Programming Language Support to Context-Aware Adaptation—A Case-Study with Erlang*

Carlo Ghezzi
DEEPSE Group
DEI, Politecnico di Milano
Piazza L. Da Vinci, 32
Milano, Italy
carlo.ghezzi@polimi.it

Matteo Pradella
DEEPSE Group
CNR IEIIT-MI
Via Golgi, 42
Milano, Italy
pradella@elet.polimi.it

Guido Salvaneschi
DEEPSE Group
DEI, Politecnico di Milano
Piazza L. Da Vinci, 32
Milano, Italy
salvaneschi@elet.polimi.it

ABSTRACT

Software applications are increasingly situated in a world where context changes continuously. At the same time, applications need to provide continuous service, and the service provided often needs to change in order to adapt to the new contexts. Context-aware adaptation can be greatly facilitated by using programming languages that natively support high-level features to deal with contexts, context changes, and context-aware behaviors. Although context-oriented programming has been around for a while, most existing efforts focus on incorporating context-oriented features in languages that are not primarily oriented to concurrency, distribution, and dynamic reconfiguration. These features, however, characterize most pervasive context-aware situations. In this work, we illustrate how context-aware programming primitives may introduced in the parallel and distributed Erlang programming language. We also present an extended example, which illustrates the benefits of using our extension (*ContextErlang*) to design context-aware pervasive applications.

Categories and Subject Descriptors

D.1 [**Software**]: Programming Techniques—*Object-oriented Programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages, Design

Keywords

Context, Self-adaptative software, Context-oriented programming, Erlang, OTP platform

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

SEAMS '10, May 2-8, 2010, Cape Town, South Africa

1. INTRODUCTION

The need for a complex system to dynamically adapt to a changing context is becoming common to a wide range of scenarios. Managing context-awareness became especially critical with the advent of mobile and ubiquitous computing applications. Since context-depending behaviors typically crosscut the system functionalities, managing context-dependent features in a systematic and effective way became a key software challenge.

The problem of dynamic software adaptation to respond to context changes has been mainly tackled so far from a software architecture standpoint [9, 20, 24, 14, 23]. This paper investigates instead a different and complementary approach by focusing on programming language support. *Context-Oriented Programming* (COP) [18] has been recently proposed as a viable approach and language-level support for context management. Although the natural application of COP is in ubiquitous systems [6], because of the strong influence of context on the behavior of such applications, so far a lot of work has been done with languages that are not specifically aimed at highly distributed and concurrent programming.

The contribution of this work is the introduction of context-oriented programming techniques in a language that natively supports distribution and concurrency and the investigation of the effectiveness of the approach to support dynamic context-aware adaptation. Specifically, we introduce COP in the concurrent and distributed Erlang programming language [1]. Erlang comes with the OTP platform, which provides a rich set of libraries supporting reliable, large-scale, distributed, dynamically evolvable applications. We propose a solution for code aggregation across the crosscutting concerns of contexts and for the key features of dynamic variation activation, dynamic variation composition, and variation transmission, that naturally fits the standard architecture of the Erlang/OTP applications. The resulting *ContextErlang* enables the development of COP systems with a high degree of availability, thanks to the well known fault-tolerance features provided by the OTP architecture.

The paper is organized as follows. In Section 2 we introduce COP and its main features. In Section 3 we analyze related work. Section 4 presents Erlang and the OTP platform. ContextErlang is presented in Section 5. An extended illustrative example is in Section 6. The conclusions and future work are discussed in Section 8.

2. CONTEXT-ORIENTED PROGRAMMING

COP addresses the need for applications to behave differently accordingly to the changing run-time context in which they are embedded. This goal is achieved by providing the abstractions that enable application context-awareness without hard-wired conditional statements spread over the application code [10]. Because context dependency is a cross-cutting concern for a system, one of the key elements of COP deals with modularization in stating the partition of programs into *behavioral variations*.

COP provides means for the dynamic activation and composition of such behavioral variations [4]. Behavioral variations are grouped in *layers*, which are first-class entities that can be referenced at run time. The adaptation to a context is obtained through *layer activation*. Ad-hoc language constructs ensure that the partial definitions inside the layer are activated at run time and thus can influence the behavior of the program accordingly.

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 cross-cutting concerns. A discussion of the difference between the two approaches can be found in [10] and [18].

3. RELATED WORK

As mentioned earlier, software engineering research in the area of context-aware software mostly addressed the issues of dynamic adaptation from a software architecture viewpoint, both exploring the architectural styles that best support adaptation and the mechanisms that can be exploited to achieve adaptation, given a specific architectural model or a specific style. For example, [20] shows how the C2 architectural style enables dynamic evolution. C2 introduces a sharp distinction between computation and communication and constrains communication among components to minimize their inter-dependencies. Components can consume data from only one connector and produce output data only on one connector. Connectors, instead, can accommodate any number of components or other connectors. Every communication is carried out in an asynchronous way through connectors. Adaptation can be carried out without suspending computation, and is achieved through addition or removal of components.

Other interesting examples of dynamically adaptable architectures are the Rainbow framework [14] and the planning-based approach presented by [23]. Architectural solutions supporting self-adaptation, inspired by other natural domains (such as biology or physics), are also currently investigated. For example, [7] investigates an architectural style for large networks, based on a formal mathematical study of crystal growth. Di Nitto et al. [13] study a set of the most relevant bio-inspired principles that may be applied to achieve dynamic self-adaptation. Cavallaro et al. [8] present an approach for designing self-adaptive service-oriented applications based on self-matching components called *service tiles*. Taylor et al. [24] analyze different architectural styles from an evolution and adaptation viewpoint, trying to understand how and how much the system's behavior can be changed.

This paper addresses the problem of dynamic context-aware software adaptation from a programming language viewpoint, which provides a different–yet complementary–perspective. Context-aware programming has been recently explored, mostly within the functional programming languages community, starting from the pioneering work on ContextL by P. Costanza [10, 11, 12]. ContextL is an extension of CLOS (the Common Lisp Object System) and built on top of the CLOS Meta-object Protocol. In ContextL, a language entity can be declared to exist only inside a certain layer. When the layer is explicitly activated, the related entities are visible in the activation scope, and they can be referred from code and participate in code execution. Classes, generic functions, methods, and the values bound to certain slots can be defined inside layers. When a layer is activated, the associated entities interact with the other CLOS entities to enable context-aware behavior. For example, if a layer containing a class is activated, that class takes part in the inheritance chain and becomes part of the method dispatching mechanism.

A different solution is adopted by AmOS [15], a prototype-based object system built on top of Common Lisp and independent from CLOS. In AmOS, for any method call, applicable target methods are firstly looked up in the current activation, then in further enclosing lexical scopes. As the top level activation scope is reached, rather than stopping and returning an error, search continues in a graph of *context objects* delegating to each other. A context-manager thread can change dynamically the delegation relationships among context objects in order to achieve a context-dependent behavior.

COP is also explored in the context of existing scripting languages. ContextPy [22] is a Python implementation of COP that directly moves the concepts of layers and dynamic layer activation to the Python language. PyContext [25] is another COP framework for Python. It provides an extension to the concept of method layers to support implicit activation of layers, and proposes dynamic variables as mechanisms to access context-dependent state. Implementations in other programming languages such as Smalltalk [17], Ruby [21], JavaScript and Groovy [2], or Java [18, 5] have been also developed. A fairly complete comparison of the existing languages with a performance evaluation of the available solutions can be found in [4].

Our current work pushes COP techniques into the distributed, concurrent, and functional programming language Erlang. The result is a very promising tool for the design and implementation of dynamically adaptable, context-aware software systems.

4. ERLANG AND THE OTP PLATFORM

Erlang [1] is a general-purpose functional programming language and concurrent runtime system designed at the Ericsson Computer Science Laboratory since 1986 and released under an open source license in 1998. While the language provides the basic functionalities for software development, practically any Erlang application is based on the OTP platform which is a powerful library and a set of procedures for structuring real-world applications.

Erlang is based on some strong features: single assignment, dynamic typing, and an actor model for concurrency. It provides support for writing distributed programs, since processes can be distributed over different machines. Many features of this language such as concurrent processes, scheduling, memory management, distribution or networking are commonly associated with operating systems and middle-

wares, rather than to programming languages. Here is a short list of its most appreciated features.

- *Concurrency* - Erlang processes are extremely lightweight, have no shared memory, and communicate by asynchronous message passing. Erlang supports applications with a large number of concurrent processes. No requirements for concurrency support are expected from the host operating system.
- *Distribution* - Erlang is designed to run over a virtual machine (thus independently from the underlying operating system) in a distributed environment. An Erlang virtual machine is called a *node* and a distributed system is a network of nodes. Communication is transparent with respect to distribution.
- *Robustness* - Erlang support for fault-tolerant systems is proved by a well known history of high availability communication systems. This is achieved through the structure of the Erlang applications, where some processes work as *monitors*. Processes in a distributed system can be configured to fail-over to other nodes in case of failures and automatically migrate back to recovered nodes.
- *Hot code upgrade* - Usually code upgrade in a system implies a downtime during which the upgrade is performed. Erlang was created with telecommunications systems in mind, which cannot be stopped for software maintenance. Erlang allows program code to be changed in a running system. During the transition, both old code and new code can coexist. It is thus possible to install bug fixes and upgrades in a running system with limited or no interference with the running application. This feature is key to long-lives and ever-running pervasive systems.

Erlang comes with the so called OTP (Open Telecom Platform), which is a set of libraries and procedures used for implementing fault-tolerant, large-scale, distributed applications. Complete tools are available such as a web and FTP server, a CORBA Object Request Broker, and the distributed Mnesia database.

The OTP documentation is based upon the OTP Design Principles, which is a set of rules for how to structure an Erlang application in terms of process interaction, code modularization and overall architecture. An Erlang/OTP application should follow the pattern of a *supervision tree* (see Figure 1). In this model processes are structured in a hierarchical fashion and are divided in two sets: *workers* and *supervisors*. Workers are processes which perform computations (e.g. servers are workers), while supervisors are processes whose role is monitoring the behavior of workers. For example a supervisor can restart a worker if it crashes. This design principle is the base for designing fault-tolerant applications in Erlang.

The concept of *behavior* is also central in OTP. The basic idea is that in a supervision tree many processes enact similar patterns. For example, workers are typically servers that exchange messages, or event handlers such as error loggers, while the supervisors only differ with respect to the processes they supervise. The OTP generalizes these common patterns and gives a ready implementation of the generic structure (i.e., in OTP jargon, *the behavior*), while the user

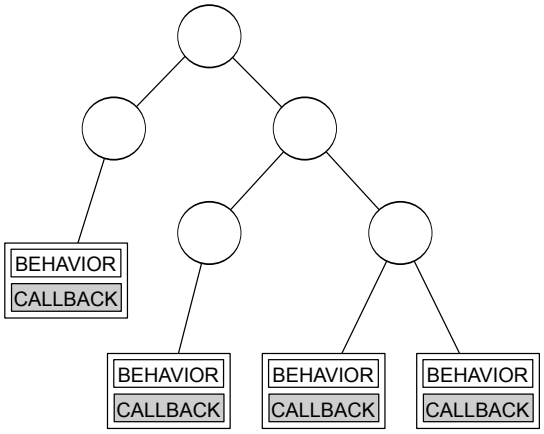

Figure 1: The general structure of an Erlang/OTP application. Internal nodes of the tree represent supervisors. Leaves represent workers.

has to implement only the specific part that exports a predefined set of functions (the so called *callback* module). These are standard behaviors provided by Erlang/OTP:

- `gen_server` for implementing the server of a client-server architecture.
- `gen_fsm` for finite state machines.
- `gen_event` for event handling functionalities.
- `supervisor` for a supervisor module in a supervision tree.

This kind of code structuring makes programs easier to understand by other programmers, and prescribes a general architecture that should be common to all OTP applications. In the case of a server, the behavioral module provides functionalities for message passing, error handling and fault-tolerance, while the callback module implements the actual actions the server has to perform when a request is issued.

A simplified example of a behavior module for a generic server and a callback module are shown in Figure 2. The server manages message passing with the clients, receiving messages and sending back the answer. *Call* messages are used for synchronous requests, while *cast* messages are the asynchronous ones (i.e. the client just sends a request but does not wait for an answer). Of course, generic modules in real applications typically support features such as error handling, process monitoring, hot code replacement or fault tolerance and are thus much more complex. The callback module contains the actual implementation of each single functionality. A client usually calls a function on the server using the API exposed by the callback module (it is an OTP convention that the callback module defines both the callback and the API functions), for example:

```
server:alloc(Item, Key).
```

Otherwise, the server can be called directly by using its registered name (in this case simply `server`):

```
gen_server:call(server, {alloc, Item, Key}).
```

When a client performs a call to the server, procedure call is turned into a message which is sent to and then handled by the server, waiting in a *receive loop*. Note that although only two functions are shown due to space reason, the code in a complete callback module does not add any particular complexity to the example given, being a simple collection of back-end functions, because all the advanced features are implemented in the generic component.

5. CONTEXT ORIENTED PROGRAMMING IN ERLANG

COP is generally considered as an extension of object-oriented programming [4]. Being Erlang a functional language, we had to adapt the required COP paradigm to its functional model. However, inside an Erlang system a sort of object-oriented structure can be seen in the fact that its architecture is made of several autonomous entities that typically hold a state and exchange messages during processing.

In an OTP-compliant application, as we have seen before, these components are made of a generic part and a callback module. While the callback module implements the specific functionalities of the component, and it is directly influenced by a context change, the functionalities associated with the generic module, such as message management, error handling, or fault tolerance, are in general not context-dependent. A ContextErlang application is designed as a set of components, each made of a single behavior module and several callback modules. Each callback module is used to implement a behavioral variation for the component and it is bound at run time to the behavior module.

We refer to these different callback modules as *variations*. A variation contains the declarations of all the functions which implement a behavioral change; these functions take effect when the variation is *activated*. Activation occurs when the variation is dynamically bound to the application. It is worth noticing that the behaviors and the whole supervisor tree structure do not change during the execution even if a variation activation occurs. What changes according to the context is only the lower part of each leaf of the supervisor tree (i.e. the variation), which is referenced on-the-fly.

In order to clarify our approach we give a concrete example of our ideas. We suppose that a mobile phone may receive signals from an access point or from the other phones. Occasionally, it can work as a "bridge": if another phone is out of the range of the access point, the first phone can receive the signal from the access point and forward it to the other phone. Of course, this behavior is not necessarily always desired, for example because for security reasons the phone might not trust other phones.

We can model this scenario from a context perspective view. Message receiving is not influenced by the external context because both the messages directed to the phone and messages for the other phones must be handled. However, after message reception, context-awareness comes into play. Depending on the external context, the phone can adopt a "cooperative behavior" in which it acts as a bridge, or a "selfish behavior" in which the phone ignores all the incoming messages that are not directed to it. As mentioned, the selfish behavior mode is selected in an untrusted situation. It can also be selected in a context where the battery level of the phone acting as a bridge is below a certain threshold. In the application, this scenario can be modeled by a component `phone` that is context-aware. `phone` includes a generic module implementing the receive loop and waiting for incoming messages; this module is the context-invariant part of the component. The "cooperative" and "selfish" behaviors can be modeled as variations of the `phone` component, which are callback modules associated to the generic module. After a message reception, the generic module invokes a callback module for message processing. When the `cooperative_variation` is active, messages are forwarded to the proper destination, when `selfish_variation` is active, messages directed to the phone itself are processed and other messages are dropped. The switch from one behavior to the other is obtained through a variation activation, which is driven by the phone itself on the basis of a certain policy. When a variation activation occurs, the generic module of the `phone` component remains the same; a different callback module (i.e. a variation) is referenced on-the-fly by the generic module.

While in the previous example we showed a case with only one variation per component at a time involved (`cooperative_variation` *or* `selfish_variation`), a *set* of variations can be activated on a component: the resulting effect is a combination of the features implemented by the different variations. This case is discussed in the sequel.

The introduction of variations is part of our proposal for adding COP features to Erlang/OTP. Variations represent the behavioral changes at the component level. The activation of a variation triggers a modification that affects only a single leaf of the supervision tree. However, in general, the reaction of the application to an external context change can require that more than one component is adapted. We address this issue introducing in ContextErlang the concept of *layer*. In our solution a layer indicates the set of variations which are activated on each component of the application in reaction to a certain external context change. We indicate this application-wide change that involves a set of components as a *layer activation*.

A well known point addressed by COP is code modularization with respect to context adaptation. Two options have been proposed in literature [18]: the modularization along *layers*, that groups behavioral variations associated with the same layer, or along *classes*, which fragments each layer by putting its elements in the related classes. Since Erlang is not an OO language, the previous classification cannot be applied strictly. However the modularization along classes is probably more suitable to describe ContextErlang variations because in ContextErlang the application is modularized in component and the behavior of each component can be modified by component-specific variations. Note that while a variation is specific for a given component, *variation activation* (see Section 5.1) is done on per-process base, since there can be different processes running different *instances* of the same component.

While ContextErlang variations are component-specific, ContextErlang layers are application-wide behavioral adaptations. This is in conformance with COP layers which are first class entities that can affect different elements of an application. For example in ContextJ each class defines how its instances react to a layer activation; when a context is activated all the objects in the activation scope adapt themselves according to the what specified by their class. ContextErlang layers are built on top of the variations, because each layer is mapped on a set of variations for each specific

```
-module(gen_server).
...
loop(Mod, State) ->
receive
  {call, From, Request} ->
    {Response, State2} =
        Mod:handle_call(Request, State),
      From ! {Mod, Response},
      loop(Mod, State2);
  {cast, Request} ->
    State2 = Mod:handle_cast(Request, State),
      loop(Mod, State2)
end.
...

call(Name, Request) ->
  Name ! {call, self(), Request}.
...
```

```
-module(server).

%----------------
% API functions

alloc(Item, Key) ->
  gen_server:call(?MODULE, {alloc, Item, Key}).

free(Key) ->
  gen_server:call(?MODULE, {free, Key}).
...

%---------------------
% Callback functions

handle_call({alloc, Item, Key}, State) ->
  do_alloc(Item, Key, State).

handle_cast({free, Key}, State) ->
  do_free(Key, State).
...
```

Figure 2: A simplified behavioral part of a generic server (left) and its callback module (right).

component. When a layer is activated, the activation affects all the related components of the system: a context manager maps the layer activation on the variations that must be activated for each component.

As a final remark, we notice that our notion of *context* is the complete set of behavioral variations that are activated at a given instant inside the application. Because variation activations are the way each module reacts to a layer activation, the context of the application can be seen as the set of the active variations of each component of the system, or, with an application perspective, as the active layers.

Since more than one variation at a time can be active in a component, in the next sections we describe how multiple variations interact in changing the behavior of a component. Also, we show how variations can be provided to the components of a remote Erlang node so that those components can be enabled to react to unforeseen situations.

5.1 Dynamic variation activation

A key feature of context-oriented programming is the run time activation and composition of behavioral variations of an application. This can be achieved in various ways, for example by dynamically changing the delegation relationship between objects [16] or the inheritance relationships among classes [10]. However solutions for context activation leveraging on run time change of inheritance relationships and dynamic method dispatching are typically OO-based, and clearly not applicable in the Erlang language.

While in the OTP design principles a single callback module is associated with each component of the application, in our solution we allow different callback modules to be used at the same time by the same component. Since we implement variations as callback modules, we have a sequence of variations that affect the component behavior at the same time, and this sequence depends on the external context. A variation starts by affecting the behavior of the application when is *activated*. Variation activation occurs when an external component (i.e. a *context manager*) sends a `change_variations` request to a context-enabled process stating explicitly which variations must be active.

When the behavior module receives a call request, the corresponding function to be executed is searched inside the ordered sequence of the active variations. The composition logic is given by the lookup algorithm inside our modified "context-oriented" behavior modules. Being each variation implemented as a different callback module, the resulting code organization meets the requirements of context-oriented programming in partitioning the application code according to the context crosscutting concern.

The ordered sequence of variations results in a simple composition schema in which variations are arranged hierarchically, in a stack. The function to be called for a given message is searched in the top-of-stack variation: if the search is successful then the call is performed, otherwise the search goes on over the subsequent variations down along the stack. Variations aimed at managing more general contexts are arranged at the bottom of the stack, while more specific variations are in the upper part of the stack and thus they have precedence in call dispatching. The idea of searching over the modules is used here only to better explain our model; in the implementation, a list of the context-enabled functions exported from the variation is kept into a data structure, and subsequently used for function dispatching.

The stack solution enables a simple set of desirable behaviors. First of all a variation can overload functions declared in other (more generic) variations by simply being placed in an upper part of the stack. Function overriding can be obtained in the same manner implementing the new version of the function in an upper variation. Adding new capabilities to a module is straightforward: if a function declared in a variation is not located anywhere else in the variation stack, the presence of that context simply adds that function to the capabilities of the component. Consider the following situation, which illustrates the bindings established when a certain context is entered and how the function calls issued by agents are resolved accordingly.

```
-module(variation1).
-behavior(context_gen_server)
  ...
handle_call({funA}, State) ->
        io:format("[variation1]: funA executed~n"),
        Reply = ok,
        {reply, Reply, State};

handle_call({funB}, State) ->
        io:format("[variation1]: funB executed~n"),
        Reply = ok,
        {reply, Reply, State};
  ...
```

```
-module(variation2).
  ...
handle_call({funA}, State) ->
        io:format("[variation2]: funA executed~n"),
     Reply = ok,
     {reply, Reply, State};

handle_call({funB}, State) ->
        io:format("[variation2]: funB executed~n"),
     Reply = ok,
     {reply, Reply, State};

handle_call({funC}, State) ->
        io:format("[variation2]: funC executed~n"),
     Reply = ok,
     {reply, Reply, State};
  ...
```

Figure 3: The code of the callbacks modules *variation1* and *variation2*, simplified for better readability. The *behavior* attribute is declared in the callback module that provides the initialization and the cleanup functions.

Figure 3 shows an example of the concepts seen so far. Suppose that `variation1` implements the functions `funA()` and `funB()`, and only `variation1` was activated in component `agent`.

```
%% In an external context manager
[agent:change_context([variation1]).]

agent:funA().
agent:funB().
agent:funC().

$>[variation1]: funA executed
$>[variation1]: funB executed
$>Error: funC not exported
```

Since `variation1` contains an implementation of both `funA` and `funB` and `variation1` is the only variation present in the component, functions `funA` and `funB` are directly called. The call to `funC` fails because such function is not present. Now suppose that another component that manages the context switches in the system adds a new `variation2` on top of `variation1` (Figure 4). Let us assume that `variation2` implements `funC()` and its own version of `funB()`. While the component behaves the same upon receiving a request for the execution of `funA()`, the definition of `funB()` in `variation2` overrides the one in `variation1`. The following fragment illustrates the context-dependent bindings established in the new situation.

```
%% In an external context manager
[agent:change_context([variation2, variation1]).]

agent:funA().
agent:funB().
agent:funC().

$>[variation1]: funA executed
$>[variation2]: funB executed
$>[variation2]: funC executed
```

When `funA` is called, `variation2` is searched for the function, but the search fails. The search continues in lower

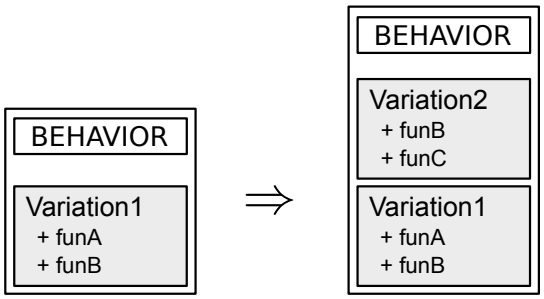

Figure 4: The activation of *variation2* over *variation1*. The resulting module implements the union of the functions in the two variations. Functions with the same signature implemented in both variations are taken from the upper variation.

variations, and succeeds in `variation1`. The same happens with `funB`, but it is immediately found in `variation2`, so this version is executed. The call to `funC` is now performed because it is implemented in `variation2`.

5.2 Variation Composition

Context composition is not only obtained through the context-stack mechanism in the sense that the behavior exposed by a component comes from the arrangement of all the active variations, but it also provided by the `proceed()` call from inside a callback function. `proceed()` calls the subsequent eligible callback function in the context-stack. Suppose that a function `f()` is implemented in different variations of the stack: only the first one is called because of the mechanism described previously. However, if inside the called version of the function a call to `proceed()` is performed, the stack is searched for the next variation implementing `f()` and that implementation is called.

The following example is taken and adapted from [18]. Suppose that a component `person` is part of the system, whose callback module `person_base_variation` implements a `display` function that simply shows some information:

```
-module(person_base_variation).
  ...
handle_call({display}, From, State) ->
io:format("[Person] Name: John; Surname: Smith"),
     Reply = ok,
     {reply, Reply, State};
  ...
```

The module `employment_variation` implements a `display` function that prints some data related to the employment and make use of the `proceed()` call:

```
-module(employment_variation).
  ...
handle_call({display}, From, Tab) ->
        proceed(),
        io:format("[Employer] Name: aFirm;
                                City: London"),
        Reply = ok,
        {reply, Reply, Tab}.
  ...
```

When invoking `display` on the component `person`, the call is dispatched to `person_base_variation` which is the only callback module available. After the first call to `display`, an external component activates the "employment" variation. Now, the call to `display` is dispatched to the "employment" variation because it is the first one in the variation stack. When in `employment_variation`, `proceed()` is called, the next variation in the stack implementing the `display` function is searched and `display` is called.

```
person:display().

%% In an external context manager
[person:change_context(
    [employment_variation,
     person_base_variation, person]).]

person:display().

$>[Person] Name: John; Surname: Smith
$>
$>[Person] Name: John; Surname: Smith
$>[Employer] Name: aFirm; City: London
```

By using this feature, one may add functionalities to an existing function simply by introducing a new variation that wraps the calls to the function. In this sense `proceed()` is similar to the call to `super()` in object-oriented languages that performs calls to the same method implemented by a parent class.

5.3 Variation Transmission

A component of the system could reach a state in which it cannot cooperate usefully with the other components because it is not able to manage either the interaction with them or with the environment. In other words the component could lack the support for the context that it is required to manage. Note that the notion of context may include both the interaction with other components, and events coming from the environments.

Because of Erlang's dynamic code loading, the actual binding between a fully-qualified call (i.e. a call with the syntax `module:function()`) and the module implementing the call is done at run time. Therefore variations can be dynamically loaded during system execution.

A complex Erlang system can be distributed over several different machines, each of which can execute one or more Erlang nodes. After a context change, if the behavior of a component of the system must be modified by activating a variation currently unavailable on that node, the variation can be simply sent by a context manager and loaded on-the-fly. This is done by performing the following call:

```
context:send_variation(node@host, newVariation)
```

After that, the new variation can be activated, thus affecting the behavior of the node. Remote activation is done via the *rpc* call:

```
rpc:call(node@host,component,
    change_variation,[newVariation,groudVariation])
```

In certain scenarios, a system must be able to react also to a sequence of events that was not originally considered, when the system was built or deployed. So, it is important to design a solution that is able to adapt to such unpredicted situation. The variation transmission feature is clearly vital to this end, since it is possible now to design and build applications in which the required behavior is not completely stated at deployment time.

6. AN EXTENDED SCENARIO

In this section we consider an emergency scenario in which we apply the concepts presented before. This scenario is part of a set of use cases that are being used to validate our research results in the area of self-managing situational computer applications [3]. A mountaineer, during a climb, falls off a cliff and is severely injured. He has a mobile phone he can use to contact a base station asking for help. The base station is not meant to manage rescue operations, so it contacts an emergency management center. The mountain is rigged up with sensors (e.g. webcams, or to detect conditions that may lead to landslides or avalanches), and all the sensors, the base station, and the instrumentation taken by the rescue people are able to communicate with each other.

In normal conditions, the sensors interact with tourists, for example to send them data or turistic information. Also the sensors are devoted to collect meteorological data. This "normal context" can be simply modeled using a `tourist_service` variation and a `meteo_info` variation. These variations are both active and implement different functions, exposing an overall behavior which is the union of the available functions. When a request for help is received, the whole system switches to an "emergency context", because all the components must behave in a different manner. In fact all the sensors now must cooperate in order to facilitate the rescue and the base station must coordinate them and manage the communication with the rescue team. For example, if one of the sensors were in fact a videocamera, it could act as a remote "eye" for the emergency management center to localize the mountaineer. It is reasonable that in order to concentrate all the resources in the rescue operations, request from a tourist could be served during this phase. The switch from the "normal context" to the "emergency context" can be achieved deactivating the `tourist_service` and the `meteo_info` variations activating an emergency variation each component of the system.

For example the `sensor_emergency_rescue` variation on the sensors is able to manage the requests from the base station and from a rescue team aimed at collecting information useful for the rescue operations. Instead a request from a tourist is simply discarded with a *service not available* answer. A situation can be envisaged in which, in addition to the rescue team dispatched by the emergency management center, help can be found in the rescue area in the form of other mountaineers. The phones in the hands of the mountaineers are obviously not equipped with all the applications that can be useful in an emergency context. For example in this context a phone should directly interact with the phones of other rescuers, acting as a bridge for a low signal from the injured mountaineer and the base station. However a `rescue-bridge` variation cannot be simply activated on the phone but it must be sent to the phone by the base station and then activated on-the-fly on the phone.

The application might also look for trusted people in the vicinity of the rescue area, who also happen to have the capability to help the injured mountaineer (e.g. because they are doctors). In this case the phone should help the rescuer to locate the mountaineer, or provide medical support to the occasional rescuer. For example the phone could load an application that receives messages from the base station that guide him to the position of the injured person. This can be obtained through the remote loading of a `rescue_guide` variation that simply displays an arrow that shows to the occasional rescuer the direction to follow to find the injured mountaineer. In case of availability of good communications facilities, an additional `broad_band_rescue_guide` variation can be activated on top of the previous one. Some functions are redefined in the upper variation resulting in the overriding of the bottom variation functions. For example a *display* function can now draw an entire map on the screen of the phone instead of a simple arrow. Similar considerations are valid for an application that helps the occasional rescuer to provide some medical support, where instructions can be simply textual, the voice of a remote operator, or some support of images and animations can be available if the bandwidth gets broader and the associate variations are subsequently activated.

If the injured mountaineer does not have a phone, the sensors distributed in the area can be used to find him using a camera installed on each sensor. The base station can send a request for the presence of a man in the area of vision of the sensor. The `sensor_emergency_rescue` variation contains a `find_man` function that analyzes the pictures taken by the sensor and can answer to the request of the base station. However in presence of fog, the sensor cannot be sure that there is a man near it or simply a shadow. This scenario can be addressed with the use of a variations composition. A `foggy_search_man` variation can be activated on top of the `sensor_emergency_rescue` variation. When the request from the base station arrives, it is dispatched to the upper variation, that tries to detect the presence of a man calling `proceed()` (i.e. activating the `find_man` function in the lower variation). Because of the presence of uncertainty in this knowledge, instead of simply returning the answer to the base station, the nearest sensors are queried for the presence of a man. The `foggy_search_man` variation can implement an agent that starts a negotiation algorithm so that the decision of the presence of the man is the result of a voting phase. When an agreement is achieved, the answer is given back to the base station.

7. IMPLEMENTATION AND PERFORMANCE

We implemented our COP extension to the Erlang/OTP platform as an OTP behavior module `context_gen_srv`. It provides all the functionalities of a `gen_server` extended with the variation activation and composition described in the previous paragraphs.

We implemented context awareness in the `gen_server` behavior because we believe that this is the most influenced by context issues among the OTP behaviors, since external context impacts on the way a request is managed. However as a future work we plain to consider if it is sensible a context-aware version of the other OTP behaviors.

The programmer is expected to simply implement the variation modules that can be dynamically activated during execution. None of the callback modules implementing different variations has a privileged role. This means that in principle *all* the behaviors of the component can be put in dynamically activated variations. In any case one of the variations must contain an `init` function, which is expected to initialize the process and the `code_change` function used for hot code replacement. It is probably a good practise to keep one callback module almost unchanged, and implement these functions inside that "ground" variation. Moreover while a module implemented through a generic behavior can be directly called from clients through a registered name, it is common in the OTP design principles to put in the callback module the API calls that wraps the calls t the server. Therefore a client simply invokes a function on the callback module and the callback modules interacts with the server. According to this paradigm, the "ground" callback module should expose the functions exported by all the behaviors, hiding the interaction with the server. This is of course no option if a variation is sent to a node at run time (see Section 5.3) and adds some new callback functions.

Our implementation introduces a performance overhead, because a function call requires to be dispatched over the several active variations. We give two evaluations of the performance of our system. In the first case we consider a `context_gen_server` that receives a function call with five active variations with only the last variation implementing the required function. The cost of this call is compared with a call to an OTP generic server with a single callback module implementing the function (Table 1, first row).

In the second case we analyze the overhead introduced by the `proceed()` call. We consider five active variations; when the top one is called it calls `proceed()` on the `context_gen_server` which dispatches the call to the next variation. The process is repeated up to the call to the last variation which simply returns. This implementation is compared with the same five variations simply calling each other without any "smart" dispatching mechanism (Table 1, second row).

The results show that the introduction of the variations mechanism introduces a significant overhead, which is in line with other COP implementations [4]. However the tests were executed on a first implementation and a wide space for optimization is available, such as implementing the function lookup using an hash table. All tests were done on desktop hardware (Intel Core 2 Duo T9500 2.60GHz with

	OTP		ContextErlang	
	Mean	**Median**	**Mean**	**Median**
Proceed Test	16	18	34	32
Call Test	8	8	19	18

Table 1: Performance Evaluation of ContextErlang compared with a pure OTP implementation. All values are in microseconds.

4GB RAM).

8. CONCLUSION AND FUTURE WORK

This work presented an approach for Context Oriented Programming in Erlang, called *ContextErlang*. COP provides support for the implementation of applications whose behavior depends on the context in which they are executing. Erlang is an industrial-strength programming language for the development of distributed fault-tolerant systems. It is therefore natural to combine them to satisfy the typical requirements of distributed, ubiquitous, and adaptive applications. We have introduced COP in Erlang through the concepts of dynamic variation activation, dynamic variation composition and variation transmission. In the future we plan to continue with the development of ContextErlang, covering all the OTP behaviors and realizing an infrastructure for context-aware applications based on ContextErlang.

At present our prototype implementation works under the hypothesis that context-change messages are issued by some components of the system that actually manage the context status of the system. An aspect yet to be clearly investigated is how structure, model, and obtain in practice contextual information from the environment. As far as these issues are concerned, our approach provides a simple yet strong foundation upon which we plan to build and integrate possible solutions defined in the related literature.

9. REFERENCES

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

[2] *http://www.swa.hpi.uni-potsdam.de/cop/implementations/index.html*.

[3] *Self Managing Situated Computing*. ERC Advanced Investigator Grant N. 227977 [2008-2013], *http://www.erc-smscom.org/*.

[4] M. Appeltauer, R. Hirschfeld, M. Haupt, J. Lincke, and M. Perscheid. A comparison of context-oriented programming languages. In *COP '09: International Workshop on Context-Oriented Programming*, pages 1–6, New York, NY, USA, 2009. ACM.

[5] M. Appeltauer, R. Hirschfeld, M. Haupt, and H. Masuhara. Contextj: Context-oriented programming with java. In *Proceedings of the JSSST Annual Conference 2009*, Shimane University, Matsue, Shimane, Japan, September 16, 2009.

[6] M. Appeltauer, R. Hirschfeld, and T. Rho. Dedicated programming support for context-aware ubiquitous applications. In *UBICOMM '08: Proceedings of the 2008 The Second International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, pages 38–43, Washington, DC, USA, 2008. IEEE Computer Society.

[7] Y. Brun and N. Medvidovic. An architectural style for solving computationally intensive problems on large networks. In *SEAMS '07: Proceedings of the 2007 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, page 2, Washington, DC, USA, 2007. IEEE Computer Society.

[8] L. Cavallaro, E. Di Nitto, C. A. Furia, and M. Pradella. A tile-based approach for self-assembling service compositions. In *15th IEEE International Conference on Engineering of Complex Computer Systems*, St. Anne's College, University of Oxford, 2010. (To appear).

[9] B. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee. Software engineering for self-adaptive systems: A research road map. In *Dagstuhl Seminar Proceedings*, volume 8031. Springer, 2008.

[10] P. Costanza. Language constructs for context-oriented programming. In *In Proceedings of the Dynamic Languages Symposium*, pages 1–10. ACM Press, 2005.

[11] P. Costanza. Context-oriented programming in contextl: state of the art. In *LISP50: Celebrating the 50th Anniversary of Lisp*, pages 1–5, New York, NY, USA, 2008. ACM.

[12] P. Costanza and R. Hirschfeld. Reflective layer activation in contextl. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1280–1285, New York, NY, USA, 2007. ACM.

[13] E. di Nitto, D. J. Dubois, and R. Mirandola. On exploiting decentralized bio-inspired self-organization algorithms to develop real systems. In *International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 68–75, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[14] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[15] S. González, K. Mens, and A. Cádiz. Context-oriented programming with the ambient object system. *j-jucs*, 14(20):3307–3332, 2008.

[16] S. González, K. Mens, and A. Cádiz. Context-Oriented Programming with the Ambient Object System. In *European Lisp Symposium*, pages 17–31, 2008.

[17] R. Hirschfeld, P. Costanza, and M. Haupt. An introduction to context-oriented programming with contextS. In R. Lämmel, J. Visser, and J. Saraiva, editors, *GTTSE*, volume 5235 of *Lecture Notes in Computer Science*, pages 396–407. Springer, 2007.

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

[19] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. pages 2–28. Springer-Verlag, 2003.

[20] 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, Washington, DC, USA, 1998. IEEE Computer Society.

[21] G. Schmidt. *ContextR and ContextWiki*. Master's thesis, Hasso-Plattner-Institut, Potsdam, 2008.

[22] C. Schubert. *ContextPy and PyDCL - Dynamic*

Contract Layers for Python. Master's thesis, Hasso-Plattner-Institut, Potsdam, 2008.

[23] D. Sykes, W. Heaven, J. Magee, and J. Kramer. From goals to components: a combined approach to self-management. In *SEAMS '08: Softw. eng. for adaptive and self-managing systems*, pages 1–8, New York, NY, USA, 2008. ACM.

[24] R. N. Taylor, N. Medvidovic, and P. Oreizy. Architectural styles for runtime software adaptation. In *WICSA/ECSA 09*, 2009.

[25] M. von Löwis, M. Denker, and O. Nierstrasz. Context-oriented programming: Beyond layers. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*, pages 143–156. ACM Digital Library, 2007.