ContextErlang: Introducing Context-oriented Programming in the Actor Model *

Guido Salvaneschi, Carlo Ghezzi, Matteo Pradella
DEEPSE Group, DEI, Politecnico di Milano, Piazza L. Da Vinci, 32, Milano, Italy
{salvaneschi, ghezzi, pradella}@elet.polimi.it

Abstract

Self-adapting systems are becoming widespread in emerging fields such as autonomic, mobile and ubiquitous computing. Context-oriented programming (COP) is a promising language-level solution for the implementation of context-aware, self-adaptive software. However, current COP approaches struggle to effectively manage the asynchronous nature of context provisioning.

We argue that, to solve these issues, COP features should be designed to fit nicely in the concurrency model supported by the language. This work presents the design rationale of ContextErlang, which introduces COP in the Actor Model. We provide evidence that ContextErlang constitutes a viable solution to implement context-aware software in a highly concurrent and distributed setting. We discuss a case study and an evaluation of run-time performance.

Categories and Subject Descriptors D.1.3 [*Software*]: Programming Techniques—Concurrent Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages, Design

Keywords Context-oriented programming, Self-adaptive software, Erlang, OTP platform

1. Introduction

Dynamic adaptation of a software system to a changing context has emerged as a common need in a wide range of scenarios. For example, autonomic computing is about providing a system the means to be self-managing in changing conditions. Mobile and ubiquitous computing often require applications to adapt to the external environment.

Since context-depending behaviors must be activated at run time and typically crosscut the system functionalities, managing context-dependent features in a systematic and effective way became a key software challenge. The context-oriented programming paradigm (COP) introduced by Costanza and Hirschfeld [9] has emerged as a viable approach and language-level support for context management. The key idea of COP is to provide specific language abstractions that enable context-adaptability through well engineered modularization of *behavioral variations*, whose dynamic activation and composition changes the basic program behavior to support context-aware adaptation [17].

The Actor Model – originally proposed by Hewitt [16] – is an alternative solution over traditional thread-and-lock concurrency approaches. There is growing interest around languages that are based on this paradigm, such as Erlang [1] and Scala [2] which easily allow to take advantage of increasing hardware parallelism. In this scenario, actors offer an interesting solution to the new challenges of self-adaptive and context-aware software. We argue that the Actor Model strongly fits the requirements of context awareness. Asynchronous message passing offers an intuitive representation for context provisioning and agents are a natural abstraction for context-adaptable units inside an application.

Hereafter, we briefly provide the motivations that lead to the development of ContextErlang, an extension we designed and developed for the Erlang programming language. Most COP languages implement a *programmatic* and *synchronous* variation activation model *on a specific control flow*. Variations are activated and composed through an explicit statement such as

```
with(variationList) { codeBlock }
```

and activation is scoped to the dynamic extent of the code block. This model has two shortcomings. On the one hand *all* the objects in the control flow are automatically adapted. This precludes fine-grain adaptation on single entities of the application. Fine-grain control is needed in many real-world applications where adaptation is required (see Section 2). On the other hand, dynamically scoped activation fails to

* This research has been funded by the European Community's IDEAS-ERC Programme, Project 227977 (SMSCom).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AOSD'12, March 25–30, 2012, Potsdam, Germany.
Copyright © 2012 ACM 978-1-4503-1092-5/12/03. . . $10.00

manage event-specific context changes, i.e. context changes that are asynchronously delivered to the application. Event-based context changes can impact several control flows and it is not possible to react to them through conventional COP `with`-driven activations occurring at fixed points in the code. Event-specific context changes have a prominent role in real world applications where the adaptation is driven by changes discovered via environmental monitoring, user interaction, or the insurgence of internal system conditions.

Our approach abandons the traditional per-control-flow dynamic scoped activation mechanism of thread-based COP languages. Because of the asynchronous nature of context-change notifications that generate event-specific context changes, we propose a solution that is tightly coupled with the adopted concurrency model.

Specifically, we leverage the agent-based model of Erlang to support context-adaptations. CONTEXTERLANG is based on the concept of context-aware reactive agents. Context-adaptable agents have a basic behavior which can be altered by *variations*, i.e. behavioral units that can be *activated* on the agent. Variations can be *composed* to produce the actual behavior of the agent. Variation activation and the other context-related operations are performed by sending *ad-hoc* messages to the agent. Therefore, in CONTEXTERLANG, asynchronous activation, as required by real-world adaptive systems, is the norm. Instead of the dynamically scoped activation model of COP languages, we adopt *per-agent* variation activation and composition. This gives the programmer full control over fine-grained adaptation of the application components.

CONTEXTERLANG also supports *variation transmission*. An agent on a remote Erlang node can be provided with a new behavior by sending a variation to the node and activating it on the agent. This introduces a very useful support for systems that must adapt to unforeseen situations. Since we wanted to rely on a robust implementation, compatible with existing Erlang applications, we developed CONTEXTERLANG as part of the OTP platform, on which practically any real-world Erlang application is based.

To summarize, the main contribution of this paper is the introduction of COP in the Actor Model, through the design and the implementation of CONTEXTERLANG. More precisely in our work we achieved the following results:

- Integration of COP concepts with the Actor concurrency model.
- Implementation as part of Erlang OTP, an industrial-strength language for distributed and concurrent applications.
- Experimental validation of our approach through prototypes of significant complexity and performance evaluation.

The paper is organized as follows. In Section 2 we discuss the motivation this work. In Section 3 we describe CONTEXTERLANG and its design. Section 4 discusses the validation of our approach. Section 5 discusses the related work. Section 6 draws some conclusions and presents future research.

2. Motivation

To motivate the design choices behind our work, we introduce a non-trivial example called ContextChat, our prototype of an instant messaging server. We discuss possible designs and implementations of ContextChat using existing COP languages and CONTEXTERLANG. More details of the implementation will be presented along the paper to illustrate CONTEXTERLANG's features.

In ContextChat, the connected clients can exchange messages in real time. The server also implements some advanced features, which can be dynamically activated. When users go offline, received messages are stored on the server and delivered later when the addressee connects. An optional backup can be enabled by the user to save both the received and sent messages on a remote server. Additionally, the system can activate a tracing functionality to collect information on client communications. In a distributed environment, this allows for self-adaptive behavior, moving users who often exchange messages on the same physical machine and reducing cross-node communications.

An abstract view of the application is sketched in Figure 1. For each user i an always-alive component U_i embodies the user even when he or she is offline (e.g. U_4). Border components B_i are created when clients C_i connect. Each border component is in charge of the network connection with the client and controls the always-alive component. Consider the scenario in which the client C_1 sends a message to the client C_2. C_1 communicates the message to the border component (e.g. via some protocol over HTTPS). The border component B_1 decodes the "send_msg" command and controls U_1. B_1 activates the *send message* functionality on U_1. U_1 forwards the message to U_2 and through B_2 the message reaches C_2.

Context-oriented programming. In ContextChat, the variations to the basic behavior are clearly identified, should be separated from the rest in the codebase, must be dynamically activated, and depend on the current *context* of the application – as we explain in a while. Therefore, COP looks like the natural solution for the requirements of ContextChat. While in traditional OO programming method dispatching is two-dimensional, depending on the message and on the receiver, COP adds a further dimension: methods may also be dispatched according to the current context [17]. In COP, the notion of context is abstract and general. *Every computationally accessible information* can be considered as context. The user condition (e.g., online/offline, enabled backup) can be considered its current *context*. To enable run time adaptation, COP supports dynamic context composition and its abstractions avoid cluttering the code with *if* statements to express context dependency. Therefore using COP to auto-

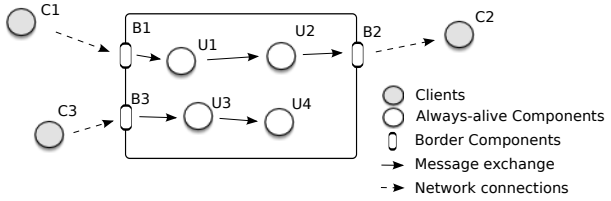

Figure 1. The ContextChat application.

matically select and combine the proper behaviors is an appealing solution.

Figure 2 shows a possible implementation of the `User` object implementing an U_i component in a COP language extension to Java, such as ContextJ [5]. For the benefit of the reader we use this example also to shortly introduce the typical COP features. A more conceptual analysis and an overview of COP can be found in [17]. COP provides language-level abstractions to modularize context-dependent behavioral variations and dynamically activate and combine them. In COP languages, behavioral variations are reified in *layers*[1], abstractions which group *partial method definitions*. For example, in Figure 2 the `tracing` layer contains a partial definition of the `receive_msg` and of the `send_msg` methods. When a method is called, the implementation to execute is chosen according to the active layers. The `proceed` keyword allows dynamic combination. It is similar to `proceed` in aspect-oriented programming and calls the partial definition in the next active layer or the basic definition. Layer activation has dynamic extent and it is done through the `with` statement: in the control flow all the method calls are dispatched according to the active layers.

Dynamic scope is a powerful mechanism for variations activation, since it allows remote effect, setting the active layers once and automatically adapting all the objects in the execution flow. This behavior has already proved useful in several application scenarios [5, 9, 26]. However, implementing ContextChat with the traditional COP dynamically scoped activation highlights some inconveniences. We argue that these problems are due to the asynchronous nature of context provisioning, to the concurrent nature of the application and to its non-trivial complexity. Therefore the issues analyzed in the rest are likely to be encountered in any sufficiently large self-adaptive application which needs to be organized in several functional modules, and are not specific of this example.

First, a context change is often an *asynchronous* event coming from *outside* the execution flow. Since layers are activated when the control flow reaches the statement, the `with` construct is inherently synchronous and is not suitable for these cases. For example, the `tracing` layer is activated by an external engine in charge of implementing the autonomic behavior. The same holds for the activation of the backup functionality which can be performed anytime by the client while `User` objects are exchanging messages with other users. A possible solution is to adopt inversion of control [22] and first class layers. For example, a `setActiveLayers` callback method can be implemented in the `User` class to notify the change of the active layers and store them locally. However, this solution increases the complexity the application making it less readable. Indeed, in this case, inversion of control does not capture the design intention. Conceptually, the programmer's intention is to cause an entity adaptation and not to notify an entity letting it perform the activation at the next `with` statement. In addition to that, in some applications, it is not possible to identify unique entry points for the control flow. As already noticed by COP researchers [6], in this cases, layer composition statements must be scattered and replicated across all the possible control flows, such as all the callback methods in a GUI application.

Second, in a highly concurrent environment, the control flow can follow complex paths. These paths hardly map on dynamically scoped program sections, i.e. contextual regions whose adaptation condition is known where the region is entered. For example, the `User` object (Figure 2) may be traversed by several control flows, and the information of which behavioral variation to activate is not directly available to all of them. The backup functionality is enabled by the client C_1 and therefore the associated border component B_1 can trigger the backup behavior by interacting with the `User` object U_1 in the dynamic scope of a `with` statement.

```
public class User {
  layer offline {
    void receive_msg(User source,M msg){
      store_chats.store_message(source, msg);
  } }
  layer tracing { ...
    void receive_msg(User source,M msg){
      // send msg to the tracing listener
      proceed(source, msg);
    }
    void send_msg(User source,M msg){
      // send msg to the tracing listener
      proceed(source, msg);
  } }
  layer backup { ...
    void receive_msg(User source,M msg){
      // send msg to the remote server
      proceed(source, msg);
  } }
    ... // Other methods
  void receive_msg(User source,M msg){
    //forward msg to my border component
  }
  void send_msg(User dest,M msg){
    // forward to dest client
} }
```

Figure 2. An implementation of the chat server in ContextJ.

[1] For continuity with our previous work, ContextErlang keeps the name *variation* also to indicate the language abstraction. CONTEXTERLANG variations are quite similar to COP layers; a comparison between the two is in Section 5.

However, when the `User` object U_1 is called from another `User` object U_2 to receive a message, U_2 does not know if the `backup` layer should be activated on U_1.

A third issue is that in a complex application with several components, dynamic scope is difficult to control and can extend *too far*. For example, in case a border component B_1 delivers a message through the associated `User` object U_1 and the client C_1 activated the backup feature on U_1, the backup functionality is propagated along the flow to the other `User` object U_2.

COP researchers have already investigated the limitations of dynamically scoped variation activation. ContextJS [20] is an open implementation of COP which supports user-defined activation strategies, such as indefinite scope or per-object activation. Per-object activation is performed calling a `setWithLayer` method on the instance. Per-object activation solves the problem of the activation along the execution path, since objects identify the boundaries in which layer activation is constrained. This solution nicely fits in the OO model, resembling the way other design problems have been solved for objects. For example, in Java, concurrency is addressed at the language level by assigning a monitor to each object. Similarly, in per-object activation, a list of currently active layers is associated to each object.

EventCJ [29] is a Java COP extension which supports declarative layer transitions and implicit activation through pointcut-like predicates. The issue of asynchronous activation, discussed previously, is solved by AspectJ-like statements: when a pointcut-like event occurs, a layer transition is triggered. Layers are activated on per-object basis. Figure 3 shows a possible implementation of an `User` object in EventCJ. Events and layer transitions are declared inside `direction` modules. When the `onStatusChanged` method is called, the `StatusOffline` or the `StatusOnline` events are triggered, depending on the parameters. These events trigger layer `transitions` from `Online` to `Offline` and vice versa. The approach solves the problem of asynchronous activation by introducing points in the program execution which implicitly activate layers.

However, none of the existing COP languages leverages the concurrency model to easily support asynchronous context propagation. As a result, the layer activation mechanism can be quite complex (Figure 3).

As we have seen, the backup and the tracing functionalities in the example are activated by a different thread than the one actually affected by them. This aspect is not peculiar of our example, but is common to many self-adaptive applications. The MAPE-K model (*Monitor, Analyze, Plan, Execute-Knowledge*), conceived by the autonomic computing community, decouples the adaptive application in a managed element, which implements the application logic and an autonomic manager, which collects data from sensors and plans the adaptive behavior [18]. So, these subsystems are not only conceptually separated, but usually run in separate threads and communicate asynchronously. However, the relation between context-adaptation and the language concurrency model has not been investigated so far in COP research. Even more advanced COP languages are quite traditional in this sense. ContextJS is single-threaded, since it extends JavaScript, a single-threaded language; while EventCJ adopts the standard Java share-and-lock concurrency model. By leveraging the integration of COP with the Actor Model, CONTEXTERLANG directly addresses the issue of context propagation in concurrent systems, it allows asynchronous context provisioning directly in the language, without pointcut-like expressions, and solves in a natural way the problem of context confinement adopting actors as context boundaries.

```
public class User {
  void onStatusChanged(Status s){...}
    ...
}
direction UserLayerActivations{
  declare event StatusOffline(User u)
    :after call(onStatusChanged(Status s)) && target(u)
        && args(s) && if(s==Status.OFFLINE) :sendTo(u);
  declare event StatusOnline(User u)
    :after call(onStatusChanged(Status s)) && target(u)
        && args(s) && if(s==Status.ONLINE) :sendTo(u);
   ... // Other events
  transition StatusOffline: Offline switchTo Online;
  transition StatusOnline: Online switchTo Offline;
   ... // Other transitions
}
```

Figure 3. ContextChat in EventCJ.

ContextErlang. CONTEXTERLANG mainly differs from other COP languages in the way it supports the activation of context-specific functionalities. To address the issue of asynchronous context provisioning, variations are activated through messages. This approach nicely reflect the design intention and avoids the cluttering of control inversion. To cope with the complexity of a concurrent application organized in several behavioral units, in CONTEXTERLANG, variations are activated on *per-agent* basis, and each agent can be controlled individually. This also eliminates the risk of unintended adaptation propagation. After activation, variations are implicitly associated with the agent. They are managed transparently and do not need dedicated local variables or other boilerplate code.

To make the benefits of such design more concrete, hereafter we illustrate how ContextChat is designed in CONTEXTERLANG. Always-alive components are context-aware `user` agents exchanging Erlang messages. Border components are standard Erlang agents, since no special adaptation is required. The `offline`, `online`, `backup` and `tracing` variations implement the dynamically activatable features for the `user` agents. Other agents can directly control the adaptation state of a `user` agent. For example when a client C_i closes the connection, the border agent sends a context-related message to the associated agent U_i, which has the effect of

```
-module(cache).
-behavior(gen_server).
    ...
start() ->
  gen_server:start_link({local, ?MODULE},
                        ?MODULE, [], []).
get(Name) ->
  gen_server:call(?MODULE, {get, Name}).
add(Name, Item) ->
  gen_server:cast(?MODULE, {add, Name, Item}).

init([]) ->
    % ... initialization here
    {ok, State}.
handle_call({get, Name}, From, State) ->
    % ... retrieve Item from the state
    Reply = Item, {reply, Reply, State}.
handle_cast({add, Name, Item}, State) ->
    % ... add Item to the state
    {noreply, State}.
terminate(Reason, State) ->
    % ... manage shutdown here
    ok.
```

Figure 4. A callback module of an OTP `gen_server`.

activating the `offline` variation. In a similar way B_i activates the backup of the conversations and the autonomic engine activates the `tracing` variation. Active variations can be dynamically combined to allow coexisting multiple adaptations. For example, the `backup` variation proceeds to either the `online` or the `offline` variation to send a chat to a backup server and then either forward or store it locally.

In order to present a peculiar feature of CONTEXTERLANG, i.e. *variation transmission*, we augment the ContextChat application with an additional functionality. A client can apply a customizable filter to its outgoing messages such as capitalizing all the first letters of sentences or adding emoticons to each message. Despite its triviality, this feature is interesting because the type of filter cannot be forecast in advance. In CONTEXTERLANG this kind of situation is specifically addressed by variation transmission, which allows one to send a variation to a remote agent and dynamically load it. In this way the agent can react to unforeseen situations.

Further insights into the details of the ContextChat implementation in CONTEXTERLANG are provided in the following sections.

3. Design of ContextErlang

In this section, we describe and motivate the basic concepts and the language constructs introduced by CONTEXTERLANG. To achieve the high quality standards of Erlang applications, CONTEXTERLANG is built on the OTP platform – a library and a set of procedures for structuring fault-tolerant, large-scale, distributed applications. We provide a minimal description of the Erlang syntax and a short introduction to the OTP.

3.1 OTP in a Nutshell

While the language provides the basic functionalities for software development, practically any real-world Erlang application is based on the OTP platform. The concept of *behavior* is central in OTP and is based on the idea that, in an application, many processes enact similar patterns, such as serving requests, handling events, or monitoring other processes. OTP generalizes these common patterns, and gives a ready implementation of the generic structure (called the *behavior*), which provides features such as message passing, error handling and fault-tolerance. The user only needs to implement the specific part in a *callback* module, which exposes a predefined interface. This kind of code structuring makes programs easier to understand, and prescribes a general architecture that should be common to all OTP applications. In the paper we use the term *behavior* also to indicate the way an agent behaves with respect to the software system. To avoid confusion, we will use the term *OTP behavior* to disambiguate.

In Figure 4 we present a callback module for the most common OTP behavior, the `gen_server`, a process which stands waiting for requests from other processes. An Erlang module starts with attributes introduced by "`-`". They state the module name, the exported functions and other declarations. Functions implementation follows. A function body is started by "`->`". Curly braces indicate tuples of fixed length, square braces indicate lists. Variables start with an uppercase letter, other literals are atoms, i.e. literal constants. The `?MODULE` literal macroexpands to the module name. In the example, the process implements a cache that allows for adding and retrieving items. The `gen_server` process is spawned with the `start` function. It is common practice in OTP that the callback hides the interaction with the behavior, providing an API to the user. In this case, the callback exposes the `get` and the `add` functions that in turn interact with the spawned process. A call to the `get` function invokes `call` on the `gen_server` module, which causes a message to be sent to the created process. When the message is received, the corresponding callback function `handle_call({get, Name}, From, State)` is invoked (notice that this function is executed in a different process with respect to `call`). The returned result is sent back through a message and the `gen_server:call` function ends. All the machinery associated with message passing, possible message loss, timeout, and dispatching over callback functions, is hidden from the programmer. `call` functions are used for synchronous messages expecting a return value, `cast` functions are asynchronous and do not return a value to the caller.

3.2 CONTEXTERLANG Basics

To support fast development of self-adaptive applications, CONTEXTERLANG provides context-aware agents through the OTP `context_agent` behavior. According to the OTP conventions, the programmer only needs to define the callback

```
-module(user).
-behavior(context_agent).
-include("context_agent_api.hrl"). % contextual API
% API
receive(AgentId, Source, Msg) ->
context_agent:cast(AgentId, {receive_msg, Source, Msg}).
add(AgentId, Dest, Msg) ->
context_agent:cast(AgentId, {send_msg, Dest, Msg}).

handle_cast({receive_msg, Source, Msg}, State) ->
    % ... forward to my client
    {noreply, State}.
handle_cast({send_msg, Dest, Msg}, State) ->
    % ... forward to dest client
    {noreply, State}.
% startup, shutdown and other auxiliary functions
```

Figure 5. The callback for the user agents in ContextChat.

```
-module(offline).
-context_cast([receive_msg/2]). % Contextual dispatch
    ...
handle_cast({receive_msg, Source, Msg}, State) ->
    store_chats:store_message(Source, Msg),
    {noreply, State}.
```

Figure 6. The `offline` variation in ContextChat.

module containing the functions for the core functionalities. We refer to these functions as *handle* functions [2].

Behavioral adaptation of context-aware agents is performed in CONTEXTERLANG through *variations*. A variation encapsulates a set of changes that modify the way an agent reacts to messages. Variations are combined in a stack fashion through `proceed`. When the agent receives a request message, the function to execute is searched along the stack of *active* variations up to the callback. This design is substantially similar to the layer combination in other COP languages. It clearly separates the basic behavior of an agent from the variations, making the application easier to understand and maintain, and supports reuse through combination of variations.

Figure 5 shows the callback of the context-aware agents that implement `user` agents inside the ContextChat server. The callback declares a function for receiving messages and a function for sending them to a different agent. Based on this example, hereafter, we analyze how the programmer can interact with variations in CONTEXTERLANG. Then we discuss how variations are declared and activated and how they can be sent to another node, changing the behavior of remote agents.

Variation creation. A variation is an Erlang module defining a set of handle functions exposed to the contextual dispatching. Implementing variations as Erlang modules has

[2] In the OTP terminology, functions inside callback modules are commonly referred to as *callback functions*. Since in CONTEXTERLANG functions like `handle_call` and `handle_cast` appear both in callback and in variation modules, we indicate them uniformly with the term *handle* functions to avoid confusion.

```
-module(backup).
-context_cast([receive_msg/2]).
  ...
handle_cast({receive_msg, Source, Msg}, State) ->
    % send Msg to the remote server
        ...
    ?proceed_cast({receive_msg, Source, Msg}, State),
    {noreply, State}.
```

Figure 7. The `backup` variation in ContextChat.

several advantages. It makes their development invaluably simple. It does not require syntax extensions, increasing the chances of acceptance by the programmers and avoiding the risk of breaking tool compatibility. Finally improves extensibility, since new variations can be added by implementing new modules without modifying the existing code.

The `offline` variation (Figure 6) defines an asynchronous `receive_msg` function, which at the moment of the activation, overrides the corresponding function in the callback module. In Figure 7, the `backup` variation redefines the `receive_msg` function in order to forward the message to a remote server in charge of the backup. If the `backup` variation is activated on top of the `user` callback, a call to `receive` causes the implementation inside `backup` to be called. The `proceed` call resolves to the implementation of `receive_msg` inside the callback module.

Variations can require an initialization or a shutdown phase to work properly. For example, if the `offline` variation in ContextChat saves the conversations on disk, a file must be created and opened. CONTEXTERLANG allows a variation to declare the `on_activation` and the `on_deactivation` functions, which are guaranteed to be called when the variation is respectively made active or deactivated. Initialization and cleanup code is placed inside these functions.

Variation activation. To allow asynchronous contextual adaptation, variation activation is performed in an imperative way by a different agent (an exception is discussed in Section 3.4). A common pattern is that a single agent enacts the role of context manager, and activates variations on the other agents depending on the context conditions. We expect that with the development of agent-based context-aware applications, new patterns arise. For example, agents could be organized in communities sharing a *local* context manager, while global context managers supervise other managers, in a hierarchical fashion.

The modification of the behavior of context-aware agents is exposed by the API of the `context_agent` module. The `activate_variations` function activates a list of variations on a given agent. In this example, the `offline` variation is activated on the agent `AgentId`. Then the `backup` variation is activated on top of the `offline` variation:

```
context_agent:call(
    {activate_variations, AgentId, [offline]}),
  ...
context_agent:call(
    {activate_variations, AgentId, [backup, offline]}),
```

```
CONTEXT_SPEC  ::=  [ SLOT_SPEC* ]
SLOT_SPEC  ::=  { Slotname, SLOT }
SLOT  ::=  SWITCH_SLOT
         |  ACTIVATABLE_SLOT
         |  FREE_SLOT
SWITCH_SLOT  ::=
         [ (Varname1,)* { Varname2, active } (,Varname3)*]
ACTIVATABLE_SLOT  ::=
         { Varname, active } | { Varname }
FREE_SLOT  ::=  free_slot
```

Figure 8. The syntax specification of a context ADT.

The same updating mechanism can then be used for variations deactivation. We require the atoms in the list to be valid names of modules available to the Erlang virtual machine. Beside direct interaction with the `context_agent`, we adhere to the OTP convention of hiding the interaction with OTP behaviors inside the callback and referencing the agent with the callback name (Section 3.1). The following code equivalent to the first call in the previous example:

```
user:activate_variations(AgentId, [offline]),
```

This is achieved thanks to a `context_agent_api.hrl` module which makes the API is available when imported by the callback.

Variation transmission. Variation transmission is a powerful mechanism to implement software which reacts to unforeseen conditions. For example, our previous work [11] shows how this feature can be used to adapt PDA devices to support rescue operations in an emergency scenario.

To design variation transmission and variation dynamic loading we leveraged advanced Erlang VM features, such as run time code manipulation, dynamic module loading and remote procedure call. The `variation_code` module provides the API for the functionalities concerning variation transmission. The following call sends a variation `var` to a remote node `node2`, and loads `var` in the virtual machine of `node2`:

```
% on node1@machine1
variation_code:send_var(node2@machine2, var)
```

The `send_var` call requires the `var` module to be available to the `node1` virtual machine. In the case of the ContextChat server, variation transmission can be used to allow clients to create a filter variation that manipulates the characters of their messages. The variation is then dynamically loaded and activated on the `user` agent. To include the filter in a variation, on the fly compilation is obtained using the Erlang compiler API. After this process completes, the variation can be activated on an agent as usual:

```
user:activate_variations(AgentId, [text_effect])
```

Of course, loading a module created from a user-defined filter is potentially dangerous and proper input validation is required to avoid security flaws.

3.3 Coherence among variations: context ADT

COP behavioral variations are activated and combined while the application is running. As a result, the issue of the consistency among variations arises. For example, the `offline` and the `online` variations in the ContextChat example should not be active at the same time. COP researchers have already investigated this problem. For example, reflection [10] has been leveraged to dynamically check the constraints. Other solutions use domain specific languages (DSL) to express *declarative* constraints on layers [8], in a way similar to feature diagrams in software product lines. The violation of a constraint raises an error which must be interactively managed by the programmer, so the need for human intervention limits the applicability of this approach. Subjective-C [13] also introduces a DSL to express context dependencies. The system inspects all the user-defined relations, possibly triggering an activation if needed. Another approach is to employ formal verification to statically guarantee layer constraints [29].

Our solution starts from the observation that organizing adaptability concerns in an application, and mapping them to variations and meaningful variation combinations, always requires careful design. For this reason, in practice, the programmer defines in advance which variation combinations are required. To explicitly capture these design choices, CONTEXTERLANG introduces a context abstract data type (ADT). The context ADT encapsulates the variations that can be activated on an agent, organizing the possible variation combinations and enforcing constraints on their activation. In this way the user of the context ADT instance is forced by the interface to activate only valid combinations (i.e. those designed in advance by the ADT programmer). The creation of an unforeseen combination, required by remote variation transmission, is made explicit. Note that the context ADT solution is not specific to ContextErlang and in principle could be ported to layer-based COP languages.

The `context_ADT` module creates a context data type instance from a given specification. The context ADT is organized as a fixed-size stack. Each level of the stack, referred as a *slot*, has a name for direct access. Three types of slots are defined. *Activatable* slots contain a single variation which can be active or not. *Switch* slots contain one or more variations, only one of which can be active at a certain instant of time. *Free* slots contain a single variation which is left undefined and can be assigned later. Free slots are the way variations transmitted by remote nodes can be used. In the following example a context ADT is created for the variations of a `user` agent.

```
Spec = [{persistency, {backup, active}},
        {tracing, {trace, active}},
        {status, [{offline, active}, online]},
        {text_effect, free_slot},
        {base_behavior, {user, active}}],

Context = context_ADT:create(Spec),
```

```
user:start_link(AgentId, Context)
```

The specification required to create a context ADT must obey the rules in Figure 8.

To start an agent with a given context ADT, the ADT is passed to the `start_link` function which spawns a new agent. The management of the variations in the context ADT is performed through the API we provide. The following call performs a switch on the `status` slot, activating the `online` variation.

```
user:in_cur_context_switch(AgentId, online, status)
```

Similar functions are provided for activating and deactivating the variation in an activatable slot, and for filling a free slot with a variation sent from a remote agent. After filling, the variation in the free slot can be activated normally.

The introduction of a context ADT has some drawbacks. Most noticeably, ADT specifications require an extra effort to be designed. However, the impact on complexity is kept minimum by using a DSL. In addition to that, introducing a variation into an existing context ADT instance requires to change the specification, forcing the programmer to think how to combine variations in a coherent way. In any case, the effort required is similar to write a new set of layer transitions in EventCJ when a new layer is added. Another observation should be made about the choice of limiting the stack size and forcing variations to obey certain constrains, which possibly limits variation capabilities. This design choice advantages safety against flexibility. However, in our experience, more flexibility is not really required. For example, changing active variations by specifying the list of all the active ones (like we showed in the previous sections) is a highly dynamic and flexible mechanism, which gives the programmer more freedom than it is really needed in most scenarios. Even in the examples provided in COP literature, most activation schemas are quite simple and encompass only few variations often in mutual exclusion [5, 6, 17, 29]. Nonetheless, in the spirit of leaving open the exploration of more dynamic solutions, we decided to maintain both activation mechanisms.

The context ADT solution is different from other COP proposals essentially in that it limits the ADT user to correct configurations instead of allowing free interaction and *ex-post* constraints check. However, analogies in the kind of constraints the context ADT allows to express can be envisaged. The context ADT in CONTEXTERLANG resembles approaches based on feature diagrams to express constraints among layers such as the *one-of* or the *all-of* relations. E.g., the `switch` slot clearly allows to express a *one-of* constraint. Finally, note that none of the approaches proposed so far, including the context ADT, relies on some automatically inferred semantics to check variations configurations. Instead, they only ensure constraints determined by the programmer. Investigation in this direction is an open research problem.

3.4 Concurrency: consistency with context change

The combination of COP with concurrency is not easy. While Erlang offers invaluable support for concurrency, integrating actors with run time behavioral change requires careful comprehension of how these aspects interact. In this section we clarify some fundamental points.

A crucial requirement is that behavioral change is *safe*, i.e. a change of the active variations does not corrupt the task in execution. As we will explain shortly, this cannot be achieved by simply forbidding a context change in the middle of message-triggered computations. In fact, this functionality is sometimes required. Our solution is based on shaping CONTEXTERLANG around the following principles:

- **Non interference.** The context of a running computation cannot be altered by a contextual message.
- **Agent authority.** An agent retains ultimate authority on its current context.

The first rule states that if an agent A sends a message to an agent B, triggering a computation cmp on B, no agent (not even A) can change the context of B in the middle of cmp by sending a contextual message to B. This is achieved by processing context-related and other messages one at a time, picking them up from the agent mailbox. Therefore, it is not possible that context-related messages interfere with the execution activated by a standard message.

The second rule states that an agent can change its context arbitrarily during a computation. This principle reminds OO programming, where an object is ultimately responsible for how it responds to messages. The second rule is required in some practical scenarios with non-trivial concurrency patterns. For example, when a client connects to the ContextChat server, some data structures in the `user` agent can be required to be initialized. Examples are the source IP, the client version, or a status variable that must be set to *online*. In addition to these operations, since the client is now connected, the `online` variation must be activated and the `offline` one deactivated. Now consider the case in which these actions (state changes and variation activation) are performed by two subsequent calls from another agent. With an unlucky interleaving, a call coming form a third agent can fall between these two, and find the agent with the status set to *online* but still with the `offline` variation active.

In general, it is not possible, with the functions for variation management seen so far, to execute a variation manipulation atomically with a set of operations. Of course, agents can coordinate to enforce this constraint at higher level, implementing some synchronization mechanism. However this solution requires a lot of error-prone code even for trivial tasks. For this reason in CONTEXTERLANG all the functions like `in_cur_context_switch` have an *immediate* counterpart which has effect on the context-aware agent that calls them. For example when the `user` agent receives the `init` mes-

sage, it atomically initializes its internal data structures and activates the `online` variation atomically.

```
-module(user).
  ...
  handle_cast({init, Data}, State) ->
    % ... initialize the data structure
    user:in_my_cur_context_switch(online, status),
    {noreply, State}.
```

While a message is served, other messages are queued and cannot interrupt it. So atomicity is guaranteed. Interestingly, immediate activation is more general than message-based activation, since context-related messages could be implemented as standard messages which triggers the execution of an immediate activation. To alleviate the programmer from this annoying task we maintain both versions.

4. Validation

In this section, we discuss how we validated CONTEXTERLANG. To demonstrate that CONTEXTERLANG is effective in the development of real-world applications we implemented two prototypes: one is the ContextChat extensively presented in the previous sections, and the other is an autonomic storage server which will be analyzed in the rest. To prove that a language is usable, we studied the critical performance aspects of CONTEXTERLANG through a micro-benchmark and we compared its performance with other COP languages. Then we reimplemented the autonomic storage server in plain Erlang and we compared its performances with the CONTEXTERLANG version.

4.1 Case Study

The design of CONTEXTERLANG was done in conjunction with the development of the ContextChat prototype which has been an immediate testbed for our choices. This approach helped us to design CONTEXTERLANG with the concrete problems or run time adaptation in mind, however, further validation is required to avoid overfitting. Indeed, the instant messaging domain somehow puts a message-based approach in a position of possibly unfair advantage.

The development of the storage server allowed to gain better confidence that CONTEXTERLANG is effective in general. The storage server is an autonomic application which provides a space for generic resources such as web pages or serialized data structures. The application behaves like a key-value map: keys allow to retrieve resources or modify their value. Resources can be stored in memory or on disk. Autonomicity provides that the most requested entities are moved into memory to reduce service time. The disk is used for other resources to avoid excessive memory consumption.

Each resource is implemented as a context-aware agent which reacts to messages like `set_value` and `get_value`. These details are hidden from the user which interacts only with an API module. The implementation of each resource with an agent is normal in Erlang OTP due to the extremely lightweight Erlang processes [21]. This makes the application scalable by simply spawning agents on several machines, because Erlang manages remote messaging in a transparent way. The `on_disk` and the `in_memory` variations can be dynamically activated on each agent. An optional `logging` variation provides a trace of the system execution. Autonomic behavior is implemented in a decentralized fashion: each agent migrates the resource to memory depending on the frequency of the requests it receives.

The development of the application confirmed the design choices of CONTEXTERLANG. Since the `on_disk` and the `in_memory` variations are in mutual exclusion they were managed through a *switch* slot of the context ADT. The `logging` variation occupies an *activatable* slot. The support for initialization and shutting down of variations (Section 3.3) is required to automatically initialize the needed files when the `on_disk` variation is activated and to move the resource in memory when it is deactivated. Since each agent adapts autonomously, the `in_memory` variation activation is performed by the agent itself through the immediate API (see Section 3.3). Note that moving the autonomic capabilities to a centralized engine would require that the adaptation is driven by context-related messages.

As a final remark we observe that all the considerations that motivated our work (see Section 2) could be repeated almost unchanged for the storage server example.

4.2 Performance

Our implementation introduces a performance overhead, because a function call requires to be dispatched over possibly several active variations. CONTEXTERLANG is a prototype and a wide space for optimization is available, e.g. hashing the function lookup. However our evaluation shows that the approach is feasible and already usable. All tests were performed on a laptop equipped with an Intel Core 2 Duo T9500 2.60GHz, 4GB RAM, and GNU/Linux OS. Concerning the languages, the version numbers are: Erlang R13 hipe, Ruby 1.8.7, ContextR 1.0.2, JavaScript Chrome 16.0.912.63, ContextJS Lively Kernel 2, Python 2.7, ContextPy 1.1, PyContext 1.0, SBCL 1.0.45.0, ContextL 0.61.

Microbenchmark. We compare the overhead introduced by CONTEXTERLANG with respect to other COP implementations [4]. The purpose is to compare the message dispatching slowdown introduced by each COP extension. We decided to keep our methodology as simple as possible, following the approach elaborated in [15] for AOP micro-benchmarking: compare methods performance without aspects (i.e. a non-advised method) and with aspects deployed. To remove a source of variability, the comparison is limited to dynamically typed languages.

We assume a message delivery in a non-layered method, as a reference (Table 1, second column). Then we evaluate the time required to dispatch a layered method with 0 to 5 active proceeding layers/variations (columns from 3 to 8). Each method and each partial method increments a global

Language	Basic Call	0	1	2	3	4	5
ContextErlang	540.65 (OTP) / 90.58 (PA) / 9.38 (FC)	815.33	1071.14	1311.59	1531.77	1819.07	2074.73
ContextR	43.52 (Ruby)	768.58	1768.58	2768.58	3768.58	4768.58	5768.58
ContextJS	0.40 (JavaScript)	85.90	158.60	211.00	256.80	299.20	338.30
ContextPy	24.22 (Python)	406.85	661.01	873.50	1163.31	1397.62	1623.49
PyContext	24.48 (Python)	410.66	854.66	1265.21	1668.65	2073.56	2472.16
ContextL	2.2 (Common Lisp)	2.50	3.50	4.30	5.30	6.40	7.40

Table 1. Performance of COP languages in the microbenchmark. All values are in milliseconds.

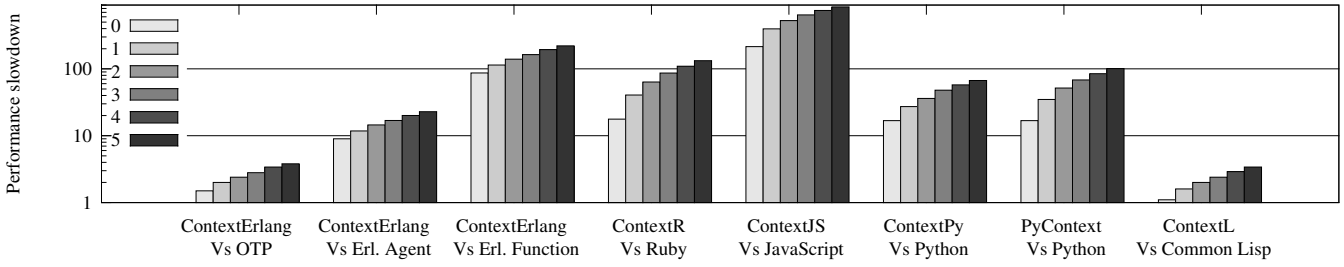

Figure 9. Performance of layered methods compared to the basic methods in various COP languages.

variable (in the CONTEXTERLANG benchmark we used an agent-local variable, since Erlang has no shared state by design). All benchmarks are executed 10^5 times taking the mean over 10 executions, with a complete dry run (therefore 10^6 executions) to achieve steady state of the runtime. Information about warm-up times for each implementation is not easy to find. However benchmarks are running for minutes, and we observed a x10 time factor from 10^5 to 10^6 executions, increasing our confidence on the steady state of the runtimes. In the case of CONTEXTERLANG, COP functionalities are implemented in the OTP library which adds many time-consuming operations due to the built-in fault-tolerance support. Therefore, it has scarce significance to compare message sending to a CONTEXTERLANG context-aware agent with a basic function call. For this reason, we compare it (Table 1, line 2, column 2) not only with a pure Erlang function call (FC), but also with a message to a pure Erlang agent (PA), and with a message to a standard `gen_server` OTP agent. Figure 9 shows the ratio between the time required to call a layered method and a basic method for various languages (note the logarithmic scale). For CONTEXTERLANG we report the comparison with all the three cases.

Previous work [4] highlighted a huge performance impact of COP and motivated research on possible optimizations [3]. Our evaluation confirms this result. Our results also show that CONTEXTERLANG introduces a non-negligible overhead, which, however, is not dissimilar from other COP languages. For example, a CONTEXTERLANG message to a context-aware agent is approximately 87 times slower than a function call in Erlang and 1.5 times slower than a message sent to a `gen_server` standard OTP agent. Note that Figure 9 should be read carefully. For example, results of ContextJS are due to the aggressive optimization of JavaScript compiler and VM which makes basic methods extremely efficient [20]. This leads to the apparently poor performance of the ContextJS COP implementation compared to the basic language in Figure 9. Nevertheless, ContextJS is among the fastest COP extension in our test (Table 1).

Performance on the case study. To overcome the obvious limitations of micro-benchmarking, we estimated the overhead of CONTEXTERLANG in a complete application. We implemented the autonomic storage server in plain Erlang. Variations are simulated by `if` chains switching between different behaviors. Active behavioral variations are stored in each agent's state. Since the logging functionality introduces a uniform overhead, we left it off. The Erlang version resulted in 390 non-comment LOC instead of the 380 non-comment LOC of the CONTEXTERLANG version. While this value is not impressive, it is likely to significantly increase with the number of variations and variations combinations.

In the experiment, each resource is initially created, it is requested 10 times, and then deleted. This is equivalent to starting an agent, delivering 10 messages, and shutting down the agent. We tuned the autonomic behavior so that the resource is initially stored on disk and after the first 2 requests is moved to memory. The measures were taken by repeating this process on all the resources for a variable number of resources, from 1 up to 1000. For each run we took the mean among 10 executions. Figure 10 shows the results. To make the graph more readable we plot the trend of the two executions as the mean over 100 values. The analysis shows that the significant overhead detected by the micro-benchmark becomes almost negligible in a real application.

5. Related work

The problem of dynamic software adaptation to respond to context changes has been extensively tackled from a software architecture standpoint [23]. Over the years, however,

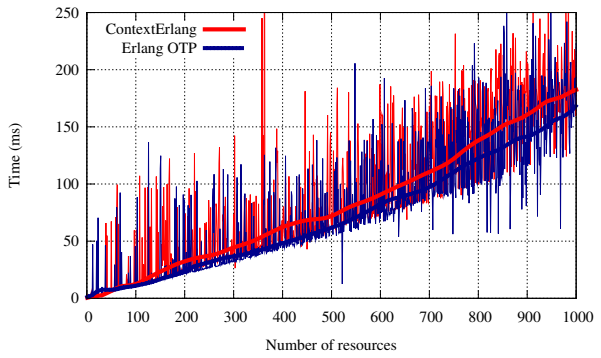

Figure 10. Performance comparison for the autonomic storage server.

language-centered techniques have been progressively investigated leading to the development of *ad-hoc* programming paradigms for context adaptation.

Context oriented programming. COP has been recently explored, starting from the pioneering work on ContextL [9, 10] based on the CLOS metaobject protocol. Over the time, many COP extensions have been developed for different languages such as Python, Smalltalk, Ruby, JavaScript and Groovy. This effort has been extended to less dynamic languages, in which COP extensions are more difficult to implement due to the limited reflective capabilities, such as Java [5, 6, 17, 27, 29]. A comparison of the existing COP languages with a performance evaluation of the available solutions can be found in [4].

CONTEXTERLANG is in the COP tradition since it supports modularization, dynamic activation, and combination of behavioral variations. It differentiates from most COP approaches, since behavioral variations are activated on per-agent bases through context-related messages rather than in a dynamic scope. CONTEXTERLANG *variations* are similar to COP *layers*. The difference is that layers usually contain partial definitions associated with different classes. While nothing prevents a CONTEXTERLANG variation from containing partial definitions referring to different agents, this is scarcely used in applications, since the variation must be activated singularly on each agent. Therefore a CONTEXTERLANG variation is usually associated with a specific agent and contains the partial definition for that agent.

Ambience is a COP language based on AmOS, an object system built on top of Common Lisp [14]. Ambience – designed simultaneously with ContextL – is alternative approach to layer-based COP languages, leveraging multimethods dispatching and context objects. In [14] the authors recognize the need for variations activation by an external monitoring thread. In Ambience the context – and therefore the active variations – is global and shared among all the threads. A monitoring thread can asynchronously change the context of the whole application. In CONTEXTERLANG each agent can adapt individually, as we believe that in certain scenarios this feature is required. For example, in the ContextChat server, per-agent adaptation is crucial to adapt to each single client. As stated by the authors of Ambience, asynchronous activation exposes the system to the risk of behavioral inconsistency. CONTEXTERLANG enforces consistency by design, avoiding that variation activations conflict with other computations (Section 3.3).

Event-based COP. The need for event-based composition and activation has been recognized as an emerging need for COP in our previous work [12], in which we presented the initial implementation of CONTEXTERLANG as a promising solution. As already discussed (Section 2) Kamina et al. [29] also tackled this issue in the EventCJ Java COP extensions.

Jcop [6] is a Java COP extension which introduces two constructs. *Declarative layer composition* allows to express variation activation declaratively through joinpoint quantification. *Conditional composition* activates variations depending on a run time condition. So the developer is relieved from specifying variation activation programmatically in the code. Jcop allows the compact representation of otherwise scattered `with` activation statements, a problem that emerged in the development of ContextChat (Section 2). However, activation in Jcop is always dynamically scoped and can lead to the problem of excessive adaptation propagation.

Aspect oriented programming. COP has a certain degree of similarity with Aspect-Oriented Programming [19], which may be viewed as a general term indicating a family of approaches that support modularization of crosscutting concerns. The main contribution of COP with respect to AOP is to provide specific abstractions for context adaptation. AOP can be indirectly applied for the same purposes and some COP language implementations rely on AOP [6, 27, 29]. However, although AOP frameworks exist which support dynamic aspect activation, such as Prose [24], AOP focuses on compile time feature selection and combination, while the COP core concept is run time activation and combination of behavioral variations. A detailed comparison of the two approaches can be found in [6, 9, 17].

Event-based programming. Event-driven or event-based programming is a programming paradigm in which the flow of control is determined by events that can be triggered and listened according to the Observer pattern. This approach is a contribution to address the problem of concerns not amenable to modularization along the main dimension of decomposition. Implicit invocation (II) languages [22] offer a linguistic support for this mechanism, obtaining better encapsulation of crosscutting concerns and decoupling from other code. The Ptolemy language [25] combines ideas from AOP and II languages. In Ptolemy code blocks are bound to events as closures which can be executed inside the event handler. Since basic behavior can be written in the closure and observers can execute code *around* the execution of the closure, Ptolemy seems to be the II language that most resembles COP techniques.

Other language-level techniques. Subjective dispatch [28] adds a dimension to the receiver-based method dispatch of OO languages, considering also the sender in the dispatch mechanism. COP conceptually operates in a similar way, taking into account the context as a dispatching dimension. Feature oriented programming (FOP) targets crosscutting concerns with the goal of synthesizing programs in software product lines [7] from single units of functionality conventionally called *features*. Features are selected and combined at compile time while COP variations, due to the volatile nature of the context, are activated and combined dynamically.

6. Conclusions and Future Work

Model through the development of CONTEXTERLANG, a COP extension of the Erlang language based on the OTP platform. We discussed the mechanisms through which CONTEXTERLANG supports dynamic software adaptation. We argue that due to the asynchronous nature of context provisioning, context adaptation should be designed taking into account the concurrency model of the language. CONTEXTERLANG constitutes a first contribution in this direction.

Our purposes for the future are twofold. On the one hand we plan to further improve the CONTEXTERLANG implementation, for example by optimizing the code in order to minimize the overhead introduced by the context management. On the other hand we are considering the option of exploiting the coupling between event-based context adaptability and concurrency model investigated in this paper in other agent-based languages, such as Scala.

References

[1] *http://erlang.org*. Reference website for Erlang.

[2] *http://www.scala-lang.org/*. Reference website for Scala.

[3] M. Appeltauer, M. Haupt, and R. Hirschfeld. Layered method dispatch with INVOKEDYNAMIC: an implementation study. COP '10, pages 4:1–4:6, 2010.

[4] M. Appeltauer, R. Hirschfeld, M. Haupt, J. Lincke, and M. Perscheid. A comparison of context-oriented programming languages. In *COP '09*, pages 1–6, 2009.

[5] M. Appeltauer, R. Hirschfeld, M. Haupt, and H. Masuhara. ContextJ: Context-oriented Programming with Java. *Information and Media Technologies*, 6(2):399–419, 2011.

[6] M. Appeltauer, R. Hirschfeld, H. Masuhara, M. Haupt, and K. Kawauchi. Event-specific software composition in context-oriented programming. In B. Baudry and E. Wohlstadter, editors, *Software Composition*, volume 6144 of *LNCS*. 2010.

[7] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, Washington, DC, USA, 2003.

[8] P. Costanza and T. D'Hondt. Feature descriptions for context-oriented programming. In *Software Product Lines, 12th International Conference (SPLC)*, pages 9–14, September 2008.

[9] P. Costanza and R. Hirschfeld. Language constructs for context-oriented programming: an overview of ContextL. In *Proceedings of the 2005 symposium on Dynamic languages*, DLS '05, 2005.

[10] P. Costanza and R. Hirschfeld. Reflective layer activation in contextL. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, 2007.

[11] C. Ghezzi, M. Pradella, and G. Salvaneschi. Programming language support to context-aware adaptation - a case-study with Erlang. *SEAMS: Software Engineering for Adaptive and Self-Managing Systems,International Workshop, ICSE 2010*.

[12] C. Ghezzi, M. Pradella, and G. Salvaneschi. Context-oriented programming in highly concurrent systems. In *Proceedings of the 2nd International Workshop on Context-Oriented Programming*, COP '10, New York, NY, USA, 2010. ACM.

[13] S. González, N. Cardozo, K. Mens, A. Cádiz, J.-C. Libbrecht, and J. Goffaux. Subjective-C: Bringing context to mobile platform programming. In *Proceedings of the International Conference on Software Language Engineering*, Eindhoven, The Netherlands, 2010.

[14] S. González, K. Mens, and P. Heymans. Highly dynamic behaviour adaptability through prototypes with subjective multimethods. In *Proceedings of the 2007 symposium on Dynamic languages*, DLS '07, pages 77–88, 2007.

[15] M. Haupt and M. Mezini. Micro-measurements for dynamic aspect-oriented systems. In M. Weske and P. Liggesmeyer, editors, *Object-Oriented and Internet-Based Technologies*, volume 3263 of *LNCS*. Springer Berlin / Heidelberg, 2004.

[16] C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *IJCAI'73: Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann.

[17] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), Mar. 2008.

[18] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36:41–50, 2003.

[19] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In J. Knudsen, editor, *ECOOP 2001 – Object-Oriented Programming*, volume 2072 of *LNCS*, pages 327–354. Springer Berlin / Heidelberg, 2001.

[20] J. Lincke, M. Appeltauer, B. Steinert, and R. Hirschfeld. An open implementation for context-oriented layer composition in ContextJS. *Sci. Comput. Program.*, 76:1194–1209, 2011.

[21] M. Logan, E. Merritt, and R. Carlsson. *Erlang and OTP in Action*. Manning Publications, 2010.

[22] D. Notkin, D. Garlan, W. G. Griswold, and K. Sullivan. Adding implicit invocation to languages: Three approaches. In *Object Technologies for Advanced Software, First JSSST International Symposium*, volume 742 of *LNCS*, 1993.

[23] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 177–186. IEEE Computer Society, 1998.

[24] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st international conference on Aspect-oriented software development*, AOSD '02, 2002.

[25] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In J. Vitek, editor, *ECOOP 2008, Cyprus*, volume 5142 of *LNCS*, pages 155–179, Berlin, July 2008.

[26] G. Salvaneschi, C. Ghezzi, and M. Pradella. *Context-Oriented Programming: A Programming Paradigm for Autonomic Systems*. Technical Report, arXiv:1105.0069, 2011.

[27] G. Salvaneschi, C. Ghezzi, and M. Pradella. *JavaCtx: Seamless Toolchain Integration for Context-Oriented Programming*. COP '11. 2011.

[28] R. B. Smith and D. Ungar. A simple and unifying approach to subjective objects. *TAPOS*, 2(3):161–178, 1996.

[29] K. Tetsuo, A. Tomoyuki, and H. Masuhara. EventCJ: A context-oriented programming language with declarative event-based context transition. In *Proceedings of the 10nd international conference on Aspect-oriented software development*, AOSD '11, 2011.