Context Oriented Programming in Highly Concurrent Systems *

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

Context Oriented Programming (COP) allows modularization of programs according to the cross-cutting concern of contexts. Context depending features are grouped in layers which can be activated at run time by triggering the associated behavioral variations.

COP extensions have been provided for different languages. However all of them enforce a thread, shared-memory based concurrency model. In this paper we discuss how the COP paradigm can be applied to message-based concurrent systems which support the agents paradigm. The discussion is supported by the case of *ContextErlang*, our COP-inspired contextual version of Erlang.

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

1. INTRODUCTION

Over recent years, SMP architectures became very common. After a phase in which more computational power was obtained leveraging on an increase of the processor clock frequency, the increase of clock rate slowed down due to technological problems, and processor vendors have turned towards multi-core processors for gaining processing power.

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

COP'10, June 22, 2010, Maribor, Slovenia

Modern desktop hardware is usually provided with two or four cores, while it is expected that this number grows up to tens or hundreds in years [11]. As expected, applications are becoming increasingly concurrent in order to effectively take advantage of the new computational power. However the traditional support that mainstream programming languages provide, which is based on threads and locks, is appearing inadequate more and more. Concurrent programming with threads and locks is in general cumbersome and error-prone.

The Actor Model, proposed by Hewitt [10] and improved, among others, by Agha [5] takes a different approach to concurrency. Actors have a behavior and a mailbox. Upon receiving a message, the behavior of the actor is executed, while messages are buffered in the mailbox. Actors communicate (only) through asynchronous messages; i.e. after sending a message the actor continues with its execution. This means that there is no shared memory between actors, which greatly simplifies coding concurrent applications.

In the last few years, an increasing interest has centered around languages that adopt the actor model, such as Erlang. Moreover, many recently developed languages such as Scala [4], F# [3] and Go [2], enforce this paradigm.

COP is a recent programming technique which addresses the problem of adapting software behavior dynamically to the current execution context by providing suitable language abstractions. Starting from the pioneering work on the Lisp extension ContextL by Costanza [7], several COP extensions have been developed for different languages, such as ContextPy and PyContext for Python, ContextS for Squeak, ContextR for Ruby, and COP extensions for JavaScript, Groovy, Scheme and Java. Complete references for these languages can be found in [6].

Despite the trend mentioned above, COP extensions have been provided only for languages whose concurrency model is based on shared memory and locks. In this paper we discuss how the COP can be applied to languages that leverage on the actor model. In Section 2 the main aspects of COP layers are analyzed, in Section 3 we discuss the reasons why the features of layers explored so far do not fit the actor model, in Section 4 we expose a possible alternative, and in Section 5 we briefly present our COP variant of an agent-based language.

2. LAYERS IN COP

Layers are the abstraction used to modularize cross-cutting behavioral variations in context oriented languages. Lay-

ers are sets of *partial program definitions* implementing the functionalities of a behavioral variation: when a layer is *activated*, the partial definitions contained in it start having influence on the behavior of the program.

We point out three specific aspects of layers: declaration strategies, activation, and relationship with concurrency:

Layer declaration. Two layer declaration strategies have been explored so far in literature: *class-in-layer* and *layer-in-class*. In the class-in-layer pattern layers are defined outside the lexical scope of the code unit (usually classes) for which they provide behavioral variations. In the layer-in-class pattern the declaration of a layer is in the lexical scope of the module it augments.

Layer activation. In COP languages layer activation can be obtained through the use of ad hoc language primitives such as `(with-active-layers ({layer-name}*) body)` in ContextL, `with activelayer(layer-name)` in ContextPy, and `with(LayerList){BlockStatement*}` in ContextJ. Layer activation is dynamically scoped: activated layers affect both direct and indirect function invocations. The only exception to the dynamically scoped activation is given by global activations primitives. In ContextL the `ensure-active-layer` function enables global activation of layers without dynamic scope, while in the Ambience programming language [9] a context manager is in charge of updating the global context in real time.

Concurrency. Layer activation usually has an effect that is restricted to the thread that performs the activation. This choice is motivated by the need for avoiding race conditions and conflicts between different threads. However some languages have primitives that allow global layer activation for all threads without dynamic scope. This is the case of Ambience in which the context manager runs in separate thread, and in ContextLisp with the `ensure-active-layer` function mentioned above. With the exception of global activation primitives, layer activation is a synchronous operation: the thread calls a `with(Layers) Block` primitive and after the layer activation, the following code block is executed.

3. CRITICISM

In highly concurrent systems, a high number of processes in the form of agents interact with each other by exchanging messages. What makes these systems different with respect to the traditional shared memory model is that processes are not simply spawn when a parallel task must be carried on, but processes become a unit that *structures* programs. In fact, in languages that heavily leverage on agents as a programming paradigm, such as Erlang, the process becomes an abstraction that has a modularization/encapsulation role which is similar to the one of objects in object oriented languages. In a certain sense, at runtime, the constitutive elements of an application are processes rather than objects.

In this scenario it seems natural that variations are activated on a per-process basis so that the behavior of each single process can be modified. However, dynamically scoped variation activation do not seem to be the most suitable mechanism.

Take as an example a server; each time a client connects, a `user` process is spawned which serves all the successive client requests. This process is implemented as a server-side agent that interacts with the other `user` processes representing different clients or with other agents that encapsulate resources inside the server. It is natural that the context of the process is somehow related to the status of the client-side user who interacts with it. If the client is connected using a mobile device, the bandwidth availability can be limited and it can be useful to adopt a protocol which limits the amount of data transmitted, maybe at the price of reducing graphical effects or other not essential features.

In the COP paradigm this can be achieved by activating the proper variations on the `user` process so that it interacts with the client using a lighter protocol. Dynamic scope activation as it is supposed by current languages, does not fit well this case. First, we wish the activation of a layer to be active indefinitely, until a new context change triggers activation of a new layer. Second, we wish to be able to represent the common case in which the process that triggers a layer activation is not the same on which the variations are activated. In concurrent systems with many interacting agents this is a common condition if the adaptation is performed as a response to external conditions. For example an environment monitoring process can decide that some variations must be activated on certain other processes. Dynamic scope activation limits the variation effect to a set of operations which are in the fixed scope of the activation. Moreover the variation activation with dynamic scope is synchronous, while a request from another process can come in an asynchronous manner.

In the example above a temporary bandwidth reduction can be detected by a process which acts as a net monitor. After detection, the `user` process must be informed of the variation activations that must be performed. This type of activation is asynchronous with respect to the calls that the `user` agent receives from all the other processes. Consider the processes inside the server that manage the resources requested by the `user` processes. These processes can keep resources in memory, reducing the response time, or in case of heavy load of the server they can keep the resources on disk, increasing the time required to retrieve the data, but freeing as much memory as possible. It is reasonable that a process acts as a system monitor, keeping track of the memory consumption. When the free memory goes below a fixed bound, the system monitor process must inform the other processes that they have to activate a `save_memory` variation. These processes are not waiting for a variation activation, but they receive messages coming from other processes and the variation activation is asynchronous with respect to these requests.

4. VARIATIONS IN CONCURRENT SYSTEMS

Since in agent-based systems inter process communication is based on messages, the same mechanism can be applied for variation activation and deactivation. Per-process activation can be obtained by implementing context-aware agents that react to certain special *context-related* messages activating or deactivating their variations. The agents keep the active variations as an internal state, and the computation triggered by successive *standard* messages is affected by the presence of the variations. Variation activation has indefinite scope in the sense that from activation onwards the partial program definitions inside the variation affect the program behavior until a different variation activation occurs.

An interesting issue concerns which paradigm between the class-in-layer and the layer-in-class applies better to the agent model. Our experience with *ContextErlang* (see Sec-

tion 5) showed that a hybrid technique can give the advantages of both solutions.

A useful approach in structuring complex COP agent-based applications is to reason in term of *components*. Each component groups the modules that are executed by an agent, one with the basic behavior, and the other which are *variations*, i.e. sets of partial program definitions that can be activated on the agent. With this approach it is easier to raise the level of abstraction in structuring complex applications by considering each component as the basic constitutive unit. This makes it natural to think of a component as the equivalent of a class in an OO language and the variation modules as internal to the component, which is typical of the layer-in-class paradigm. Here the advantage is the ability of encapsulating in a component the basic behavior of an agent of a certain type and its variations.

It is useful that each component-specific variation is implemented as a single module which is proper of the class-in-layer approach. The main advantage of the class-in-layer model is adaptability in an evolving system. Being implemented as a single module, variations can in fact follow the loading rules of modules, such as being added and loaded at run time in the system, if the language supports this feature. For example, ContextErlang leverages this feature in order to achieve a great flexibility. In a distributed system such as the case of many interacting network nodes, a variation can be sent to a remote node and activated, so that the system can adapt to conditions that were not known when it was deployed.

5. CONTEXTERLANG

ContextErlang [8] is COP-inspired contextual extension to Erlang [1], a single assignment, dynamic typed, functional language with an actor model for concurrency and very lightweight processes.

Practically any real-world Erlang application is based on the OTP platform which is a library and a set of procedures for structuring fault-tolerant, large-scale, distributed applications. Since many processes enact similar patterns, such as serve requests, handle events, or monitor other processes, OTP generalizes these common patterns and gives a ready implementation of the generic structure (the *behavior*), while the user needs to implement only the specific part that exports a predefined set of functions (the *callback* module). 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 case of a `gen_server` (a generic process that serves requests), 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. While the callback module implements specific functionalities and it is directly influenced by a context change, the functionalities associated with the generic module 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; indeed, these different callback modules are used to implement *variations*. A variation contains the declarations of all the functions that implement a behavioral change; when the variation is activated (i.e. the variation is dynamically bound to a context-enabled behavior), these functions take effect overriding the functions defined in the basic callback module. As said before, a variation is activated on a specific process, which means that each context-enabled process has its set of active variations. Since more than one variation at a time can be active on an agent, active variations interact in changing the behavior of a component.

Variations are activated in a certain order, so that they conceptually create a stack. When the agent receives a call request message, the stack 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. If a `proceed()` call is performed from inside the called function in a variation, it is called the function in the subsequent eligible variation in the variations-stack. This is a simple mechanism for composing variations, inspired by other context-oriented languages.

An interesting feature of ContextErlang, based on the dynamic code loading capabilities of Erlang, is the variation transmission. With it, variations can be provided to the components of a remote Erlang node by sending and dynamic loading them so that those components can be enabled to react to unforeseen situations.

6. REFERENCES

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

[2] *http://golang.org/*. Website for the GO language.

[3] *http://research.microsoft.com/fsharp*.

[4] *http://www.scala-lang.org/*.

[5] G. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press: Cambridge, MA, USA, 1990.

[6] 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.

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

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

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

[10] 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 Publishers Inc.

[11] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41:33–38, 2008.