Types for Coordinating Secure Behavioural Variations*

Pierpaolo Degano, Gian-Luigi Ferrari, Letterio Galletta, and Gianluca Mezzetti

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

Abstract. Context-Oriented programming languages provide us with primitive constructs to adapt program behaviour depending on the evolution of their operational environment. We are interested here in software components, the behaviour of which depend on the following: their actual operating context; the security policies that control accesses to their resources and the potential interactions with the external environment. For that, we extend a core functional language with mechanisms to program behavioural variations, to manipulate resources and to enforce security policies over both variations and resource usages. Additionally, there are message passing primitives to interact with the environment, also subject to a simple policy. Changes of the operational context are triggered both by the program and by the exchanged messages. Besides a definition of the dynamic semantics, we introduce a static analysis for guaranteeing programs to safely operate in any admissible context, and to correctly interact with the environment they comply with.

1 Introduction

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 some solutions have been proposed. We refer to [1,2,3] for a more comprehensive discussion. In this paper, we adopt a programming languages approach, that allows us to describe fine-grain adaptability mechanisms.

We consider the Context-Oriented Programming (COP)[4] paradigm, that extends standard programming languages with suitable constructs to express context-dependent behaviour in a modular fashion. The design and development of ubiquitous and autonomic systems would greatly benefit from such languages [5]. The fundamental concept in COP is that of *behavioural variation*. A behavioural variation is a chunk of behaviour that can be activated depending on the current working environment so to dynamically modify the execution. The current working environment is represented by the notion of *context*. The context is a stack of layers, i.e. properties identifying the actual structure of the environment. In this setting a programmer can (de)activate layers to represent

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

changes in the environment. This (de)activation mechanism is the engine of context evolution. Usually, behavioural variations are bound to layers: the (de)activation of a layer correspond to the (de)activation of a behavioural variation.

The development of complex adaptive systems presents issues that cannot be tackled only by COP primitives. Indeed, a complex adaptive system is made up of a massive number of interacting components. Each component is able to modify its behaviour, it can access a private set of resources and it has security constraints (security policies). Policies govern behaviour adaptation, access to resources, interaction with other components. A system behaves correctly when each component respects its own security policies and interacts with others by respecting the communication protocol.

We aim at contributing to the design of language-based methods and techniques that support the development of complex adaptive components. An adaptive component has (*i*) mechanisms to manipulate the context, (*ii*) security policies governing behaviour and resource usages, (*iii*) an abstract, declarative representation of the operational environment. We adopt a *top-down* approach [1] to describe the interactions with other components, because we do not want to wire a component to a specific communication infrastructure. Our communication model is based on a bus through which messages are exchanged.

The main contribution of this paper is the introduction of a method to program adaptive components. This proposal suitably extends and integrates together techniques from COP, type theory and model-checking. In particular, it consists of 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, we extend the COP functional language ContextML [6] with constructs for resource manipulation, following [7]. Also, our extension of ContextML has mechanisms to declare and enforce security policies by adopting the local sandbox approach of [7]. Finally, another novel feature is the introduction of message passing constructs for the communication with external parties (Section 3).

II Next, we design a type and effect system for ContextML (Section 5). 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 *History Expressions* (Section 4), describes the sequences of resource manipulation 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 6).

In Section 2 we introduce a motivating example, that is also instrumental in displaying our methodology at a glance.

2 A motivating example: an e-library app

Consider a simple scenario consisting of a smartphone app 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 app. 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 app 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

$$\textbf{with}(\texttt{getBatteryProfile()})\ \textbf{in}\ exp_1 \tag{1}$$

activates the layer obtained by calling `getBatteryProfile`. The scope of this activation is the expression exp_1 in Fig. 1(a). In lines 2-10, there is the following *layered expression*:

$$\begin{array}{l}\texttt{ACMode}.\ \langle \text{DO SEARCH} \rangle,\\ \texttt{BatMode}.\ \langle \text{DO SOMETHING ELSE} \rangle\end{array}$$

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 (Fig. 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 Fig. 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 $\varphi[\dots]$ 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 $\texttt{write}(\texttt{library})$ is forbidden (so only reading is allowed). The framing guarantees that the execution of foreign code does not alter the remote library. 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

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

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 $\tau' = \texttt{unit} \xrightarrow{\mathbb{P}|H} \tau''$, 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 H (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 (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 (1). This is because the type of `getBatteryProfile` guarantees that one among them will be activated in the context by the **with**.

An effect H (history expression) represents (an over-approximation of) the sequences of events, i.e. of resource manipulation or layer activations or communication actions.

```
1   fun g x =
2     ACMode.
3       IsCloud.search(),
4       IsLocal.
5         LibrarySynced.search(y),
6         LibraryUnsynced.
7           send_τ(ACMode);
8           send_τ'(g);
9           receive_τ''
10    BatMode. ⟨DO SOMETHING ELSE⟩
11  g()
```

(a) The definition of exp_1

```
1 fun f x =
2   let lyr = receive_τ in
3   let g = receive_τ' in
4   φ[with(lyr) in
5     let res = g() in
6     send_τ''(res)
7   ]; f()
8 f()
```

(b) The code for a service

Fig. 1. Fragments of an App and of a service in the cloud

The effect H 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 (a context-free language) against the policy φ (a regular language). 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 considering communications. Thus, the effect of exp_1 becomes:

$$H_{sr} = send_{\tau} \cdot send_{\tau'} \cdot receive_{\tau''}$$

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

3 ContextML: a context-oriented ML core

ContextML [6] is a fragment of ML designed 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. For simplicity, we do not provide ContextML with constructs for dynamically creating resources, but these can be added following [7,8].

We enforce security properties by protecting expressions with policies: $\varphi[e]$. This mechanism is known in the literature as *policy framing* [8]. 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; more details in Section 6.

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, Res a finite set of resources identifiers and Act a finite set of actions for manipulating resources. Then, the syntax of ContextML is:

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

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

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 the seminal paper [4]). 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 $\varphi[e]$ guaranteeing that the computation satisfies the policy φ. Of course, policy framings can be nested. The communication is performed by $\textbf{send}_\tau$ and $\textbf{receive}_\tau$. They allow us to interact with the external environment by writing/reading values of type τ (see Section 5) to/from the bus.

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, $\textbf{let}\, x = \textbf{receive}_\tau\, \textbf{in}\, x + 1$ is closed but it reads an unknown value v from the bus. To give meaning to such programs, we have an early input similar to that of the π-calculus [9].

Our semantics is history dependent. Program histories are sequences of events, namely *histories*, occurring during program execution. Events *ev* indicate (de)activation layers, selection of behavioural variations and program actions, be they resource accesses, entering/exiting policy framing and communication. The syntax of events *ev*

and programs histories η is the following:

$$ev ::= (\!|_L \mid |\!)_L \mid \{_L \mid \}_L \mid \mathsf{Disp}(L) \mid \alpha(r) \mid send_\tau \mid receive_\tau \mid [_\varphi \mid]_\varphi \quad (2)$$

$$\eta ::= \varepsilon \mid ev \mid \eta\,\eta \quad (3)$$

The event ($|\!)_L$) $(\!|_L$ marks that we (end) begin the evaluation of a **with** body in a context where the layer L is (de)activated; symmetrically, the event ($\}_L$) $\{_L$ signals that we (end) begin the evaluation of a **unwith** body in a context where the layer L is (un)masked; 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 $send_\tau$/$receive_\tau$ indicates that we have sent/read a value of type τ over/from the bus; the event ($]_\varphi$) $[_\varphi$ marks that we (end) begin the enforcement of the policy φ.

A context C is a stack of active layers with two operations. The first $C-L$ remove a layer L from the context C if present, the second $L :: C$ pushes L over $C-L$. Formally:

Definition 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*

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

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.

Most of semantic rules are inherited from ML. Fig. 2 shows the ones for new constructs. We briefly comment on them.

The rules for (**unwith**(e_1) **in** e_2) **with**(e_1) **in** e_2 evaluate e_2 in a context where the layer obtained evaluating e_1 is (de)activated. Additionally, we store in the history the events $(\!|_L$ and $|\!)_L$ ($\{_L$ 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 using $\overline{L}$).

When a layered expression $e = L_1.e_1,\dots,L_n.e_n$ 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

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

that returns the first layer in the context $[L'_0,L'_1,\dots,L'_m]$ 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

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

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

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

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

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

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

$$\text{unwith}_4\ \frac{}{C \vdash \eta, \mathbf{unwith}(\bar{L})\ \mathbf{in}\ v \to \eta \}_L, v}$$

$$\text{lexp}\ \frac{L_i = Dsp(C, \{L_1, \ldots, L_n\})}{C \vdash \eta, L_1.e_1, \ldots, L_n.e_n \to \eta\, \mathsf{Disp}(L_i), e_i} \qquad \text{action}\ \frac{}{C \vdash \eta, \alpha(r) \to \eta\, \alpha(r), ()}$$

$$\text{send}_1\ \frac{C \vdash \eta, e \to \eta', e'}{C \vdash \eta, \mathbf{send}_\tau(e) \to \eta', \mathbf{send}_\tau(e')} \qquad \text{send}_2\ \frac{}{C \vdash \eta, \mathbf{send}_\tau(v) \to \eta\ send_\tau, ()}$$

$$\text{receive}\ \frac{}{C \vdash \eta, \mathbf{receive}_\tau \to \eta\ receive_\tau, v} \qquad \text{framing}_1\ \frac{\eta^{-[]} \models \varphi}{C \vdash \eta, \varphi[e] \to \eta[_\varphi, \overline{\varphi}[e]}$$

$$\text{framing}_2\ \frac{C \vdash \eta, e \to \eta', e' \qquad \eta'^{-[]} \models \varphi}{C \vdash \eta, \overline{\varphi}[e] \to \eta', \overline{\varphi}[e']} \qquad \text{framing}_3\ \frac{\eta^{-[]} \models \varphi}{C \vdash \eta, \overline{\varphi}[v] \to \eta]_\varphi, v}$$

Fig. 2. Semantic rules for new constructs

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, $\mathbf{send}_\tau(e)$ evaluates e and sends the obtained value over the bus. Additionally, the history is extended with the event $send_\tau$. A $\mathbf{receive}_\tau$ 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 [9].

The rules for framing say that an expression $\varphi[e]$ can reduce to $\varphi[e']$, provided that the resulting history η' obeys the policy φ, in symbols $\eta'^{-[]} \models \varphi$ (see Section 4 and Section 6 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 $[_\varphi /]_\varphi$ the point where we begin/end the enforcement of φ.

$$\frac{}{\varepsilon \cdot H \xrightarrow{\varepsilon} H} \qquad \frac{}{\alpha(r) \xrightarrow{\alpha(r)} \varepsilon} \qquad \frac{}{\mu h.H \xrightarrow{\varepsilon} H\{\mu h.H/h\}}$$

$$\frac{H_1 \xrightarrow{\alpha(r)} H_1'}{H_1 \cdot H_2 \xrightarrow{\alpha(r)} H_1' \cdot H_2} \qquad \frac{H \xrightarrow{\alpha(r)} H'}{H_1 + H_2 \xrightarrow{\alpha(r)} H_1'} \qquad \frac{H_2 \xrightarrow{\alpha(r)} H_2'}{H_1 + H_2 \xrightarrow{\alpha(r)} H_2'}$$

Fig. 3. Transition system of History Expressions.

4 History Expressions

History Expressions [10,7,8] are a simple process algebra providing an abstraction over the set of histories that a program may generate. We recall here the definitions and the properties of [8] but we consider histories with a different set of events *ev*, also endowing communication, layer activation and dispatching.

Definition 2 (History Expressions). *History Expressions are defined as follows:*

$H, H_1 ::=$ ε	*empty*		$H_1 + H_2$	*sum*
ev	*events in (2)*		$H_1 \cdot H_2$	*sequence*
h	*recursion variable*		$\mu h.H$	*recursion*
$\varphi[H]$	*safety framing, abbrev. for* $[_\varphi \cdot H \cdot]_\varphi$			

The signature defines sequentialization, sum and recursion operations over sets of histories containing events; μh is a binder for the recursion variable h.

The following definition exploits the labelled transition system in Fig. 3.

Definition 3 (Semantics of History Expressions). *Given a closed H (i.e. without free variables), we define its semantics* $[\![H]\!]$ *to be the set of histories* $\eta = w_1 \dots w_n$ $(w_i \in ev \cup \{\varepsilon\}, 0 \leq i \leq n)$ *such that* $\exists H'.\, H \xrightarrow{w_1} \cdots \xrightarrow{w_n} H'$.

We remark that the semantics of a history expression is a prefix closed set of histories.

Back to the example in Section 2, assume that H is the history expression over-approximating the behaviour of function g. Then, the history expression of the fragment of the cloud service (Fig. 1(b)) is $\mu h.receive_\tau \cdot receive_{\tau'} \cdot \varphi[(_{\mathtt{ACMode}} \cdot H \cdot send_{\tau''})_{\mathtt{ACMode}}] \cdot h$ assuming $\tau = ly_{\mathtt{ACMode}}$.

Closed history expressions are partially ordered: $H \sqsubseteq H'$ means that the abstraction represented by H' is less precise than the one by H. The structural ordering $\sqsubseteq$ is defined over the quotient induced by the (semantic preserving) equational theory presented in [7] as the least relation such that $H \sqsubseteq H$ and $H \sqsubseteq H + H'$. Clearly, $H \sqsubseteq H'$ implies $[\![H]\!] \subseteq [\![H']\!]$.

Validity of History Expressions Given a history η we denote with $\eta^{-[]}$ the history purged of all framings events $[_\varphi,]_\varphi$. For details and examples, see [7].

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

$$ap(\varepsilon) = \{\} \qquad ap(\eta\, [_\varphi) = ap(\eta) \cup \{\varphi\}$$

$$ap(\eta\gamma) = ap(\eta) \quad \gamma \in ev \setminus \{[_\varphi,]_\varphi\} \qquad ap(\eta\,]_\varphi) = ap(\eta) \setminus \{\varphi\}$$

The validity of a history η ($\models \eta$ in symbols) is inductively defined as follows, assuming the notion of policy compliance $\eta \vDash \varphi$ of Section 6.

$$\vDash \varepsilon \quad \text{and} \quad \vDash \eta' w \qquad w \in ev \quad \text{if} \;\; \vDash \eta' \text{ and } (\eta' w)^{-[]} \vDash \varphi \;\; \text{for all } \varphi \in ap(\eta' w)$$

A history expression H is *valid* when $\models \eta$ for all $\eta \in [\![H]\!]$.

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

Property 1. If a history η is valid, then each prefix of its is valid.

The semantics of ContextML (in particular the rules for framing) ensure that the histories generated at runtime are all valid.

Property 2. If $C \vdash \varepsilon, e \to \eta', e'$, then η' is valid.

5 ContextML types

We provide here ContextML with a type and effect system. We use it for over-approximating the programs behaviour and for ensuring that the dispatch mechanism always succeeds at runtime. The associated effect is a history expression representing all the histories that a program may generate. Here, we only give a logical presentation of our type and effect system, and we are confident that an inference algorithm can be developed, along the lines of [10].

Our typing judgements have the form $\langle \Gamma; C \rangle \vdash e : \tau \triangleright H$. This means that in "in the type environment Γ and in the context C the expression e has type τ and effect H".

Types are integers, unit, layers and functions:

$$\sigma \in \wp(\mathsf{LayerNames}) \qquad \mathbb{P} \in \wp(\wp(\mathsf{LayerNames}))$$

$$\tau, \tau_1, \tau' ::= \texttt{int} \mid \texttt{unit} \mid ly_\sigma \mid \tau_1 \xrightarrow{\mathbb{P}|H} \tau_2$$

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 $\tau_1 \xrightarrow{\mathbb{P}|H} \tau_2$, $\mathbb{P}$ is a set of *preconditions* υ. Each $\upsilon \in \mathbb{P}$ over-approximates the set of layers that must occur in the context to apply the function. The history expression H is the latent effect, i.e. the sequence of events generated while evaluating the function.

Fig. 4 introduces the rules for subeffecting ($H \sqsubseteq H'$) and 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 H. 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.

Fig. 5 shows the rules of our type and effect system. Most of them are inhered from that of ML, so we only comment in detail on the rules for the new 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' satisfying υ. A

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

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

Fig. 4. Subtyping rules

context satisfies the precondition υ whenever it contains all the layers in υ, in symbols $|C'| \subseteq \upsilon$, where $|C'|$ denotes the set of layers active in the context C'. 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 H, 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. The effect is obtained by concatenating the ones of e_2 and e_1 and the latent effect H. To better explain how preconditions work, consider the technical example in Fig. 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 $\mathbf{with}(e_1)\ \mathbf{in}\ e_2$ 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 in σ. The special events $(\!|_L$ and $|\!)_L$ 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 H_i preceded by $Disp(L_i)$.

The expression $\mathbf{send}_\tau(e)$ has type `unit` and its effect is that of e extended with event $send_\tau$. The expression $\mathbf{receive}_\tau$ has type τ and its effect is the event $receive_\tau$. 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 6).

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

$$\text{(Tbphi)}\,\frac{\langle \Gamma; C\rangle \vdash e : \tau \triangleright H}{\langle \Gamma; C\rangle \vdash \overline{\varphi}[e] : \tau \triangleright H \cdot]_\varphi} \qquad \text{(Tbwith)}\,\frac{\langle \Gamma; L :: C\rangle \vdash e_2 : \tau \triangleright H}{\langle \Gamma; C\rangle \vdash \mathbf{with}(\overline{L})\ \mathbf{in}\ e_2 : \tau \triangleright H \cdot |\!)_L}$$

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

Our type system enjoys the following soundness results.

$$\text{(TVar)}\ \frac{\Gamma(x)=\tau}{\langle\Gamma;C\rangle\vdash x:\tau\triangleright\varepsilon}\quad \text{(Tint)}\ \frac{}{\langle\Gamma;C\rangle\vdash n:\mathtt{int}\triangleright\varepsilon}\quad \text{(Tunit)}\ \frac{}{\langle\Gamma;C\rangle\vdash ():\mathtt{unit}\triangleright\varepsilon}$$

$$\text{(Tly)}\ \frac{}{\langle\Gamma;C\rangle\vdash L:ly\{L\}\triangleright\varepsilon}\quad \text{(Talpha)}\ \frac{}{\langle\Gamma;C\rangle\vdash \alpha(a):\mathtt{unit}\triangleright\alpha(a)}$$

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

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

$$\text{(Tif)}\ \frac{\langle\Gamma;C\rangle\vdash e_0:int\triangleright H\qquad\langle\Gamma;C\rangle\vdash e_1:\tau\triangleright H_1\qquad\langle\Gamma;C\rangle\vdash e_2:\tau\triangleright H_2}{\langle\Gamma;C\rangle\vdash\mathbf{if}\ e_0\ \mathbf{then}\,e_1\,\mathbf{else}\,e_2:\tau\triangleright H\cdot(H_1+H_2)}$$

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

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

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

$$\text{(Tapp)}\ \frac{\langle\Gamma;C\rangle\vdash e_1:\tau_1\xrightarrow{\mathbb{P}|H}\tau_2\triangleright H_1\qquad\langle\Gamma;C\rangle\vdash e_2:\tau_1\triangleright H_2\qquad\exists\upsilon\in\mathbb{P}.\upsilon\subseteq|C|}{\langle\Gamma;C\rangle\vdash e_1e_2:\tau_2\triangleright H_2\cdot H_1\cdot H}$$

$$\text{(Top)}\ \frac{\langle\Gamma;C\rangle\vdash e_1:\mathtt{int}\triangleright H_1\qquad\langle\Gamma;C\rangle\vdash e_2:\mathtt{int}\triangleright H_2}{\langle\Gamma;C\rangle\vdash e_1\,\mathbf{op}\,e_2:\mathtt{int}\triangleright H_1\cdot H_2}\quad \text{(Tphi)}\ \frac{\langle\Gamma;C\rangle\vdash e:\tau\triangleright H}{\langle\Gamma;C\rangle\vdash\varphi[e]:\tau\triangleright[_\varphi\cdot H\cdot]_\varphi}$$

$$\text{(Trec)}\ \frac{}{\langle\Gamma;C\rangle\vdash\mathbf{receive}_\tau:\tau\triangleright receive_\tau}\quad \text{(Tsend)}\ \frac{\langle\Gamma;C\rangle\vdash e:\tau\triangleright H\qquad H'=H\cdot send_\tau}{\langle\Gamma;C\rangle\vdash\mathbf{send}_\tau(e):\mathtt{unit}\triangleright H'}$$

Fig. 5. Typing rules

Theorem 1 (Subject reduction). *Let e be a closed expression, if $\langle\Gamma;C\rangle\vdash e:\tau\triangleright H$ and $C\vdash\eta,e\to\eta',e'$, then $\langle\Gamma;C\rangle\vdash e':\tau\triangleright H'$ with $\eta H\sqsupseteq\eta' H'$*

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

Corollary 1 (Over-approximation). *If $\langle\Gamma;C\rangle\vdash e:\tau\triangleright H$ and $C\vdash\varepsilon,e\to^*\eta,e'$, then $\eta\in[\![H]\!]$.*

We also have the following result, where $C\vdash\eta,e\nrightarrow$ means that e is stuck.

Theorem 2 (Progress). *Let e be a closed expression such that $\langle\Gamma;C\rangle\vdash e:\tau\triangleright H$. If $C\vdash\eta,e\nrightarrow$ and ηH is valid and with balanced policy framings, then e is a value.*

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

Corollary 2. *If* $\langle \emptyset; C\rangle \vdash e.\tau \triangleright H$ *and* H *is valid and with balanced policy framings, then* $C \vdash \varepsilon, e \rightarrow^* \eta', v$.

This corollary guarantees that a well-typed expression will eventually be reduced to a value, provided that its H is valid and that it complies with the communication protocol.

6 Model Checking

In this section we introduce a model-checking machinery for verifying whether a history expression 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 will be actually a safety property [11], expressing that nothing bad will occur during a computation. Policies are expressed through standard Finite State Automata (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. Let $L(\varphi)$ denote the language of φ.

We depict in the left part of Fig. 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 $\eta \models \varphi$, completing the definition of validity presented in Section 4.

Definition 4 (Policy compliance). *Let* η *be a history without framing events, then* $\eta \models \varphi$ *iff* $\eta \notin L(\varphi)$.

The semantics of a history expression may contain histories with redundant framings, i.e. nesting of the same policy framing. For instance, $\mu h.\,(\varphi[\alpha(r)h] + \varepsilon)$ generates $[_\varphi \alpha(r)[_\varphi \alpha(r)]]$. 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 4 uses $\eta^{-[]}$. Actually, given H there is a *regularisation* algorithm returning his regularized version $H{\downarrow}$ such that *(i)* each history in $[\![H{\downarrow}]\!]$ has no redundant framing, *(ii)* $H{\downarrow}$ is valid if and only if H

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

Fig. 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.

is valid [7]. Hence, checking validity of a history expression H can be reduced to the problem of checking validity of a history expression $H{\downarrow}$ without redundant framings.

Our approach fits into the standard *automata based* model checking [12]. Indeed, there is an efficient and fully automata based method for checking the $\models$ relation for a regularized history expression H.

Let $\{\varphi_i\}$ be the set of all policies φ_i occurring in H. From each φ_i it is possible to obtain a *framed automata* $\varphi_i^{[]}$ such that η is valid iff $\eta \notin L(\bigcup \varphi_i^{[]})$. The detailed construction of framed automata 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 $[_\varphi$ from the first copy to the second and $]_\varphi$ in the opposite direction. So when a framing gets activated we can also reach an offending state. Fig. 7 shows the framed automaton used to model check the policy φ_2.

Validating a regularized history expression H amounts to verifying $[\![H]\!] \cap \bigcup L(\varphi_i^{[]})$ is empty. Using the fact that for any history expression H there exists a pushdown automaton $B(H)$ (see [8]) that recognizes the semantics of H, we can state the following:

Theorem 3 (Model checking policies). *A given history expression H is valid if and only if $L(B(H{\downarrow})) \cap \bigcup L(\varphi_i^{[]}) = \emptyset$.*

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 [13] the above 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 $send_\tau$ and $receive_\tau$ actions designating the coordination interactions, possibly repeated (in symbols S^*), as defined below:

$$P ::= S \mid S^* \qquad\qquad S ::= \varepsilon \mid send_\tau.S \mid receive_\tau.S$$

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

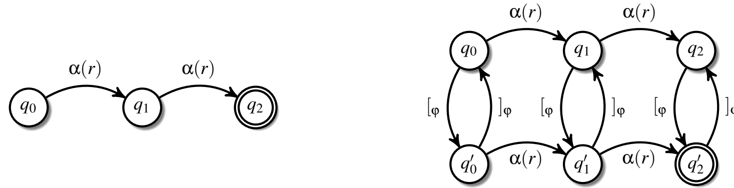

Fig. 7. On the left: a policy φ_2 expressing that two consecutive actions α on r at the beginning of the computation are forbidden. On the right: the framed automaton obtained from φ_2.

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 $send_\tau$ and $receive_\tau$ appear) belong to $L(P)$.

Let H^{sr} be a projected history expression where all non $send_\tau$,$receive_\tau$ events have been removed. Then we define compliance to be:

Definition 5 (Protocol compliance). *Let e be an expression such that $\langle \Gamma, C \rangle \vdash e : \tau \triangleright H$, then e is compliant with P if $[\![H^{sr}]\!] \subseteq L(P)$.*

This theorem provides us with a decidable model checking procedure to establish protocol compliance. In its statement we write $\overline{L(P)}$ for the complement of $L(P)$. Note that the types annotating $send_\tau$/$receive_\tau$ can be kept finite in both $L(P)$ and $\overline{L(P)}$, because we only take the types occurring in the effect H under checking.

Theorem 4 (Model checking protocols). *Let e be an expression such that $\langle \Gamma, C \rangle \vdash e : \tau \triangleright H$, then e is compliant with P iff*

$$L(B(H^{sr})) \cap L(P) \neq \emptyset \wedge L(B(H^{sr})) \cap \overline{L(P)} = \emptyset$$

We remark that, in our model, protocol compliance cannot be expressed only through security policies introduced above. As a matter of fact, $L(B(H^{sr})) \cap \overline{L(P)} = \emptyset$ expresses that H has no forbidden communication patterns, and this is a requirement much similar to a default-accept policy. However $L(B(H^{sr})) \cap L(P) \neq \emptyset$ requires that some communication pattern in compliance with P *must* be done. This highlights the different nature of security policies and protocols in our framework.

7 Conclusions

This paper is an initial step in defining language-based methods for the development of complex adaptive systems. 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.

This work crosses the boundaries of several research fields. Space limitation prevents us to make a comprehensive discussion of related works. Therefore we only point the reader to some papers — we apologise for our omissions. On the foundational aspects of COP, we only cite to [14,15,6], [16,4] that focus on implementation issues, while [5] is on the methodological side. The ContextML primitives for resources usage control are borrowed from [17,7]. Here we additionally deal with layer activation and dispatching, and with a restricted form of communication. Indeed, our communication model can be read as a minimal coordination paradigm, as it only requires the knowledge of the flow of the exchanged messages and their types. Other work in the literature presents richer coordination models; among others, see [18,19].

Our types are a simple form of dependent types, while effects and protocols may be seen as a form of *behavioural types* [20].

We plan to extend the present work by dealing with a more powerful coordination model including concurrency, distribution and asynchrony. It would be also interesting to investigate the relationships between our notion of protocol and *session types* [21].

Acknowledgments. We would like to thank the anonymous referees for their comments that pointed us inaccuracies and guided us to improve the quality of our paper.

References

1. Cheng, B.H.C., 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., Springer (2009) 1–26
2. Bruni, R., Corradini, A., Gadducci, F., Lluch Lafuente, A., Vandin, A.: A conceptual framework for adaptation. In: FASE 2012. Volume to appear of LNCS., Springer (2012)
3. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. TAAS **4**(2) (2009)
4. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. Journal of Object Technology, March-April 2008, ETH Zurich **7**(3) (2008) 125–151
5. Salvaneschi, G., Ghezzi, C., Pradella, M.: Context-oriented programming: A programming paradigm for autonomic systems. CoRR **abs/1105.0069** (2011)
6. Degano, P., Ferrari, G.L., Galletta, L., Mezzetti, G.: Typing context-dependent behavioural variations. In: PLACES 2012. Volume to appear in EPTCS. (2012)
7. Bartoletti, M., Degano, P., Ferrari, G.L., Zunino, R.: Local policies for resource usage analysis. ACM Trans. Program. Lang. Syst. **31**(6) (2009)
8. Bartoletti, M., Degano, P., Ferrari, G.L.: Planning and verifying service composition. Journal of Computer Security **17**(5) (2009) 799–837
9. Sangiorgi, D., Walker, D.: The Pi-Calculus - a theory of mobile processes. Cambridge University Press (2001)
10. Skalka, C., Smith, S., Horn, D.V.: Types and trace effects of higher order programs. Journal of Functional Programming **18**(2) (2008) 179–249
11. Hamlen, K.W., Morrisett, J.G., Schneider, F.B.: Computability classes for enforcement mechanisms. ACM Trans. on Programming Languages and Systems **28**(1) (2006) 175–205
12. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification (preliminary report). In: LICS, IEEE Computer Society (1986) 332–344
13. Hopcroft, J., Motwani, R., Ullman, J.: Introduction to automata theory, languages, and computation. Volume 2. Addison-wesley Reading, MA (1979)
14. Clarke, D., Sergey, I.: A semantics for context-oriented programming with layers. In: International Workshop on Context-Oriented Programming. COP '09, New York, NY, USA, ACM (2009) 10:1–10:6
15. Hirschfeld, R., Igarashi, A., Masuhara, H.: ContextFJ: a minimal core calculus for context-oriented programming. In: Proceedings of the 10th international workshop on Foundations of aspect-oriented languages, ACM (2011) 19–23
16. Costanza, P.: Language constructs for context-oriented programming. In: In Proceedings of the Dynamic Languages Symposium, ACM Press (2005) 1–10
17. Igarashi, A., Kobayashi, N.: Resource usage analysis. In: POPL. (2002) 331–342
18. Proença, J., Clarke, D., de Vink, E.P., Arbab, F.: Decoupled execution of synchronous coordination models via behavioural automata. In Mousavi, M.R., Ravara, A., eds.: FOCLASA. Volume 58 of EPTCS. (2011) 65–79
19. Bonsangue, M., Clarke, D., Silva, A.: Automata for context-dependent connectors. In: Coordination Models and Languages, Springer (2009) 184–203
20. Nielson, H.R., Nielson, F.: Higher-order concurrent programs with finite communication topology (extended abstract). In: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '94, New York, NY, USA, ACM (1994) 84–97
21. Honda, K., Vasconcelos, V., Kubo, M.: Language primitives and type discipline for structured communication-based programming. Programming Languages and Systems (1998) 122–138