for $\tilde{\texttt{x}}$ (recall it is e'), where $dsp(C,\ e') = dsp(C, \texttt{G}_1.\texttt{F}_5, \texttt{G}_2.\,\texttt{F}_2) = (\texttt{F}_2, \theta)$, for a suitable substitution θ. Note in passing that facts are values, so we can bind them to parameters and test their equivalence by a conditional expression.

The application of the behavioural variation $\#(e_1, e_2)$ evaluates the subexpressions until e_1 reduces to $(x)\{Va\}$ and e_2 to a value v. Then, the rule (VaApp3) invokes the dispatching mechanism to select the relevant expression e from which the computation proceeds after v is substituted for x. Also in this case the computation gets stuck, if the dispatching mechanism fails. As an example, consider the behavioural variation $(x)\{G_1.c_1, G_2.x\}$ and apply it to the constant c in a context C that satisfies the goal G_2, but not G_1. Since $dsp(C, (x)\{G_1.c_1, G_2.x\}) = (x,\ \theta)$ for some substitution θ, we get

$$\rho \vdash C,\ \#((x)\{G_1.c_1, G_2.x\}, c)\ \rightarrow\ C,\ c$$

The rule for $tell(e)^l/retract(e)^l$ evaluates the expression e until it reduces to a fact F, which is a value of $\mathrm{ML}_{\mathrm{CoDa}}$. The new context C', obtained from C by adding/removing F, is checked against the security policy Φ. Since Φ is a Datalog goal, we can easily reuse our dispatching machinery, implementing the check as a call to the function dsp where the first argument is C' and the second one is the trivial variation $\texttt{phi}.()$. If this call produces a result, then the evaluation yields the unit value $()$ and the new context C'.

The following example shows the reduction of a *retract* construct violating the policy Φ, of Section 2. Let the context be $C = \{F_3,\ F_4,\ F_5\}$ and apply the function $\texttt{f} = \lambda\texttt{x}.\,\texttt{if}\,\texttt{e}_1\,\texttt{then}\,\texttt{F}_5\,\texttt{else}\,\texttt{F}_4$ to unit, assuming that the evaluation of e_1 reduces to $\texttt{false}$ without changing the context:

$$\rho \vdash C,\ \texttt{retract(f ())}^l \rightarrow^* C,\ \texttt{retract(F}_4\texttt{)}^l \not\rightarrow$$

$$\frac{}{C,\, (\ni \cdot H)^l \to C,\, H} \qquad \frac{}{C,\, \epsilon^l \to C,\, \ni} \qquad \frac{}{C,\, tell\, F^l \to C \cup \{F\},\, \ni}$$

$$\frac{}{C,\, retract\, F^l \to C \backslash \{F\},\, \ni} \qquad \frac{C, H_1 \to C',\, H_1'}{C, (H_1 + H_2)^l \to C',\, H_1'} \qquad \frac{C, H_2 \to C',\, H_2'}{C, (H_1 + H_2)^l \to C',\, H_2'}$$

$$\frac{C,\, H_1 \to C',\, H_1'}{C,\, (H_1 \cdot H_2)^l \to C',\, (H_1' \cdot H_2)^l} \qquad \frac{}{C, (\mu h.H)^l \to C, H[(\mu h.H)^l/h]}$$

$$\frac{C \vDash G}{C,\, (ask\, G.H \otimes \Delta)^l \to C,\, H} \qquad \frac{C \nvDash G}{C,\, (ask\, G.H \otimes \Delta)^l \to C,\, \Delta}$$

Fig. 3. Semantics of History Expressions

Since the policy requires that the fact F_4 always holds, every attempt to remove it from the context violates Φ. Consequently, the evaluation gets stuck because $dsp(C \smallsetminus \{F_4\}, \texttt{phi}.())$ fails.

Instead, if e_1 reduces to `true`, there is no violation of the policy and the evaluation reduces to unit:

$$\rho \vdash C,\, \texttt{retract(f ())}^l \to^* C,\, \texttt{retract(F}_5\texttt{)}^l \to C \smallsetminus \{\texttt{F}_5\}, ()$$

4 Type and Effect System

We now associate $\mathrm{ML}_{\mathrm{CoDa}}$ expressions with a type, an abstraction called *history expression*, and a function called *labelling environment*. During the verification phase, the virtual machine uses this history expression to ensure that the dispatching mechanism will always succeed at runtime. Then, the labelling environment is used to drive us in instrumenting the code with security checks. First, we briefly present history expressions and labelling environments, and then the rules of our type and effect system.

History Expressions They are a simple process algebra used to soundly abstract the execution histories that a program may generate [4]. Here, history expressions approximate the sequence of actions that a program may perform over the context at runtime, i.e., asserting/retracting facts, and asking if a goal holds.

To support the following formal development, we assume that history expressions are uniquely labelled on a given set of Lab_H. Labels allow us to go back from static actions in histories to the corresponding actions inside the code. The syntax of history expressions is described below:

$$H ::= \ni \mid \epsilon^l \mid h^l \mid (\mu h.H)^l \mid tell\, F^l \mid retract\, F^l \mid (H_1 + H_2)^l \mid (H_1 \cdot H_2)^l \mid \Delta$$
$$\Delta ::= (ask\, G.H \otimes \Delta)^l \mid fail^l$$

The empty history expression abstracts programs which do not interact with the context. For technical reasons, we syntactically distinguish when the empty history expression comes from the syntax (ϵ^l) and when it is instead obtained by reduction in the semantics ($\ni$). The history expression $\mu h.H$ represents possibly recursive functions, where h is the recursion variable; the "atomic" history expressions $tell\, F$ and $retract\, F$ are for the analogous expressions of $\mathrm{ML_{CoDa}}$; the non-deterministic sum $H_1 + H_2$ abstracts the conditional expression *if-then-else*; the concatenation $H_1 \cdot H_2$ is for sequences of actions, that arise, e.g., while evaluating applications; Δ mimics our dispatching mechanism, where Δ is an *abstract variation*, defined as a list of history expressions, each element H_i of which is guarded by an $ask\, G_i$.

For example, the history expression computed for the behavioural variation `url` in the function `getExhibitData` of Section 2, is $H_{url} = ask\, G_1.H_1 \otimes ask\, G_2.H_2 \otimes fail$, where the goals $G_1 = \leftarrow$ `direct_comm()` and $G_2 = \leftarrow$ `use_qrcode (decoder), camera(cam)` and where H_1 is the effect of the expression guarded by G_1 and H_2 is the effect of the one guarded by `G2` . Intuitively, H_{url} says that at least one between G_1 or G_2 must be satisfied by the context in order to successfully apply the behavioural variation `url`.

Given a context C, the behaviour of a history expression H is formalised by the transition system, inductively defined in Fig. 3. Configurations have the form $C, H \rightarrow C', H'$ meaning that H reduces to H' in the context C and yields the context C'. Most rules are similar to the ones presented in [4]: below we only comment on those dealing with the context. An action $tell\, F$ reduces to $\ni$ and yields a context C' where the fact F has just been added; similarly for $retract\, F$. Differently from what we do in the semantic rules, here we do not consider the possibility of a policy violation: history expressions approximate how the application would behave in absence of any type of check. The rules for Δ scan the abstract variation and look for the first goal G satisfied in the current context; if this search succeeds, the whole history expression reduces to the history expression H guarded by G; otherwise the search continues on the rest of Δ. If no satisfiable goal exists, the stuck configuration $fail$ is reached, to indicate that the dispatching mechanism fails.

Labelling Environment We assume as given the function $h : Lab_H \rightarrow H$ that recovers a construct in a given history expression from a label l. Then, we will define a way of going back from a `tell/retract` in a history expression to the corresponding operations in the code, by exploiting their labels in the set Lab_C (see Section 3). As an example, consider the history expression H_a of Section 2, and the correspondence given there between its labels and those in the code: $\{1 \mapsto 1, 2 \mapsto 2, 4 \mapsto 3, 5 \mapsto 4, 8 \mapsto 5\}$. The function below will do that and will be computed by our type and effect system.

Definition 1 (Labelling environment). *A* labelling environment *is a (partial) function $\Lambda : Lab_H \rightarrow Lab_C$, defined only if $h(l) \in \{tell(F), retract(F)\}$.*

Typing rules Here, we only give a logical presentation of our type and effect system. We assume that our Datalog is typed, i.e., that each predicate has a

fixed arity and a type (see [21]). From here onwards, we simply assume that there exists a Datalog typing function γ that, given a goal G, returns a list of pairs (x, type-of-x), for all the variables x in G.
The rules of our type and effect systems have:

- the usual environment Γ binding the variables of an expression:

$$\Gamma ::= \emptyset \mid \Gamma, x : \tau$$

  where $\emptyset$ denotes the empty environment and $\Gamma, x : \tau$ denotes an environment having a binding for the variable x (x does not occur in Γ).
- a further environment K that maps a parameter $\tilde{x}$ to a pair consisting of a type and an abstract variation Δ. The information in Δ is used to resolve the binding for $\tilde{x}$ at runtime. Formally:

$$K ::= \emptyset \mid K, (\tilde{x}, \tau, \Delta)$$

  where $\emptyset$ denotes the empty environment and $K, (\tilde{x}, \tau, \Delta)$ denotes an environment having a binding for the parameter $\tilde{x}$ ($\tilde{x}$ does not occur in K).

Our typing judgements have the form $\Gamma; K \vdash e : \tau \triangleright H; \Lambda$, expressing that in the environments Γ and K the expression e has type τ, effect H and yields a labelling environment Λ.

The syntax of types is

$$\tau_c \in \{int, bool, unit, \ldots\} \qquad \phi \in \wp(Fact)$$
$$\tau ::= \tau_c \mid \tau_1 \xrightarrow{K|H} \tau_2 \mid \tau_1 \xRightarrow{K|\Delta} \tau_2 \mid fact_\phi$$

We have basic types (*int*, *bool*, *unit*), functional types, behavioural variations types, and facts. Some types are annotated for analysis reason. In the type $fact_\phi$, the set ϕ soundly contains the facts that an expression can be reduced to at runtime (see the semantics rules (Tell2) and (Retract2)). In the type $\tau_1 \xrightarrow{K|H} \tau_2$ associated with a function f, the environment K is a precondition needed to apply f. Here, K stores the types and the abstract variations of parameters occurring inside the body of f. The history expression H is the latent effect of f, i.e., the sequence of actions which may be performed over the context during the function evaluation. Analogously, in the type $\tau_1 \xRightarrow{K|\Delta} \tau_2$ associated with the behavioural variation $bv = (x)\{Va\}$, K is a precondition for applying bv, while Δ is an abstract variation. The variation Δ represents the information that the dispatching mechanism uses at runtime to apply bv.

We now introduce the *orderings* $\sqsubseteq_H, \sqsubseteq_\Delta, \sqsubseteq_K, \sqsubseteq_\Lambda$ on H, Δ, K and Λ, respectively (often omitting the indexes when unambiguous). We define:

- $H_1 \sqsubseteq_H H_2$ if and only if $\exists H_3$ such that $H_2 = H_1 + H_3$;
- $\Delta_1 \sqsubseteq_\Delta, \Delta_2$ if and only if $\exists \Delta_3$ such that $\Delta_2 = \Delta_1 \otimes \Delta_3$ (note that the concatenation Δ_2 has a single trailing term *fail*);

$$\text{(TFACT)}\quad \frac{}{\Gamma;\, K \vdash F \,:\, fact_{\{F\}} \triangleright \epsilon;\, \bot} \qquad \text{(TPAR)}\quad \frac{K(\tilde{x}) = (\tau,\, \Delta)}{\Gamma;\, K \vdash \tilde{x} \,:\, \tau \triangleright \Delta;\, \bot} \qquad \text{(TSUB)}\quad \frac{\Gamma;\, K \vdash e \,:\, \tau' \triangleright H'; \Lambda' \quad \tau' \leq \tau \quad H' \sqsubseteq H \quad \Lambda' \sqsubseteq \Lambda}{\Gamma;\, K \vdash e \,:\, \tau \triangleright H; \Lambda}$$

$$\text{(TIF)}\quad \frac{\Gamma;\, K \vdash e_1 : int \triangleright H_1;\, \Lambda \qquad \Gamma;\, K \vdash e_2 : \tau \triangleright H_2;\, \Lambda \qquad \Gamma;\, K \vdash e_3 : \tau \triangleright H_3;\, \Lambda}{\Gamma;\, K \vdash if\ e_1\ then\ e_2\ else\ e_3 : \tau \triangleright H_1 \cdot (H_2 + H_3); \Lambda}$$

$$\text{(TLET)}\quad \frac{\Gamma;\, K \vdash e_1 \,:\, \tau_1 \triangleright H_1;\, \Lambda_1 \qquad \Gamma,\, x : \tau_1;\, K \vdash e_2 : \tau_2 \triangleright H_2;\, \Lambda_2}{\Gamma;\, K \vdash let\ x = e_1\ in\ e_2 : \tau_2 \triangleright H_1 \cdot H_2;\, \Lambda_1 \uplus \Lambda_2}$$

$$\text{(TTELL)}\quad \frac{\Gamma;\, K \vdash e \,:\, fact_\phi \triangleright H;\, \Lambda}{\Gamma;\, K \vdash tell(e)^l \,:\, unit \triangleright \left(H \cdot \left(\sum_{F_i \in \phi} tell\, F_i^{l_i}\right)\right)^{l'} ;\, \Lambda \biguplus_{F_i \in \phi} [l_i \mapsto l]}$$

$$\text{(TRETRACT)}\quad \frac{\Gamma;\, K \vdash e \,:\, fact_\phi \triangleright H;\, \Lambda}{\Gamma;\, K \vdash retract(e)^l \,:\, unit \triangleright \left(H \cdot \left(\sum_{F_i \in \phi} retract\, F_i^{l_i}\right)\right)^{l'} ;\, \Lambda \biguplus_{F_i \in \phi} [l_i \mapsto l]}$$

$$\text{(TVARIATION)}\quad \frac{\begin{array}{c} \forall i \in \{1, \ldots, n\} \qquad \gamma(G_i) = \overrightarrow{y_i} \,:\, \overrightarrow{\tau_i} \\ \Gamma, x : \tau_1, \overrightarrow{y_i} \,:\, \overrightarrow{\tau_i}; K' \vdash e_i \,:\, \tau_2 \triangleright H_i;, \Lambda_i \qquad \Delta = ask\, G_1.H_1 \otimes \cdots \otimes ask\, G_n.H_n \otimes fail \end{array}}{\Gamma;\, K \vdash (x)\{G_1.e_1, \ldots, G_n.e_n\} \,:\, \tau_1 \xrightarrow{K'|\Delta} \tau_2 \triangleright \epsilon;\, \biguplus_{i \in \{1,\ldots,n\}} \Lambda_i}$$

$$\text{(TVAPP)}\quad \frac{\Gamma;\, K \vdash e_1 \,:\, \tau_1 \xrightarrow{K'|\Delta} \tau_2 \triangleright H_1;\, \Lambda_1 \qquad \Gamma;\, K \vdash e_2 \,:\, \tau_1 \triangleright H_2;\, \Lambda_2 \qquad K' \sqsubseteq K}{\Gamma;\, K \vdash \#(e_1, e_2) \,:\, \tau_2 \triangleright H_1 \cdot H_2 \cdot \Delta;\, \Lambda_1 \uplus \Lambda_2}$$

$$\text{(TAPPEND)}\quad \frac{\Gamma;\, K \vdash e_1 \,:\, \tau_1 \xrightarrow{K'|\Delta_1} \tau_2 \triangleright H_1;\, \Lambda_1 \qquad \Gamma;\, K \vdash e_2 \,:\, \tau_1 \xrightarrow{K'|\Delta_2} \tau_2 \triangleright H_2;\, \Lambda_2}{\Gamma;\, K \vdash e_1 \cup e_2 \,:\, \tau_1 \xrightarrow{K'|\Delta_1 \otimes \Delta_2} \tau_2 \triangleright H_1 \cdot H_2;\, \Lambda_1 \uplus \Lambda_2}$$

$$\text{(TDLET)}\quad \frac{\Gamma, \overrightarrow{y} \,:\, \overrightarrow{\tau};\, K \vdash e_1 \,:\, \tau_1 \triangleright H_1;\, \Lambda_1 \qquad \Gamma;\, K, (\tilde{x},\, \tau_1,\, \Delta') \vdash e_2 \,:\, \tau \triangleright H;\, \Lambda_2}{\Gamma;\, K \vdash dlet\, \tilde{x} \,=\, e_1\, when\, G\, in\, e_2 \,:\, \tau \triangleright H;\, \Lambda_1 \uplus \Lambda_2}$$

where $\gamma(G) = \overrightarrow{y} \,:\, \overrightarrow{\tau}$
if $K(\tilde{x}) = (\tau_1,\, \Delta)$ then $\Delta' = G.H_1 \otimes \Delta$
else (if $\tilde{x} \notin K$ then $\Delta' = G.H_1 \otimes fail$)

Fig. 4. Typing rules for new constructs

- $K_1 \sqsubseteq_K K_2$ if and only if ($(\tilde{x}, \tau_1, \Delta_1) \in K_1$ implies $(\tilde{x}, \tau_2, \Delta_2) \in K_2$ and $\tau_1 \leq \tau_2 \wedge \Delta_1 \sqsubseteq_\Delta \Delta_2$);
- $\Lambda_1 \sqsubseteq_\Lambda \Lambda_2$ if and only if $\exists \Lambda_3$ such that $\Lambda_2 = \Lambda_1 \uplus \Lambda_3$.

Most of the rules of our type and effect system are inherited from those of ML, and those for the new constructs are in Fig. 4, together with some other which are relevant. A few comments are in order.

Subtyping and subeffecting We have rules for subtyping and sub-effecting (displayed Fig. 4, top). As expected, these rules say that subtyping relation is reflexive (rule (SREFL)); a type $fact_\phi$ is a subtype of a type $fact_{\phi'}$ whenever $\phi \subseteq \phi'$ (rule (SFACT)); functional types are contravariant in the types of arguments and covariant in the result type and in the annotations (rule (SFUN)); analogously for behavioural variations types (rule (SVA)). Also, we can add elements to Λ, provided that there is no clash.

Type and effect of expressions The rule (TSUB) allows us to freely enlarge types and effects by applying the subtyping and subeffecting rules. Rule (TPAR) looks for the type and the effect of the parameter $\tilde{x}$ in the environment K. We determine the type for each subexpression e_i under K' and the environment Γ extended by the type of x and of the variables $\overrightarrow{y_i}$ occurring in the goal G_i (recall that the Datalog typing function γ returns a list of pairs (z, type-of-z) for all variable z of G_i). Note that all subexpressions e_i have the same type τ_2. We also require that the abstract variation Δ results from concatenating $ask\, G_i$ with the effect computed for e_i. The type of the behavioural variation is annotated by K' and Δ.

Consider, e.g., the behavioural variation $bv_1 = (x)\{G_1.e_1, G_2.e_2\}$. Assume that the two cases of this behavioural variation have type τ and effects H_1 and H_2, respectively, under the environment $\Gamma, x : int$ (goals have no variables), and that the guessed environment is K'. Hence, the type of bv_1 will be $int \xrightarrow{K'|\Delta} \tau$ with $\Delta = ask\, G_1.H_1 \otimes ask\, G_2.H_2 \otimes fail$ and the effect will be empty.

The rule (TVAPP) type-checks behavioural variation applications and reveals the role of preconditions. As expected, e_1 is a behavioural variation with parameter of type τ_1 and e_2 has type τ_1. We get a type if the environment K', which acts as a precondition, is included in K according to $\sqsubseteq$. The type of the behavioural variation application is τ_2, i.e., the type of the result of e_1, and the effect is obtained by concatenating the ones of e_1 and e_2 with the history expression Δ, occurring in the annotation of the type of e_1. Consider, e.g., bv_1 above, its type and its empty effect ϵ. Assume to type-check $e = \#(bv_1, 10)$ in the environments Γ and K. If $K' \sqsubseteq K$, the type of e is τ and its effect is $\epsilon \cdot \Delta = ask\, G_1.H_1 \otimes ask\, G_2.H_2 \otimes fail$.

The rule (TAPPEND) asserts that two expressions e_1,e_2 with the same type τ, except for the abstract variations Δ_1, Δ_2 in their annotations, and effects H_1 and H_2, are combined into $e_1 \cup e_2$ with type τ, and concatenated annotations and effects. More precisely, the resulting annotation has the same precondition of e_1 and e_2 and abstract variation $\Delta_1 \otimes \Delta_2$, and effect $H_1 \cdot H_2$. Consider, e.g., again

l	1 2 3 4 5 6 7 8 9
$\Lambda(l)$	1 2 ⊥ 3 4 ⊥ ⊥ 5 ⊥

l	1 2 3 4 5 6 7 8 9
$\Lambda'(l)$	1 1 ⊥ 2 3 ⊥ ⊥ ⊥ ⊥

Fig. 5. The labelling environments Λ for the expression e_a and its history expression H_a of Section 2; and a non-injective environment Λ'.

the above bv_1, its type $int \xrightarrow{K'|\Delta} \tau$; let $bv_2 = (w)\{G_3.c_2\}$, and let its type be $int \xrightarrow{K'|\Delta'} \tau$ and its effect be H_2. Then the type of $bv_1 \cup bv_2$ is $int \xrightarrow{K'|\Delta\otimes\Delta'} \tau$, while the effect is H_2.

The rule (TDLET) requires that e_1 has type τ_1 in the environment Γ extended with the types for the variables $\overrightarrow{y}$ of the goal G. Also, e_2 has to type-check in an environment K, extended with the information for parameter $\tilde{x}$. The type and the effect for the overall *dlet* expression are the same of e_2.

Handling the labelling environment The labelling environment generated by the rules (TFACT) and (TPAR) is ⊥, because there is no *tell* or *retract*. Instead (TTELL) updates the current environment Λ by associating all the labels of the facts which e can evaluate to, with the label l of the $tell(e)$ being typed; similarly for (TRETRACT).

All the other rules behave inductively as briefly discussed below.

The rule (TLET) produces an environment Λ that contains all the correspondences of Λ_1 and Λ_2 coming from e_1 ed e_2; note that unicity of the labelling is guaranteed by the condition $dom(\Lambda_1) \cap dom(\Lambda_2) = \emptyset$. As an example, Fig. 5 (left part) shows the correspondence of the labels in the expression e_a and those of its history expression H_a of Section 2.

Note that a labelling environment needs not to be injective. Consider, e.g., the ambient Λ' in Fig. 5 (right part), computed for

$$e'_a = \texttt{let x = tell(if y then F}_1\texttt{ else F}_2\texttt{)}^1\texttt{ in ask F}_5\texttt{.retract F}_8^2\texttt{, ask F}_3\texttt{.retract F}_4^3$$

and for its history expression

$$H'_a = ((tell\, F_1^1 + tell\, F_2^2)^3 \cdot (ask\, F_5.retract\, F_8^4 \otimes (ask\, F_3.retract\, F_4^5 \otimes fail^6)^7)^8)^9$$

Soundness Our type and effect system is sound with respect to the operational semantics. It is convenient to introduce the following technical definitions.

Definition 2 (Typing dynamic environment). *Given the type environments Γ and K, we say that the dynamic environment ρ has type K under Γ (in symbols $\Gamma \vdash \rho : K$) iff $dom(\rho) \subseteq dom(K)$ and $\forall \tilde{x} \in dom(\rho)$. $\rho(x) = G_1.e_1, \ldots, G_n.e_n$ $K(\tilde{x}) = (\tau, \Delta)$ and $\forall i \in \{1, \ldots, n\}$. $\gamma(G_i) = \overrightarrow{y_i} : \overrightarrow{\tau_i}$ $\Gamma, \overrightarrow{y_i} : \overrightarrow{\tau_i}; K_{\tilde{x}} \vdash e_i : \tau' \triangleright H_i$ and $\tau' \leq \tau$ and $\bigotimes_{i\in\{1,\ldots,n\}} G_i.H_i \sqsubseteq \Delta$.*

Definition 3. *Given H_1, H_2 then $H_1 \preccurlyeq H_2$ iff one of the following cases holds*

(a) $H_1 \sqsubseteq H_2$; *(b)* $H_2 = H_3 \cdot H_1$ *for some* H_3;

(c) $H_2 = \bigotimes_{i \in \{1,\dots,n\}} ask\, G_i.H_i \otimes fail \;\wedge\; H_1 = H_i,\; i \in [1..n]$.

Intuitively, the above definition formalises the fact that the history expression H_1 could be obtained from H_2 by evaluation.

The soundness of our type and effect system easily derives from the following standard results.

Theorem 1 (Preservation). *Let e_s be a closed expression; and let ρ be a dynamic environment such that $dom(\rho)$ includes the set of parameters of e_s and such that $\Gamma \vdash \rho : K$.*
If $\Gamma; K \vdash e_s : \tau \triangleright H_s; \Lambda_s$ and $\rho \vdash C,\, e_s \to C',\, e'_s$ then $\Gamma; K \vdash e'_s : \tau \triangleright H'_s; \Lambda'_s$ and there exist H'_s, $\overline{H}$, such that $\overline{H} \cdot H'_s \preccurlyeq H_s$ and $C, \overline{H} \cdot H'_s \to^{+} C',\, H'_s$ and $\Lambda'_s \sqsubseteq \Lambda_s$.

The Progress Theorem assumes that the effect H does not reach *fail*, i.e., that the dispatching mechanism succeeds at runtime. We take care of ensuring this property in Section 5 (we write $\rho \vdash C,\, e \nrightarrow$ to intend that there exists no transition outgoing from C, e).

Theorem 2 (Progress).
Let e_s be a closed expression such that $\Gamma; K \vdash e_s : \tau \triangleright H_s; \Lambda_s$; and let ρ be a dynamic environment such that $dom(\rho)$ includes the set of parameters of e_s, and such that $\Gamma \vdash \rho : K$.
If $\rho \vdash C,\, e_s \nrightarrow \;\wedge\; C,\, H_s \nrightarrow^{+} C',\, fail$ then e_s is a value.

The following corollary ensures that the history expression obtained as an effect of e over-approximates the actions that may be performed over the context during the evaluation of e.

Corollary 1 (Over-approximation). *Let e be a closed expression.*
If $\Gamma; K \vdash e : \tau \triangleright H; \Lambda_s \;\wedge\; \rho \vdash C,\, e \to^{\star} C',\, e'$, for some ρ such that $\Gamma \vdash \rho : K$, then there exists a sequence of transitions $C,\, H \to^{\star} C',\, H'$, for some H'.

Note that the type of e' is the same of e, because of Theorem 1, and the obtained label environment is included in Λ_s.

5 Loading-time Analysis

Our execution model for $\mathrm{ML_{CoDa}}$extends that of [13]: the compiler produces a quadruple $(C_p,\, e_p,\, H_p,\, \Lambda_p)$ composed by the application context, the object code, the history expression over-approximating the behaviour of e_p and the labelling environment associating labels of H_p with those in the code. Given $(C_p,\, e_p,\, H_p,\, \Lambda_p)$, at loading time, the virtual machine performs:

- a *linking* phase, in which the virtual machine of $\mathrm{ML_{CoDa}}$ resolves system variables and constructs the initial context C (combining C_p and the system context); and

– a *verification* phase, in which, a graph $\mathcal{G}$ describing the possible evolutions of C is built, starting from H_p.

We exploit $\mathcal{G}$ in order to (i) verify whether applications adapt to all evolutions of C, i.e., that all dispatching invocations will always succeed (only programs which pass this verification phase will be run), as done in [13]; and (ii) detect which `tell`/`retract` may lead to a violation of the system policy (see Section 6).

Technically, we compute $\mathcal{G}$ through a static analysis, specified in terms of Flow Logic [22]. Below, we describe the specification of our analysis, and we introduce the notion of viable history expressions. Intuitively, a history expression is viable for an initial context if the dispatching mechanism always succeeds.

To support the formal development, we assume that all bound variables occurring in a history expression are distinct. So we can define a function $\mathbb{K}$ mapping a variable h^l to the history expression $(\mu h.H_1^{l_1})^{l_2}$ that introduces it.

Analysis The static approximation is represented by a pair $(\Sigma_\circ, \Sigma_\bullet)$, called *estimate* for H, with $\Sigma_\circ, \Sigma_\bullet \colon Lab \to \wp(Context \cup \{\bullet\})$ and where $\bullet$ is the distinguished "failure" context representing a dispatching failure. For each label l,

– the set $\Sigma_\circ(l)$ over-approximates the set of contexts that may arise before evaluating H^l (call it *pre-set*); while
– $\Sigma_\bullet(l)$ over-approximates the set of contexts that may result from the evaluation of H^l (call it *post-set*).

The analysis is specified in terms of a set of clauses that operate upon judgments in the form $(\Sigma_\circ, \Sigma_\bullet) \vDash H^l$, where

$$\vDash \subseteq \mathcal{AE} \times \mathbb{H}$$

and $\mathcal{AE} = (Lab \to \wp(Context \cup \{\bullet\}))^2$ is the domain of the results of the analysis and $\mathbb{H}$ the set of history expressions. The judgment $(\Sigma_\circ, \Sigma_\bullet) \vDash H^l$, expresses that $\Sigma_\circ$ and $\Sigma_\bullet$ is an acceptable analysis estimate for the history expression H^l.

The notion of acceptability will then be used in Definition 5 to check whether the history expression H_p, hence the expression e it is an abstraction of, will never fail in a given initial context C.

In Fig. 6 we give the set of inference rules that validate the correctness of a given estimate. Now, we comment on them, where $\mathcal{E}$ denotes the estimate $(\Sigma_\circ, \Sigma_\bullet)$.

Intuitively, the estimate components take into account the possible dynamics of the language evaluation. The checks in the clauses mimic the semantic evolution of contexts, by modelling the semantic preconditions and the consequences of the possible reductions.

In the rule (Atell) the analysis checks whether the context C is in the preset, and the context $C \cup \{F\}$ is in the post-set; similarly for(Aretract), where $C \backslash \{F\}$ shold be in the post-set.

The rule (Anil) says that every pair of functions is an acceptable estimate for the "semantic" empty history expression $\ni$. The estimate $\mathcal{E}$ is acceptable for the "syntactic" ϵ^l if the pre-set is included in the post-set (rule (Aeps)).

The rules (ASEQ1) and (ASEQ2) handle the sequential composition of history expressions. The rule (ASEQ1) states that $(\Sigma_\circ, \Sigma_\bullet)$ is acceptable for $H = (H_1^{l_1} \cdot H_2^{l_2})^l$ if it is valid for both H_1 and H_2. Moreover, the pre-set of H_1 must include that of H and the pre-set of H_2 includes the post-set of H_1; finally, the post-set of H includes that of H_2. The rule (ASEQ2) states that $\mathcal{E}$ is acceptable for $H = (\ni \cdot H_1^{l_2})^l$ if it is acceptable for H_1 and the pre-set of H_1 includes that of H, while the post-set of H includes that of H_1.

By the rule (ASUM), $\mathcal{E}$ is acceptable for $H = (H_1^{l_1} + H_2^{l_2})^l$ if it is valid for H_1 and H_2; the pre-set of H is included in the pre-sets of H_1 and H_2; and the post-set of H includes those of H_1 and H_2.

The rules (AASK1) and (AASK2) handle the abstract dispatching mechanism. The first states that the estimate $\mathcal{E}$ is acceptable for $H = (askG.H_1^{l_1} \otimes \Delta^{l_2})^l$, provided that, for all C in the pre-set of H, if the goal G succeeds in C then the pre-set of H_1 includes that of H and the post-set of H includes that of H_1. Otherwise, the pre-set of Δ^{l_2} must include the one of H and the post-set of Δ^{l_2} is included in that of H. The rule (AASK2) requires $\bullet$ to be in the post-set of $fail$.

By the rule (AREC) $\mathcal{E}$ is acceptable for $H = (\mu h.H_1^{l_1})^l$ if it is acceptable for $H_1^{l_1}$ and the pre-set of H_1 includes that of H and the post-set of H includes that of H_1.

The rule (AVAR) says that a pair $(\Sigma_\circ, \Sigma_\bullet)$ is an acceptable estimate for a variable h^l if the pre-set of the history expression introducing h, namely $\mathbb{K}(h)$, is included in that of h^l, and the post-set of h^l includes that of $\mathbb{K}(h)$.

We are now ready to introduce when an estimate for a history expression is valid for an initial context.

Definition 4 (Valid analysis estimate). *Given $H_p^{l_p}$ and an initial context C, we say that a pair $(\Sigma_\circ, \Sigma_\bullet)$ is a* valid analysis estimate *for H_p and C iff $C \in \Sigma_\circ(l_p)$ and $(\Sigma_\circ, \Sigma_\bullet) \models H_p^{l_p}$.*

Semantic properties The following theorems state the correctness of our approach. The first guarantees that there exists a minimal valid analysis estimate, showing that the set of acceptable analyses forms a Moore family [22].

Theorem 3 (Existence of solutions). *Given H^l and an initial context C, the set $\{(\Sigma_\circ, \Sigma_\bullet) \mid (\Sigma_\circ, \Sigma_\bullet) \models H^l\}$ of the acceptable estimates of the analysis for H^l and C is a Moore family; hence, there exists a minimal valid estimate.*

As expected, we have a standard subject reduction theorem, saying that the information recorded by a valid estimate is correct with respect to the operational semantics of history expressions.

Theorem 4 (Subject Reduction). *Let H^l be a closed history expression such that $(\Sigma_\circ, \Sigma_\bullet) \models H^l$.*
If for all $C \in \Sigma_\circ(l)$ it is $C, H^l \to C', H'^{l'}$ then $(\Sigma_\circ, \Sigma_\bullet) \models H'^{l'}$ and $\Sigma_\circ(l) \subseteq \Sigma_\circ(l')$ and $\Sigma_\bullet(l') \subseteq \Sigma_\bullet(l)$.

(ANIL)
$$\frac{}{(\Sigma_\circ, \Sigma_\bullet) \models \backepsilon}$$

(AEPS)
$$\frac{\Sigma_\circ(l) \subseteq \Sigma_\bullet(l)}{(\Sigma_\circ, \Sigma_\bullet) \models \epsilon^l}$$

(ATELL)
$$\frac{\forall C \in \Sigma_\circ(l) \qquad C \cup \{F\} \in \Sigma_\bullet(l)}{(\Sigma_\circ, \Sigma_\bullet) \models \mathit{tell}\, F^l}$$

(ARETRACT)
$$\frac{\forall C \in \Sigma_\circ(l) \qquad C \backslash \{F\} \in \Sigma_\bullet(l)}{(\Sigma_\circ, \Sigma_\bullet) \models \mathit{retract}\, F^l}$$

(ASEQ1)
$$\frac{\begin{array}{c}(\Sigma_\circ, \Sigma_\bullet) \models H_1^{l_1} \\ (\Sigma_\circ, \Sigma_\bullet) \models H_2^{l_2} \qquad \Sigma_\circ(l) \subseteq \Sigma_\circ(l_1) \qquad \Sigma_\bullet(l_1) \subseteq \Sigma_\circ(l_2) \qquad \Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l)\end{array}}{(\Sigma_\circ, \Sigma_\bullet) \models (H_1^{l_1} \cdot H_2^{l_2})^l}$$

(ASEQ2)
$$\frac{(\Sigma_\circ, \Sigma_\bullet) \models H_2^{l_2} \qquad \Sigma_\circ(l) \subseteq \Sigma_\circ(l_2) \qquad \Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l)}{(\Sigma_\circ, \Sigma_\bullet) \models (\backepsilon \cdot H_2^{l_2})^l}$$

(ASUM)
$$\frac{\begin{array}{c}(\Sigma_\circ, \Sigma_\bullet) \models H_1^{l_1} \qquad \Sigma_\circ(l) \subseteq \Sigma_\circ(l_1) \qquad \Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l) \\ (\Sigma_\circ, \Sigma_\bullet) \models H_2^{l_2} \qquad \Sigma_\circ(l) \subseteq \Sigma_\circ(l_2) \qquad \Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l)\end{array}}{(\Sigma_\circ, \Sigma_\bullet) \models (H_1^{l_1} + H_2^{l_2})^l}$$

(AASK1)
$$\frac{\begin{array}{ll}\forall C \in \Sigma_\circ(l) & (C \models G \implies (\Sigma_\circ, \Sigma_\bullet) \models H^{l_1} \qquad \Sigma_\circ(l) \subseteq \Sigma_\circ(l_1) \qquad \Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)) \\ & (C \not\models G \implies (\Sigma_\circ, \Sigma_\bullet) \models \Delta^{l_2} \qquad \Sigma_\circ(l) \subseteq \Sigma_\circ(l_2) \qquad \Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l))\end{array}}{(\Sigma_\circ, \Sigma_\bullet) \models (askG.H^{l_1} \otimes \Delta^{l_2})^l}$$

(AASK2)
$$\frac{\bullet \in \Sigma_\bullet(l)}{(\Sigma_\circ, \Sigma_\bullet) \models \mathit{fail}^l}$$

(AREC)
$$\frac{(\Sigma_\circ, \Sigma_\bullet) \models H^{l_1} \qquad \Sigma_\circ(l) \subseteq \Sigma_\circ(l_1) \qquad \Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)}{(\Sigma_\circ, \Sigma_\bullet) \models (\mu h.H^{l_1})^l}$$

(AVAR)
$$\frac{\mathbb{K}(h) = (\mu h.H^{l_1})^{l'} \qquad \Sigma_\circ(l) \subseteq \Sigma_\circ(l') \qquad \Sigma_\bullet(l') \subseteq \Sigma_\bullet(l)}{(\Sigma_\circ, \Sigma_\bullet) \models h^l}$$

Fig. 6. Specification of the analysis for History Expressions

	$\Sigma^1_\circ$	$\Sigma^1_\bullet$
1	$\{\{F_2, F_5, F_8\}\}$	$\{\{F_1, F_2, F_5, F_8\}\}$
2	$\{\{F_1, F_2, F_5, F_8\}\}$	$\{\{F_1, F_5, F_8\}\}$
3	$\{\{F_2, F_5, F_8\}\}$	$\{\{F_1, F_5, F_8\}\}$
4	$\{\{F_2, F_5, F_8\}\}$	$\{\{F_2, F_5\}\}$
5	$\{\{F_2, F_5, F_8\}\}$	$\{\{F_2, F_5\}\}$
6	$\emptyset$	$\emptyset$
7	$\emptyset$	$\{\bullet\}$
8	$\emptyset$	$\emptyset$
9	$\{\{F_2, F_5, F_8\}\}$	$\{\{F_1, F_5, F_8\},\{F_2, F_5\}\}$

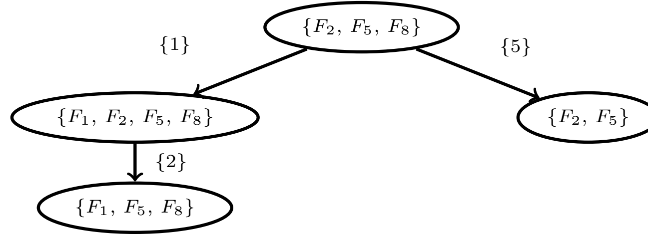

Fig. 7. The analysis result (on top) and the evolution graph (on bottom) for the context $C = \{F_2, F_5, F_8\}$ and the history expression $H_p = ((tell\, F_1^1 \cdot retract\, F_2^2)^3 + (ask\, F_5.retract\, F_8^5 \otimes ask\, F_3.retract\, F_4^6 \otimes fail^7)^4)^8$.

Viability of history expressions We now define when a history expression H_p is viable for an initial context C, i.e., when it passes the verification phase. Below, let $lfail(H)$ be the set of labels of the *fail* sub-terms in H:

Definition 5 (Viability). *Let H_p be a history expression and C be an initial context. We say that H_p is* viable *for C if there exists the minimal valid analysis estimate $(\Sigma_\circ, \Sigma_\bullet)$ such that $\forall l \in dom(\Sigma_\bullet)\setminus lfail(H_P)$ it is $\bullet \notin \Sigma_\bullet(l)$.*

We present now a couple of examples to illustrate how viability is checked. Since the focus here is on the technical details of the analysis of behavioural variations, we resort to *ad-hoc* examples. Consider the history expression

$$H_p = ((tell\, F_1^1 \cdot retract\, F_2^2)^3 + (ask\, F_5.retract\, F_8^5 \otimes (ask\, F_3.retract\, F_4^6 \otimes fail^7)^8)^4)^9$$

and the initial context $C = \{F_2, F_5, F_8\}$, consisting of facts only. For each label l occurring in H_p, Fig. 7 shows the corresponding values of $\Sigma^1_\circ(l)$ and $\Sigma^1_\bullet(l)$, respectively. We can observe, e.g., that the pre-set for the *tell* labelled with 1 includes $\{F_2, F_5, F_8\}$, while the post-set includes $\{F_1, F_2, F_5, F_8\}$, while the pre-set for the *remove* labelled with 5 includes $\{F_2, F_5, F_8\}$, while the post-set includes $\{F_2, F_5\}$. The column describing $\Sigma_\bullet$ contains $\bullet$ only for $l = 7$ which is the label of *fail*, so H_p is viable for C.

	$\Sigma_\circ^2$	$\Sigma_\bullet^2$
1	$\{\{F_2,\, F_5,\, F_8\}\}$	$\{\{F_1,\, F_2,\, F_5,\, F_8\}\}$
2	$\{\{F_1,\, F_2,\, F_5,\, F_8\}\}$	$\{\{F_1,\, F_5,\, F_8\}\}$
3	$\{\{F_2,\, F_5,\, F_8\}\}$	$\{\{F_1,\, F_5,\, F_8\}\}$
4	$\{\{F_2,\, F_5,\, F_8\}\}$	$\{\bullet\}$
5	$\emptyset$	$\emptyset$
6	$\{\{F_2,\, F_5,\, F_8\}\}$	$\{\bullet\}$
7	$\{\{F_2,\, F_5,\, F_8\}\}$	$\{\{F_1,\, F_5,\, F_8\}, \bullet\}$

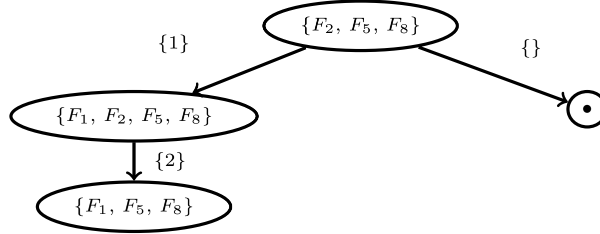

Fig. 8. The analysis result (on top) and the evolution graph (on bottom) for the context $C = \{F_2,\, F_5,\, F_8\}$ and the history expression $H'_p = ((tell\, F_1^1 \cdot retract\, F_2^2)^3 + (ask\, F_3.retract\, F_4^5 \otimes fail^6)^4)^7$

Now consider the following history expression that fails to pass the verification phase, when put in the same initial context C used above:

$$H'_p = ((tell\, F_1^1 \cdot retract\, F_2^2)^3 + (ask\, F_3.retract\, F_4^5 \otimes fail^6)^4)^7$$

Indeed H'_p is not viable, because the goal F_3 does not hold in C, and this is reflected by the occurrences of $\bullet$ in $\Sigma_\bullet^2(4)$ and $\Sigma_\bullet^2(7)$, as shown in Fig. 8.

Now we exploit the result of the above analysis to build up the evolution graph $\mathcal{G}$. The graph describes how the initial context C evolves at runtime, paving our way to security enforcement. Intuitively, $\mathcal{G}$ is a direct graph, whose nodes are sets of contexts, and where an arc between two nodes C_1 and C_2 records that C_2 is obtained from C_1, through telling or removing a fact F.

In the following let $Fact^*$ and Lab_H^* be the set of facts and the set of labels occurring in H_p, i.e., the history expression under verification.

Definition 6 (Evolution Graph). *Let H_p be a history expression, C be an initial context, and $(\Sigma_\circ, \Sigma_\bullet)$ be a valid analysis estimate.*
The evolution graph of C is $\mathcal{G} = (N, E, L)$, where

$$N = \bigcup_{l \in Lab_H^*} (\Sigma_\circ(l) \cup \Sigma_\bullet(l))$$

$$E = \{(C_1, C_2) \mid \exists F \in Fact^*,\ l \in Lab_H^* \ s.t.\ C_1 \in \Sigma_\circ(l) \wedge C_2 \in \Sigma_\bullet(l) \wedge (h(l) \in \{tell(F),\, retract(F)\} \vee (C_2 = \bullet))\}$$

$$L \; : \; E \to \mathcal{P}(Labels)$$
$$\forall t = (C_1, C_2) \in E, \; l \in L(t) \; \textit{iff} \; C_1 \in \Sigma_\circ(l) \wedge C_2 \in \Sigma_\bullet(l) \wedge h(l) \neq \; fail$$

As examples of evolution graph, consider the context C and the history expressions H_p and H'_p introduced in the examples above. The evolution graph of C for H_p is in Fig. 7. From the initial context there is an arc with label 1 to the context $C \cup F_1$, because of the $tell^1$. There is also an arc labelled 5 to the context without F_8, because of $retract^5$.

Clearly, the evolution graph $\mathcal{G}$ tell us when the dispatching mechanism always succeeds: it is sufficient to verify that the failure context $\bullet$ is not reachable from the initial context C. Back to our examples, it is easy to see that H_p is viable for C, because the node $\bullet$ is not reachable from C in the graph for H_p. Instead, H'_p is not viable, because $\bullet$ is reachable in the evolution graph for H'_p, displayed in Fig. 8.

Note that labels of $\mathcal{G}$ indicate which *tell*/*retract* may lead to a context violating the security policy Φ. To enforce Φ, we will exploit the correspondence between these labels and the labels in the code.

6 Code instrumentation

We preliminarily detect which are the potential risky operations the application can perform through a static analysis of the evolution graph $\mathcal{G}$. The occurrence of these risky actions will then be guarded by our *runtime monitor*; on the others the monitor will be switched off.

We proceed as follows. First, since a node n of $\mathcal{G}$ represents a context that the application may reach during its execution, we verify whether n satisfies Φ. If this is not the case, we consider all the edges with target n and the set $R = \{l_i\}$ of their labels. The labelling environment Λ, computed while type checking the application, determines those portions of the code that require to be monitored during the execution, indexed by the set $Risky = \Lambda(R)$.

The actual implementation of our runtime monitor and the way to switch it on and off requires, however, to consider all the `tell`/`retract` and to single out which operations are risky and which are not. To do that, the compiler (labels the source code as seen in Section 2 and) generates specific calls to *trampoline-like* procedures. More in detail, we will define below a procedure for verifying whether the policy Φ is satisfied or not, called `check_whether_policy_violation(l)`, that takes a label `l` as parameter and has *unit* as return type. Our compilation schema requires to replace every `tell(e)`^{l} in the source code with the following:

```
let z = tell(e) in
  check_whether_policy_violation(l)
```

where `z` is a fresh name; and to do in a similar way for every `retract`. Note, in passing, that we have a lightweight form of code instrumentation that does not operate on the object code, differently from the standard instrumentation.

At linking time, a global mask `risky[]` will be assigned for each label $\mathtt{l} \in Lab_C$, using the information stored in the graph $\mathcal{G}$ and in the set *Risky*, as follows:

$$\mathtt{risky[l]} = \begin{cases} \mathtt{true} & \text{if } \mathtt{l} \in \mathit{Risky} \\ \mathtt{false} & \text{otherwise} \end{cases}$$

Now we can specify the procedure `check_whether_policy_violation`. Intuitively, it looks at `risky[l]`: if the value is `false`, then the procedure returns to the caller and the execution goes on normally; otherwise it calls for a check on the policy Φ. Its code in a pseudo $\mathrm{ML}_{\mathrm{CoDa}}$ could be:

```
fun check_whether_policy_violation l =
  if risky[l] then
    ask phi.()
  else
    ()
```

Note that the call `ask phi.()` triggers a call to the dispatching mechanism to check the policy Φ: if this call fails then a policy violation has been observed and the computation is aborted. This is exactly what we require to a runtime monitor, i.e., to stop the application when a policy violation is about to occur.

It is easy to speed up the mechanism above, avoiding to invoke the procedure `check_whether_policy_violation` when *Risky* is empty, i.e., when the analysis of the evolution graph ensures that all occurrences of `tell`/`retract` are perfectly safe, because there is no execution path leading to a policy violation. To do that, we introduce the flag `always_ok`, whose value will be computed at linking time: if it turns out to be true, no check is needed. Then, we change the previously compilation schema by testing `always_ok` before calling our check procedure. All the occurrences $\mathtt{tell(e)}^l$ ($\mathtt{retract(e)}^l$, respectively) in the source code are now replaced by

```
let z = tell(e) in
  if not(always_ok) then
    check_whether_policy_violation(l)
```

In this way, the execution time is likely to be reduced, because some costly, and useless security checks are not performed

7 Conclusions

Following the Context-Oriented Programming paradigm, we considered the language for adaptive programming $\mathrm{ML}_{\mathrm{CoDa}}$ [14]. Here, we addressed security issues, by suitably extending the two-phase static analysis for $\mathrm{ML}_{\mathrm{CoDa}}$ [13]. Our methodology and our main contributions can be summarised as follows.

- We introduced $\mathrm{ML}_{\mathrm{CoDa}}$, a core of ML extended with COP features, coupled with Datalog for dealing with contexts. We showed here that the Datalog component of the language suffices for expressing and for enforcing context-dependent security policies.

- We presented a *type and effect system* for $\mathrm{ML}_{\mathrm{CoDa}}$ for ensuring that programs adequately respond to context changes, and for computing as effect an abstract representation of the overall behaviour. This representation, in the form of *history expressions*, abstractly describes the sequences of dynamic actions that a program may perform over the context. Our present extension also establishes a correspondence between the abstract actions in effects and the actual ones in the code, relevant to security.
- We further developed the approach introduced in [13], where the effects are exploited at loading time to verify that the application can adapt to all contexts possibly arising at runtime. More precisely, we built above a graph and a further *static analysis* that identifies the actions that may lead to contexts which violate the required *policy*.
- We defined a *runtime monitor* that stops an application when about to violate the policy to be enforced. The monitor exploits the link between the effects and the code. It is switched on and off, depending on the information collected by the static analysis mentioned above.

Future Work Still, our proposal is far from being definitive and many improvements are possible, especially on the security side. Our efforts are now addressed at implementing a *smarter* runtime monitor. For instance, our static analysis can detect *safe contexts*. Once reached a safe context, we are guaranteed that no policy violation will ever occur in the future. As a consequence, we could definitely turn off the runtime monitor, when the execution reaches one of those contexts.

Furthermore, we are thinking of providing the user with a kind of *recovery* mechanism for behavioural variations. The idea is to give the possibility to undo some risky actions and make different choices in some portions of the code, labelled by the user as particularly sensitive.

References

1. Achermann, F., Lumpe, M., Schneider, J., Nierstrasz, O.: PICCOLA—a small composition language. In: Formal methods for distributed processing. pp. 403–426. Cambridge University Press (2001)
2. Al-Neyadi, F., Abawajy, J.: Context-based e-health system access control mechanism. Advances in information security and its application pp. 68–77 (2009)
3. Appeltauer, M., Hirschfeld, R., Haupt, M., Masuhara, H.: ContextJ: Context-oriented programming with java. Computer Software 28(1) (2011)
4. Bartoletti, M., Degano, P., Ferrari, G.L., Zunino, R.: Local policies for resource usage analysis. ACM Trans. Program. Lang. Syst. 31(6) (2009)
5. Bonatti, P., De Capitani Di Vimercati, S., Samarati, P.: An algebra for composing access control policies. ACM Transactions on Information and System Security 5(1), 1–35 (2002)
6. Campbell, R., Al-Muhtadi, J., Naldurg, P., Sampemane, G., Mickunas, M.D.: Towards security and privacy for pervasive computing. In: Proc. of the 2002 Mext-NSF-JSPS international conference on Software security: theories and systems (ISSS'02). Lecture Notes in Computer Science, vol. 2609, pp. 1–15. Springer-Verlag (2003)

7. Cardelli, L., Gordon, A.D.: Mobile ambients. Theor. Comput. Sci. 240(1), 177–213 (2000)
8. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about datalog (and never dared to ask). IEEE Trans. on Knowl. and Data Eng. 1(1), 146–166 (1989)
9. Costanza, P.: Language constructs for context-oriented programming. In: Proc. of the Dynamic Languages Symposium. pp. 1–10. ACM Press (2005)
10. Deng, M., Cock, D.D., Preneel, B.: Towards a cross-context identity management framework in e-health. Online Information Review 33(3), 422–442 (2009)
11. DeTreville, J.: Binder, a Logic-Based Security Language. In: Proc. of the 2002 IEEE Symposium on Security and Privacy. pp. 105–113. SP '02, IEEE Computer Society, Washington, DC, USA (2002)
12. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive datalog. ACM Transactions on Database Systems 5(1), 1–35 (1997)
13. Galletta, L., Degano, P., Ferrari, G.L.: A staged static analysis for reliable adaptation, Submitted for publication - `http://www.cli.di.unipi.it/~galletta/staged_analysis.pdf`
14. Galletta, L., Degano, P., Ferrari, G.L.: A two-component language for context oriented programming, Submitted for publication - `http://www.cli.di.unipi.it/~galletta/mlcoda.pdf`
15. Heer, T., Garcia-Morchon, O., Hummen, R., Keoh, S., Kumar, S., Wehrle, K.: Security challenges in the IP-based internet of things. Wireless Personal Communications pp. 1–16 (2011)
16. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. Journal of Object Technology, March-April 2008 7(3), 125–151 (2008)
17. Hulsebosch, R., Salden, A., Bargh, M., Ebben, P., Reitsma, J.: Context sensitive access control. In: Proc. of the tenth ACM symposium on Access control models and technologies. pp. 111–119. ACM (2005)
18. Kamina, T., Aotani, T., Masuhara, H.: Eventcj: a context-oriented programming language with declarative event-based context transition. In: Proc. of the tenth international conference on Aspect-oriented software development (AOSD '11). pp. 253–264. ACM, New York, NY, USA (2011)
19. Li, N., Mitchell, J.C.: DATALOG with Constraints: A Foundation for Trust Management Languages. In: Proc. of the 5th International Symposium on Practical Aspects of Declarative Languages. pp. 58–73. PADL '03, Springer-Verlag, London, UK, UK (2003)
20. Loke, S.W.: Representing and reasoning with situations for context-aware pervasive computing: a logic programming perspective. Knowl. Eng. Rev. 19(3), 213–233 (2004)
21. Mycroft, A., O'Keefe, R.A.: A polymorphic type system for prolog. Artificial Intelligence 23(3), 295 – 307 (1984)
22. Nielson, H.R., Nielson, F.: Flow logic: a multi-paradigmatic approach to static analysis. In: Mogensen, T.A., Schmidt, D.A., Sudborough, I.H. (eds.) The essence of computation. Lecture Notes in Computer Science, vol. 2566, pp. 223–244. Springer-Verlag (2002)
23. Orsi, G., Tanca, L.: Context modelling and context-aware querying. In: Moor, O., Gottlob, G., Furche, T., Sellers, A. (eds.) Datalog Reloaded, Lecture Notes in Computer Science, vol. 6702, pp. 225–244. Springer (2011)
24. Pfleeger, C., Pfleeger, S.: Security in computing. Prentice Hall (2003)

25. Román, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R., Nahrstedt, K.: Gaia: a middleware platform for active spaces. ACM SIGMOBILE Mobile Computing and Communications Review 6(4), 65–67 (2002)
26. Wrona, K., Gomez, L.: Context-aware security and secure context-awareness in ubiquitous computing environments. In: XXI Autumn Meeting of Polish Information Processing Society (2005)
27. Zhang, G., Parashar, M.: Dynamic context-aware access control for grid applications. In: Proc. of Fourth International Workshop on Grid Computing, 2003. pp. 101–108. IEEE (2003)