A semantics for disciplined concurrency in COP[⋆]

Matteo Busi, Pierpaolo Degano, and Letterio Galletta

Dipartimento di Informatica, Università di Pisa
m.busi@studenti.unipi.it, {degano,galletta}@di.unipi.it

Abstract. A concurrent extension of the recent COP language $\mathrm{ML}_{\mathrm{CoDa}}$ is presented. We formalise its operational semantics and we propose a run time verification mechanism that enforces a notion of non-interference among concurrent threads. More precisely, this mechanism prevents an application from modifying the context so as to dispose some resources or to contradict assumptions upon which other applications rely.

1 Introduction

Modern software have to run in highly dynamic and open heterogeneous environment, often referred as the context. The context abstracts the communication infrastructure and the available resources, so as to make them seem less heterogeneous, unlimited and fully dedicated to their users. Programming these systems thus requires new programming language features and effective mechanisms to deal with context-awareness, i.e. sensing the context, reacting and properly adapting the program behaviour to changes of the actual context. Recently the last two authors proposed $\mathrm{ML}_{\mathrm{CoDa}}$ a two-component Context-oriented Programming (COP) language [10]. It has a logical constituent for specifying and manipulating the context and a functional one for computing. Separation of concerns drove the design choices. Indeed, one specifies the context and its evolution, using its own specific mechanisms and rules, that are typically different from those used in programming applications. Designing a context requires skills different from those needed for applications, and it is usually programmed by requirements engineers [19]. The declarative approach allows requirements engineers to express *what* information the context has to include, leaving to the virtual machine *how* this information is actually collected and managed.

In $\mathrm{ML}_{\mathrm{CoDa}}$ a context is a Datalog knowledge base [15]. Thus, verifying whether a given property holds in the context simply consists in querying a Datalog goal. During the needed deductions the relevant information is also retrieved. The

⋆ Partially supported by the Università di Pisa PRA_2016_64 Project *Through the fog.*

choice of a functional language was driven by the popularity of this paradigm (see e.g. F#, Scala), by its formal elegance and the conciseness of its programs. The first mechanism takes care of those program variables that assume different values depending on the different properties of the current context. The notion of *context-dependent binding* makes that explicit. The second one extends standard *behavioural variations*, that are chunks of code that are dynamically activated depending on the context, so adapting the behaviour of the program. In ML_{CoDa} behavioural variations are a first-class construct, so they can be referred to by identifiers, and passed to and returned by functions. This helps in programming dynamic adaptation patterns, as well as reusable and modular code. Also, a behavioural variation can be supplied by the context, and then composed with existing ones.

We study these aspects from a basic point of view, in [10] a single application within a context was considered. ML_{CoDa} was equipped with an operational semantics which provided us with the basis for a prototypical implementation in F# [5]. Since adaptive applications may misbehave because at design time an unknown environment was not considered, a static analysis ensures that this kind of run time errors never occur, e.g. because the actual hosting environment lacks a required capability. The analysis is performed in two phases: a type and effect system (*at compile time*) and a Control Flow Analysis (*at load time*). Type-checking a program also computes an *effect* that over-approximates the capabilities required by an application at run time. When entering a new context, *before* running the program, this abstraction is exploited to check that the actual context, and those resulting from its evolution, support the capabilities required by the application. Note that this last analysis can only be done at load time, because at compile time the possible hosting contexts are still unknown.

A first extension of ML_{CoDa} with concurrency is in [11], where there is a two-threaded system: the context and the application. The first virtualizes 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. The semantics of [11] also offered a way of preventing a context change to affect the validity of a choice of a behavioural variation. Also a recovery mechanism is triggered at need.

Here we extend this approach by explicitly describing many applications that execute in a context, and possibly exchange information through it and asynchronously update it. 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.

The novelty of the semantics proposed here consists in addressing this problem through a run time verification mechanism. We assume our applications be typed as in [10], and the resulting effect, called *history expression* is carried on by the code. Roughly, a history expression collects the sequence of context modifications that an application may perform, as well as the Datalog goals it queries. Intuitively, the effects of the running applications are checked to make 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. Dually, also the other threads are checked to verify that they cause no harm to the application entering in a behavioural variation. Differently than in [10], the verification is not done at load time, but it occurs at run time when a behavioural variation is about to be evaluated. All the checks sketched above are performed on the effects computed at compile time using the Control Flow Analysis of [10]. Note that performing the verification mechanism at load time will result is a huge loss of precision of the analysis due to the inherent non-determinism. At the moment, we designed no recovery mechanisms for when a possible inconsistency is predicted, and we only leave stuck the application responsible for that.

Structure of the paper The next section intuitively presents our COP language and the verification mechanism through an example. In Section 3 we formally define the syntax and the operational semantics of this extension of $\mathrm{ML}_{\mathrm{CoDa}}$. The last section concludes and discusses some related work.

2 An example: competing for visors

Here we elaborate on the example of [10] describing a museum guide. First, we briefly recall the features of the functional component of $\mathrm{ML}_{\mathrm{CoDa}}$, omitting the Datalog constituent that is fully standard. We refer the reader to [10] for a full description of the language.

The original functional component of $\mathrm{ML}_{\mathrm{CoDa}}$ provides two main mechanisms for adaptation. The first is *context-dependent binding* through which a programmer declares variables whose values depend on the context. The construct `dlet x = e1 when G in e2` means that the variable `x` (called *parameter* hereafter) may denote different objects, with different behavior depending on the different properties of the current context, checked by evaluating the goal `G`. If the goal `G` is true in the current context, the variable `x` is bound the result of evaluation of the expression `e1`.

The second mechanism is based on the notion of *behavioural variations*. Basically, it is a list of pairs `(x){G1.e1, ...,Gn.en}`, similar to a case statement, that alters the control flow of applications according to which goal holds in the context, so as to dynamically adapt the running application. Behavioural variations are similar to functions: they take arguments (e.g. `x`) and are (high-order) values so facilitating programming dynamic adaptation patterns. To run a behavioural variation we need to apply it through the application operator `#(bv, v)` where `bv = (x){G1.e1, ...,Gn.en}`. The application triggers a *dispatching mechanism* that visits the cases in textual order and selects the first expression `ei` whose goal `Gi` holds; then `ei` evaluates in a environment with a new binding between `x` and `v`. If no goal holds then the application cannot adapt to the context and a run time error occurs. The interaction with the Datalog context is not limited to queries, but one can change the knowledge base through `tell`/`retract` operations that add/remove facts.

We illustrate now how the linguistic extensions to $\mathrm{ML}_{\mathrm{CoDa}}$ we are proposing help in designing an adaptive museum guide application. To make our point, it suffices to consider two concurrent applications that are hosted in the shared context offered by the museum intranet.

Each visitor registers at entrance and gets credentials to access the museum intranet and to download the guide application to his smartphone. This guide adapts to the device (e.g. enabling/disabling particular features like HD videos or NFC communication) and to the user's preferences (e.g. accessibility options for blind or deaf people) and has the ability to interact with (some of) those exhibits of the museum which are interactive. Differently than in [10] we here explicitly consider applications that are deployed at an interactive exhibit and reply to user's questions, e.g. about the author of the exhibit.

Since the museum resources can typically be concurrently accessed by a limited number of users, the activities performed by the guide applications and those done by the context itself have to be coordinated.

Here we focus on the operations performed by the guide applications to access the shared interactive exhibits. We assume that applications communicate with a central server; and as in [11] that the shared context provides applications with a communication infrastructure accessible through the `tell`/`retract` operations that update the context, as well as through suitable remote procedure calls (RPCs).

2.1 The context

Abstractly the context could be thought as a heterogeneous collection of data coming from different sources and having different representations. As we said, the context in $\mathrm{ML}_{\mathrm{CoDa}}$ is a knowledge base implemented as a Datalog program, i.e. a set of facts that predicate over a possibly rich data domain, and a set of logical rules to deduce further implicit properties of the context itself. Below, we briefly introduce some aspects of the context of the museum where the multimedia guide is plugged in.

Suppose we have the museum context presented in [10], that includes information about the user profiles, their device capabilities, the ticketing policies, access points to the intranet etc. Here we enrich the museum context with some facts about the exhibits, e.g. the following fact declares that exhibit `x` is interactive:

```
is_interactive(x)
```

A specific exhibit can interact with a visitor through a virtual reality visors that plays a video. This feature can be expressed in the context by the following fact

```
play_video(x,visor) :-
 is_interactive(x), has_visors(x,visor)
```

Acquiring a visor requires to check if it is available, i.e. that the following Datalog goal holds

```
← ¬busy(x, visor)
```

If this is the case, the application can lock the visor by inserting the fact `busy(x, visor)` in the context through the `tell` construct. Symmetrically, releasing the visor is done by removing the fact through a `retract`.

2.2 The guide and the exhibit application

We now show the relevant code concerning the interaction among the multimedia guide, the exhibit application and the shared environment. For readability we use a sugared syntax of our extended ML$_{\text{CoDa}}$ which will be formally introduced in Section 3. Assume that a new GUI element in the guide is enabled when data are downloaded from the interactive exhibit. Once active it allows a user to visualize the data of the exhibit.

Suppose that a user `U` wants to interact with the virtual reality exhibit `ie` with two visors `v1` and `v2`. As expected `U` can acquire a visor if it is available and cannot if in use until it is released. The following code implements the above:

```
fun interact () =
 let get_visor = (){
  ← ¬busy(ie, v1).
   showMessage "Please use the first visor"
   enable_first()
  ← busy(ie, v1), ¬busy(ie, v2).
   showMessage "Please use the second visor"
   enable_second()
  ← busy(ie, v1), busy(ie, v2).
   showMessage "Please wait..."
 }
 in #(get_visor, ())
```

In the code we define the behavioural variation `get_visor` with no argument that queries the context to get information on availability of visors. The behavioural variation is applied in the last line through the `#` construct. Each case of `get_visor` is driven by a goal, e.g. `¬busy(ie, v1)`. The application interacts with visors via RPCs.

In the exhibit the implementation of the RPC function `enable_first` is straightforward:

```
fun enable_first () =
 tell busy(exhibitID,v1)
 (* Code for interacting with the user *)
 retract busy(exhibitID,v1)
```

where `exhibitID` identifies the current exhibit; the code for the function `enable_second` is analogous.

2.3 Executing the guide

Assume Alice is in front of one of the interactive exhibits and wants to play with it. She taps on the relevant button to launch the function `interact`, causing the behavioural variation `get_visor` to run. If the visor `v1` is available, the first goal succeeds and the RPC `enable_first` is invoked.

Now Bob arrives and wants to interact with the same exhibit. Three different situation may occur, depending on the execution point reached by Alice's application when Bob starts to execute the behavioural variation `get_visor`:

- Alice has still to execute `tell(busy(v1))` and thus also Bob could get the visor `v1`. The runtime of ML_{CoDa} first inspects the history expression H associated with Bob at compile time, and discovers the potential damage to Alice. Indeed H records that Bob will change the context through `tell busy(exhibitID,v1)`, so falsifying the goal ¬`busy(ie, v1)` that Alice has just checked. In this case, the runtime prevents Bob from performing the harmful operation;
- Alice completed the execution of `tell(busy(v1))` and is interacting with the exhibit. Then Bob will simply find the visor `v1` busy and the second case of his behavioural variation will be selected;
- Alice has released the visor `v1` through `retract(busy(v1))` and so Bob can acquire it.

As intuitively described above, the extended runtime support of ML_{CoDa} embeds a verification mechanism at run time, so enforcing a sort of non-interference property among threads. Of course, the simple situation above can be extended to the case with many visitors interacting with the same exhibit.

3 Regulating concurrency in ML_{CoDa}

This section presents our extension of ML_{CoDa} with concurrency. As in [10] the context provides applications with information and resources they need. Here, the context works also as a shared memory through which applications interact. Additionally, our semantics makes sure that when an application modifies the context, it falsifies no hypothesis that drove the selection of the running behavioural variations of other applications.

Syntax The Datalog part is standard: a program is a finite set of (ground) facts and clauses. As defined in [8], we assume that each program is *safe* and *stratified*, so negation is allowed.

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

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

The novelties w.r.t. [10] are that the goals of behavioural variations $(x)\{Va\}$ and of the context dependent binding **dlet** have labels $l \in Lab$ to link them with their abstract counterparts in history expressions (see below). These labels are mechanically attached (in the abstract syntax tree) and uniquely identify sub-expressions. They do not alter the semantics of [10]: at run time, the first goal G_i^l satisfied by the context determines the expression e_i to be run (*dispatching*). Also the **tell**/**retract** constructs, which insert/remove facts from the context, carry labels. The application of a behavioural variation $\#(e_1, e_2)$ which applies e_1 to its argument e_2 is the same as in [10]: the dispatching mechanism is triggered to query the context and to select from e_1 the expression to run. In the formal development we record the goal selected by the dispatching mechanism through the auxiliary expression $\lfloor e \rfloor_G$.

Semantics We assume that our systems are made of some expressions running concurrently in a context $C \in Context$. Here we inherit the standard top-down semantics [8] for Datalog under the Closed World Assumption to deal with negation. We write $C \vDash G\, with\, \theta$ when the goal G, under a ground substitution θ, is satisfied in the context C. The concurrent semantics of a system is defined by a hierarchy of three SOS transition systems. The first one is for expressions with no free variables, but possibly with free parameters, thus allowing for openness. It is a slight modification of the one in [10], where the environment $\rho\colon DynVar \to Va$ maps parameters to variations Va. A first novelty is that transitions are labelled to record the actions performed (irrelevant labels will be omitted). For example we have the following axiom that specifies how the fact F is added to the context C. It also records where this happens through the label that identifies the specific **tell** responsible for that. This information will be used later on to link the actual code with its history expression, computed by the type and effect system, and it helps the verification made at run time.

$$\frac{}{\rho \vdash C,\, \mathbf{tell}(F)^l \xrightarrow{l} C \cup \{F\},\, ()}\ \textsc{Tell2}$$

A second novelty concerns the rules that query the context. Through the dispatching mechanism (see below), we detect a case e of a behavioural variation which will be selected and run (suitable instantiated). Also here the transition records the label ℓ of the goal G satisfied, for future use. Additionally, the goal G indexes the selected case, giving raise to the auxiliary expression $\lfloor e \rfloor_G$. Indeed,

G has to always hold along the execution of e, until it reduces to a value v; in other words, $\lfloor e \rfloor_G$ reduces to v.

$$\frac{\rho(\tilde{x}) = Va \qquad dsp(C,\ Va) = (e, \{\overrightarrow{c}/\overrightarrow{y}\}, G^l)}{\rho \vdash C,\ \tilde{x} \xrightarrow{l} C,\ \lfloor e\{\overrightarrow{c}/\overrightarrow{y}\}\rfloor_G} \ \textsc{Dyvar}$$

where dispatching is essentially the same of [10]:

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

Also the rules for behavioural variation applications are modified similarly. Labels are preserved by the inference rules.

The second level provides the third one with the relevant information to guarantee that no applications modify a resource needed by another one. To do that, we exploit the behavioural abstraction of the application computed by the type and effect system of [10] in order to perform run time checks. We recall from [10] the syntax of the abstractions, called history expressions $H \in \mathbb{H}$, that here carry labels $\ell \in \widehat{Lab}$, for simplicity disjoint from Lab.

$$H ::= \epsilon \mid h \mid \mu h.H \mid tell\, F^\ell \mid retract\, F^\ell \mid H_1 + H_2 \mid H_1 \cdot H_2 \mid \varDelta$$
$$\varDelta ::= ask\, G^\ell.H \otimes \varDelta \mid fail$$

History expressions abstractly represent the activities performed: *tell*/*retrect* are obvious, $\mu h.H$ is for recursion, $+$ abstracts conditionals, $\cdot$ sequential compositions and $\varDelta$ represents the dispatching mechanism. As in [4], our type and effect system associates with an expression e a (standard) type, an effect H and a function $\varLambda$ that records the correspondence of labels ℓ in H with those in e. The semantics of history expression is trivially extended to take care of labels.

There are three rules in the second level. The first follows:

$$\frac{\emptyset \vdash C, e \xrightarrow{l} C', e' \qquad C, H \to^* C, H'' \xrightarrow{\ell} C', H'}{C,\ e : (H, \omega) \to C',\ e' : (H', \omega \wedge \widehat{G})} \ \varLambda(\ell) = l$$

where $\widehat{G} = \begin{cases} G & \text{if } ask\, G^\ell.H \text{ is a sub-history of } H \\ true & \text{otherwise} \end{cases}$

We write $e : (H, \omega)$, when H is the abstraction of e and ω is the conjunction of all the goals (holding in C) of the behavioural variations still in execution. The case $\widehat{G} = true$ holds when l labels a *tell* or a *retract*.

The second rule governs the termination of a behavioural variation and the elimination of the relevant goal:

$$\frac{\emptyset \vdash C, \lfloor v \rfloor_G \to C, v}{C,\ e : (H, \omega \wedge G) \to C,\ v : (H, \omega)}$$

The third rule considers the case when the context does not change (we do not track the changes in H):

$$\frac{\emptyset \vdash C, e \to C, e'}{C,\, e : (H, \omega) \to C,\, e' : (H, \omega)}$$

The top-level transition system takes care of the (interleaved) concurrent behaviour of systems. Here we assume the standard congruences of $\|$, the parallel operator, e.g. commutativity. The first rule of this level is

$$\frac{C, e_0 : (H_0, \omega_0) \to C, \lfloor e_0' \rfloor_G : (H_0', \omega_0 \wedge G) \qquad \alpha_1 \qquad \alpha_2}{\begin{array}{c} C,\, \|_{i=1}^n\, e_i : (H_i, \omega_i) \;\|\; e_0 : (H_0, \omega_0) \to \\ C,\, \|_{i=1}^n\, e_i : (H_i, \omega_i) \;\|\; \lfloor e_0' \rfloor_G : (H_0', \omega_0 \wedge G) \end{array}}$$

where α_1 and α_2 are the following conditions:

$$\alpha_1 = \forall C'' s.t.\ C, H_0 \to^* C'', H_0''.\ C'' \models \bigwedge_{i=1}^{n} \omega_i$$
$$\alpha_2 = \forall i \in [1, n]\, \forall C'' s.t.\ C, H_i \to^* C'', H_i''.\ C'' \models \omega_0 \wedge G$$

The first condition says that no actions of e_0 will falsify any of the goals of the e_i. Symmetrically, the second one guarantees that the goals of e_0 will hold along the execution of the other threads.

There is a rule for when the context changes because of a *tell*/*retract*:

$$\frac{C, e_0 : (H_0, \omega_0) \to C', e_0' : (H_0', \omega_0) \qquad \alpha_1 \qquad \alpha_2}{C,\, \|_{i=1}^n\, e_i : (H_i, \omega_i) \;\|\; e_0 : (H_0, \omega_0) \to C',\, \|_{i=1}^n\, e_i : (H_i, \omega_i) \;\|\; e_0' : (H_0', \omega_0)}\; C \neq C'$$

The last rule is for when the context does not change, and no violations may then occur

$$\frac{C, e_0 : (H_0, \omega_0) \to C, e_0' : (H_0, \omega_0)}{C,\, \|_{i=1}^n\, e_i : (H_i, \omega_i) \;\|\; e_0 : (H_0, \omega_0) \to \quad C,\, \|_{i=1}^n\, e_i : (H_i, \omega_i) \;\|\; e_0' : (H_0, \omega_0)}$$

The mechanism specified by the last two inference rules prevents all the applications running concurrently to misbehave so causing adaptivity errors each other. The properties of the context and the resources acquired by an application in order to execute a behavioural variation are guaranteed to hold until the behavioural variation itself is not completed, regardless of any update made by other applications. The conditions α_1 and α_2 are crucial for performing these checks at run time. These conditions can be verified through the control flow analysis of [10]. Essentially, exploiting the history expressions H_i and H_0 the analysis results in a graph $\mathcal{G}$ that describes the possible evolutions of the context C. Technically, the graph $\mathcal{G}$ is obtained as solution of a set of constraints following the Flow Logic approach [16]. Very roughly, these constraints express how a *tell* or a *retract* inside a history expression modifies a context into a new one. Note that there this static analysis is done at load time, while here it has to be performed at *run time*, because one only knows the acquired resources while executing. Note also that condition α_1 constrains the effects of the running application e_0 on the other applications, but not on itself, otherwise a wanted, and fairly acceptable behaviour could be discarded. An example of this is discussed in Section 2: the RPC function `enable_first` above falsifies the goal $\leftarrow$ `¬busy(ie, v1)` driving the first case of the behavioural variation of the function `interact`. Summing up our concurrent semantics embeds a sort of non-interference mechanism.

4 Conclusions

Our starting point has been the two-component COP language $\mathrm{ML}_{\mathrm{CoDa}}$ [10], in which the context is a Datalog knowledge base and the application code is ML with specific adaptation constructs; in particular, the dispatching mechanism is driven by Datalog goals. Here, we have extended $\mathrm{ML}_{\mathrm{CoDa}}$ to operate in a concurrent environment. A major contribution of this paper is the run time verification mechanism embedded in the semantics. It is triggered when a behavioural variation is about to start and it enforces a sort of non-interference among the running applications.

Our proposal relies on a formal operational semantics of the extended language, as well as on a type and effect system that associates each application with a safe abstraction of its run time behaviour, namely a history expression. The verification mechanism uses the history expressions of the application ready to evaluate a behavioural variation and checks that none of its future actions may invalidate the assumptions driving the execution of the other threads. Analogously, the application is protected against actions done by other threads. The verification can be performed by simply moving at run time the Control Flow Analysis done in [10] at load time. As a matter of fact, our mechanism for non-interference has been inspired by the classical notion of critical section. In our case, the resources to protect are the properties of the context that are relevant for the execution of applications: a behavioural variation plays here a role similar to that of a critical section.

Future work includes extending our prototypical implementation of $\mathrm{ML}_{\mathrm{CoDa}}$ [5] with concurrency and the run time verification. Since history expressions are over-approximations of the behaviour of applications, in some cases the verification mechanism unnecessarily suspends the execution of a thread, e.g. when there is a conditional and only one branch may lead to troubles. We would like to investigate whether it will be possible to live dangerously in a partially inconsistent context. However, in this case some notions of compensation (or recovery like in [11]) would be in order to prevent an application to crash definitely. A different approach would be analysing a history expression to detect where the code may perform dangerous activities, and dynamically instrument it accordingly, as proposed in [4].

Related Work Most research efforts in COP have been directed toward the design and the implementation of concrete languages; see [2] for an analysis of some implementations. Below, we focus on papers that are strictly related to our proposal and on those proposing concurrency and verification mechanism; see [17] for a broad survey on primitives and possible language designs.

Many COP languages are object oriented, thus behavioural variations are often implemented as partially defined methods, and are not values as it is the case in $\mathrm{ML}_{\mathrm{CoDa}}$. The most notable exception is *ContextL* [9], that is based on *Common Lisp*, from which it inherits higher-order features.

Typically, the context is a stack of layers, that can be activated and deactivated at run time. One can simply and intuitively view a layer as an elementary property (a proposition) of the current context. $\mathrm{ML}_{\mathrm{CoDa}}$ differs from this approach having distinct formalisms for specifying the context and the applications. Others papers in the literature do the same. The language *Javanese* [14] supplies primitives for declaring a context and its properties in a logical manner through a temporal logic. In *Javanese* the context represents properties of the system that are "activated by an action and held active until another action that deactivates it occurs". This is similar to our vision where the system running an application is part of the context and where a fact inserted into the context holds until explicitly retracted. Also *Subjective-C* [13] is equipped with a domain-specific language for specifying the contents of what is called a *set of contexts*. A context of *Subjective-C* is just a single property holding in the working environment of an application, behaving much like our facts. Similarly, a context is activated when particular circumstances occur in the environment. Furthermore, *Subjective-C* proposes constructs for specifying relationships and constraints over contexts, e.g. inclusion and conflict. This approach is very similar to ours, and Datalog can also express these kinds of relations through logical rules.

As far as we know a limited number of papers have considered concurrency in COP. In [12] *ContextML*, a predecessor of $\mathrm{ML}_{\mathrm{CoDa}}$, is proposed. It extends ML with layers, layered expressions, and scoped activation mechanisms for layers (`with` and `without`). Applications are made of many components with a local context interacting through message passing. Similarly to the present proposal a type and effect system computes application abstractions, but there they are statically model-checked to enforce communication compliance and security policies. In [18] the formal semantics of *ContextErlang* describes the behaviour of the constructs for adaptation within a distributed and concurrent framework, based on message passing. Similar to ours, that semantics ensures a non-interference property among *Erlang* actors.

As regards event-driven adaptation, *EventCJ* [1] is a Java-based language which combines mechanisms from COP with event based changes of the context. It provides constructs to declare both the events thrown by an application and the transition rules specifying how to change the context when an event is received. The language *Flute* [3] is designed for programming reactive adaptive software. *Flute* constrains the execution of a procedure with certain contextual properties specified by a developer. If any of these properties is no longer satisfied, the execution is suspended until the property holds again. Stopping the execution is like in our approach when goal of a behavioural variation has been falsified because of a context change.

In [6] a run time verification mechanism based on symbolic execution is proposed. Differently from ours the verification step is performed just before activating/deactivating a layer in the context, in order to check whether adaptation is possible. A different approach to verify applications is proposed in [7]: the structure of contexts is a(n enriched) Petri net which is analysed through existing tools.

References

1. Aotani, T., Kamina, T., Masuhara, H.: Featherweight EventCJ: a core calculus for a context-oriented language with event-based per-instance layer transition. pp. 1:1–1:7. COP '11, ACM, New York, NY, USA (2011)
2. Appeltauer, M., Hirschfeld, R., Haupt, M., Lincke, J., Perscheid, M.: A comparison of context-oriented programming languages. In: COP '09. pp. 6:1–6:6. ACM, New York, USA (2009)
3. Bainomugisha, E.: Reactive method dispatch for Context-Oriented Programming. Ph.D. thesis, Comp. Sci. Dept., Vrije Universiteit Brussel (2012)
4. Bodei, C., Degano, P., Galletta, L., Salvatori, F.: Linguistic Mechanisms for Context-aware Security. In: Ciobanu, G., Méry, D. (eds.) ICTAC 2014. LNCS, vol. 8687. Springer (2014)
5. Canciani, A., Degano, P., Ferrari, G.L., Galletta, L.: A context-oriented extension of F#. In: FOCLASA 2015. EPTCS, vol. 201 (2015)
6. Cardozo, N., Christophe, L., De Roover, C., De Meuter, W.: Run-time validation of behavioral adaptations. In: COP'14s. pp. 5:1–5:6. ACM, New York, NY, USA (2014)
7. Cardozo, N., González, S., Mens, K., Straeten, R.V.D., Vallejos, J., D'Hondt, T.: Semantics for consistent activation in context-oriented systems. Information and Software Technology 58, 71 – 94 (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 (Mar 1989)
9. Costanza, P., Hirschfeld, R.: Language Constructs for Context-oriented Programming: An Overview of ContextL. In: DSL '05. pp. 1–10. ACM, New York, NY, USA (2005)
10. Degano, P., Ferrari, G.L., Galletta, L.: A two-component language for adaptation: Design, semantics, and program analysis. IEEE Trans. Software Eng. 42(6), 505–529 (2016)
11. Degano, P., Ferrari, G.L., Galletta, L.: Event-driven adaptation in COP. In: PLACES 2016. EPTCS, vol. to appear
12. Degano, P., Ferrari, G.L., Galletta, L., Mezzetti, G.: Types for coordinating secure behavioural variations. In: Sirjani, M. (ed.) COORDINATION 2012. LNCS, vol. 7274, pp. 261–276. Springer (2012)
13. González, S., Cardozo, N., Mens, K., Cádiz, A., Libbrecht, J.C., Goffaux, J.: Subjective-c. In: Malloy, B., Staab, S., van den Brand, M. (eds.) Software Language Engineering, LNCS, vol. 6563, pp. 246–265. Springer (2011)
14. Kamina, T., Aotani, T., Masuhara, H.: A unified context activation mechanism. In: COP'13. pp. 2:1–2:6. ACM, New York, NY, USA (2013)

15. Loke, S.W.: Representing and reasoning with situations for context-aware pervasive computing: a logic programming perspective. Knowl. Eng. Rev. 19(3), 213–233 (Sep 2004)
16. Nielson, H.R., Nielson, F.: Flow logic: A multi-paradigmatic approach to static analysis. In: Mogensen, T., Schmidt, D.A., Sudborough, I. (eds.) The Essence of Computation, Lecture Notes in Computer Science, vol. 2566, pp. 223–244. Springer Berlin Heidelberg (2002)
17. Salvaneschi, G., Ghezzi, C., Pradella, M.: An analysis of language-level support for self-adaptive software. ACM Trans. Auton. Adapt. Syst. 8(2), 7:1–7:29 (Jul 2013)
18. Salvaneschi, G., Ghezzi, C., Pradella, M.: Contexterlang: A language for distributed context-aware self-adaptive applications. Sci. Comput. Program. 102, 20–43 (2015)
19. Zave, P., Jackson, M.: Four dark corners of requirements engineering. ACM Trans. Softw. Eng. Methodol. 6(1), 1–30 (Jan 1997)