Experimenting with a Context-aware Language

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

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

Abstract. Contextual information plays an increasingly crucial role in concurrent applications in the times of mobility and pervasiveness of computing. Context-Oriented Programming languages explicitly treat this kind of information. They provide primitive constructs to adapt the behaviour of a program, depending on the evolution of its operational environment, which is affected by other programs hosted therein independently and unpredictably. We discuss these issues and the challenges they pose, reporting on our recent work on $\mathrm{ML}_{\mathrm{CoDa}}$, a language specifically designed for adaptation and equipped with a clear formal semantics and analysis tools. We will show how applications and context interactions can be better specified, analysed and controlled, with the help of some experiments done with a preliminary implementation of $\mathrm{ML}_{\mathrm{CoDa}}$.

1 Introduction

Today there is a growing trend in having software systems able to operate *every time* and *everywhere*, and applications are working side by side, either in a cooperative or in a competitive way. Ubiquitous and pervasive computing scenarios are typical of the Internet of Things (IoT), a cyber-physical communication infrastructure, made of a wide variety of interconnected and possibly mobile devices. As a consequence, modern software systems have to cope with changing operational environments, i.e. their *context*. At the same time, they must never compromise their intended behaviour and their non-functional requirements, typically security or quality of service. Thus, programming languages need effective mechanisms to become *context-aware*, so as to detect the changes in the context where the application is plugged in, and to properly *adapt* to them, with little or no user involvement. Accordingly these mechanisms must maintain the functional and non-functional properties of applications after the adaptation steps. For example, suppose you want to have just a quick look at your mail and at your social platforms when in a hotel: 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.

The context is crucial for adaptive software and typically it includes different kinds of computationally accessible information 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.). The literature proposes many different programming languages that support dynamic adjustments and tuning of programs,

e.g. [20,17,18,24,25,26] (a detailed discussion on the great deal of work in this area is in [23,13]). In this field, Context Oriented Programming (COP) [9,15,16,1] offers a neat separation between the working environment and the application. Indeed, the COP linguistic paradigm explicitly deals with contexts, by providing programming adaptation mechanisms to support dynamic changes of behaviour, in reaction to changes in the context (see [2,23] for an overview). In this paradigm, programming adaptation is specified using *behavioural variations*, chunks of code that can be automatically selected depending on the current context hosting the application, dynamically modifying its execution.

To address adaptivity we defined $\mathrm{ML}_{\mathrm{CoDa}}$ [10,11,13,3], a core of ML with COP features. It has two tightly integrated components: a declarative constituent for programming the context and a functional one for computing. The bipartition reflects the separation of concerns between the specific abstractions for describing contexts and those used for programming applications [22]. The context in $\mathrm{ML}_{\mathrm{CoDa}}$ is a knowledge base implemented as a (stratified, with negation) Datalog program [21,19]. Applications inspect the contents of a context by simply querying it, in spite of the possibly complex deductions required. The behavioural variations of $\mathrm{ML}_{\mathrm{CoDa}}$ are 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. Note that the choice depends on both the application code and the "open" context, unknown at development time. If no alternative is viable then a *functional failure* occurs, as the application cannot adapt to the current context. *Non-functional failures* are also possible, when the application does not meet some requirements, e.g. about quality of service or security.

The execution model of $\mathrm{ML}_{\mathrm{CoDa}}$ assumes that the context is the interface between each application it hosts and the system running it. Applications interact with the system using a predefined set of APIs that provide handles to resources and operations on them. Also, they interact with each other via the context. The system and the applications do not trust each other, and may act maliciously, e.g. one application can alter some parts of the context so driving another in an unsafe state. The application designer would like to detect both functional and non-functional failures as early as possible, and for that $\mathrm{ML}_{\mathrm{CoDa}}$ has a two-phase static analysis, one at compile and one at load-time [14,11,13,4], briefly summarised below. The static analysis takes care of failures in adaptation to the current context (*functional failures*), dealing with the fact that applications operate in an "open" environment. Indeed, 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*. Then the effect is used at load time to verify that the resources required by the application are available in the actual context, and in its future modifications. To do that, the effect of the application

is suitably combined with the effect of the APIs provided by the context that are computed by the same type and effect system. If an application passes this analysis, then no functional failure can arise at run time. The results of the static analysis also drive an instrumentation of the original code, so as to monitor its execution and block dangerous activities [4].

In addition to the formal aspects of $\mathrm{ML}_{\mathrm{CoDa}}$, a main feature of our approach is that a single and fairly small set of constructs is sufficient enough for becoming a practical programming language, as shown in [7]. $\mathrm{ML}_{\mathrm{CoDa}}$ can easily be embedded in a real programming eco-system as .NET, so preserving compatibility with future extensions and with legacy code developed within this framework. Being part of a well supported programming environment minimises the learning cost and lowers the complexity of deploying and maintaining applications. In [7] a prototypical implementation of $\mathrm{ML}_{\mathrm{CoDa}}$ is presented as an extension of the (ML family) functional language F#. Indeed, no modifications at all were needed to the available compiler and to its runtime. The F# metaprogramming facilities are exploited, such as code introspection, quotation and reflection, as well as all the features provided by .NET, including a vast collection of libraries and modules. In particular, we used the Just-In-Time mechanism for compiling to native code. As a consequence, $\mathrm{ML}_{\mathrm{CoDa}}$ is implemented as a standard .NET library. In the path towards the implementation a crucial role has been played by the formal description of the language and by its formal semantics, which highlight and explain how the two components of $\mathrm{ML}_{\mathrm{CoDa}}$ interact. Furthermore they helped in identifying and describing the crucial parts of the implementation toolchain, compilation, generated code and runtime structures.

Here, we will survey on some applications we developed in $\mathrm{ML}_{\mathrm{CoDa}}$ 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 $\mathrm{ML}_{\mathrm{CoDa}}$, with the help of our first case study. Two more case studies are summarised in Section 3. Section 4 shortly illustrates the Just-In-Time compiler of $\mathrm{ML}_{\mathrm{CoDa}}$. In Section 5 we conclude and discuss the planned extensions, in particular those required to handle many applications running concurrently.

2 A First Example: an e-Healthcare System

Here, we illustrate the main features of $\mathrm{ML}_{\mathrm{CoDa}}$ by considering an e-healthcare system with a few aspects typical of the Internet of Things. A more detailed description of this case study is in [7], and its full executable definition is in `https://github.com/vslab/fscoda`.

In our scenario each physician can retrieve a patient's clinical record using a smartphone or a tablet, which also tracks the current location. Got the relevant data, the doctor decides which exams the patient needs and the system helps scheduling them. In addition, the system checks whether the doctor has the competence and the permission to actually perform the required exam, otherwise it suggests another physician who is enabled to, possibly coming from another

department. Moving from a ward to another, the operating context changes and allows the doctor to access the complete clinical records of the patients therein. The application must adapt to the new context and it may additionally provide different features, e.g. by disabling rights to use some equipment and by acquiring access to new ones. Indeed, location-awareness of devices is exploited to tune access policies.

The e-healthcare context. We consider below a small part of the context, in particular that for storing and making some data available about the doctors' location, information on their devices, the patients' records and the ward medical equipment. Some 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 example, the fact that Dr. Turk is in the cardiology ward is rendered as

```
physician_location("Dr. Turk" "Cardiology").
```

The following inference rule permits to deduce that the clinical data of patients can be accessed by the doctors in the same department where patients are. It states that the predicate on the left hand-side of the implication operator `:-` holds when the conjunction of the predicates (`physician_location` and `patient_location`) in the right hand-side yields true, i.e. when the physician and patient's location coincide.

```
physician_can_view_patient(Physician, Patient) :-
        physician_location(Physician, Location),
        patient_location(Patient, Location).
```

The $\mathrm{ML}_{\mathrm{CoDa}}$ context is quite expressive and can model fairly complex situations. Typically, some medical exams can only be performed after some others. To compute this list of exams, all the dependencies among them are to be considered. This could be expressed by the following recursive rules:

```
patient_needs_result(Patient, Exam) :-
        patient_has_been_prescribed(Patient, Exam).

patient_needs_result(Patient, Exam) :-
        exam_requirement(TargetExam, Exam),
        patient_needs_result(Patient, TargetExam).
```

The first rule means that the prescription of an exam implies that the involved patient needs the results of the test. The second rule says that whenever a patient needs an exam, so are also needed all the screenings the exam depends on. Datalog can conveniently model recursive relations like the dependency among exams, which may require involved queries with standard relational databases.

The next rule dictates that a patient has to do an exam if the two clauses in the right hand-side are true. The first has been already discussed above, while the second clause says that a patient should *not* do an exam if its results are already known (in the rule below the operator `\+` denotes the logical *not*, dealt with in our version of Datalog [8]).

```
patient_should_do(Patient, Exam) :-
        patient_needs_result(Patient, Exam),
        \+ patient_has_result(Patient, Exam).
```

In addition, we can declaratively describe physical objects in quite a similar, homogeneous manner. The following (simplified) rule specifies when a device can display a certain exam, by checking whether it has the needed capabilities:

```
device_can_display_exam(Device, Exam) :-
    device_has_caps(Device, Capability),
    exam_view_caps(Exam, Capability).
```

By listing a set of facts, we can easily assert the capabilities of a device, e.g.

```
device_has_caps('iPhone 5', '3D acceleration').
device_has_caps('iPhone 5', 'Video codec').
device_has_caps('iPhone 5', 'Text display').
device_has_caps('Apple Watch', 'Text display').
```

Adaptation constructs. Now we focus on context-dependent bindings and behavioural variations. These adaptation constructs allow specifying program behaviour, which depends on the context in our e-healthcare system. When entering a ward, the patients' records under treatment can be displayed on the doctor's personal device. Moreover, the e-healthcare system computes the list of the clinical exams a patient should do and that the doctor can perform. The following code (in a F#-like syntax) shows how the adaptation constructs are used to implement these functionalities. The `display` function, given a doctor `phy` and a patient `pat`, prints the information about the patient's exams on the screen.

```
let display phy pat =
  match ctx with
  | _ when !- physician_can_view_patient(phy, pat) ->
    match ctx with
    | _ when !- patient_has_result(pat, ctx?e) ->
      printfn "%s sees that %s has done:" phy pat
      for _ in !-- patient_has_result(pat, ctx?exam) do
        display_exam phy ctx?exam
    | _ ->
      printfn "%s sees that %s has done no exam" phy pat

    let next_exam = "no exam" |- True
    let next_exam = ctx?exam |-
                      (physician_exam(phy, ctx?exam),
                       patient_active_exam(pat, ctx?exam))
    printfn "%s can submit %s to %s" phy pat next_exam
  | _ ->
    printfn "%s cannot view details on %s" phy pat
```

Behavioural variations change the program flow according to the current context. They have the form `match ctx with | _ when !- Goal -> expression`, where the sub-expression `match ctx with` explicitly refers to the context; the part

`| _ when !- Goal` introduces the goal to solve; and `-> expression` is the sub-expression to evaluate when the goal is true.

Using the outermost behavioural variation (starting at line 2), we check whether the doctor `phy` is allowed to access the data of the patient `pat`, when the goal `physician_can_view_patient(phy, pat)` at line 3 holds.

With the nested behavioural variation (line 4), we check if the patient has got the results of some exams, using the predicate `patient_has_result`. If this is the case, the **`for`** construct extracts the list of exam results from the context (line 7). The statement **`for _ in`** `!-- Goal` **`do`** `expression` iterates the evaluation of `expression` over all the solutions of the `Goal`. It works as an iterator on-the-fly, driven by the solvability of the goal in the context. The predicate `patient_has_result` at line 7 contains the *goal variable* `ctx?exam`: if the query succeeds, at each iteration `ctx?exam` is bound to the current value satisfying `Goal`. A goal variable is introduced in a goal, defining its scope, using the syntax `ctx?var_name`.

Finally, through **`let`** `x = expression1 |- Goal [`**`in`**`] expression2` (the context dependent binding), the function `display` shows an exam that the physician `phy` can do on the patient `pat`. At lines 12-13 we declare by cases the parameter `next_exam`, referred to in line 16. Only at run time when the actual context is known, we can determine which case applies and which value will be bound to `next_exam` when the parameter is used. If the goal in lines 14-15 holds, then `next_exam` assumes the value retrieved from the context, otherwise it gets the default value `"no exam"`.

Note that 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 classify this new kind of runtime errors as *adaptation failures*. For example, the following function assumes that given the identifier of a physician, it is always possible to retrieve the physician's location from the context using the `physician_location` predicate:

```
let find_physician phy =
  let loc = ctx?location |-
              physician_location(phy, ctx?location) in
  loc
```

The context-dependent binding may find no solution for the goal, e.g. when `find_physician` is invoked on a physician whose location is not in the context. If this is the case, the current implementation throws a runtime exception.

```
let find_physician phy =
  try
    let loc = ctx?location |-
                  physician_location(phy, ctx?location) in  loc
  with e ->  printfn "WARNING: cannot locate %s:\n%A" phy e
                  "unknown location"
```

As described in [11], we may adopt a more sophisticated approach where for statically determining whether the adaptation might fail and reporting it before running the application.

Finally, the interaction with the Datalog context is not limited to queries: it is possible indeed to program the modifications to the knowledge base on which it performs deduction, by adding or removing facts with the **tell** and **retract** operations, as in:

```
tell <| patient_has_result("Jordan", "CT scan")
```

Some execution examples. We now show how the functions defined above give different results when invoked in different contexts, parts of which are only described intuitively. For instance, in a context where Dr. Turk is not in the same ward as Bob, the result of the invocation `display "Dr. Turk""Bob"` is `Dr. Turk cannot view details on Bob`. This is because physicians are only allowed to see data about the patients in the department where they are. Indeed, the behavioural variation introduced at line 3 on `physician_can_view_patient` finds out that accessing data is not allowed. If instead Dr. Cox is in the same department where Bob is, the call `display "Dr. Cox""Bob"` correctly prints the details about Bob (actually stored in the Datalog knowledge base):

```
Dr. Cox sees that Bob has done no exam
Dr. Cox can submit Bob to Blood test
```

In this case the outermost behavioural variation (starting at line 2) confirms that Dr. Cox can view the data. The nested one (starting at line 4), driven by `patient_has_result`, finds no exam for Bob, hence the function displays the no-exam message (line 10). Furthermore, the program finds out that Dr. Cox could do a blood test on Bob, as he is enabled to; then it additionally finds out that Bob needs no pre-screening and so that exam can be done immediately, because the predicate at line 15 holds.

Suppose now to have a slightly more complex situation, in which the context itself is modified. Patient Jordan has already performed an EEG test, and doctors prescribed her a CT and nothing else. Dr. Kelso is in Jordan's room, is enabled to do only CT tests and carries a device on which he can visualise the results. In this context, the invocation `display "Dr. Kelso""Jordan"` outputs

```
Dr. Kelso sees that Jordan has done:  EEG
Dr. Kelso can submit Jordan to CT scan
```

Differently from the case above, Jordan has already performed an exam, listed by the iteration construct. Once Dr. Kelso have performed a CT scan on Jordan, the context has to be accordingly changed, by asserting the fact

```
tell <| patient_has_result("Jordan", "CT scan")
```

Now the query `display "Dr. Kelso""Jordan"` has a different output in the modified context: besides displaying a longer list of exam results, the application shows Dr. Kelso that Jordan needs him to perform no other exam:

```
Dr. Kelso sees that Jordan has done:  EEG, CT scan
Dr. Kelso can submit Jordan to no exam
```

Suppose now that Dr. Cox moves to Jordan room and checks her medical report, but he has a device that cannot show CT images. The `display_exam` function warns the doctor and possibly presents the results in a more limited form, e.g. a static thumbnail. So, the result of the query `display "Dr. Cox""Jordan"` is

```
Dr. Cox sees that Jordan has done: EEG, CT scan
                (current device cannot display the exam data)
Dr. Cox can submit Jordan to no exam
```

3 Further Case Studies

The following case studies illustrate how ML_{CoDa} can be used to specify small-sized real context-aware applications. Afterwards, we outline some internals of our preliminary compiler.

Fsc-Rover. We now briefly describe the implementation in ML_{CoDa} of a small rover robot, endowed with two wheels, engine control, foto and video camera and a distance sensor, done by Riccardo Rolla, a master student of our research group (see `https://github.com/riccardorolla/rpi-iot-fscoda`). The rover moves in a building, and detects the objects therein and some of their features. Also, it interacts with other applications that use the information it collects, by exchanging messages on the Internet. The rover can perform either its actions, called *local*, or actions issued by other applications, called *remote*. Each kind of action has a different set of parameters and the rover has to identify the right values for them, by inspecting the properties of the objects in the context.

Besides getting a formal executable specification of a rover, differently from the other case studies this one makes it evident that the context provides effective support to uniformly handle both local and remote activities. The control loop of the rover, shown below, is really quite standard: it 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. 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.

```
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
```

FSEdit editor. Here, we briefly survey the implementation of FSEdit, a context-aware text editor implemented in ML_{CoDa}. This case study was a workbench for testing how our implementation deals and interacts with pure F# code, in particular against the standard GUI library provided by .NET. Besides playing with contexts, this case study also helped finding some little flaws in the way our compiler treated some pieces of code using the object oriented features of F#. Furthermore, it also allowed us to identify some programming patterns that may be considered as idiomatic of ML_{CoDa} programs (see below).

The editor supports three different execution modes: *rich text editor*, *text editor* and *programming editor*. A context switch among the different modes changes the GUI of the editor, by offering e.g. different tool-bars and menus.

In the first mode, the GUI allows the user to set the size and the face of a font; to change the color 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 C source files.

The context of FSEdit contains the current execution mode and other information that directly depend on it, as shown by the predicates below:

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

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

For example, the predicate `tokens` only holds in the programming mode and returns the keywords of the programming language selected by the user to perform syntax highlighting. For simplicity, the editor currently supports the C programming language only. The second piece of information is about the kind of files supported by the editor in the different modes. For instance, *.rtf files in rich text mode, *.txt files in text mode and *.c in programming mode.

As said before, the execution mode affects the behaviour of the editor. For instance, in the following piece of code we invoke the syntax highlighter procedure when the user changes the text, if the editor is in the right mode.

```
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
```

As anticipated, this code snippet is interesting because it shows an idiomatic use of the context dependent binding. Indeed, there are two definitions of the identifier `f_body`: the first one represents the basic behaviour of the editor that is independent of the context; the second one extends the basic behaviour with the features that are to be provided when the editor is in the programming mode. Notice in particular the use of `f_body` on the right-hand side in the last-but-one line of the snippet. Although it may seem a recursive definition it is not; it is instead an invocation of `f_body` defined in the previous line, i.e. the one specifying the basic behaviour of the editor.

4 A Glimpse on $\mathrm{ML}_{\mathrm{CoDa}}$ Compiler

The $\mathrm{ML}_{\mathrm{CoDa}}$ compiler `ypc` is based on the integration of the functional language F# with a customised version of YieldProlog[1] serving as Datalog engine.

Our compiler ahead-of-time compiles each Datalog predicate into a .NET method, whose code enumerates one by one the solutions, i.e. the assignments of values to variables that satisfy the predicate. In this way, the interaction and

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

the data exchange between the application and the context is fully transparent to the programmer because the .NET type system is uniformly used everywhere.

The functional part of ML_{CoDa} that extends F# is implemented through just-in-time compilation. To do that, a programmer annotates these extensions with *custom attributes*, among which the most important is `CoDa.Code`. When a function annotated by it is to be executed, the ML_{CoDa} runtime is invoked to trigger the compilation step. Since the operations needed to adapt the application to contexts are transparently handled by our runtime support, the compiler `fsharpc` works as it is. Actually, `CoDa.Code` is an alias for the standard `ReflectedDefinitionAttribute` that marks modules and members whose abstract syntax trees are used at runtime through reflection. Of course ML_{CoDa} specific operations are only allowed in methods marked with this attribute; otherwise an exception is raised when they are invoked.

5 Conclusions, Discussion and Open Problems

We have surveyed the COP language ML_{CoDa} and we have reported on the experiments carried on some case studies. These proved ML_{CoDa} expressive enough to support the designer of real applications, although admittedly simplified in some details. The formal description of the dynamic and the static semantics of ML_{CoDa} drove a preliminary implementation of a compiler and of an analysis tool. Especially, we found that the bipartite nature of ML_{CoDa} permits the designer to clearly separate the design of the context from that of the application, yet maintaining their inter-relationships. This is particularly evident in the rover case study of Section 3, where the context provides the mechanism to virtualise and abstract from the communication infrastructure, thus making the logic of the control of the rover fully independent from the actual features of the communication infrastructure.

At the same time, the people working with ML_{CoDa} asked for more functionalities to make ML_{CoDa} more effective, and below we discuss some lines of improvement, both pragmatic and theoretical.

Non-functional properties. A crucial aspect that arose when designing the e-healthcare system concerns context-aware security and privacy, which we approached still from a formal linguistic viewpoint. We equipped ML_{CoDa} with security policies and with mechanisms for checking and enforcing them [4]. It turns out that policies are just Datalog clauses and that enforcing them reduces to asking goals. As a matter of fact, the control of safety properties, like access control or other security policies, requires extensions to the knowledge base and to its management that are not too heavy. We also extended the static analysis mentioned above to identify the operations that may violate the security policies in force. Recall that this step can only be done at load time, because the execution context is only known when the application is about to run, and thus our static analysis cannot be completed at compile time. Yet we have been able to instrument the code of an application, so as to incorporate in it an *adaptive reference monitor*, ready to stop executions when a policy to be enforced is

about to be violated. When an application enters a new context, the results of the static analysis mentioned above are used to suitably drive the invocation of the monitor that is switched on and off upon need.

Further work will investigate other non-functional properties that however are of interest in real applications. A typical example is quality of service, which requires enriching both our logical knowledge base and our applications with quantitative information, *in primis* time. Such an extension would also provide the basis for evaluating both applications and contexts. For instance, statistical information about performance can help in choosing the application that better fits our needs, as well as statistical information on the usage of contexts or reliability of resources therein can be used for suggesting the user the context that guarantees more performance. A further approach to statically reason about resource usage, typically acquisition and release, is in [5].

Coherency of the context and interference. Other issues concern the context, in particular the operations to handle it and to keep it coherent. When developing and testing the e-healthcare system discussed in Section 2, it was necessary to extend the Datalog deduction machinery in order to get the entire list of the solutions to a given query.

Pursuing coherency at any cost can instead hinder adaptation, e.g. when an application e can complete its task even in a context that became partially incoherent. This is a pragmatically very relevant issue. Consider for example the case when a resource becomes unavailable, but was usable by e in the context when a specific behavioural variation started. At the moment we implemented our language in a strict way that prevents e even to start executing. For sure this guarantees that no troubles will arise, but also precludes to run an application that only uses that resource when available, e.g. at the very beginning, and never again, so no adaptation error will show up at run time. While a continuous run time monitoring can handle this problem, but at a high cost, finding a sound and efficient solution to this issue is a hard challenge from a theoretical point of view. Indeed, it involves a careful mix of static analysis and of run time monitoring of the applications that are executing in a context. Also, "living in an incoherent context" is tightly connected with the way one deals with the needed recovery mechanisms that should be activated without involving the users.

Concurrency. The above problem is critical in the inherently concurrent systems we are studying. Indeed, an application does not perform its task in isolation, rather it needs some resources offered by a context where plenty of other applications are running therein (often competing for those resources). For instance, the implementation of the rover described in Section 3 posed concurrency issues, because 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 exploits the management of threads offered by the operative system, and it is not yet fully integrated in $\mathrm{ML}_{\mathrm{CoDa}}$.

A first extension of $\mathrm{ML}_{\mathrm{CoDa}}$ with concurrency is in [12], where there is a two-threaded system: the context and the application. The first virtualises the

resources and the communication infrastructure, as well as other software components running within it. Consequently, the behaviour of a context, describing in particular how it is updated, abstractly accounts for all the interactions of the entities it hosts. The other thread is the application and the interactions with the other entities therein are rendered as the occurrence of asynchronous events that represent the relevant changes in the context. A more faithful description of concurrency requires to explicitly describing the many applications that execute in a context, that exchange information using it and that asynchronously update it. This is the approach followed in [6]. Nonetheless, the well known problem of interference now arises, because one thread can update the context possibly making unavailable some resources or contradicting assumptions that another thread relies upon. Classical techniques for controlling this form of misbehaviour, like locks, are not satisfying, because they contrast with the basic assumption of having an open world where applications appear and disappear unpredictably, and freely update the context. However, application designers are only aware of the relevant fragments of the context and cannot anticipate the effects a change may have. Therefore, the overall consistency of the context cannot be controlled by applications, and "living in an incoherent context" is unavoidable. The semantics proposed in [6] addresses this problem using a run time verification mechanism. Intuitively, the effects of the running applications are checked to guarantee that the execution of the selected behavioural variation will lead no other application to an inconsistent state, e.g. by disposing a shared resource. Dually, also the other threads are checked to verify that they are harmless with respect to the application entering in a behavioural variation.

Recovery mechanisms. We already mentioned briefly the need of recovery mechanism when run time errors arise, in particular when adaptation failures prevent an application to complete its task. Recovery mechanisms are especially needed to adapt applications that raise security failures, in case of policy violations. Recovery should be carried on with little or no user involvement, and this imposes on the system running the applications to execute parts of their code "atomically." A typical way is to consider those pieces of code as all-or-nothing transactions, and to store auxiliary information for recovering from failures. If the entire transaction is successfully executed, then the auxiliary information can be disposed, otherwise it has to be used to restore the application in a consistent state, e.g. the one holding at the start of the transaction. To this end, we plan to investigate recovery mechanisms appropriate for behavioural variations, to allow the user to undo some actions considered risky or sensible, and force the dispatching mechanism to make different, alternative choices.

However, in our world the context might have been changed in the meanwhile, and such a state might not be consistent any longer. A deep analysis is therefore needed of the interplay between the way applications use contextual information to adapt or to execute, and the highly dynamic way in which contexts change. A possible line of investigation can be giving up with the quest for a coherent *global* context, while keeping coherent portions of it, i.e. *local* contexts where applications run and, so to speak, posses for a while.

References

1. Appeltauer, M., Hirschfeld, R., Haupt, M., Masuhara, H.: ContextJ: Context-oriented programming with Java. Computer Software 28(1) (2011)
2. 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)
3. Bodei, C., Degano, P., Ferrari, G.L., Galletta, L.: Last mile's resources. In: Semantics, Logics, and Calculi. LNCS 9560, Springer (2016)
4. 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)
5. Bodei, C., Dinh, V.D., Ferrari, G.L.: Checking global usage of resources handled with local policies. Sci. Comput. Program. 133, 20–50 (2017)
6. Busi, M., Degano, P., Galletta, L.: A semantics for disciplined concurrency in COP. In: Proceedings of the ICTCS 2016. pp. 177–189. CEUR Proceedings 1720 (2016)
7. Canciani, A., Degano, P., Ferrari, G.L., Galletta, L.: A context-oriented extension of F#. In: FOCLASA 2015. pp. 18–32. EPTCS 201 (2015)
8. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about datalog (and never dared to ask). IEEE Trans. on Knowl. and Data Eng. 1(1), 146–166 (1989)
9. Costanza, P.: Language constructs for context-oriented programming. In: Proc. of the Dynamic Languages Symposium. pp. 1–10. ACM Press (2005)
10. 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)
11. 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. pp. 347–362. LNCS 8702, Springer (2014)
12. Degano, P., Ferrari, G.L., Galletta, L.: Event-driven adaptation in COP. In: PLACES 2016. EPTCS 211 (2016)
13. 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. (2016)
14. Galletta, L.: Adaptivity: linguistic mechanisms and static analysis techniques. Ph.D. thesis, University of Pisa (2014), `http://www.di.unipi.it/~galletta/phdThesis.pdf`
15. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. Journal of Object Technology, March-April 2008 7(3), 125–151 (2008)
16. 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)
17. Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. IEEE Computer 36(1), 41–50 (2003)
18. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: An Overview of AspectJ. In: Knudsen, J. (ed.) ECOOP 2001 — Object-Oriented Programming, pp. 327–354. LNCS 2072, Springer (2001)

19. 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)
20. Magee, J., Kramer, J.: Dynamic structure in software architectures. SIGSOFT Softw. Eng. Notes 21(6), 3–14 (Oct 1996)
21. Orsi, G., Tanca, L.: Context modelling and context-aware querying. In: Moor, O., Gottlob, G., Furche, T., Sellers, A. (eds.) Datalog Reloaded, pp. 225–244. LNCS 6702, Springer (2011)
22. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst. 4(2), 14:1–14:42 (2009)
23. Salvaneschi, G., Ghezzi, C., Pradella, M.: Context-oriented programming: A software engineering perspective. Journal of Systems and Software 85(8), 1801–1817 (2012)
24. Spinczyk, O., Gal, A., Schröder-Preikschat, W.: Aspectc++: An aspect-oriented extension to the c++ programming language. pp. 53–60. CRPIT '02, Australian Computer Society, Inc. (2002)
25. Walker, D., Zdancewic, S., Ligatti, J.: A Theory of Aspects. SIGPLAN Not. 38(9), 127–139 (Aug 2003)
26. 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 (Sep 2004)