Formalising Security in Ubiquitous and Cloud Scenarios *

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

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

Abstract. We survey some critical issues arising in the ubiquitous computing paradigm, in particular the interplay between context-awareness and security. We then overview a language-based approach that addresses these problems from the point of view of Formal Methods. More precisely, we briefly describe a core functional language extended with mechanisms to express adaptation to context changes, to manipulate resources and to enforce security policies. In addition, we shall outline a static analysis for guaranteeing programs to securely behave in the digital environment they are part of.

1 Introduction

We can be connected at any time and anywhere. Internet is *de facto* becoming the infrastructure providing us with wired or wireless access points for our digitally instrumented life. A great variety of activities and tasks performed by individuals are mediated, supported and affected by different heterogenous digital systems that in turn often cooperate each other without human intervention. These digital entities can be any combination of hardware devices and software pieces or even people, and their activities can change the physical and the virtual environment where they are plugged in. For example, in a smart house, a sensor can proactively switch on the heater to regulate the temperature. An emerging line is therefore integrating these entities into an active and highly dynamic digital environment that hosts end-users, continuously interacting with it. Consequently, the digital environment assumes the form of a communication infrastructure, through which its entities can interact each other in a loosely coupled manner, and they can access resources of different kinds, e.g., local or remote, private or shared, data or programs, devices or services. The name *ubiquitous computing* is usually adopted to denote this phenomena.

Some illustrative, yet largely incomplete, cases of computational models and technologies towards the realisation of this approach have been already developed and deployed. Among these, the most significant are Service Oriented Computing, the Internet of Things and Cloud Computing. Each of them is fostered by

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

and addresses different aspects of the implementation and the usage of ubiquitous computing as follows.

In the Service Oriented Computing approach, applications are open-ended, heterogenous and distributed. They are built by composing software units called services, which are published, linked, and invoked on-demand by other services using standard internet-based protocols. Moreover, applications can dynamically reconfigure themselves, by re-placing the services in use with others. Finally, services are executed on heterogeneous systems and no assumptions can be taken on their running platforms. In brief, a service offers its users access remote resources, i.e. data and programs.

A further step towards ubiquitous computing is when software pervades the objects of our everyday life, e.g. webTV, cars, smartphones, ebook readers, etc. These heterogenous entities often have a limited computational power, but are capable of connecting to the internet, coordinating and interacting each other, in the so-called "plug&play" fashion. The real objects, as well as others of virtual nature (programs, services, etc.), which are connected in this way, form the Internet of Things. Objects become points where information can be collected and where some actions can be performed to process it, so changing the surrounding environment.

Cloud computing features facilities that are present on both the approaches above. Indeed, it offers through the network a hardware and software infrastructure on which end-users can run their programs on-demand. In addition, a rich variety of dynamic resources, such as networks, servers, storage, applications and services are made available. A key point is that these resources are "virtualised" so that they appear to their users as fully dedicated to them, and potentially unlimited.

The three approaches briefly surveyed above share some peculiar aspects, in that the objects they manipulate are highly dynamic, open-ended, available on-demand, heterogeneous, and always connected. At the same time, also the infrastructure hosting the digital entities needs to efficiently support connectivity, elastic re-configuration, and resource access and handling. Mechanisms are therefore in order to adapt the shape and modalities of the interactions among digital entities, making their behaviour context-aware. These can join and leave the digital environment at will, so the infrastructure must protect itself and the users by supplying security guarantees.

Many different techniques and approaches are being proposed to tackle the issues typical of the ubiquitous computing scenario. In spite of their importance, we shall completely neglect the social and legal aspects, and refer the interested reader to [96]. In this paper, we shall instead rely on *Formal Methods*, in particular from a language-based perspective. Being formal offers the mathematical bases supporting a well-established repertoire of techniques, methods and tools for a rigorous development of applications. In turn, these are to be implemented in a programming language with high-level constructs, endowed with a clear semantics.

More precisely, we shall focus on adaptivity and security. Adaptivity is the capability of digital entities to fit for a specific use or situation; in a pervasive computing scenario this is a key aspect. Security is mandatory because the apparent simplicity of use of the new technologies hides their not trivial design and implementation, that become evident only when something goes wrong. In general, the risk is exchanging simplicity for absence of attention. For example, [71] reports on an easy attack to a wireless insulin pump, that enables the intruder, who bypasses authentication, to dangerously quadruple the dose remotely.

The next section will briefly survey the security issues of context-aware systems developed in the fields of Service Oriented Computing, the Internet of Things and Cloud computing. In Section 3 we shall discuss the interplay between context-awareness and security, and in Section 4 we shall introduce our recent proposal through a running example. Section 5 briefly overviews our proposal more technically [53]. We describe a core functional language extended with mechanisms to express adaptation to context changes, to manipulate resources and to enforce security policies. In addition, we shall outline a static analysis for guaranteeing programs to safely adapt their behaviour to the changes of the digital environment they are part of, and to correctly interact with it.

2 State of the Art and Challenges

There is a very rich literature about ubiquitous computing, from different points of view, including social, political, economical, technological and scientific ones. By only considering the approaches within the last two viewpoints, a large number of technological and scientific communities grew in a mesh of mostly overlapping fields, each one with its own methodologies and tools.

Below, we focus on three branches, and related technologies, that we consider pivotal in ubiquitous computing from a developer perspective. We think that security and context-awareness are among the main concerns of ubiquitous computing, and so we mainly report on the results of the formal method community on these aspects.

2.1 Service Oriented Computing

Service Oriented Computing (SOC) is a well-established paradigm to design distributed applications [81,80,79,50]. In this paradigm, applications are built by assembling together independent computational units, called *services*. Services are stand-alone components distributed over a network, and made available through standard interaction mechanisms.

The main research challenges in SOC are described in [80]. An important aspect is that services are *open*, in that they are built with little or no knowledge about their operating environment, their clients, and further services therein invoked. Adaptivity shows up in the SOC paradigm at various levels. At the lower one, the middleware should support dynamically reconfigurable run-time architectures and dynamic connectivity. Service composition heavily depends on

which information about a service is made public; on how those services are selected that match the user's requirements; and on the actual run-time behaviour of the chosen services. Service composition demands then autonomic mechanisms also driven by business requirements. The service oriented applications, made up by loosely coupled services, also require self management features to minimise human intervention: self-configuring, self-adapting, self-healing, self-optimising, in the spirit of autonomic computation [67].

A crucial issue concerns defining and enforcing non-functional requirements of services, e.g. security and service level agreement ones. In particular, service assembly makes security imposition even harder. One reason why is that services may be offered by different providers, which only partially trust each other. On the one hand, providers have to guarantee the delivered service to respect a given security policy, in any interaction with the open operational environment, and regardless of who actually called the service. On the other hand, clients may want to protect their sensible data from the services invoked. Furthermore, security may be breached even when all the services are trusted, because of unintentional behaviour due, e.g. to design or implementation bugs, or because the composition of the services exhibits some unexpected and unwanted behaviour, e.g. leakage of information.

Web Services [8,88,94] built upon XML technologies are possibly the most illustrative and well developed example of the SOC paradigm. Indeed, a variety of XML-based technologies already exists for describing, discovering and invoking web services [45,28,12,2]. There are several standards for defining and enforcing non-functional requirements of services, e.g. WS-Security [14], WS-Trust [11] and WS-Policy [29]. The kind of security taken into account in these standards only concerns end-to-end requirements about secrecy and integrity of the messages exchanged by the parties.

Assembly of services can occur in two different flavours: *orchestration* or *choreography*. Orchestration describes the interactions from the point of view of a single service, while choreography has a global view, instead. Languages for orchestration and choreography have been proposed, e.g. BPEL4WS [12,72] and WS-CDL [70]. However these languages have no facilities to explicitly handle security of compositions, only being focussed on end-to-end security. Instead, XACML [3] gives a more structured and general approach, because it allows for declaring access control rules among objects that can be referenced in XML.

It turns out that the languages mentioned above do not describe many non-functional requirements, especially the ones concerning the emerging behaviour obtained by assembling services.

The literature on formal methods reports on many (abstract) languages for modelling services and their orchestration, see [56,27,60,73,20,77,74,97,38,34] just to cite a few; [23] is a recent detailed survey on approaches to security and related tools, especially within the process calculi framework, [76] provides a formal treatment for a subset of XACML. An approach to the secure composition of services is presented in [19,20]. Services may dynamically impose policies on resource usage, and they are composed guaranteeing that these policies will

actually be respected at run-time. The security control is done efficiently at static time, by exploiting a type and effect system and model-checking.

The problem of relating orchestration and choreography is addressed in [40,46], but without focusing on security issues.

Recently, increasing attention has been devoted to express service contracts as behavioural or session types [65]. These types synthesise the essential aspects of the interaction behaviour of services, while allowing for efficient static verification of properties of composed systems. Through session types, [91] formalises compatibility of components and [33] describes adaptation of web services. Security has also been studied using session types, e.g. by [26,17,16].

2.2 Internet of Things

In 1988, Mark Weiser described his vision about the coming age of ubiquitous computing.

> "Ubiquitous computing names the third wave in computing, just now beginning. First were mainframes, each shared by lots of people. Now we are in the personal computing era, person and machine staring uneasily at each other across the desktop. Next comes ubiquitous computing, or the age of calm technology, when technology recedes into the background of our lives."

In this world, sensors and computers are commodities available everywhere and surrounding people anytime. This idea has given rise to what is now called the "Internet of Things" [15] or "Everyware" [58]. Due to the pervasive integration of connectivity and identification tags, real objects are represented by digital entities in the virtual environment supported by dynamic opportunistic networks [82] or by the Internet. This is the case, e.g., of a fridge, that becomes an active entity on the Internet ready to be queried for its contents.

Such an intelligent space is then made of smart things, physical and endowed with software pieces, or fully virtual, that are highly interconnected and mutually influence their behaviour.

The software of intelligent spaces has then to be aware of the surrounding environment and of the ongoing events, to reflect the idea of an active space reacting and adapting to a variety of different issues, e.g. arising from people, time, programs, sensors and other smart things. Besides being assigned a task, a device can also take on the responsibility of proactively performing some activities with or for other digital entities.

The *Ambient calculus* [41] is among the first proposals to formalise those aspects of intelligent, virtual environments that mainly pertain to the movement of (physical devices and) software processes. The processes are hosted in virtual, separated portions of the space, called *ambients*. Processes can enter and leave ambients, and be executed in them. This calculus has been used, e.g. in [36] to specify a network of devices, in particular to statically guarantee some properties

of them, e.g. for regulating rights to the provision and discovery of environmental information.

Security plays a key role in the Internet of Things, not only because any digital entity can plug in, but also because the smart things may be devices, with also specific physical dimensions. The key point here is that information and physical security became interdependent, and traditional techniques that only focus on digital security are inadequate [39]. For example, there are some papers facing these issues with suitable extensions of the Ambient calculus, among which [75,37,31,30,93], but these proposals do not address the protection of the physical layer of smart things.

In addition, since spaces are active, the digital entities composing them may easily collect sensible information about the nearby people and things. Privacy may therefore be violated, because people are not always aware that their sensible data may be collected, nor that their activities are always context-aware, either. Even worse: it is possible through data mining to disclose pieces of information, possibly confidential, so originating a tension with a tacit assumption or an explicit guarantee of privacy. For example, the analysis of behavioural patterns over big data can infer the actual identity of the individuals, violating their assumed anonymity [89].

Although some workshops and conferences are being organised on the new security topics of the Internet of Things, to the best of our knowledge little work is done in the area of formal methods, except for studies on protocols that guarantee anonymity (for brevity, we only refer the reader to the discussion on related work and to the references of [99]).

2.3 Cloud computing

The US National Institute of Standards and Technology defines Cloud computing as follows:

> Cloud computing is a model for enabling convenient, on-demand network access a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

Cloud computing refers therefore to both the applications therein deployed and made available on the Internet and to the hardware infrastructures that makes this possible. The key characteristics of Cloud computing include on-demand self-service, ubiquitous network access, resource pooling. Also, Cloud systems are characterised by some peculiar adaptivity characteristics called *elasticity* and *measured services* [10]. These are related to the two different viewpoints of the Cloud that customers and providers have. Elasticity refers to the ability of the provider to adapt its hardware infrastructure with minimal effort to the requirements of different customers, by scaling up and down the resources assigned to them. Measured services indicates load and resource usage optimisation. It

refers then to scalability from the provider point of view, usually achieved by *multi-tenancy*, i.e. by the capability of dynamically and accurately partitioning an infrastructure shared among various consumers.

The Cloud offers different models of services: "Software as a Service", "Platform as a Service" and "Infrastructure as a Service", so subsuming SOC. These three kinds can be organised in a stack hierarchy where the higher ones are built on the lower ones. At the lowest level, the Cloud provider offers an infrastructure made up by virtual machines, storages, network interconnections, etc. The whole application environment is granted to the user, who takes the responsibility for it. When suppling a platform as a service, the provider gives a basic software stack, usually including the operating system and a programming environment. At the highest level, we have Software as a Service, i.e. the provider enables its users to run on-demand one of its software service.

These aspects have been recently tackled within the formal methods approach. A formal calculus for defining the architecture of virtualised systems has been proposed in [22]. Elasticity has been studied and formalised in [32] through a calculus of processes with a notion of groups. A core functional language extended with explicit primitives for computational resource usage has been proposed in [24]. A process calculus with explicit primitives to control the distributed acquisition of resources has been presented in [25].

The most relevant security issues in Cloud computing deal with access to resources, their availability and confidentiality [90]. Due to the distributed nature of the computation carried on in the Cloud, one challenge is to ensure that only authorised entities obtain access to resources. A suitable identity management infrastructure should be developed, and users and services are required to authenticate with their credentials. However such a feature may affect the level of interoperability that a Cloud needs to have, because of the management of the identity tokens and of the negotiation protocols. In addition, providers should guarantee the right level of isolation among partitioned resources within the same infrastructure, because multi-tenancy can make unstable the borders between the resources dedicated to each user. Virtualisation usually helps in controlling this phenomenon.

Moreover users should protect themselves from a possibly malicious Cloud provider, preventing him from stealing sensible data. This is usually achieved by encryption mechanisms [10] and identity management. It is worth noting that whenever a program running in the Cloud handles some encrypted data, and the attacker is the provider itself, encryption is useless if also the encoding key is on the Cloud. For a limited number of cases, this problem can be circumvented by using homomorphic encryption schemata [57,6]. There are providers that support working groups (e.g. [1]) who are aiming at making efficient homomorphic encryption so to commercially exploit it in the Cloud [78].

Note in passing that the Cloud can be misused and support attacks to security. Indeed, the great amount of computational resources made easily available can be exploited for large-scale hacking or denial of service attacks, and also to perform brute force cracking of passwords [4].

In the currently available systems, the responsibility for dealing with security issues is often shared between customer and providers. The actual balance depends on the service level at which security has to be enforced [9].

3 Adaptivity and Security

In the previous sections, we overviewed a few technological and foundational features of ubiquitous computing, focussing on the Service Oriented Computing, the Internet of Things and the Cloud paradigms, from the developers' perspective. An emergent challenge is integrating adaptivity and security issues to support programming of applications and systems in the ubiquitous setting.

Adaptivity refers to the capability of a digital entity, in particular of its software components, to dynamically modify its behaviour reacting to changes of the surrounding active space, such as the location of its use, the collection of nearby digital entities, and the hosting infrastructure [86,35]. Software must therefore be aware of its running environment, represented by a declarative and programmable *context*.

The notion of context assumes different forms in the three computational models discussed earlier. The shape of contexts in Service Oriented Computing is determined by the various directories where services are published and by the end-points where services are actually deployed and made available, as well as by other information about levels of service, etc. In the Internet of Things, the context is a (partial) representation of the active space hosting and made of the digital entities; so each entity may have its own context. In the Cloud, a context contains the description of the computational resources offered by the centralised provider, and also a measure of the available portion of each resource; note that the context has to show to the provider the way computational resources are partitioned, for multi-tenancy.

A very short survey of the approaches to context-awareness follows, essentially from the language-based viewpoint (see, e.g. SCEL [51]). Other approaches range on a large spectrum, from the more formal description logics used to representing and querying the context [43,95,59] to more concrete ones, e.g. exploiting a middleware for developing context-aware programs [84].

Another approach is Context Oriented Programming (COP), introduced by Costanza [49]. Also subsequent work [63,5,69,13] follow this paradigm to address the design and the implementation of concrete programming languages. The notion of *behavioural variation* is central to this paradigm. It is a chunk of behaviour that can be activated depending on the current working environment, i.e. of the context, so to dynamically modify the execution. Here, the context is a stack of *layers*, and a programmer can activate/deactivate layers to represent changes in the environment. This mechanism is the engine of context evolution. Usually, behavioural variations are bound to layers: activating/deactivating a layer corresponds to activating/deactivating a behavioural variation. Only a few papers in the literature give a precise semantic description of the languages

within the Context Oriented Programming paradigm. Among these, we refer the reader to [48,68,64,47,52], that however do not focus on security issues.

Security issues, instead, have been discussed in [42], even though this survey mainly considers specific context-aware applications, but not from a general formal methods viewpoint. Combining security and context-awareness requires to address two distinct and interrelated aspects. On the one side, security requirements may reduce the adaptivity of software. 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 [98,39], and we outline below two possible 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 avaiability [83] and at developing techniques for guaranteeing them [98]. Contexts may contain sensible data of the working environment (e.g. information about surrounding digital entities), and therefore to grant confidentiality this contextual information should be protected from unauthorised access. Moreover, the integrity of contextual information requires mechanisms for preventing its corruption by any entity in the environment. 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, that may 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 definition and enforcement of high-level policies that talk about, are based on, and depend on the notion of dynamic context. The policies most studied in the literature control the accesses to resources and smart things, see among the others [98,66,100]. Some e-health applications show the relevance of access control policies based on the roles attached to individuals in contexts [7,54].

Most of the work on securing context-aware systems and on context-aware security aims at implementing various features at different levels of the infrastructures, e.g. in the middleware [84] or in the interaction protocols [62]. Indeed, the basic mechanisms behind security in adaptive systems have been studied much less. Moreover, the two dual aspects of context-aware security sketched above are often tackled separately. We lack then a unifying concept of security.

Our proposal faces the challenges pointed out above, by formally endowing a programming language with linguistic primitives for context-awareness and security, provided with a clear formal semantics. We suitably extend and integrate together techniques from COP, type theory and model-checking. In particular, we develop a static technique ensuring that a program: (*i*) adequately reacts to context changes; (*ii*) accesses resources in accordance with security policies; (*iii*) exchanges messages, complying with specific communication protocols.

4 An example

In this section a working example intuitively illustrates our methodology, made of the following three main ingredients:

1. a COP functional language, called ContextML [53,52], with constructs for resource manipulation and communication with external parties, and with mechanisms to declare and enforce security policies, that talk about context, including roles of entities, etc. We consider regular policies, in the style of [61], i.e. safety properties of program traces;
2. a type and effect system for ContextML. 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*, describes the sequences of resource manipulation and communication with external parties in a succinct form;
3. a model check on the 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.

Consider a typical scenario of ubiquitous computing. A smartphone app remotely uses a Cloud as a repository to store and synchronise a library of ebooks. Also it can execute locally, or invoke remotely customised services. In this example we consider a simple 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 be not synchronised. The synchronisation is triggered on demand and it depends on several factors: the actual bandwidth available for connection; the free space on the device; etc.

This example shows that our programmable notion of context can represent some of the environmental information briefly discussed in Section 3. In particular, the context is used to represent the location where the full-text search is performed; the status of the device and of the resources (synchronised or not) offered by the Cloud; and the actual role of the service caller. We specify below the fragment of the app that implements the search over the user's library.

Consider 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*:

`ACMode`. ⟨DO SEARCH⟩,
`BatMode`. ⟨DO SOMETHING ELSE⟩

This is the way context-dependent *behavioural variations* are declared. Roughly, a layered expression is an expression defined by cases. The cases are given by 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 locally, 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(b)), 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 at line 4), the synchronisation state of the library, at lines 5,6, and the active energy profile at lines 2,10. The smartphone communicates with the Cloud system over the bus through message passing primitives, at lines 7-9. The search is performed locally only if the library is fully synchronised and the smartphone is plugged in. If the device is plugged in, but the library is not fully synchronised, then the code of function g is sent to the Cloud and executed remotely by a suitable server.

Lines 7-10 specify some communications of the service, in quite a simple model. Indeed, we adopt a *top-down* approach [44] to describe the interactions between programs, based on a unique channel of communication, the *bus*, through which messages are exchanged. For simplicity, we assume the operational environment to give the protocol P governing the interactions.

In Fig. 1(a) we show a fragment of the environment provided by the cloud. The service considered offers generic computational resources to the devices connected on the bus, by continuously running f. The function f listens to the bus for an incoming request from a user identified by *id*. Then the provider updates the billing information for the customer and waits for incoming code (a function) and an incoming layer. Finally, it executes the received function in a context extended with the received layer. Note that before executing the function, the Cloud switches its role from Root, with administrator rights, to the role Usr.

In the code of the Cloud there are two security policies φ, φ', the scopes of which are expressed by the security framings $\varphi[\dots], \varphi'[\dots]$. Intuitively, they cause a sandboxing of the enclosed expression, to be executed under the strict monitoring of φ and φ' respectively. The policy φ specifies the infrastructural rules of the Cloud. Among the various controls, it will inhibit a Usr to become Root. Indeed, being context-aware, our policies can also express role-based or

location-based policies. Instead φ' is enforced right before running the received function g and expresses that writing on the library `write(library)` is forbidden (so only reading is allowed). In this way, we guarantee that the execution of external code does not alter the remote library.

The viable interactions on the bus are constrained by a given protocol P. We assume that the given protocol P is indeed an abstraction of the behaviour of the various parties involved in the communications. We do not address here how protocols are defined by the environment and we only check whether a program respects the given protocol.

In our example the app must send the identifier of the user, the layer that must be active active and the code implementing the search. Eventually, the app receives the result of the search. This sequence of interactions is precisely expressed by the following protocol.

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

The values to be sent/received are represented in the protocol by their type: τ_{id}, τ, τ', τ''. We will come back later on the usage of these types. The symbol * means that this sequence of actions can be repeated any number of times.

Function `getBatteryProfile` returns either layer value `ACMode` or `BatMode`. So the function is assigned the type $ly_{\{\texttt{ACMode},\texttt{BatMode}\}}$ meaning that the returned layer is one between the two mentioned above.

Function g takes the type `unit` and returns the type τ'', assuming that the value returned by the `search` function has type τ''. The application of g depends on the current context. For this reason we enrich the type with a set of preconditions $\mathbb{P}$ where each precondition $\upsilon \in \mathbb{P}$ is a set of layers. In our example the precondition $\mathbb{P}$ is

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

In order to apply g, the context of application must contains all the layers in υ, for a preconditions $\upsilon \in \mathbb{P}$. Furthermore, we annotate the type with the latent effect H. It is a history expression and represents (a safe over-approximation of) the sequences of resource manipulation or layer activations or communication actions, possibly generated by running g. To summarise the complete type of g is $\texttt{unit} \xrightarrow{\mathbb{P}|H} \tau''$.

Our type system guarantees that, if a program type-checks, the dispatching mechanism always succeeds at run-time. In our example, the expression (1) will be well-typed whenever the context in which it will be evaluated contains either `IsLocal` or `IsCloud`, and either `LibraryUnsynced` or `LibrarySynced`. The preconditions over `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 construct **with**.

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 policies φ and φ' is done by standard model-checking the effect

```
1  : fun f x =
2  :   φ[with(Root) in
3  :     let id = receive_τid in
4  :       cd(/billing); write(bid_id)
5  :       cd(../lib_id)
6  :   ;
7  :   with(Usr) in
8  :     let lyr = receive_τ in
9  :     let g = receive_τ′ in
10 :     φ′[with(lyr) in
11 :     let res = g() in
12 :       send_τ″(res)
13 :     ]
14 :   ]; f()
```

(a)

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

(b)

Fig. 1. Two fragments of a service in the Cloud (a) and of an app (b)

of g (a context-free language) against the policies (regular languages). Since in our example the app never writes and never changes the actual role to Root, the policies φ' and φ are satisfied (under the hidden assumption that the code for the `BatMode` case has an empty effect).

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

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

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 $P = (send_{\tau_{id}} send_{\tau} send_{\tau'} receive_{\tau''})^*$. This is indeed the case here.

5 ContextML

Context and adaptation features included in ContextML are borrowed from COP languages [63] discussed above. Indeed, ContextML is characterised by: (i) a declarative description of the working environment, called *context*, that is a stack of *layers*; (ii) layers describing properties about the application current environment; (iii) constructs for activating layers; (iv) mechanism for defining behavioural variations.

The resources available in the system are represented by identifiers and can be manipulated by a fixed set of actions. For simplicity, we here omit a construct for dynamically creating resources, that can be dealt with following [20,18].

We enforce security properties by protecting expressions with policies: $\varphi[e]$. This construct is called *policy framing* [18] and it generalises the standard *sandbox* mechanism. Roughly, it means that during the evaluation of the program

e the computation performed so far must respect the policy φ in the so-called history-dependent security.

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.

5.1 Syntax and Semantics

Let $\mathbb{N}$ be the naturals, Ide be a set of identifiers, LayerNames be a finite set of layer names, Policies be a set of security policies, Res be a finite set of resources (identifiers) and Act be a finite set of actions for manipulating resources. The syntax of ContextML is as follows:

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

v, v' ::=		*values*	
	$\mid n \mid \mathbf{fun} f\ x\ \Rightarrow e \mid () \mid L$		
$lexp$ ::=		*layered expressions*	
	$\mid L.e \mid L.e, lexp$		
e, e' ::=		*expressions*	
	$\mid \mathbf{v}$	value	Core ML
	$\mid \mathbf{x}$	variable	Core ML
	$\mid e_1\ e_2$	application	Core ML
	$\mid e_1\ \mathbf{op}\ e_2$	operation	Core ML
	$\mid \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2$	conditional expression	Core ML
	$\mid \mathbf{with}(e_1)\ \mathbf{in}\ e_2$	with	Context
	$\mid lexp$	layered expressions	Context
	$\mid \alpha(r)$	access event	Resource
	$\mid \varphi[e]$	policy framing	Policy
	$\mid \mathbf{send}_\tau(e)$	send	Communication
	$\mid \mathbf{receive}_\tau$	receive	Communication

In addition, we adopt the following standard abbreviations: $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \triangleq (\mathbf{fun}\_\ x \Rightarrow e_2)\, e_1$ and $e_1; e_2 \triangleq (\mathbf{fun} f\ x \Rightarrow e_2)\, e_1$, with x, f not free in e_2.

The core of ML is given by the functional part, modelled on the call-by-value λ-calculus.

The primitive **with** models the evaluation of the expression e_2 in the context extended by the layer obtained by the evaluation of e_1. Behavioural variations are defined by layered expression ($lexp$), expressions defined by cases each specifying a different behaviour depending on the actual structure of the context.

The expression $\alpha(r)$ models the invocation of the access operation α over the resource r, causing side effects. Access labels specify the kind of access operation, e.g. read, write, and so on.

A security policy framing $\varphi[e]$ defines the scope of the policy φ to be enforced during the evaluation of e. Policy framings can also be nested.

The communication is performed by $\mathbf{send}_\tau$ and $\mathbf{receive}_\tau$ that allow the interaction with the external environment by writing/reading values of type τ (see Subsection 5.3) to/from the bus.

Dynamic Semantics We endow ContextML with a small-step operational semantics, only defined, as usual, for closed expressions (i.e. without free variables).

Our semantics is history dependent. Program *histories* are sequences of events that occur during program execution. Events ev indicate the observation of critical activities, such as activation (deactivation) of layers, selection of behavioural variations and program actions, like resource accesses, entering/exiting policy framings and communication. The syntax of events ev and programs histories η is the following:

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

$$\eta ::= \epsilon \mid ev \mid \eta\,\eta \qquad (\epsilon \text{ is the empty history}) \tag{3}$$

A history is a possibly empty sequence of events occurring at runtime. The event $(\!|_L$ ($|\!)_L$) marks that we begin (end) the evaluation of a **with** body in a context where the layer L is activated (deactivated), the event $\mathsf{Disp}(L)$ signals that the layer L has been selected by the dispatching mechanism. The event $\alpha(r)$ marks that the action α has been performed over the resource r; the event $[_\varphi$ ($]_\varphi$) records that we begin (end) the enforcement of the policy φ; the event $send_\tau$/$receive_\tau$ indicates that we have sent/read a value of type τ over/from the bus.

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

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

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

$$L :: C = [L, L_1, \ldots, L_n] \text{ where } [L_1, \ldots, L_n] = C - L$$

The semantic rules of the core part are inherited from ML and we will omit them. Below, we will show and comment only the ones for the new constructs.

The transitions have the form $C \vdash \eta, e \to \eta', e'$, meaning that in the context C, starting from a program history η, in one evaluation step the expression e may evolve to e', extending the history η to η'. Initial configurations have the form (ϵ, e). We write $\eta \vDash \varphi$ when the history η obeys the policy φ, in a sense that will be made precise in the Subsections 5.2 and 5.4.

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

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

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

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

The rules for $\mathbf{with}(e_1)\ \mathbf{in}\ e_2$ evaluate e_1 in order to obtain the layer L (with_1) that must be activated to eventually evaluate the expression e_2 (with_3). In addition, in the history, the events $(\!|_L$ and $|\!)_L$ mark the beginning (with_2) and the end (with_4) of the evaluation of e_2. Note that being within the scope of layer L activation is recorded by using $\overline{L}$ (with_2). When the expression is just a value the **with** is removed (with_4).

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

In evaluating a layered expression $e = L_1.e_1, \ldots, L_n.e_n$ (rule lexp), the current context is inspected top-down to select the expression e_i that corresponds to the layer L_i, selected by the dispatching mechanism illustrated below. The history is updated by appending $\mathsf{Disp}(L_i)$ to record that the layer L_i has been selected. The dispatching mechanism is implemented by the partial function Dsp, defined as

$$Dsp([L_0', L_1', \ldots, L_m'], A) = \begin{cases} L_0' & \text{if } L_0' \in A \\ Dsp([L_1', \ldots, L_m'], A) & \text{otherwise} \end{cases}$$

It returns the first layer in the context $[L_0', L_1', \ldots, L_m']$ which matches one of the layers in the set A. If no layer matches, then the computation gets stuck.

$$\text{action} \frac{}{C \vdash \eta, \alpha(r) \to \eta\, \alpha(r), ()}$$

The rule (action) describes the evaluation of an event $\alpha(r)$ that consists in extending the current history with the event itself, and producing the unit value $()$.

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

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

$$\text{framing}_3 \frac{\eta^{-[]} \vDash \varphi}{C \vdash \eta, \overline{\varphi}[v] \to \eta]_\varphi, v}$$

The policy framing $\varphi[e]$ enforces the policy φ on the expression e, meaning that the history must respect φ at each step of the evaluation of e and each event issued within e must be checked against φ. More precisely, each resulting history η' must obey the policy φ (in symbols $\eta'^{-[]} \vDash \varphi$). When e is just a value, the security policy is simply removed (framing$_3$). As for the **with** rules, placing a bar over φ records that the policy is active. Also here, in the history the events $[_\varphi /]_\varphi$ record the point where we begin/end the enforcement of φ. Instead, if η' does not obey φ, then the computation gets stuck.

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

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

$$\text{receive} \frac{}{C \vdash \eta, \mathbf{receive}_\tau \to \eta\ receive_\tau, v}$$

The rules that govern communications reflect our notion of protocol, that abstractly represents the behaviour of the environment, showing the sequence of the pair direction/type of messages. Accordingly, our primitives carry types as tags, rather than dynamically checking the exchanged values. In particular, there is no check that the type of the received value matches the annotation of the primitive **receive**. Our static analysis in Subsection 5.4 will guarantee the correctness of this operation. More in detail, $\mathbf{send}_\tau(e)$ evaluates e (send$_1$) and sends the obtained value over the bus (send$_2$). In addition, 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 [85].

5.2 History Expressions

To statically predict the histories generated by programs at run-time, we introduce history expressions [87,20,18], 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 given in [18], extended to cover histories with a larger set of events ev, also endowing layer activation, dispatching and communication.

Definition 2 (History Expressions). *History expressions are defined by the following syntax:*

$H, H_1 ::=$	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, stands for* $[_\varphi \cdot H \cdot]_\varphi$		

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

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

Fig. 2. Transition system of History Expressions.

The signature defines sequentialisation, 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. 2.

Definition 3 (Semantics of History Expressions). *Given a closed history expression H (i.e. without free variables), its semantics $[\![H]\!]$ is the set of histories $\eta = w_1 \dots w_n$ $(w_i \in ev \cup \{\epsilon\}, 0 \leq i \leq n)$ 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. For instance, $\mu h.(\alpha(r) + \alpha'(r) \cdot h \cdot \alpha''(r))$ comprises all the histories of the form $\alpha'(r)^n \alpha(r) \alpha''(r)^n$, with $n \geq 0$.

Back to the example in Section 4, assume that H is the history expression over-approximating the behaviour of the function g. Then, assuming $\tau = ly_{\texttt{ACMode}}$, the history expression of the fragment of the Cloud service (Fig. 1(b)) is

$$\begin{aligned}
&\mu h.\varphi[\\
&\quad (\!|_{\texttt{Root}} \cdot receive_{\tau_{id}} \cdot \mathrm{cd}(/billing) \cdot \mathrm{write}(bid_id) \cdot \mathrm{cd}(../lib_{id}) \cdot |\!)_{\texttt{Root}} \cdot\\
&\quad (\!|_{\texttt{Usr}} \cdot receive_{\tau} \cdot receive_{\tau'} \cdot \varphi'[(\!|_{\texttt{ACMode}} \cdot H \cdot send_{\tau''}|\!)_{\texttt{ACMode}}]|\!)_{\texttt{Usr}}\\
&\quad] \cdot h
\end{aligned}$$

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 [20] 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 instance, if $\eta = \alpha(r)[_\varphi \alpha'(r)[_{\varphi'}[\alpha''(r)]_\varphi]_{\varphi'}$ then $\eta^{-[]} = \alpha(r)\alpha'(r)\alpha''(r)$. For details and other examples, see [20].

Given a history η, the multiset $ap(\eta)$ collects all the policies φ still active, i.e. whose scope has been entered but not exited yet. These policies are called *active policies* and are defined as follows:

$$\begin{aligned}
&ap(\epsilon) = \{\,\} && ap(\eta\,[_\varphi) = ap(\eta) \cup \{\varphi\}\\
&ap(\eta\, ev) = ap(\eta) \quad ev \neq [_\varphi,]_\varphi && ap(\eta\,]_\varphi) = ap(\eta) \setminus \{\varphi\}
\end{aligned}$$

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 Subsection 5.4.

$$\models \epsilon$$

$$\models \eta' ev \qquad \text{if } \models \eta' \text{ and } (\eta' ev)^{-[]} \models \varphi \text{ for all } \varphi \in ap(\eta' ev)$$

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

If a history is valid, also its prefixes are valid, i.e. validity is a prefix-closed property, as stated by the following lemma.

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

For instance, if the policy φ amounts to "no $read(r)$ after $write(r)$", the history $write(r)\varphi[read(r)write(r)]$ is not valid because $write(r)read(r) \not\models \varphi$ and remains not valid after, e.g. also $write(r)read(r)write(r) \not\models \varphi$. Instead, the history $\varphi[read(r)]write(r)$ is valid because both $\epsilon \models \varphi$, $read(r) \models \varphi$ and $read(r)write(r) \models \varphi$. The semantics of ContextML (in particular the rules for framing) ensures that the histories generated at runtime are all valid.

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

5.3 ContextML types

We briefly describe here our type and effect system for ContextML. We use it for over-approximating the program behaviour and for ensuring that the dispatching mechanism always succeeds at runtime. Here, we only give a logical presentation of our type and effect system, but we are confident that an inference algorithm can be developed, along the lines of [87]. All the technical properties that show the correctness of our type systems are detailed in [53]. Here, we only state the most intuitive results.

Our typing judgements have the form $\langle \Gamma; C \rangle \vdash e : \tau \triangleright H$. This reads as "in the type environment Γ and in the context C the expression e has type τ and effect H." The associated effect H is a history expression representing all the possible histories that a program may generate.

Types are integers, unit, layers and functions:

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

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

We annotate layer types with sets of layer names σ for analysis reason. In ly_σ, σ safely 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, such that each υ 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. a safe over-approximation of the sequence of events generated while evaluating the function.

The rules of our type and effect system are in Fig. 3. We show and comment in detail only the rules for the new constructs; the others are directly inherited

$$\text{(Tly)}\ \frac{}{\langle \Gamma; C\rangle \vdash L : ly_{\{L\}} \triangleright \epsilon}$$

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

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

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

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

$$\text{(Talpha)}\ \frac{}{\langle \Gamma; C\rangle \vdash \alpha(r) : \mathtt{unit} \triangleright \alpha(r)} \qquad \text{(Tphi)}\ \frac{\langle \Gamma; C\rangle \vdash e : \tau \triangleright H}{\langle \Gamma; C\rangle \vdash \varphi[e] : \tau \triangleright [_\varphi \cdot H \cdot]_\varphi}$$

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

Fig. 3. Typing rules

from that of ML. For the sake of simplicity, we also omit some auxiliary rules and the rules for subeffecting and for subtyping that can be found in [53].

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 preconditions $\upsilon \in \mathbb{P}$, we also guess a context C' that satisfies υ, i.e. that 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 those of the resulting function. In addition, 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 $\upsilon \in \mathbb{P}$ 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 the use of preconditions, consider the technical example in Fig. 4. There, the function $\mathbf{fun} f\ x \Rightarrow L_1.0$ is shown to have type $\mathit{int} \xrightarrow{\{L_1\}} \mathit{int}$ (for the sake of simplicity, we ignore the effects). This means that in order to apply the function, the layer L_1 must be active, i.e. must occur in the context.

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 the evaluation of the body. This evaluation is

$$\dfrac{\dfrac{\dfrac{\langle \Gamma, x : int, f : \tau \xrightarrow{\{|C'|\}} int; C' \rangle \vdash 0 : int \qquad L_1 \in C'}{\langle \Gamma, x : int, f : \tau \xrightarrow{\{|C'|\}} int; C' \rangle \vdash L_1.0 : int}}{\langle \Gamma; C \rangle \vdash \mathbf{fun} f\ x \Rightarrow L_1.0 : int \xrightarrow{\{|C'|\}} int} \qquad \langle \Gamma; C \rangle \vdash 3 : int \qquad |C'| \subseteq |C|}{\langle \Gamma; C \rangle \vdash (\mathbf{fun} f\ x \Rightarrow L_1.0)\, 3 : int}$$

Fig. 4. 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 ignore effects.

carried on the different contexts obtained by extending C with one of the layers in σ. The special events $(\!|_L$ and $|\!)_L$ mark the limits of layer activation.

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 rule (Talpha) gives expression $\alpha(r)$ type `unit` and effect $\alpha(r)$. In the rule (Tphi) the policy framing $\varphi[e]$ has the same type as e and $[_\varphi \cdot H \cdot]_\varphi$ as effect.

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 (Subsection 5.4).

The history expression H obtained as effect of an expression e safely over-approximates the set of histories η that may actually be generated during the execution of e. More formally:

Theorem 1 (Correctness).
If $\langle \Gamma; C \rangle \vdash e : \tau \triangleright H$ *and* $C \vdash \epsilon, e \to^* \eta, e'$, *then* $\eta \in [\![H]\!]$.

In [53] we also proved a more general result, long to state here: our type and effect system is *sound*. Its direct consequence is that a well-typed program may go wrong only because of policy violations or because the communication protocol is not respected. We take care of these two cases in the next subsection.

5.4 Model Checking

We discuss here our model-checking machinery for verifying whether a history expression is compliant with respect to a policy φ and a protocol P. To do this we will use the history expression obtained by typing the program.

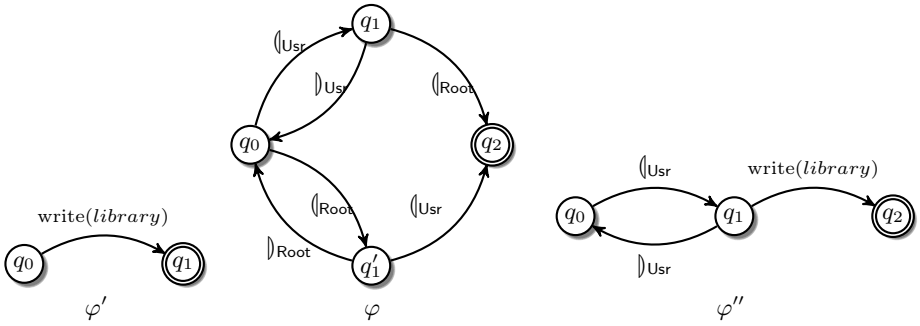

Fig. 5. Some examples of security policies. For clarity we omit the self loops in each state, that however are necessary to capture security unrelated events; they are labelled with those actions not in any outgoing edge.

Policy checking A policy φ is actually a safety property [61], expressing that nothing bad will occur during a computation. Policies are expressed through 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.

The policies φ', φ described in the example of Section 4 are in Fig. 5. The automaton for φ' expresses that writing in the library is forbidden. The one for φ describes as offending the traces that activate a layer Root (event $\langle_{\mathsf{Root}}$) while Usr is still active (after $\langle_{\mathsf{Usr}}$ but before $\rangle_{\mathsf{Usr}}$), or viceversa. The automaton for φ'' is an example of context-aware policy. In particular it states a role-based policy where writing in the library is forbidden when Usr is impersonated. Note that also φ'' is an infrastructural policy, just as it was φ in the example of Section 4.

We now complete the definition of validity of a history η, $\eta \vDash \varphi$, anticipated in Section 5.2. Recall also that a history expression is valid when all its histories are valid.

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

To check validity of a history expression we first need to solve a couple of technical problems. The first is because the semantics of a history expression may contain histories with nesting of the same policy framing. For instance, $H = \mu h.\,(\varphi[\alpha(r)h] + \epsilon)$ generates $[_\varphi \alpha(r)[_\varphi \alpha(r)]_\varphi]_\varphi$. This kind of nesting is *redundant* because the expressions monitored by the inner framings are already under the scope of the outermost one (in this case the second $\alpha(r)$ is already under the scope of the first φ). In addition, policies cannot predicate on the events of opening/closing a policy framing (because definition of validity in Subsection 5.2 uses $\eta^{-[]}$). More in detail, a history η has *redundant framing* whenever the active policies $ap(\eta')$ contain multiple occurrences of φ for some prefix η' of η. Redundant

framings can be eliminated from a history expression without affecting the validity of its histories. For example, we can rewrite H as $H{\downarrow} = \varphi[\mu h.\,(\alpha(r)h + \epsilon)] + \epsilon$ that does not generate histories with redundant framings. Given H, there is a *regularisation* algorithm returning his regularised 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 is valid [20]. Hence, checking validity of a history expression H can be reduced to checking validity of a history expression $H{\downarrow}$.

The second technical step makes a local policy to speak globally, by transforming it so to trap the point of its activations. Let $\{\varphi_i\}$ be the set of all the policies φ_i occurring in H. From each φ_i it is possible to obtain a *framed automaton* $\varphi_i^{[]}$ such that a history without redundant framings η is valid ($\models \eta$) if and only if $\eta \notin L(\bigcup \varphi_i^{[]})$. The detailed construction of framed automata can be found in [20] and roughly works as follows. The framed automaton for the policy φ consists of two copies of φ. The first copy has no offending states and is used when the actions are performed while the policy is not active. The second copy is reached when the policy is activated by a framing event and has the same offending states of φ. Indeed, there are edges labelled with $[_\varphi$ from the first copy to the second and $]_\varphi$ in the opposite direction. As a consequence, when a framing is activated, the offending states are reachable. Fig. 6 shows the framed automaton used to model check a simple policy φ_2 that prevents the occurrence of two consecutive actions α on the resource r. Clearly, the framed automaton only works on histories without redundant framing. Otherwise, it should record the number of nesting of policy framings to determine when the policy is active, and this is not a regular property.

Validating a regularised history expression H against the set of policies φ_i appearing therein amounts to verifying that $[\![H{\downarrow}]\!] \cap \bigcup L(\varphi_i^{[]})$ is empty. This procedure is decidable, since $H{\downarrow}$ is context-free [18,55], $\bigcup L(\varphi_i^{[]})$ is regular; context-free languages are closed by intersection with regular languages; and emptiness of context-free languages is decidable.

Note that our approach fits into the standard *automata-based* model checking [92]. Therefore there is an efficient and fully automata-based method for checking validity, i.e. $\models$ relation for a regularised history expression H [21].

Protocol compliance We are now ready to check whether a program will well-behave when interacting with other parties through the bus. The idea is that the

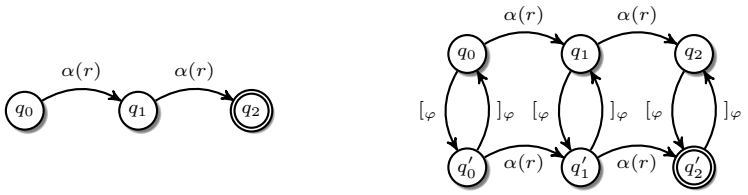

Fig. 6. On the left: a policy φ_2 that expresses that two consecutive actions α on r are forbidden. On the right: the framed automaton obtained from φ_2.

environment specifies P, and only accepts a user to join that follows P during the communication. We take a protocol P to be a sequence S of $send_\tau$ and $receive_\tau$ actions, possibly repeated (in symbols S^*), that designs the coordination interactions, as defined below:

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

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 specified interaction. The language $L(P)$ of P turns out to be a prefix-closed set of histories, obtained by considering all the prefixes of the sequences defined by P. Then we only require that all the histories generated by a program (projected so that only $send_\tau$ 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 $[\![H^{sr}]\!] \subseteq L(P)$.*

As for policy compliance, also protocol compliance can be established by using a decidable model checking procedure.

Note that, in our model, protocol compliance cannot be expressed only through the security policies introduced above. As a matter of fact, we have to check that H^{sr} does not include forbidden communication patterns, and this is a requirement much similar to a default-accept policy. Furthermore, we also need to check that some communication pattern in compliance with P *must* be done, cf. the check on the protocol compliance of the e-book reader program made in the example of Section 4.

References

1. Cloud cryptography group at Microsoft Research. `http://research.microsoft.com/en-us/projects/cryptocloud/`
2. UDDI technical white paper. Tech. rep., W3C (2000)
3. eXtensible Access Control Markup Language (XACML) Version 2.0. Tech. rep., OASIS (2005)
4. The future of cloud computing. Tech. rep., European Commision, Information Society and Media (2010)
5. 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)
6. Agrawal, R., Kiernan, J., Srikant, R., Xu, Y.: Order preserving encryption for numeric data. In: Proceedings of the 2004 ACM SIGMOD international conference on Management of data. pp. 563–574. SIGMOD '04, ACM, New York, NY, USA (2004), `http://doi.acm.org/10.1145/1007568.1007632`
7. Al-Neyadi, F., Abawajy, J.: Context-based e-health system access control mechanism. Advances in information security and its application pp. 68–77 (2009)

8. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services: Concepts, Architectures and Applications. Springer-Verlag (2004)
9. Amazon.com Inc.: Aws customer agreement. http://aws.amazon.com/agreement/
10. Amazon.com Inc.: Overview of Amazon Web Services. http://aws.amazon.com/whitepapers/ (2010)
11. Anderson, S., et al.: Web Services Trust Language (WS-Trust) (2005)
12. Andrews, T., et al.: Business Process Execution Language for Web Services (BPEL4WS), Version 1.1 (2003)
13. Appeltauer, M., Hirschfeld, R., Haupt, M., Masuhara, H.: ContextJ: Context-oriented programming with java. Computer Software 28(1) (2011)
14. Atkinson, B., et al.: Web Services Security (WS-Security) (2002)
15. Atzori, L., Iera, A., Morabito, G.: The internet of things: A survey. Computer Networks 54(15), 2787–2805 (2010)
16. Barbanera, F., Bugliesi, M., Dezani-Ciancaglini, M., Sassone, V.: Space-aware ambients and processes. Theor. Comput. Sci. 373(1-2), 41–69 (2007)
17. Barbanera, F., Dezani-Ciancaglini, M., Salvo, I., Sassone, V.: A type inference algorithm for secure ambients. Electr. Notes Theor. Comput. Sci. 62, 83–101 (2001)
18. Bartoletti, M., Degano, P., Ferrari, G.L.: Planning and verifying service composition. Journal of Computer Security 17(5), 799–837 (2009)
19. Bartoletti, M., Degano, P., Ferrari, G.L., Zunino, R.: Semantics-based design for secure web services. IEEE Trans. Software Eng. 34(1), 33–49 (2008)
20. Bartoletti, M., Degano, P., Ferrari, G.L., Zunino, R.: Local policies for resource usage analysis. ACM Trans. Program. Lang. Syst. 31(6) (2009)
21. Bartoletti, M., Zunino, R.: LocUsT: a tool for checking usage policies. Tech. Rep. TR-08-07, Dip. Informatica, Univ. Pisa (2008)
22. Bhargavan, K., Gordon, A.D., Narasamdya, I.: Service combinators for farming virtual machines. In: Lea, D., Zavattaro, G. (eds.) COORDINATION. Lecture Notes in Computer Science, vol. 5052, pp. 33–49. Springer (2008)
23. Blanchet, B.: Security protocol verification: Symbolic and computational models. In: POST. pp. 3–29 (2012)
24. Bodei, C., Dinh, V.D., Ferrari, G.L.: Safer in the clouds (extended abstract). In: Bliudze, S., Bruni, R., Grohmann, D., Silva, A. (eds.) ICE. EPTCS, vol. 38, pp. 45–49 (2010)
25. Bodei, C., Dinh, V.D., Ferrari, G.L.: Predicting global usages of resources endowed with local policies. In: Mousavi, M.R., Ravara, A. (eds.) FOCLASA. EPTCS, vol. 58, pp. 49–64 (2011)
26. Bonelli, E., Compagnoni, A., Gunter, E.: Typechecking safe process synchronization. In: Proc. Foundations of Global Ubiquitous Computing. ENTCS, vol. 138(1) (2005)
27. Boreale, M., et al.: SCC: a service centered calculus. In: WS-FM. Springer LNCS, vol. 4184 (2006)
28. Box, D., et al.: Simple Object Access Protocol (SOAP) 1.1. WRC Note (2000)
29. Box, D., et al.: Web Services Policy Framework (WS-Policy) (2002)
30. Braghin, C., Cortesi, A.: Flow-sensitive leakage analysis in mobile ambients. Electr. Notes Theor. Comput. Sci. 128(5), 17–25 (2005)
31. Braghin, C., Cortesi, A., Focardi, R.: Security boundaries in mobile ambients. Computer Languages, Systems & Structures 28(1), 101 – 127 (2002), http://www.sciencedirect.com/science/article/pii/S0096055102000097

32. Bravetti, M., Giusto, C.D., Pérez, J.A., Zavattaro, G.: Adaptable processes (extended abstract). In: Bruni, R., Dingel, J. (eds.) FMOODS/FORTE. Lecture Notes in Computer Science, vol. 6722, pp. 90–105. Springer (2011)
33. Brogi, A., Canal, C., Pimentel, E.: Behavioural types and component adaptation. In: Proc. Algebraic Methodology and Software Technology (AMAST). Springer LNCS, vol. 3116 (2004)
34. Bruni, R.: Calculi for service-oriented computing. In: Bernardo, M., Padovani, L., Zavattaro, G. (eds.) SFM. Lecture Notes in Computer Science, vol. 5569, pp. 1–41. Springer (2009)
35. Bruni, R., Corradini, A., Gadducci, F., Lluch-Lafuente, A., Vandin, A.: A conceptual framework for adaptation. In: de Lara, J., Zisman, A. (eds.) FASE. Lecture Notes in Computer Science, vol. 7212, pp. 240–254. Springer (2012)
36. Bucur, D., Nielsen, M.: Secure data flow in a calculus for context awareness. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models, Lecture Notes in Computer Science, vol. 5065, pp. 439–456. Springer Berlin / Heidelberg (2008)
37. Bugliesi, M., Castagna, G., Crafa, S.: Reasoning about security in mobile ambients. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR. Lecture Notes in Computer Science, vol. 2154, pp. 102–120. Springer (2001)
38. Caires, L., De Nicola, R., Pugliese, R., Vasconcelos, V.T., Zavattaro, G.: Core calculi for service-oriented computing. In: Results of the SENSORIA Project, pp. 153–188 (2011)
39. Campbell, R., Al-Muhtadi, J., Naldurg, P., Sampemane, G., Mickunas, M.D.: Towards security and privacy for pervasive computing. In: Proceedings of the 2002 Mext-NSF-JSPS international conference on Software security: theories and systems. pp. 1–15. ISSS'02, Springer-Verlag, Berlin, Heidelberg (2003), http://dl.acm.org/citation.cfm?id=1765533.1765535
40. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centred programming for web services. In: European Symposium in Programming Languages (ESOP). vol. 4421 (2007)
41. Cardelli, L., Gordon, A.: Mobile ambients. In: Nivat, M. (ed.) Foundations of Software Science and Computation Structures, Lecture Notes in Computer Science, vol. 1378, pp. 140–155. Springer Berlin / Heidelberg (1998), http://dx.doi.org/10.1007/BFb0053547, 10.1007/BFb0053547
42. Chen, G., Kotz, D.: A survey of context-aware mobile computing research. Tech. rep., Dartmouth College, Hanover, NH, USA (2000)
43. Chen, H., Finin, T., Joshi, A.: An ontology for context-aware pervasive computing environments. The Knowledge Engineering Review 18(03), 197–207 (2003)
44. Cheng, B.H.C., et al.: Software engineering for self-adaptive systems: A research roadmap. In: Software Engineering for Self-Adaptive Systems. Lecture Notes in Computer Science, vol. 5525, pp. 1–26. Springer (2009)
45. Chinnici, R., Gudgina, M., Moreau, J., Weerawarana, S.: Web Service Description Language (WSDL), Version 1.2 (2002)
46. Ciancia, V., Ferrari, G.L., Guanciale, R., Strollo, D.: Event based choreography. Sci. Comput. Program. 75(10), 848–878 (2010)
47. Clarke, D., Costanza, P., Tanter, E.: How should context-escaping closures proceed? In: International Workshop on Context-Oriented Programming. pp. 1:1–1:6. COP '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1562112.1562113

48. Clarke, D., Sergey, I.: A semantics for context-oriented programming with layers. In: International Workshop on Context-Oriented Programming. pp. 10:1–10:6. COP '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1562112.1562122
49. Costanza, P.: Language constructs for context-oriented programming. In: In Proceedings of the Dynamic Languages Symposium. pp. 1–10. ACM Press (2005)
50. Curbera, F., Khalaf, R., Mukhi, N., Tai, S., Weerawarane, S.: The next step in web services. Communications of the ACM 46(10) (2003)
51. De Nicola, R., Ferrari, G., Loreti, M., Pugliese, R.: A language-based approach to autonomic computing. In: Formal Methods for Component and Objects 2011. Lecture Notes in Computer Science (to appear), Springer (2012)
52. Degano, P., Ferrari, G.L., Galletta, L., Mezzetti, G.: Typing context-dependent behavioural variations. In: PLACES 2012. vol. to appear in EPTCS (2012)
53. Degano, P., Ferrari, G.L., Galletta, L., Mezzetti, G.: Typing for coordinating secure behavioural variations. In: Coordination Models and Languages. Lecture Notes in Computer Science, vol. 7274. Springer (2012)
54. 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)
55. Esparza, J.: Decidability of model checking for infinite-state concurrent systems. Acta Inf. 34(2), 85–107 (1997)
56. Ferrari, G., Guanciale, R., Strollo, D.: JSCL: A middleware for service coordination. In: Proc. FORTE. Springer LNCS, vol. 4229 (2006)
57. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proceedings of the 41st annual ACM symposium on Theory of computing. pp. 169–178. ACM (2009)
58. Greenfield, A.: Everyware: The dawning age of ubiquitous computing. Peachpit Press (2006)
59. Gu, T., Wang, X., Pung, H., Zhang, D.: An ontology-based context model in intelligent environments. In: Proceedings of communication networks and distributed systems modeling and simulation conference. vol. 2004, pp. 270–275 (2004)
60. Guidi, C., Lucchi, R., Gorrieri, R., Busi, N., Zavattaro, G.: SOCK: A calculus for service oriented computing. In: Proc. Service-Oriented Computing (ICSOC). Springer LNCS, vol. 4294 (2006)
61. Hamlen, K.W., Morrisett, J.G., Schneider, F.B.: Computability classes for enforcement mechanisms. ACM Trans. on Programming Languages and Systems 28(1), 175–205 (2006)
62. 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)
63. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. Journal of Object Technology, March-April 2008, ETH Zurich 7(3), 125–151 (2008)
64. 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. pp. 19–23. FOAL '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1960510.1960515
65. Honda, K., Vasconcelos, V., Kubo, M.: Language primitives and type discipline for structured communication-based programming. Programming Languages and Systems pp. 122–138 (1998)
66. Hulsebosch, R., Salden, A., Bargh, M., Ebben, P., Reitsma, J.: Context sensitive access control. In: Proceedings of the tenth ACM symposium on Access control models and technologies. pp. 111–119. ACM (2005)

67. IBM: An architectural blueprint for autonomic computing. Tech. rep. (2005)
68. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight java: a minimal core calculus for Java and GJ. ACM Trans. Program. Lang. Syst. 23(3), 396–450 (2001)
69. Kamina, T., Aotani, T., Masuhara, H.: Eventcj: a context-oriented programming language with declarative event-based context transition. In: Proceedings of the tenth international conference on Aspect-oriented software development. pp. 253–264. AOSD '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1960275.1960305
70. Kavantza, N., et al.: Web Service Coreography Description Language. http://www.w3.org/TR/ws-cdl-10/
71. Kelly, L.: The security threats of technology ubiquity. http://www.computerweekly.com/feature/The-security-threats-of-technology-ubiquity
72. Khalaf, R., Mukhi, N., Weerawarana, S.: Service oriented composition in BPEL4WS. In: Proc. WWW (2003)
73. Lapadula, A., Pugliese, R., Tiezzi, F.: A calculus for orchestration of web services. In: European Symposium in Programming Languages (ESOP). vol. 4421 (2007)
74. Lazovik, A., Aiello, M., Gennari, R.: Encoding requests to web service compositions as constraints. In: Proc. Principles and Practice of Constraint Programming (CP). Springer LNCS, vol. 3709 (2005)
75. Levi, F., Sangiorgi, D.: Mobile safe ambients. ACM Trans. Program. Lang. Syst. 25(1), 1–69 (2003)
76. Masi, M., Pugliese, R., Tiezzi, F.: Formalisation and implementation of the xacml access control mechanism. In: Barthe, G., Livshits, B., Scandariato, R. (eds.) ESSoS. Lecture Notes in Computer Science, vol. 7159, pp. 60–74. Springer (2012)
77. Misra, J.: A programming model for the orchestration of web services. In: 2nd International Conference on Software Engineering and Formal Methods (SEFM 2004) (2004)
78. Naehrig, M., Lauter, K., Vaikuntanathan, V.: Can homomorphic encryption be practical? In: Proceedings of the 3rd ACM workshop on Cloud computing security workshop. pp. 113–124. ACM (2011)
79. Papazoglou, M.P.: Service-oriented computing: Concepts, characteristics and directions. In: WISE (2003)
80. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-oriented computing: a research roadmap. Int. J. Cooperative Inf. Syst. 17(2), 223–255 (2008)
81. Papazouglou, M., Georgakopoulos, D.: Special issue on service oriented computing. Communications of the ACM 46(10) (2003)
82. Pelusi, L., Passarella, A., Conti, M.: Opportunistic networking: data forwarding in disconnected mobile ad hoc networks. Communications Magazine, IEEE 44(11), 134 –141 (2006)
83. Pfleeger, C., Pfleeger, S.: Security in computing. Prentice Hall (2003)
84. 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)
85. Sangiorgi, D., Walker, D.: The Pi-Calculus - a theory of mobile processes. Cambridge University Press (2001)
86. Schilit, B., Adams, N., Want, R.: Context-aware computing applications. In: In Proceedings of the Workshop on Mobile Computing Systems and Applications. pp. 85–90. IEEE Computer Society (1994)
87. Skalka, C., Smith, S., Horn, D.V.: Types and trace effects of higher order programs. Journal of Functional Programming 18(2), 179–249 (2008)

88. Stal, M.: Web services: Beyond component-based computing. Communications of the ACM 55(10) (2002)
89. Sweeney, L., et al.: k-anonymity: A model for protecting privacy. International Journal of Uncertainty Fuzziness and Knowledge Based Systems 10(5), 557–570 (2002)
90. Takabi, H., Joshi, J., Ahn, G.: Security and privacy challenges in cloud computing environments. Security & Privacy, IEEE 8(6), 24–31 (2010)
91. Vallecillo, A., Vansconcelos, V., Ravara, A.: Typing the behaviours of objects and components using session types. In: Proc. of FOCLASA (2002)
92. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification (preliminary report). In: LICS. pp. 332–344. IEEE Computer Society (1986)
93. Vitek, J., Castagna, G.: Seal: A framework for secure mobile computations. In: Bal, H.E., Belkhouche, B., Cardelli, L. (eds.) ICCL Workshop: Internet Programming Languages. Lecture Notes in Computer Science, vol. 1686, pp. 47–77. Springer (1998)
94. Vogels, W.: Web services are not distributed objects. IEEE Internet Computing 7(6) (2003)
95. Wang, X., Zhang, D., Gu, T., Pung, H.: Ontology based context modeling and reasoning using owl. In: Pervasive Computing and Communications Workshops, 2004. Proceedings of the Second IEEE Annual Conference on. pp. 18–22. Ieee (2004)
96. Weber, R.: Internet of things-new security and privacy challenges. Computer law & security review 26(1), 23–30 (2010)
97. Wirsing, M., Hölzl, M.M. (eds.): Rigorous Software Engineering for Service-Oriented Systems - Results of the SENSORIA Project on Software Engineering for Service-Oriented Computing, Lecture Notes in Computer Science, vol. 6582. Springer (2011)
98. 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)
99. Yang, M., Sassone, V., Hamadou, S.: A game-theoretic analysis of cooperation in anonymity networks. In: Degano, P., Guttman, J.D. (eds.) POST. Lecture Notes in Computer Science, vol. 7215, pp. 269–289. Springer (2012)
100. Zhang, G., Parashar, M.: Dynamic context-aware access control for grid applications. In: Grid Computing, 2003. Proceedings. Fourth International Workshop on. pp. 101–108. IEEE (2003)