Journal of Supercomputing manuscript No.
(will be inserted by the editor)

Programming in a Context-aware Language

Chiara Bodei · Pierpaolo Degano · Gian-Luigi Ferrari · Letterio Galletta

Received: date / Accepted: date

Abstract In the times of mobility and pervasiveness of computing, contextual information plays an increasingly crucial role in applications. This kind of information becomes a first class citizen in Context-Oriented Programming (COP) paradigm. COP languages provide primitive constructs for easily writing applications that adapt their behaviour depending on the evolution of their operational environment, namely the context. We present these new constructs, the issues and the challenges that arise, reporting on our recent work on $\mathrm{ML}_{\mathrm{CoDa}}$. It is a declarative language specifically designed for adaptation and equipped with a clear formal semantics and analysis tools. We will discuss some experiments done with a preliminary implementation of $\mathrm{ML}_{\mathrm{CoDa}}$. Through them we will show how applications and context interactions can be better specified, analysed and controlled.

Keywords Adaptive Software, Context-oriented Programming, Datalog

1 Introduction

In the time of ubiquitous and pervasive computing, software systems are required to operate *every time* and *everywhere*, always beside the users, possibly in a silent and invisible way. A typical ubiquitous scenario is that of Internet of Things (IoT), where any everyday object – a phone, a coffee machine, a street lamp, or a medical device – can be "smartified" by connecting it to a communication infrastructure. As a consequence, modern software systems should be able to interact and work together, by coping with changing conditions and highly dynamic operational environments, i.e. their *context* of use. Applications need mechanisms to sense the changes in the surrounding context, and to select the right responses, i.e. they need mechanisms that are *context-aware* and able to properly *adapt* their behaviour, with

C. Bodei · P. Degano · G.-L. Ferrari
Dipartimento di Informatica, Università di Pisa, Pisa, Italy
E-mail: {chiara,degano,gianqi}@di.unipi.it

L. Galletta
IMT - School for Advanced Studies, Lucca, Italy
E-mail: letterio.galletta@imtlucca.it

little or no user involvement. For example, if you switch the position of your mobile phone, you expect that the screen changes accordingly on its own. The phone perceives the world on behalf of the user.

At the same time, adaptation mechanisms must also maintain the functional and non-functional properties of applications, typically security or quality of service, that changes could compromise. Suppose, for instance, you are in a hotel and you want to join the wi-fi hotel network with your phone to check your bank account: you would like to connect in a secure way, but without bothering with all the details of the wireless connection, ideally in a fully transparent manner.

Contexts include any computationally accessible information that could be relevant, coming from both outside (e.g. sensor values, available devices, and code libraries offered by the environment), and from inside the application boundaries (e.g. its private resources, user profiles, etc.). In the literature, many different languages endowed with primitives that allow programmers to develop the mechanisms mentioned above have been proposed, e.g. [22,18,19,29,31,32] (a detailed discussion on the great deal of work in this area can be found in [28,14]).

In this field, Context Oriented Programming (COP) [1,10,16,17,2] provides programming adaptation primitives to support dynamic changes of behaviour, in reaction to changes in the context. This paradigm neatly separates the working environment from the application: programming adaptation is expressed using *behavioural variations*, chunks of code that can be automatically selected, depending on the context, dynamically modifying the application dynamic behaviour.

We contributed to COP programming languages, by introducing $\mathrm{ML}_{\mathrm{CoDa}}$ [11, 12,14,3], a core of ML with primitives for context-awareness. It presents two tightly integrated components: a declarative constituent for programming the context and a functional one for computing, that allows for separating the specific abstractions for describing contexts from those used for programming applications [27]. In $\mathrm{ML}_{\mathrm{CoDa}}$ a context is a knowledge base implemented as a (stratified, with negation) Datalog program [25,21]. The contents of a context can be inspected by simply querying it, in spite of the possibly complex deductions required.

The behavioural variations of $\mathrm{ML}_{\mathrm{CoDa}}$ offer a sort of pattern matching with Datalog goals as selectors. They are a first class, higher-order construct that can then be referred to by identifiers, and used as parameters in functions. This fosters dynamic, compositional adaptation patterns, as well as reusable, modular code. The selection of a goal is done by the *dispatching* mechanism that inspects the actual context and makes the right choices, by selecting the first expression whose goal holds. This choice depends on both the application code and on the "open" context, unknown at development time. If no alternative is viable at run time then a *functional failure* occurs, as the application cannot adapt to the current context. An application can also fail if it does not meet some requirements, e.g. about quality of service or security. In this case a *non-functional failure* occurs.

In the execution model of $\mathrm{ML}_{\mathrm{CoDa}}$ the context acts as an interface between each application it hosts and the system running it. Applications have a predefined set of APIs that provide handles to resources and operations to interact with the system. Also, they interact with each other via the context. The system and the applications may behave maliciously, by altering some parts of the context, e.g. one application can alter some parts of the context so driving another in an unsafe state; see a discussion on this and on non-functional failures in Section 5. We aim at detecting both functional and non-functional failures as early as possible, and

for this reason a two-phase static analysis has been proposed, one at compile and one at load-time [15,12,14,5], briefly summarised below. Functional failures are mainly due because applications operate in an open environment, and the actual value and even the presence of some elements in the current context are only known when the application is linked with it at run time. The first phase of our static analysis is based on a type and effect system that, at compile time, computes a safe over-approximation of the application behaviour, namely an *effect*. We use the effects at load-time to verify that the resources required by the application are available in the current context, and in the future ones. If an application passes this analysis, then no functional failure can arise at run time.

Besides the formal aspects, $\mathrm{ML_{CoDa}}$ lends itself to provide a single and fairly small set of constructs sufficient for becoming a practical programming language, as shown in [8]. $\mathrm{ML_{CoDa}}$ has been easily embedded in the real programming ecosystem .NET [24]. Being part of a well supported programming environment minimises the learning cost; lowers the complexity of deploying and maintaining applications; preserves compatibility with future extensions and with legacy code. The prototypical implementation of $\mathrm{ML_{CoDa}}$ in [8] extends the (ML family) functional language F#, without requiring any modification to the available compiler and to its runtime. The F# metaprogramming facilities, such as code introspection, quotation and reflection, have been exploited, as well as all the features provided by .NET, including a vast collection of libraries and modules. In particular, Just-In-Time mechanisms are used for compiling $\mathrm{ML_{CoDa}}$ constructs to .NET code. Consequently, $\mathrm{ML_{CoDa}}$ becomes a standard .NET library. The formal description of the language and its semantics, which highlight and explain how the two components of $\mathrm{ML_{CoDa}}$ interact, was crucial for driving the implementation. Moreover, they helped in identifying and describing the key parts of the implementation toolchain, compilation, generated code and runtime structures.

Here, we will discuss on some applications we developed in $\mathrm{ML_{CoDa}}$to assess our language, showing how context interactions can be better specified, analysed and controlled. We also discuss some extensions that will make our language more expressive and applicable. The next section introduces the main features of $\mathrm{ML_{CoDa}}$, with the help of our first case study. Two more case studies are summarised in Section 4. Section 3 shortly illustrates the Just-In-Time compiler of $\mathrm{ML_{CoDa}}$. We conclude in Section 5, where we discuss the planned extensions to our approach, in particular those required to handle many applications that run concurrently. A preliminary version of this paper appeared in [4].

2 A Walk through $\mathrm{ML_{CoDa}}$ on a Smart Home Scenario

We illustrate the main features of $\mathrm{ML_{CoDa}}$ by considering a typical scenario of the Internet of Things, namely a smart home, where a user exploits a tablet to interact with appliances. In particular, we assume that the capabilities of the tablet depend on its current location and on the profile of who is using it. Moreover, we show how the context represents both the physical and the virtual resources, and how programs can access and manage them through Datalog queries.

The interested reader can find further examples of $\mathrm{ML_{CoDa}}$ applications in the field of the Internet of Things in [8] and in [14]. The first simulates an e-healthcare system, where physicians use tablets to take care of patients, with

different capabilities depending on the context. The second example is about a multi-media guide for a smart museum that offers vistors a guided tour, based on their preferences.

Below, we first intuitively introduce the mechanisms for defining contexts, then those for adaptation, and finally we briefly discuss the new kind of run time errors that may arise in the COP paradigm.

Context description. We describe a small portion of the context, in particular we focus on that part concerning the user localization and profile, some virtual resources, like a mailbox, and physical ones, like a fridge. Basic data are represented by Datalog facts, and one can retrieve further information using the inference machinery of Datalog, which uses logical rules, also stored in the context.

For instance, as it is often the case, the tablet can be used by many users, who can perform different actions depending on their profiles. Assume to have two profiles (*i*) adult and (*ii*) kid. Thus, a user's profile is represented in the context by the binary fact `profile`. The following facts render that Alice and Jonh are adult, whereas Charlie is a kid:

```
profile("alice", "adult").
profile("charlie", "kid")
profile("john", "adult").
```

As a matter of fact, the user profile enables different applications and features: for example, Charlie (`kid`) can play videogames but he cannot access to his parents' office database; instead the desktop of an adult has an icon for the office database and mailbox, but none for videogames.

Furthermore, the tablet usage depends on its current location. For the sake simplicity, we suppose that the tablet is able to recognise three locations each providing access to the network: (*i*) office, (*ii*) home, and (*iii*) public spots. Information on the current location can be retrieved by querying the context, described by a set of Datalog clauses. In our case, from the following Datalog clauses we can deduce the user's local network:

```
local_network("office") :- wifi_connected("unipi").
local_network("home")   :- wifi_connected("myplace").
local_network("public")  :- wifi_connected(X).
```

These clauses state that the predicate on the left hand-side of the implication operator `:-` holds when the predicate in the right hand-side is true. When the tablet can connect to a specific network, the argument of the predicate `wifi_connected` is the network name and we use it to identifie the current location. For example, the tablet is in the office when it is connected to the *unipi* network. Whereas, when it is connected to a unknown network `X` the tablet is assumed to be in a public place, e.g., a bar.

As a matter of fact, the ML_{CoDa} context is quite expressive and can model fairly complex resources, both virtual and physical, and can manage the access to them in a declarative manner. Some of them depend on the current user's location and may be digital (a mailbox) or physical (a fridge or a television set); others are tightly coupled with the user profile.

The mailbox is a typical digital resource that can be used in different locations. However, the mailboxes could be different if the users are inside or outside their

workplaces. The following clauses prescribe that the only mailbox available inside the office is the professional one, whereas the personal mailbox is accessible outside.

```
mail_inbox(X) :- local_network("office"),
                 available("mailbox_office_server"),
                 inbox("mailbox_office_server", X).
mail_inbox(X) :- \+ local_network("office"),
                 available("my_mailbox_server"),
                 inbox("my_mailbox_server", X).
```

In the code above the predicate `available` holds when the tablet can connect to a given server and the variable `X` represents the mailbox returned by the server when the predicate `inbox` is true. In the second clause the operator `\+` denotes the logical *not*, a feature supported by our Datalog engine.

An example of resources that depend on the user's profile is the set of TV channels accessible at home: those for kids are usually different from those for their parents. The predicate `TV_chanel_list` retrieves the channels list `X` of a user through the following clauses:

```
TV_channel_list(X) :- profile("adult"), collection("adult", "TV", X).
TV_channel_list(X) :- profile("kid"), collection("kid", "TV", X).
```

where `collection` returns a playlist of a media device associated with a profile.

As an example of physical resources consider a fridge. As in a typical smart home scenario the user can retrieve the list of its content through the tablet. The ML_{CoDa} context can easily model this scenario with the clauses below:

```
current_fridge(Z, X) :-  profile("adult"), e_appliances(Z,Y),
                         fridge(X), is_in(X,Y).

contents(X,Y) :- profile("adult"), fridge(X), list(X,Y).
```

The first clause returns the identifier of the fridge `X` in the room `Z`. In particular, it checks if the user is an adult; it retrieves a list of all appliances `Y` inside `Z` (`e_appliances(Z,Y)`); and it returns `X` if it is a `fridge` occurring in `Y` (`is_in(X,Y)`). The second clause returns the list `Y` of items inside the fridge `X`.

Adaptation constructs. We now illustrate how one can express behavioural adaptation in our extended F#. The following *pay-per-view* scenario illustrates in more detail the context-dependent binding and behavioural variation constructs.

Suppose that users of our tablet can buy particular TV events on the fly, via a pay-TV provider. Each event has a unique identification number, which can be obtained by accessing the provider web page or by using a QR-code displayed in a suitable advertisement. Afterwards, to buy the event, the user can choose between paying through the web page or by sending a text message, including the event number in both cases.

In the following function `getTVEvent`, we declare the context-dependent variable `eventList` (called also *parameter* in ML_{CoDa}) to contain the list of available TV events.

```
1  let getTVEvent () =
2    let eventList =
3        let page = getPage()
4        extract(page)          |- wifi_connected("myplace")
5    in
6    let eventList =
7        let p = take_picture ctx?cam
8        decode_qr ctx?decoder p  |- use_qrcode(ctx?decoder), camera(ctx?cam)
9    in
10   selectEvent(eventList)
```

At lines 2-6 we declare by cases `eventList` using the syntax `let x = expression1 |- Goal [in] expression2`. The parameter is referrred to in line 10, and we can determine which value is bound to it only at run time, when it is known the context of use. If the goal at line 4 holds, then the tablet can directly connect and download the web page through the function `getPage()` to extract the required list. Otherwise, if the goal at line 8 is true, the tablet can take a picture of the QR-code and decodes it to get the list. In this second case, the goal contains two *goal variables* `ctx?decoder` and `ctx?cam`: if the query succeeds they will be assigned to the identifiers of the decoder and to that of the camera returned by the Datalog machinery. These identifiers are used by both the functions `take_picture` and `decode_qr` to interact with the actual tablet resources.

To change the program flow in accordance to the current context, we exploit behavioural variations. Syntactically, they have the form

```
match ctx with
| _ when !- Goal1 -> expression1
...
| _ when !- GoalN -> expressionN
```

where `match ctx with` explicitly refers to the context; the part `| _ when !- Goal` refers to the goal to solve; and `-> expression` is the sub-expression to evaluate when the goal holds. The execution triggers a so-called *dispatching mechanism* that queries the context and selects the first expression whose goal holds.

Consider the following function `buyTVEvent`, which allows a user to purchase the event `event_id`. As said, the payment can be performed via the web page or via a text message and it is implemented through a behavioural variation.

```
let buyTVEvent event_id price =
    match ctx with
    | _  when !-  payByWeb ->
         let c = getPage ()
         sendData c  event_id  usr_number
    | _  when !- payByText ->
        let c = getNumber () in
        sendText c  event_id  usr_number
    ;
    tell <| paidEvent(event_id, price)
```

Finally, besides queries, the interaction with the Datalog context may also consist of modifying the knowledge base on which the application performs deduction, by adding or removing facts with the `tell` and `retract` operations. For example, the last line of the function above records the event and the price to pay in the context through the predicate `paidEvent`. Similarly, the channel list of parents can be modified by the following expressions:

```
tell    <| TV_channel("adult", channel)
retract <| TV_channel("adult", channel)
```

where the predicate `TV_channel` holds if the specified channel is in the list of the user, provided that it is an adult.

Failures. It may happen that no goal is satisfied in a context while executing a behavioural variation or resolving a parameter. This means that the application in not able to adapt, either because the programmer assumed at design time the presence of functionalities that the current context lacks, or because of design errors. We refer to this new kind of run time errors as *adaptation failures*. For example, consider the function `getTVEvent` above in absence of wireless technology and, at the same time, of QR-decoder. Since no context will ever satisfy the goals in the definition of the parameter `eventList`, the current implementation throws a run time exception that can be handled as follows by standard exception mechanisms of F# (`try ... with e ->`)

```
let getTVEvent () =
  try
    let eventList =
        let page = getPage()
        extract(page)          |- wifi_connected("myplace")
    in
    let eventList =
        let p = take_picture ctx?cam
        decode_qr ctx?decoder p  |- use_qrcode(ctx?decoder), camera(ctx?cam)
    in
    selectEvent(eventList)
  with e ->  printfn "WARNING: cannot get the list of events"
```

As described in [12,14], it is possible to adopt a more sophisticated approach for statically determining whether the adaptation might fail and reporting it before running the application, based on a type and effect system, at compile time, coupled with a control flow analysis done at load time (see the concluding remarks).

3 The $\mathrm{ML}_{\mathrm{CoDa}}$ Compiler in a Nutshell

The $\mathrm{ML}_{\mathrm{CoDa}}$ compiler consists of two components that integrate the functional language F# with a customised version of YieldProlog[1] serving as Datalog engine.

The first component `ypc` compiles ahead-of-time each Datalog predicate into a .NET method. At run time, the generated method computes the assignments of values to variables of the predicate that make it true. This compilation stragegy supports the interaction and the data exchange between the application and the context in a fully transparent way. Note that the *impedance mismatch problem* [20] does not arise since the .NET type system is uniformly used everywhere.

The second component leverages the standard F# compiler to deal with $\mathrm{ML}_{\mathrm{CoDa}}$ adaptation primitives. This has been implemented through just-in-time compilation driven by annotation provided by the programmer. In practice, a programmer annotates the code using these primitives with *custom attributes*, the most important of which is `CoDa.Code`. These annotations prevent the F# compiler from

[1] Available at `https://github.com/vslab/YieldProlog`

generating .NET bytecode but make it to output a symbolic representation in form of *quotation* [9]. When a quoted piece of code is about to be executed, the $\mathrm{ML}_{\mathrm{CoDa}}$ runtime is invoked to transform it into bytecode using the .NET meta-programming facilities. The generated bytecode is stored in an internal cache by the $\mathrm{ML}_{\mathrm{CoDa}}$ runtime, and can be re-used when the piece of code is executed again. `CoDa.Code` is an alias for the standard `ReflectedDefinitionAttribute` that marks modules and members whose abstract syntax trees are used at run time through reflection. The specific $\mathrm{ML}_{\mathrm{CoDa}}$ operations, typically the adaptation constructs and those for interacting wityh the context, are only allowed in methods marked with `CoDa.Code`; otherwise an exception is raised when they are invoked.

Note that the compiler `fsharpc` does not require any change. This is because the operations needed to support the adaptation primitives are fully handled by our runtime support.

4 Evaluation of $\mathrm{ML}_{\mathrm{CoDa}}$ Implementation

In this section we illustrate the effectiveness of our $\mathrm{ML}_{\mathrm{CoDa}}$ implementation through two case studies of small-sized applications. The first is the context-aware editor FSEdit. It allowed us to test and evaluate how our implementation deals and interacts with existent code, in particular with the standard IO and GUI library of .NET. Furthermore, it also permitted us to identify some programming patterns that may be considered as idiomatic of $\mathrm{ML}_{\mathrm{CoDa}}$ programs (see below). The second one is a small rover robot based on a Raspberry PI [26]. It is endowed with wheels, an engine control and sensors through which can move around and recognise objects in the environment. The goal of this case study was to test the portability of our implementation on hardware typical of IoT solutions.

FSEdit editor. We have implemented FSEdit, a context-aware text editor. It supports three different execution modes: *rich text editor*, *text editor* and *programming editor*. A context switch among the different modes changes the GUI, by offering different tool-bars and menus. In the first mode, the GUI allows users to set size and face of the font; to change the colour of text; and to adjust the alignment of the paragraphs. In the second mode, the editor becomes very minimalistic and allows the user to edit pure text files, where no information of the text formatting can change. Finally, in the programming mode, the editor shows file line numbers and provides a simple form of syntax highlighting for source files.

The predicates below show that the context of FSEdit includes the current execution mode and other information that directly depend on it.

```
tokens(TS) :- tokens_(TS), execution_mode(programming).

file_dialog_filter(F) :- execution_mode(M),
                         file_dialog_filter_(F,M).
```

For instance, the first predicate only holds in the programming mode; it also returns the keywords of the programming language chosen by the user for syntax highlighting. Currently the editor supports the C programming language only. The kind of files supported by the editor in the different modes (e.g. *.rtf files in rich text mode, *.txt files in text mode and *.c in programming mode), represents another possible information concerning the context.

The execution mode impacts on the behaviour of FSEdit, as can be shown in the following piece of code. There, if the editor is in the right mode, when the user changes the text, the syntax highlighter procedure is invoked.

```
let textChanged (rt : RichTextBox) = // dlet
  let def_behaviour () = ...      // code not shown
  let f_body = def_behaviour () |- True    // Basic behaviour
  let f_body  = (f_body ; syntaxHighlighter rt) |-
                                  execution_mode("programming")
f_body
```

Observe in the snippet above the interesting and idiomatic use of context dependent binding. This coding pattern is recurring in the code of FSEdit whenever a programmer needs to add new behaviour to a basic one. In particular, the first definition of the identifier `f_body` represents the basic behaviour of the editor that is independent of the context, while the second definition extends the basic behaviour with the features that are to be provided when the editor is in programming mode. Note also the use of `f_body` on the right-hand side in the fourth line of the snippet. It is not a recursive definition: it is instead an invocation to `f_body` defined in the previous line, i.e. the one that specifyies the basic behaviour of the editor.

Fsc-Rover. We now briefly illustrate the implementation of a small rover robot developed on the Raspberry Pi hardware platform [26] by Riccardo Rolla, a master student of our research group (see `https://github.com/riccardorolla/rpi-iot-fscoda`). The rover has two wheels, an engine, a distance sensor, photo and video cameras. It moves around autonomously, and detects objects and some of their features. To perform its task, the rover interacts and exchanges messages on the Internet, using `REST` APIs. Actually, it uses Microsoft Cognitive Service [23] to recognise objects and it communicates with users through Telegram messages [30]. In particular, through Telegram a user can order the rover to execute some actions, called *remote*. Typical such actions include asking the robot to share the pictures it has collected in the chat, and to move in a prescribed direction. Besides these remote actions, the rover can perform its own actions, called *local*, e.g., those required to avoid an obstacle.

Regardless of their local or remote kind, the actions the rover can carry out are stored in the context as Datalog facts. For instance, the following snippet of code shows how we store in the context the information about remote actions to take a photo and a video:

```
tell <| Fsc.Facts.usrcmd("photo","/rpi/photo")
tell <| Fsc.Facts.cmddesc("/rpi/photo","snapshot a photo with camera")
tell <| Fsc.Facts.usrcmd("video","/rpi/video")
tell <| Fsc.Facts.cmddesc("/rpi/video","shoot a movie with camera")
```

Differently from the other case studies this formal executable specification of a rover makes it evident that the context provides effective support to uniformly handle both local and remote activities. As a matter of fact, the components of the rover have been implemented in different languages. For example, the driver of the wheels is written partly in C and partly in C#, while the logical control is implemented in ML_{CoDa}. Clearly, the context plays here the role of communication infrastructure, through which the different components interact. Crucial to the implementation of the rover has been therefore the interoperability feature of ML_{CoDa}, which is based on .NET.

As expected, the control loop of the rover, displayed below, repeats the following until *no-request* is found.

- Add to the rover program all the remote actions read from the context;
- Execute asynchronously local and remote actions;
- Collect and process data and store the results in the context;
- Send responses to remote applications.

```
while (not (get_detected "exit")) do
   for _ in !-- request(ctx?idchat,ctx?cmd) do
          array_cmd <- array_cmd |> Array.append [|ctx?cmd|]
   for _ in !-- next(ctx?cmd) do
          array_cmd <- array_cmd |> Array.append [|ctx?cmd|]
   listresult <- Async.Parallel
            [for c in  array_cmd -> execute c]
                  |> Async.RunSynchronously
   for r in listresult do
       match r with
        |cmd,res -> for _ in !-- result(cmd,ctx?out) do
                    retract <| Fsc.Facts.result(cmd, ctx?out)
                    tell <| Fsc.Facts.result(cmd, res)
....
    match ctx with
    | _ when !- (request(ctx?idchat,ctx?cmd),result(ctx?cmd,ctx?out))
                -> do
                    let result=send_message ctx?idchat ctx?cmd)
                    retract <| Fsc.Facts.request(ctx?idchat, ctx?cmd)
    | _ ->  printfn "no request"
....
   run()
```

The query `request(ctx?idchat,ctx?cmd)` extracts information from the context to assemble the list of commands to be executed. This is done by checking for messages arriving at the context from the Internet. The tag `idchat` identifies a remote application. Note that both rover commands and results are modelled as suitable facts inside the context through the **`tell`** and **`retract`** operations. The function `run` sets up the context, e.g. it turns on/off the video camera and the distance sensor. Its code, not displayed here, also invokes a configuration function that sets the sequence of local actions.

The behaviour of the rover also depends on the *obstacles* identified by the camera in the current environment. The following function is used to detect the nature of the obstacles by inspecting the context.

```
let infoimage = get_out "discovery" |> imagerecognition
    for tag in infoimage.tags do
        discovery tag.name tag.confidence
    for _ in !-- recognition(ctx?obj,ctx?value) do
```

The idea is that the objects are suitable facts in the context and that object recognition in the current image depends on the parameters of confidence of objects, such as size, rotation, etc.

Finally, this case study turned out to be useful also to tune our implementation. In particular, the cache for the bytecode generated by the just-in-time compiler improved the performance of the code implementing the control logic. The response time dropped from a few seconds to few milliseconds.

5 Conclusions, Discussion and Open Problems

After a brief survey of the main features of the COP language $\mathrm{ML}_{\mathrm{CoDa}}$, we have reported on some experiments on its usage. We considered some real case studies, although admittedly simplified in some details. These experiments helped us in assessing the effectiveness of our adaptation primitives and of their implementations in F#. They also showed that the expressive power of $\mathrm{ML}_{\mathrm{CoDa}}$ is adequate for supporting the design of real, non trivial applications. In particular, we found that its component based on Datalog is easy to use and powerful enough to describe involved contexts. Indeed, the smart home case study of Section 2 showed that Datalog predicates easily encode not only data but also complex business logic rules involving both physical and virtual resources. Furthermore, we found that the bipartite nature of $\mathrm{ML}_{\mathrm{CoDa}}$ permits developers to clearly separate the design of the context from that of the application, yet maintaining their interrelationships. This is particularly evident in the rover case study of Section 4, where the context provides the mechanisms to abstract from the communication infrastructure, thus making the code that controls the rover fully independent from the actual features of that infrastructure. Finally, our compilation strategy resulted in an implementation that is portable and interoperable with existent F# and .NET libraries. Actually, the rover case study of Section 4 proved that we can easily develop $\mathrm{ML}_{\mathrm{CoDa}}$ applications that run on hardware architectures different from standard PCs. Moreover, the FSedit case study of Section 4 showed that our adaptation constructs integrated well with standard .NET libraries, as no glue code was required. However, our case studies also revealed some limitations of our current prototypical implementation. Indeed, further functionalities are required to make $\mathrm{ML}_{\mathrm{CoDa}}$ more effective. Below we discuss some lines of improvement, both pragmatic and theoretical.

Non-functional properties. Context-aware security and privacy is a crucial aspect in pervasive IoT systems. For example, implementing the e-healthcare system discussed in [8], we found it necessary to protect patients' data and to regulate the usage of the hospital equipment by doctors and users through suitable access control policies. We approached both these problems within a linguistic point of view. Therefore, we extended $\mathrm{ML}_{\mathrm{CoDa}}$ with constructs to express security policies and with mechanisms for checking and enforcing them [5]. Actually, security policies are expressed just as logical predicates and their enforcement exploits the deduction machinery of Datalog. In addition, controlling safety properties, like access control or other security policies, only requires lightweight modifications to the knowledge base and to its management. Also we have devised a mechanism for instrumenting the code of an application, so as to incorporate in it an *adaptive reference monitor*, ready to stop the execution when a policy is about to be violated. We also extended the static analysis mentioned in Section 1 to identify the risky operations that may lead to a violation. Note that the execution context and the security policies therein are only known when the application is about to run, thus our static analysis can detect the potentially dangerous actions only at load-time, *not* at compile time. Thus, when an application enters a new context, the results of the static analysis are used to suitably drive the invocation of the reference monitor that is switched on and off depending on whether the action about to occur is risky or not. In this way we avoid redundant invocations.

There are also other non-functional properties that are of interest in real applications, and thus worth investigating. Typical example are properties concerned with quality of service. They requires enriching both our logical knowledge base and our applications with quantitative information, first of all with time. These extension are also useful for evaluating the performance of both applications and of contexts. For instance, on the one hand statistical information about performance can help in choosing the most suitable application among those functionally equivalent. On the other hand, contexts that guarantee a better performance can be suggested to the users depending on the statistical information on the usage of contexts or reliability of resources therein.

We also intend to enlarge our investigation to resources, an important and critical aspect of contexts, starting from the resource-aware model in [6].

Coherency of the context. Handling the context and keeping it coherent is another important issue, provided that coherency does not hinder adaptation. It is indeed possible that, e.g. some application e can complete its task even in a context that became partially incoherent. This is very relevant in practice. Suppose for instance that a resource becomes unavailable in the context, but was usable by e when a specific behavioural variation started. Our current approach prevents e even to start running, thus guaranteeing the absence of adaptation errors at run time. Nevertheless, it is too restrictive, because it precludes to run an application that only uses that resource when available, e.g. at the very beginning. A possible but expensive solution consists in resorting to a continuous run time monitoring. In general, finding a sound and efficient solution to this issue is pretty hard from a theoretical point of view. Indeed, solutions are obtained by carefully mixing static analysis and run time monitoring of the applications that are executing in a context. Furthermore, "living in an incoherent context" is tightly connected with how one deals with the needed recovery mechanisms that should be activated without involving the users.

Concurrency. Similar problems arise when we consider concurrent systems. Actually, it turns out that every context-aware application is inherently concurrent. Applications are not isolated when performing their tasks and usually they need the resources offered by a context where plenty of other applications are running and can compete for the same resources. An example of concurrency issues can be found in our implementation of the rover, described in Section 4, where the control activity of the robot is performed in parallel with the collection and analysis of data coming from the context. The current *ad hoc* solution is not yet fully integrated in $\mathrm{ML}_{\mathrm{CoDa}}$ and relies on the native threads of the operative system.

In [13], $\mathrm{ML}_{\mathrm{CoDa}}$ is extended to deal with concurrency, by having two threads: the context and the application. The first thread virtualises the resources, the communication infrastructure, and other software running within it. In the second thread, the interactions of the application with the other entities are rendered as asynchronous events that represent the relevant changes in the context. As a consequence, all the interactions of the context with the entities it hosts are abstractly described by its updates. More faithfully representations of concurrency require to explicitly describing the applications that execute in a context, exchange information using it and that asynchronously update it, as done in [7]. This approach leads however to the well-known problem of thread-interference, since one

thread, in updating the context, may make some resources unavailable or may contradict assumptions that another thread relies upon. The classical techniques and mechanisms, like locks, cannot solve these issues, in an open world where applications appear and disappear unpredictably, and freely update the context. Since designers are only aware of the relevant fragments of the context and cannot anticipate the effects a change may have, the overall consistency of the context cannot be controlled by applications, and "living in an incoherent context" cannot be avoided. This problem is addressed by the semantics introduced in [7], by using a run time verification mechanism. On the one hand, the effects of the running applications are checked to be sure that the execution of the selected behavioural variation will lead no other application to an inconsistent state, e.g. by disposing a shared resource. On the other hand, the other threads are checked to guarantee that they are harmless to the application entering in a behavioural variation.

Recovery mechanisms. Above, we briefly mentioned the need of specific recovery mechanisms when adaptation failures arise at run time that prevent an application from completing its task. These mechanisms are especially needed to adapt applications that raise security failures, in case of policy violations. Since recovery should be carried out without involving the user too much, the system running the applications must execute parts of their code "atomically." A typical way consists of introducing constructs that allow marking parts of code as all-or-nothing transactions, and to store auxiliary information to support the recovering mechanism. In case the entire transaction is successfully executed, the auxiliary information can be disposed; otherwise it is used to restore the application in the previous consistent state, e.g. the one at the start of the transaction. It would be interesting to investigate recovery mechanisms for behavioural variations that allow the user to undo some actions considered risky or sensible, and that force the dispatching mechanism to make different, alternative choices. However, in a concurrent setting the context might have been changed in the meanwhile, and it might not be consistent any longer. A deep analysis is thus needed to understand the interplay between the usage of contextual information by the application, and the highly dynamic way in which contexts change. A future line of investigation can be based on not requiring a coherent *global* context, but requiring only coherent portions of it, i.e. *local* contexts where applications run and stay for a while.

References

1. Appeltauer, M., Hirschfeld, R., Haupt, M., Lincke, J., Perscheid, M.: A comparison of context-oriented programming languages. In: International Workshop on Context-Oriented Programming (COP '09), pp. 6:1–6:6. ACM, New York, NY, USA (2009)
2. Appeltauer, M., Hirschfeld, R., Haupt, M., Masuhara, H.: ContextJ: Context-oriented programming with Java. Computer Software **28**(1), 272–292 (2011)
3. Bodei, C., Degano, P., Ferrari, G.L., Galletta, L.: Last mile's resources. In: Semantics, Logics, and Calculi, LNCS 9560, pp. 33–53. Springer (2016)
4. Bodei, C., Degano, P., Ferrari, G.L., Galletta, L.: Experimenting with a context-aware language. In: V. Malyshkin (ed.) Proc. of 14th Conference on Parallel Computing Technologies, LNCS 10421, pp. 3–17. Springer (2017)
5. Bodei, C., Degano, P., Galletta, L., Salvatori, F.: Context-aware security: Linguistic mechanisms and static analysis. Journal of Computer Security **24**(4), 427–477 (2016)
6. Bodei, C., Dinh, V.D., Ferrari, G.L.: Checking global usage of resources handled with local policies. Sci. Comput. Program. **133**, 20–50 (2017)

7. Busi, M., Degano, P., Galletta, L.: A semantics for disciplined concurrency in COP. In: Proceedings of the ICTCS 2016, CEUR Proceedings 1720, pp. 177–189 (2016)
8. Canciani, A., Degano, P., Ferrari, G.L., Galletta, L.: A context-oriented extension of F#. In: FOCLASA 2015, EPTCS 201, pp. 18–32 (2015)
9. Code Quotation: https://docs.microsoft.com/en-Us/dotnet/fsharp/language-reference/code-quotations
10. Costanza, P.: Language constructs for context-oriented programming. In: Proc. of the Dynamic Languages Symposium, pp. 1–10. ACM Press (2005)
11. Degano, P., Ferrari, G.L., Galletta, L.: A two-component language for COP. In: Proc. 6th International Workshop on Context-Oriented Programming. ACM Digital Library (2014)
12. Degano, P., Ferrari, G.L., Galletta, L.: A two-phase static analysis for reliable adaptation. In: Proc. of 12th International Conference on Software Engineering and Formal Methods, LNCS 8702, pp. 347–362. Springer (2014)
13. Degano, P., Ferrari, G.L., Galletta, L.: Event-driven adaptation in COP. In: PLACES 2016, EPTCS 211, pp. 37–45 (2016)
14. Degano, P., Ferrari, G.L., Galletta, L.: A two-component language for adaptation: design, semantics and program analysis. IEEE Transactions on Software Engineering, 10.1109/TSE.2015.2496941. pp. 505–529 (2016). DOI 10.1109/TSE.2015.2496941
15. Galletta, L.: Adaptivity: linguistic mechanisms and static analysis techniques. Ph.D. thesis, University of Pisa (2014). http://www.di.unipi.it/~galletta/phdThesis.pdf
16. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. Journal of Object Technology, March-April 2008 **7**(3), 125–151 (2008)
17. Kamina, T., Aotani, T., Masuhara, H.: EventCJ: a context-oriented programming language with declarative event-based context transition. In: Proc. of the 10 international conference on Aspect-oriented software development (AOSD '11), pp. 253–264. ACM (2011)
18. Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. IEEE Computer **36**(1), 41–50 (2003). DOI 10.1109/MC.2003.1160055
19. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: An Overview of AspectJ. In: J. Knudsen (ed.) ECOOP 2001 — Object-Oriented Programming, LNCS 2072, pp. 327–354. Springer (2001)
20. Lämmel, R., Meijer, E.: Revealing the X/O impedance mismatch - (changing lead into gold). In: R.C. Backhouse, J. Gibbons, R. Hinze, J. Jeuring (eds.) Datatype-Generic Programming - International Spring School, *LNCS*, vol. 4719, pp. 285–367. Springer (2007)
21. Loke, S.W.: Representing and reasoning with situations for context-aware pervasive computing: a logic programming perspective. Knowl. Eng. Rev. **19**(3), 213–233 (2004)
22. Magee, J., Kramer, J.: Dynamic structure in software architectures. SIGSOFT Softw. Eng. Notes **21**(6), 3–14 (1996)
23. Microsoft Cognitive Service: https://azure.microsoft.com/en-us/services/cognitive-services/
24. Microsoft .NET: https://www.microsoft.com/net/
25. Orsi, G., Tanca, L.: Context modelling and context-aware querying. In: O. Moor, G. Gottlob, T. Furche, A. Sellers (eds.) Datalog Reloaded, LNCS 6702, pp. 225–244. Springer (2011)
26. Raspberry Pi : https://www.raspberrypi.org/
27. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst. **4**(2), 14:1–14:42 (2009)
28. Salvaneschi, G., Ghezzi, C., Pradella, M.: Context-oriented programming: A software engineering perspective. Journal of Systems and Software **85**(8), 1801–1817 (2012)
29. Spinczyk, O., Gal, A., Schröder-Preikschat, W.: AspectC++: An aspect-oriented extension to the C++ programming language. CRPIT '02, pp. 53–60. Australian Computer Society, Inc. (2002)
30. Telegram: https://telegram.org/
31. Walker, D., Zdancewic, S., Ligatti, J.: A Theory of Aspects. SIGPLAN Not. **38**(9), 127–139 (2003)
32. Wand, M., Kiczales, G., Dutchyn, C.: A Semantics for Advice and Dynamic Join Points in Aspect-oriented Programming. ACM Trans. Program. Lang. Syst. **26**(5), 890–910 (2004)