ContextErlang: A Language for Distributed Context-Aware Self-Adaptive Applications[1]

Guido Salvaneschi[a], Carlo Ghezzi[b], Matteo Pradella[b]

[a] *Software Technology Group, TU Darmstadt, Hochschulstr. 10, 64289 Darmstadt, Germany, salvaneschi@st.informatik.tu-darmstadt.de*
[b] *DEEPSE Group, DEIB, Politecnico di Milano Piazza L. Da Vinci, 32 20133 Milano, Italy, {carlo.ghezzi, matteo.pradella}@polimi.it*

Abstract

Self-adaptive software modifies its behavior at run time to satisfy changing requirements in a dynamic environment. Context-oriented programming (COP) has been recently proposed as a specialized programming paradigm for context-aware and adaptive systems. COP mostly focuses on run time adaptation of the application's behavior by supporting modular descriptions of behavioral variations. However, self-adaptive applications must satisfy additional requirements, such as distribution and concurrency, support for unforeseen changes and enforcement of correct behavior in the presence of dynamic change. Addressing these issues at the language level requires a holistic design that covers all aspects and takes into account the possibly cumbersome interaction of those features, for example concurrency and dynamic change.

We present ContextErlang, a COP programming language in which adaptive abstractions are seamlessly integrated with distribution and concurrency. We define ContextErlang's formal semantics, validated through an executable prototype, and we show how it supports formal proofs that the language design ensures satisfaction of certain safety requirements. We provide empirical evidence that ContextErlang is an effective solution through case studies and a performance assessment. We also show how the same design principles that lead to the development of ContextErlang can be followed to systematically design contextual extensions of other languages. A concrete example is presented concerning ContextScala.

Keywords: Context-oriented programming, Context, Context-awareness, Concurrency, Distribution, Dynamic change, Formal semantics.

1. Introduction

Self-adaptive software [2] is capable of adapting to different conditions to satisfy changing requirements. This feature is necessary when applications operate in environments that change dynamically. Sources of change include spatial positions of moving hosts, which may in addition join and leave the network dynamically, variable connectivity conditions, and changing load to meet fluctuating usage profiles. The need for addressing those conditions has become more common with the widespread use of mobile devices, the availability of distributed cooperative applications, and more generally the increased complexity of computing systems. In the attempt to tackle such problems, in the last few years, autonomic computing has been extensively investigated by researchers [3]. Autonomic computing starts from the assumption that the complexity of computing systems is growing up to a point that human support is not sufficient any more and systems must self-adapt to meet their requirements with limited or no human supervision.

The characterizing feature of self-adaptive software is that the behavior of the application must change dynamically in response to different external and internal conditions. This goal can be achieved in different ways. Traditional solutions address the problem at the software architecture and middleware levels [4, 5, 6]. More recently, researchers proposed context-oriented programming (COP) as a dedicated paradigm to support dynamic software adaptation [7]. COP provides abstractions that are specifically designed for run time change and adaptation, supporting modularization and disciplined dynamic composition of behavioral variations, which constitute a crosscutting concern. In

---

[1]This paper is an extended version of a preliminary conference paper [1].

addition, COP relieves programmers of the need to implement at the machinery required to support dynamic change. As a result, COP solutions are less verbose, and, thanks to commonly accepted abstractions, clearly express the design intentions, which would otherwise be hidden in ad-hoc solutions.

Current COP languages focus on modularization of behavioral variations and their dynamic activation [8]. However, in a real-world self-adaptive system, a whole spectrum of additional requirements must be considered. For example, real-world adaptive systems are often distributed. In this scenario, concurrent components run in parallel and may need to adapt to different context conditions. Different contexts can be active at the same time in different regions of the system. Since some components are dedicated to gathering contextual conditions via sensors, they must be able to trigger behavioral change on other components. Asynchronous communication simplifies the design of complex systems, so it is desirable that activation of behavioral variations is performed by an asynchronous mechanism. Another crucial aspect is that when an application runs in a highly dynamic environment, it must be ensured that behavioral changes activated on the application do not conflict, leading to inconsistent behavior. Finally, adaptations can be hard to foresee upfront and mechanisms for dynamic loading of new variations must be provided.

Supporting all these requirements is not easy. For example dynamic change and concurrency can easily lead to inconsistencies. In addition, the activation mechanism should seamlessly interact with distribution to support remote adaptation. Finally, the scope of context adaptation must be compatible with the organization of the distributed system. In summary, it is not possible to support these requirements by offering separate independent features; a comprehensive coherent design is instead required to tackle these issues.

CONTEXTERLANG[2] is a COP language that addresses the aforementioned requirements of self-adaptive applications. Specifically, we leverage the agent-based model of Erlang[3] to support context adaptations. CONTEXTERLANG is based on the concept of context-aware reactive agents. Context-aware agents have a basic behavior which can be altered by variations, i.e., behavioral units that can be dynamically activated on the agent. The composition of such variations determines the actual behavior of the agent. However, variations are not activated in isolation. Instead, an abstract data type specified by the programmer controls the composition and introduces constraints that avoid possible conflicts. Variation activation and the other context-related operations are performed by sending special messages to the agent. Therefore, in CONTEXTERLANG, asynchronous activation – as required by real-world adaptive systems – is the norm. Since the programmer can enable adaptations at the granularity of single agents, she has full control over fine-grained adaptation of each application component. Context-aware agents and the messaging mechanism are compatible with existing Erlang applications since we developed CONTEXTERLANG as part of the Open Telecom Platform (OTP), on which practically any real-world Erlang application is based. As a result, CONTEXTERLANG applications inherit the distribution and the fault-tolerance support of OTP. In addition, CONTEXTERLANG 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 solution is needed for systems that must adapt to unforeseen situations. Since concurrency, in the presence of dynamic change can easily lead to inconsistencies, we formalize CONTEXTERLANG with a minimal calculus that defines its semantics and the behavior of the language in all circumstances.

In summary, the paper makes the following contributions:

- Introduction of COP in the Actor concurrency model through the design of CONTEXTERLANG.

- A complete implementation of CONTEXTERLANG as part of Erlang OTP, an industrial-strength language for distributed, concurrent and fault-tolerant applications.

- A formalization of the core language with an operational semantics that validates its key design principles.

- Experimental validation and evaluation of our approach through performance comparison and prototypes, among which CONTEXTSCALA, a complete implementation of another language sharing the same principles and semantics of CONTEXTERLANG.

---

[2]The implementation of CONTEXTERLANG and the software presented in the rest of the paper is available at:
http://www.guidosalvaneschi.com/wp/software/contexterlang/
[3]http://www.erlang.org/

A preliminary version of this work focused on the informal integration of COP with the Actor model [1]. This paper is a comprehensive overview of how CONTEXTERLANG meets the requirements of adaptive systems and includes a formalization of the most important features of the language.

The paper is organized as follows. In Section 2 we outline the requirements of adaptive applications and discuss the possible solutions. In Section 3 we present the design of CONTEXTERLANG, and in Section 4 its formal semantics. Section 5 presents the validation of our work. Section 6 discusses the related work and Section 7 draws some conclusions.

2. Adaptive Software and COP

In this section, we discuss the requirements of self-adaptive systems on programming language support, we introduce the main features of COP, and discuss how current COP languages only partially address the requirements and which problems are still open. Finally, we sketch the CONTEXTERLANG solution that is detailed in the next section.

Compared to traditional applications, self-adaptive systems have specific requirements that must be considered by software designers. Supporting these requirements is an important goal of the technology used to implement them. The main requirements are:

- *Dynamic adaptation.* Software applications operate in an environment that changes frequently. For this reason, adaptive software must provide mechanisms to dynamically modify its behavior depending on the changing conditions.

- *Modularization of behavioral variation.* Adaptive software can perform dynamic adaptation at different levels of abstraction [2]. For the purpose of this paper, a fundamental distinction is between parameter adaptation versus compositional adaptation – according to the terminology introduced by McKinley et al. [9]. Adaptation at the parameter level means that the same module is run with different input parameters. On the contrary, more complex adaptations that concern behaviors, like alternative algorithms or modifications of the same algorithm, are more challenging because they often crosscut the main organizing direction of the application and require proper modularization [10].

- *Asynchronous variation activation.* Adaptive systems are often modeled by the MAPE-K loop model (*monitoring*, *analyzing*, *planning*, *execution* and global *knowledge*) proposed by the autonomic computing community [3]. An *autonomic manager* controls a *managed element* to achieve the adaptive behavior. The autonomic manager collects information through *sensors* and modifies the behavior of the system through *effectors*. In a complex system, the autonomic manager and the managed element are not only conceptually separated, but they are also implemented as different components. When the size of a systems grows, there can be several autonomic managers that observe different sensors and are responsible for activating different adaptations. In this scenario, managed elements run independently of the autonomic managers and must be asynchronously notified of behavioral changes.

- *Constraints on variation composition.* As already discussed, in adaptive systems behavioral variations are activated at run time, possibly by different managers. In highly dynamic scenarios, several adaptations can be activated on the same component, so the risk of inconsistencies among variations is concrete. For example, in a mobile application that adapts to the presence of a Wi-Fi connection, the *online* and the *offline* variations should not be active at the same time.

- *Support for unforeseen adaptation.* Designers of adaptive software must equip the system with the functionalities required to properly respond to the most diverse conditions. However, designers can hardly foresee all the adaptations that can be needed when the system will become operational. The need for a new adaptation can be quite frequent due to unpredictable changes in the environment and stopping the system to introduce the required functionality could be unacceptable. For this reason, several adaptive systems adopt solutions that allow one to update the running code without incurring in the downtime of stopping the system for recompilation. For example, Aspect Oriented Programming (AOP) frameworks like PROSE [11, 12] and JAC [13, 14], support remote uploading and dynamic activation of aspect components.

- *Distribution.* In computing systems, the need for adaptation is often mainly due to changing connectivity conditions and environmental factors. Examples include changes in the physical location of mobile hosts, variability of network bandwidth, and changes of requests and load from the clients. Given these factors, it is not surprising that, in most scenarios, adaptive systems are distributed. As a result, run time adaptation must be designed also taking into account remote communication and host decoupling.

2.1. Context-oriented Programming

COP is an interesting starting point to meet the requirements outlined in the previous section since it supports dynamic adaptation and modularization of behavioral variations. In this section, we provide a short introduction to COP. The reader interested to a more general analysis and an overview of the existing COP languages can refer to the survey [8].

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 [7]. 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*. Thanks to this approach, context can be effectively used to model the variability required by adaptive systems.

COP provides *ad-hoc* language-level abstractions to modularize context-dependent behavioral variations and dynamically activate and combine them. In COP languages, behavioral variations are reified in *layers*[4], abstractions which group *partial method definitions*. For example, the following class defines two partial method definitions for the method m inside the layers l1 and l2.

```
class MyClass {
  layer l2 {
    void m() { System.out.println("m: l2"); proceed(); ...; }
  }
  layer l1 {
    void m() { System.out.println("m: l1"); proceed(); ...; }
  }
  void m(){ ...; System.out.println("m"); }
}
```

Layers are activated through an explicit statement such as

```
with(layersList) {
  codeBlock
}
```

and activation is scoped to the dynamic extent of the code block. When a method is called from `codeBlock` and partial definitions are available for that method in the active layers, the partial definitions are executed. For example, the call `m()` in

```
MyClass o = new MyClass()
with(l2,l1) {
  o.m()
}
// output:
m: l2
m: l1
m
```

executes the partial definition of m inside the layer l2. In COP, the `proceed` keyword allows dynamic combination. It is similar to `proceed` in Aspect-Oriented Programming (AOP) and calls the partial definition in the next active layer or the basic definition. As a result, the partial definition of m inside l2 proceeds to the partial definition of m inside l1, which finally proceeds to the execution of the basic behavior, i.e., method m. Thanks to the mechanism introduced by COP for dynamic context activation and composition, run time adaptation is implemented without cluttering the code with *if* statements to express context dependency.

---

[4]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 6.
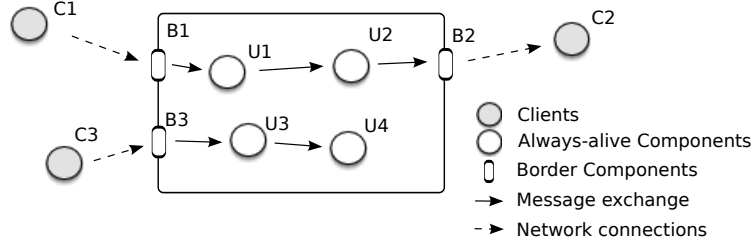
Figure 1: The ContextChat application.

In the following sections we will illustrate these concepts more thoroughly with a running example written in CONTEXTERLANG.

2.2. The ContextChat Case Study

To illustrate the design choices behind our work, we introduce a running 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 show 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.

2.3. The Context-Oriented Programming Solution

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.

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 [15]. Layers are used to model partial behavioral variations. For example, in Figure 2 the tracing layer contains a partial definition of the receive_msg and of the send_msg methods. Method calls are dispatched according to the active layers and proceed keyword allows dynamic combination of variations active at the same time.

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 [15, 16, 17]. 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 next are likely to be encountered in any sufficiently large self-adaptive application that needs to be organized in several functional modules, and are not specific to this example.

First, a context change is often signaled by 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

5

```
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.

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 [18] 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 of 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, in some applications, it is not possible to identify unique entry points for the control flow. As already noticed by COP researchers [19], in these 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. 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 could extend *too far*. For example, when 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 supporting user-defined activation strategies, such as indefinite scope or per-object activation. Per-object activation is performed by 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 de-

```
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.

sign 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 [21] is a Java COP extension supporting 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 a `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, as shown by the example in (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 designed according to the MAPE-K model, because the adaptive application is decoupled into a managed element, which implements the application logic, and the autonomic manager, which collects data from sensors and plans the adaptive behavior. 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. EventCJ instead adopts the standard Java share-and-lock concurrency model. By exploiting the integration of COP with the Actor Model, ContextErlang directly addresses the issue of context propagation in concurrent systems, and allows for asynchronous context provisioning directly in the language, without pointcut-like expressions. This approach solves in a natural way the problem of context confinement by adopting actors as context boundaries.

2.4. Overview of the ContextErlang Solution

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 reflects 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.

7

```
-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`.

To make the benefits of such design more concrete, hereafter we illustrate how ContextChat is designed in CONTEXTERLANG. Always-alive components are context-aware agents (namely `users`) 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 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.

CONTEXTERLANG provides a powerful feature: *variation transmission*. To illustrate it, 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 simplicity, 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. The Design of CONTEXTERLANG

3.1. Erlang and the Open Telecom Platform

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 software. To keep the paper reasonably self-contained, we provide a minimal description of the Erlang syntax and a short introduction to the OTP.

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 this paper we use the term *behavior* also to indicate the way an agent behaves with respect to the software system. To avoid confusion, when necessary, 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. The functions' implementation follows. A function body is started by "->". Braces indicate tuples of fixed length, brackets 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 (note 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.

The OTP platform addresses several requirements of a distributed system. For example, agents can be organized in hierarchies where supervisors (i.e., modules implementing the `supervisor` OTP behavior) can control other working agents, be notified of failures, and adopt countermeasures. Another fundamental feature is that agents can be spawned on remote hosts simply by specifying the address of the remote Erlang system. Messages can be sent to local or remote agents transparently, supporting seamless migration of applications to a distributed setting. Finally, agents are an ideal way to structure distributed software where different subsystems must be decoupled, run in parallel, and communicate asynchronously.

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 containing the functions for the core functionalities. We refer to these functions as *handle* functions[5].

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.

Modularization of variations. A variation is an Erlang module defining a set of handle functions exposed to the contextual dispatching. Implementing variations as Erlang modules has several advantages. It simplifies their development since it does not require syntax extensions, increasing the chances of acceptance by the programmers and avoiding the risk of breaking compatibility with existing tools. In addition, it improves extensibility, since new variations can be added by implementing new modules without modifying the existing code.

---

[5]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(user).
-behavior(context_agent).
-include("context_agent_api.hrl"). % contextual API
% API
receive(AgentId, Source, Msg) -> context_agent:cast(AgentId, {receive_msg, Source, Msg}).
send(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.

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 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 activated or deactivated. Initialization and cleanup code are placed inside these functions.

Variations activation model. Self-adaptive and context aware systems are inherently concurrent because, conceptually, they receive signals from the context in parallel with the execution of the application logic. For this reason, context-aware features in adaptive applications are tightly coupled with concurrency features.

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 additional patterns will be identified with the practical development of agent-based context-aware applications. 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]}),
```

```
-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.

```
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. Besides 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 available when imported by the callback.

Unforeseen adaptation with variation transmission. Variation transmission is a powerful mechanism to implement software which reacts to unforeseen conditions. For example, our previous work [22] 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.

11

3.3. Coherence Among Variations: the Context Abstract Data Type

COP behavioral variations are activated and combined while the application is running. Consistency among variations must therefore be ensured. 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 [23] has been leveraged to dynamically check the constraints. Other solutions use domain specific languages (DSL) to express *declarative* constraints on layers [24], in a way similar to selecting features in software product lines. A constraint violation 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 [25] 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 [21].

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 time instant. *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}},     % Activatable slot, backup is initally active
        {tracing, {trace, active}},          % Activatable slot, trace is initally active
        {status, [{offline, active}, online]}, % Switch slot, offline is initally active
        {text_effect, free_slot},            % Free slot, initally empty
        {base_behavior, {user, active}}],

Context = context_ADT:create(Spec),
user:start_link(AgentId, Context)
```

The formal syntax of the context ADT is reported 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 provided API. The following call performs a switch on the `status` slot, activating the `online` variation.

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

The effect of the call is shown in Figure 9. Similar functions are used to activate and deactivate the variation in an activatable slot, and for filling a free slot with a variation sent from a remote agent. After being filled, the variation in the free slot can be activated normally.

The introduction of a context ADT raises a number of critical issues. A possible drawback is that ADT specifications require an extra design effort. However, the impact on complexity is minimized by using a DSL. In addition, introducing a variation into an existing context ADT instance requires changing the specification, forcing the programmer to think about how to combine variations in a coherent way. In any case, the effort required is similar to writing a new set of layer transitions in EventCJ when a new layer is added. Another issue concerns our choice to limit the stack size and force variations to obey to certain constrains, which possibly limit variation capabilities. This design choice favors safety over 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 schema are quite simple and encompass
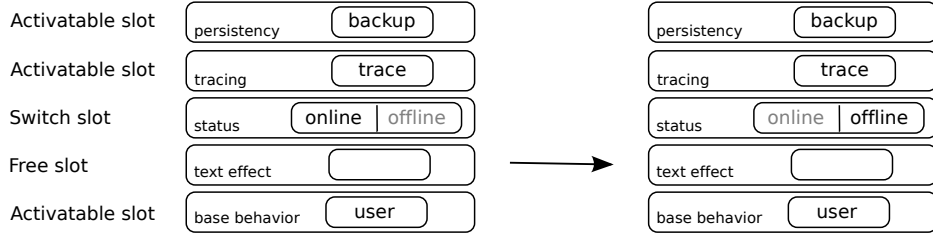
Figure 9: The context ADT in CONTEXTERLANG. Switching from the *online* to the *offline* variation.

only few variations, often in mutual exclusion [21, 7, 15, 19]. Nonetheless, in the spirit of leaving the exploration of more dynamic solutions open, we decided to keep both activation mechanisms.

The context ADT solution is different from other COP proposals because it enforces the ADT user to adopt correct configurations. Other approaches, instead, allow one to freely operate with variations. Note, however, that the content of the variations is not checked statically – a choice compliant with Erlang's dynamic typing. For this reason, run time errors due to wrong function calls can still occur, even using context ADTs. Interestingly, none of the approaches proposed so far, including the context ADT, can automatically verify the correctness of variations configurations and run-time transitions against a given specification. Investigation in this direction is an open research problem.

3.4. Concurrency: Consistency with Context Change

Combining COP with concurrency is not an easy task. Integrating Erlang's actors with run-time behavioral change requires careful scrutiny of how these aspects interact. In this section we clarify some fundamental points.

A crucial requirement is that behavioral changes should be *safe*, i.e., a change of the active variations should not corrupt the task in execution. As we will explain shortly, this cannot be achieved by simply forbidding a context change while a message-triggered computation is active. 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* while *cmp* is executed 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 is reminiscent of OO programming, where an object is ultimately responsible for how it responds to messages. This 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` variation 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, with the functions for variation management seen so far, it is not possible to execute a variation manipulation atomically with a set of operations. Of course, agents can coordinate to enforce this constraint at a higher level, implementing some synchronization mechanism. However this solution requires development of possibly 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` message, 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}.
```

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

4. Formal Semantics

We introduce ContextErlangLite, a kernel untyped language for distributed, concurrent computation. It supports context adaptation through context-aware agents, variation activation and remote variation transmission. This core language allows one to reason about the core features of ContextErlang applications, ignoring irrelevant implementation details. This flexibility can also be used to apply the founding principles of ContextErlang to other languages based on the Actor model: we will discuss this aspect in Section 5.2 by presenting a new COP language implemented on top of Scala.[6]

The formal semantics for ContextErlangLite is provided in terms of transitions between system configurations. System configurations model the possible evolutions of a set of nodes executing a given program. A ContextErlang application is made, at run time, by a set of processes holding a state and waiting for incoming messages.

Related relevant work in the field of formal semantics for the COP paradigm is discussed in Section 6. Because of the concurrent nature of ContextErlang, the approach we present hereafter was mainly influenced by non-COP core languages, such as the formal semantics of CoBoxes [26] and the work of DeBoer *et al.* on futures [27].

Conventions used in the formalization. Ordered sequences are indicated with overlined letters when they are comma-separated, as in functions parameters: $\bar{a} = a_1, \ldots a_n$. Space-separated ordered sequences are indicated with the Kleene star operator "*". The empty sequence is indicated as $\varnothing$ while ϵ indicates unassigned (*null*) values. The · operator is used to indicate the syntactic concatenation of sequences. In the semantics of the language, we use · to stress the fact that sequences are ordered, as in the case of message queues. Otherwise, $\cup$ is used to merge sequences when the order does not matter. We use the operator $\in$ to indicate the presence of an element in a sequence: $a \in Seq = a_1 a_2 \ldots a_k$ iff $\exists i : 1 \le i \le k, a = a_i$.

We define our formal framework in terms of changes between node configurations, agent configurations and system configurations. When the distinction is clear from the context, we use the terms agent/node/system and agent/node/system configurations interchangeably.

4.1. Syntax

Figure 10 shows the abstract syntax of the ContextErlangLite. Starting from the given syntax, other constructs can be added as usual. A program *P* is a sequence of node declarations, starting with the declaration of the main node. Each node declaration explicitly states the name *n* of the node and a sequence of module declarations *MlDec* the node is aware of. The main node has a special role at system startup (Section 4.4). For this reason its body, besides module declarations, contains an expression *mainexp* that triggers the entire computation.

A module declaration can be a callback *cml* declaration or a variation *vml* declaration. Both, after the `callback` or `variation` keyword, consist of a name and a set of method declarations *B*. Expressions include the `let` operator, values *v*, variables *x*, synchronous and asynchronous method invocations $e.m(\bar{e})$ and $e!m(\bar{e})$. Expressions include also the creation of a new agent `newProc`(cml, e) with a callback module *cml* on a node which results from evaluating a given expression *e*. The expression $e_1; e_2$ is syntactic sugar for `let` $x = e_1$ `in` e_2 with *x* in $FV(e_2)$, where $FV(e)$ is the set of free variables in the expression *e*. Other expressions are context-related: the `proceed` primitive for calling

$Prog ::= MainNode\ Node^*$ *program*

$Prog ::= MainNode\ Node^*$	*program*
$MainNode ::= \texttt{node}\ n = \{\ MlDec^*\ mainexp\ \}$	*main node declaration*
$Node ::= \texttt{node}\ n = \{\ MlDec^*\ \}$	*node declaration*
$mainexp ::= e$	*main expression*
$MlDec ::= \texttt{variation}\ vml\{\ B^*\ \}$	*variation declaration*
$\quad \mid \texttt{callback}\ cml\{\ B^*\ \}$	*callback declaration*
$B ::= m(\overline{x})\{e\}$	*method declaration*
$e ::= \texttt{let}\ x = e\ \texttt{in}\ e$	*let operator*
$\quad \mid v$	*value*
$\quad \mid x$	*variable*
$\quad \mid e_1; e_2$	*expression sequence*
$\quad \mid \texttt{selfNode}$	*reference to local node*
$\quad \mid e.m(\overline{e})$	*synchronous method call*
$\quad \mid e!m(\overline{e})$	*asynchronous call*
$\quad \mid \texttt{newProc}(cml, e)$	*remote agent creation*
$\quad \mid \texttt{proceed}(\overline{e})\ \underline{:\ m(\overline{x})\ :\ vml}$	*proceed call*
$\quad \mid \texttt{selfChange}(\overline{e})$	*variation change*
$\quad \mid \texttt{varSend}(e_1, e_2)$	*variation transmission*
$v ::= n \mid vml \mid cml$	*values*

$x \in$ variable names, $cml \in$ callback module names,
$vml \in$ variation module names, $m \in$ method names, $n \in$ node identifiers

Figure 10: The abstract syntax of CONTEXTERLANGLITE.

$Conf ::= N \mid P \mid Conf\ Conf \mid \epsilon$	*system configuration*	$ml ::= cml \mid vml$	*module name*
$N ::= \langle L, n \rangle_{\mathsf{N}}$	*node configuration*	$e ::= ...$	*expressions*
$P ::=$	*agent configuration*	$v ::= ... \mid p \mid ok$	*values*
$\langle self, base, A, M, sender, active, susp \rangle_{\mathsf{P}}$		$L ::= ml^*$	*loaded module names*
$msg ::= \langle p, m(\overline{v}) \rangle_{\mathsf{M}} \mid \langle m(\overline{v}) \rangle_{\mathsf{M}}$	*message*	$M ::= msg^*$	*inbox message queue*
$p ::= \langle w, n \rangle_{\mathsf{R}}$	*agent reference*	$A ::= vml^*$	*active variation names*
$self ::= p$	*agent self reference*	cml	*callback module name*
$sender ::= p \mid \epsilon$	*agent waiting for answer*	vml	*variation module name*
$active ::= e \mid \epsilon$	*agent's active task*	w	*node agent reference*
$susp ::= e \mid \epsilon$	*agent's suspended task*	n	*node reference*
$base ::= cml$	*agent's callback module name*	m	*method name*

Figure 11: The semantic entities of CONTEXTERLANGLITE.

the next eligible function in a variation stack, `selfChange(`$\overline{e}$`)` for the activation of a variation sequence on an agent, `varSend(`e_1, e_2`)` for sending a variation to a remote node.

Values that can be explicitly used in a program are node names *n*, variation names *vml*, and callback names *cml*. Following the convention in [28], underline phrases are inserted by elaboration and are not part of the surface syntax. For example, in the expression `proceed(`$\overline{e}$`)` $\underline{: m(\overline{x}) : vml}$ the annotations of the method $m(\overline{x})$ and of the variation module *vml*, in which `proceed` is called, are added statically by a code processor and are not part of the core language. This convention is adopted to simplify the semantic rules. We assume that a correct CONTEXTERLANGLITE program respects the following constraints:

- Module names for callbacks and variations *cml* and *vml* in module declarations are unique.

- Method signatures (method name and parameters arity) are unique within a module declaration.

- Node names in node declarations are unique.

- Node names used in the program are valid node names, i.e., they are names of declared nodes.

4.2. Semantic entities

This section introduces the semantic entities used to describe the behavior of the system. A configuration *Conf* is a sequence of node and agent configurations. Therefore, the top-level configuration is a snapshot of the whole system. A node configuration is a record $\langle L, n \rangle_N$, where *L* is a sequence of loaded modules and *n* is a node identifier.

An agent configuration is a tuple $\langle self, base, A, M, sender, active, susp \rangle_P$; *self* is the (unique) reference to the agent and is represented as a tuple $\langle w, n \rangle_R$, which identifies the agent with a unique identifier *w* and a reference to the node itself *n*; *base* is the name of the callback module, which constitutes the basic behavior of the agent; *A* is the sequence of names of the active variations that modify the basic behavior of the agent; *M* is the inbox of the agent, i.e., a sequence of incoming messages; *sender* is a reference to the agent that requested a method execution to this agent and is waiting for a reply; *active* is the currently active task (i.e., the code in execution); *susp* is a suspended task. Both the active and the suspended tasks are evaluation contexts, with the hole filled with the next expression to be evaluated.

There are two types of messages: $\langle p, m(\overline{v}) \rangle_M$ and $\langle m(\overline{v}) \rangle_M$, where $m(\overline{v})$ is the method call with the actual parameters and *p* is a reference to the sending agent. The former, which includes the sender's reference, stands for a synchronous call; the latter instead stands for an asynchronous call. The definition *ml* ::= *cml* | *vml* allows callback modules and variation modules to be treated in a uniform way. Expressions are the same defined in the syntax of the language. Values can be agent references *p* and the *ok* return value in addition to the values already defined in the syntax.

Evaluation contexts. We use evaluation contexts for a compact representation. In context reduction semantics, a term is decomposed into a reduction context and a redex, and semantic rules reduce the redex [29]. This allows for avoiding to provide a partitioning algorithm, relying on as few implementation details as possible. Evaluation contexts are terms with an empty part $\square$ in a certain position. We indicate with $e_\square[e']$ the replacement of the hole $\square$ in *e* with the expression e'.

$$
\begin{aligned}
e_\square ::=\ &\square \mid \texttt{let } x = e_\square \texttt{ in } e \mid e_\square ! m(\overline{e}) \mid v ! m(\overline{v}, e_\square, \overline{e}) \mid \\
&e_\square . m(\overline{e}) \mid v . m(\overline{v}, e_\square, \overline{e}) \mid \\
&\texttt{newProc}(e_\square, e) \mid \texttt{newProc}(v, e_\square) \mid \\
&\texttt{proceed}(\overline{v}, e_\square, \overline{e}) \underline{: vml : m(\overline{x})} \mid \\
&\texttt{selfChange}(\overline{v}, e_\square, \overline{e}) \mid \\
&\texttt{varSend}(e_\square, e) \mid \texttt{varSend}(v, e_\square)
\end{aligned}
$$

In our syntax, redexes can only be expressions, so holes appear only at positions where expressions are expected.

Auxiliary predicates and functions. To simplify the semantic rules, we introduce some auxiliary predicates and functions. The rules shown in Figure 12 define predicates and functions operating on the static code of the program. The rules [*callback declared in node*], [*variation declared in node*] and [*module declared in node*] define the predicate $\in_D$ which holds if a module is declared inside a node. The [*method body*] rules define the $mbody(cml, m(\overline{v}))$ function which returns the method body for a call $m(\overline{x})$, searching it in a variation module with name *cml*. A similar rule is

$$\frac{\texttt{node } n = \{ \ ... \ \texttt{callback } cml \ \{ \ ... \ \} \ ... \ \}}{cml \in_D n} \qquad\qquad [\textit{callback declared in node}]$$

$$\frac{\texttt{node } n = \{ \ ... \ \texttt{variation } vml \ \{ \ ... \ \} \ ... \ \}}{vml \in_D n} \qquad\qquad [\textit{variation declared in node}]$$

$$\frac{vml \in_D n \vee cml \in_D n}{ml \in_D n} \qquad\qquad [\textit{module declared in node}]$$

$$\frac{\begin{array}{c}|\overline{v}| = |\overline{x}| \\ \texttt{variation } vml \ \{... \ (\overline{x})e \ ...\}\end{array}}{mbody(vml, m(\overline{v})) = (\overline{x})e} \qquad \frac{\begin{array}{c}|\overline{v}| = |\overline{x}| \\ \texttt{callback } cml \ \{... \ (\overline{x})e \ ...\}\end{array}}{mbody(cml, m(\overline{v})) = (\overline{x})e} \qquad [\textit{method body}]$$

$$\frac{\begin{array}{c}|\overline{v}| = |\overline{x}| \\ \texttt{variation } vml \ \{... \ (\overline{x})e \ ...\}\end{array}}{m(\overline{v}) \in_D vml} \qquad \frac{\begin{array}{c}|\overline{v}| = |\overline{x}| \\ \texttt{callback } cml \ \{... \ (\overline{x})e \ ...\}\end{array}}{m(\overline{v}) \in_D cml} \qquad [\textit{method ownership}]$$

$$\frac{ml_i : \nexists j \ (m(\overline{v}) \in_D ml_j, \ j < i, \ 1 \le i \le n, \ 1 \le j \le n)}{getml(\ ml_1 \ ... \ ml_n, \ m(\overline{v})) = ml_i} \qquad [\textit{module selection}]$$

Figure 12: Auxiliary predicates and functions that extract information from the program code.

given for callback modules. The [*method ownership*] rules define a predicate stating that a method eligible for managing a given call is defined inside a certain module. The rule [*module selection*] defines the function *getml* that given a sequence of module names $ml_1 \ ... \ ml_k$ and a method call $m(\overline{v})$, returns the name of the first module in the sequence implementing a method compatible with the signature of the call. This function is used for method dispatching over the sequence of the active variations of an agent.

4.3. Rules

The dynamic semantics of ContextErlangLite is defined in terms of a small-step reduction relation on configurations of agents and nodes. Rules are applied to (sub)configurations. The arrow notation in the rules means that a (sub)configuration reduces in one step to another (sub)configuration. Matches of rules on configurations are modulo associativity and commutativity of node configurations and agent configurations. Hereafter, the semantic rules of ContextErlangLite are analyzed in detail. Groups of rules that conceptually participate in the same semantic operation and whose activation is usually related are described together. For terms which can assume the value ϵ we use the notation x_ϵ as a shorthand for $x \cup \epsilon$.

Let. The [*let*] rule expresses the substitution of variables within expressions in other expressions. We use $e_2[e_1/x]$ to denote the standard capture-avoiding substitution of the expression e_1 for the free variable x in the expression e_2

Agent creation. The newProc(cml, n) command creates a new agent with callback module name *cml* on the node *n*. newProc(cml, n) is reduced to a reference p to the new agent (rule [*new proc*]). The created agent belongs to the node *n*. If not already loaded, the module name *cml* is added to the names of loaded modules of the node *n* (lazy loading). The new agent is initialized with an empty set of active variation names and an empty message queue. The [*self node*] rule allows an agent to obtain a reference to the node where it is running and can be used by an agent to spawn another agent on the same node.

Method call. In the case of a synchronous method call, an agent sends a message to another agent, which adds it to its inbox queue [*sync mth call*]. Synchronous call messages contain the reference to the sender (for the delivery of the response), and the call itself i.e., the name of the method with the actual parameters. The current task of the sender

$$\langle p, cml, A, M, p_{\epsilon 1}, e_\square[\texttt{let}\ \overline{x} = \overline{v}\ \texttt{in}\ e], \epsilon\rangle_\mathsf{P}$$
$$\rightarrow \langle p, cml, A, M, p_{\epsilon 1}, e_\square[e[\overline{v}/\overline{x}]], \epsilon\rangle_\mathsf{P}$$
$$[let]$$

$$\frac{L' = L \cup \{cml_2\},\ \ cml_2 \in_D n_2 \quad\quad w\ fresh\ in\ n_2 \quad\quad p_2 = \langle w, n_2 \rangle_\mathsf{R}}{\langle p_1, cml_1, A, M, p_{\epsilon 3}, e_\square[\texttt{newProc}(cml_2, n_2)], \epsilon\rangle_\mathsf{P}\ \langle L, n_2\rangle_\mathsf{N}}$$
$$\rightarrow \langle p_1, cml_1, A, M, p_{\epsilon 3}, e_\square[p_2], \epsilon\rangle_\mathsf{P}\ \langle p_2, cml_2, \varnothing, \varnothing, \epsilon, \epsilon, \epsilon\rangle_\mathsf{P}\ \langle L', n_2\rangle_\mathsf{N}$$
$$[new\ proc]$$

$$\frac{p = \langle w, n\rangle_\mathsf{R}}{\langle p, cml, A, M, p_{\epsilon 1}, e_\square[\texttt{selfNode}], \epsilon\rangle_\mathsf{P}}$$
$$\rightarrow \langle p, cml, A, M, p_{\epsilon 1}, e_\square[n], \epsilon\rangle_\mathsf{P}$$
$$[self\ node]$$

$$\frac{msg = \langle p_1, m(\overline{v})\rangle_\mathsf{M}}{\langle p_1, cml_1, A_1, M_1, p_{\epsilon 3}, e_\square[p_2 . m(\overline{v})], \epsilon\rangle_\mathsf{P}\ \langle p_2, cml_2, A_2, M_2, p_{\epsilon 4}, e_{\epsilon 2}, e_{\epsilon susp}\rangle_\mathsf{P}}$$
$$\rightarrow \langle p_1, cml_1, A_1, M_1, p_{\epsilon 3}, \epsilon, e_\square[p_2 . m(\overline{v})]\rangle_\mathsf{P}\ \langle p_2, cml_2, A_2, msg \cdot M_2, p_{\epsilon 4}, e_{\epsilon 2}, e_{\epsilon susp}\rangle_\mathsf{P}$$
$$[sync\ mth\ call]$$

$$\frac{msg = \langle m(\overline{v})\rangle_\mathsf{M}}{\langle p_1, cml_1, A_1, M_1, p_{\epsilon 3}, e_\square[p_2 ! m(\overline{v})], \epsilon\rangle_\mathsf{P}\ \langle p_2, cml_2, A_2, M_2, p_{\epsilon 4}, e_{\epsilon 2}, e_{\epsilon susp}\rangle_\mathsf{P}}$$
$$\rightarrow \langle p_1, cml_1, A_1, M_1, p_{\epsilon 3}, e_\square[ok], \epsilon\rangle_\mathsf{P}\ \langle p_2, cml_2, A_2, msg \cdot M_2, p_{\epsilon 4}, e_{\epsilon 2}, e_{\epsilon susp}\rangle_\mathsf{P}$$
$$[async\ mth\ call]$$

$$\frac{getml(A \cdot cml, m(\overline{v})) = ml \quad\quad mbody(ml, m(\overline{v})) = (\overline{x})e}{\langle p_1, cml, A, M \cdot \langle p_2, m(\overline{v})\rangle_\mathsf{M}, \epsilon, \epsilon, \epsilon\rangle_\mathsf{P}}$$
$$\rightarrow \langle p_1, cml, A, M, p_2, e[\overline{v}/\overline{x}], \epsilon\rangle_\mathsf{P}$$
$$[sync\ exec]$$

$$\frac{getml(A \cdot cml, m(\overline{v})) = ml \quad\quad mbody(ml, m(\overline{v})) = (\overline{x})e}{\langle p_1, cml, A, M \cdot \langle m(\overline{v})\rangle_\mathsf{M}, \epsilon, \epsilon, \epsilon\rangle_\mathsf{P}}$$
$$\rightarrow \langle p_1, cml, A, M, \epsilon, e[\overline{v}/\overline{x}], \epsilon\rangle_\mathsf{P}$$
$$[async\ exec]$$

$$\langle p_1, cml_1, A_1, M_1, p_2, v_1, \epsilon\rangle_\mathsf{P}\ \ \langle p_2, cml_2, A_2, M_2, p, \epsilon, e_\square[p_1 . m(\overline{v_2})]\rangle_\mathsf{P}$$
$$\rightarrow \langle p_1, cml_1, A_1, M_1, \epsilon, \epsilon, \epsilon\rangle_\mathsf{P}\ \ \langle p_2, cml_2, A_2, M_2, p, e_\square[v_1], \epsilon\rangle_\mathsf{P}$$
$$[sync\ return]$$

$$\frac{Conf_1 \rightarrow Conf'_1}{Conf_1\ Conf_2 \rightarrow Conf'_1\ Conf_2}$$
$$[subconf]$$

Figure 13: The formal semantics of CONTEXTERLANGLITE.

$$A = vml_1 \ldots vml_i \ldots vml_k \qquad 1 \le i \le k \qquad |\overline{v}| = |\overline{x}|$$

$$\dfrac{getml(vml_{i+1} \ldots vml_k \, cml, m(\overline{v})) = ml \qquad mbody(ml, m(\overline{v})) = (\overline{x})e}{\langle p, cml, A, M, p_{\epsilon 1}, e_\square[\texttt{proceed}(\overline{v}) : m(\overline{x}) : vml_i], \epsilon \rangle_\mathsf{P}}$$

$$\to \langle p, cml, A, M, p_{\epsilon 1}, e_\square[e[\overline{v}/\overline{x}]], \epsilon \rangle_\mathsf{P}$$

[*proceed*]

$$\dfrac{A' = \overline{vml} \qquad p = \langle w, n \rangle_\mathsf{R} \qquad L' = L \cup \{ml \in \overline{vml} \mid ml \in_D n, \ ml \notin L\}}{\langle p, cml, A, M, p_{\epsilon 1}, e_\square[\texttt{selfChange}(\overline{vml})], \epsilon \rangle_\mathsf{P} \ \langle L, n \rangle_\mathsf{N}}$$

$$\to \langle p, cml, A', M, p_{\epsilon 1}, e_\square[ok], \epsilon \rangle_\mathsf{P} \ \langle L', n \rangle_\mathsf{N}$$

[*self var change*]

$$\dfrac{n_1 \ne n_2 \qquad p = \langle w, n_1 \rangle_\mathsf{R} \qquad vml \in_D n_1 \wedge vml \in L_1 \qquad L'_2 = L_2 \cup vml}{\langle p, cml, A, M, p_{\epsilon 3}, e_\square[\texttt{varSend}(vml, n_2)], \epsilon \rangle_\mathsf{P} \ \langle L_1, n_1 \rangle_\mathsf{N} \ \langle L_2, n_2 \rangle_\mathsf{N}}$$

$$\to \langle p, cml, A, M, p_{\epsilon 3}, e_\square[ok], \epsilon \rangle_\mathsf{P} \ \langle L_1, n_1 \rangle_\mathsf{N} \ \langle L'_2, n_2 \rangle_\mathsf{N}$$

[*variation sending*]

Figure 14: The formal semantics of ContextErlangLite: rules associated with context-awareness.

is placed in the suspended task field, while the sender is blocked waiting for a response. In the case of a synchronous call ([*sync mth call*]), the message contains only the method to call and the actual parameters. The message is added to the queue of the receiver; the sender immediately returns and can continue its computation.

When an agent is idle (i.e., the last three fields of the agent record are ϵ), a message can be removed from the incoming queue to be served. In the case of a synchronous message [*sync exec*], the sender of the message is added to the waiting agent field. Formal parameters are replaced with the actual parameters, and the expression in the method body is added to the active task field. In the case of an asynchronous message [*async exec*], since no answer is expected by the sender, the waiting agent field remains empty. After the parameter substitution, the expression in the method body is added to the active task field.

The rule [*sync return*] is triggered when the receiving agent has finished the computation associated with the message. In that case, the reference to the waiting agent is removed from the receiver. The sender's original computation is reactivated by moving it from the suspended task field to the active field and reducing the method call to the return value computed by the callee.

Subconfiguration reduction. Reduction applies to a subconfiguration by the rule [*subconf*]. Rules application on a partial configuration involves the reduction of the other partial configurations up to the top level configuration.

Proceed. The [*proceed*] rule formalizes the semantics of proceed, which calls the next eligible method in the active variations sequence. A proceed call from inside a method m belonging to the variation of name vml_i, works as follows. The sequence of the variations following vml_i in the sequence of the active variations names of the agent concatenated with the callback module name $vml_{i+1} \ldots vml_k \, cml$ is searched for a method with the same signature of m. The retrieved method is then executed binding the actual parameters of the proceed call. Note that the proceed primitive is semantically different with respect to the standard method calls, since it is synchronous and it is immediately executed without interaction with the inbox queue of the called agent.

Variation change. An agent can change the sequence of its own active variations by invoking the selfChange primitive. Like proceed, this primitive does not add a message to the inbox of the agent, but is immediately executed. The variation names list in the call becomes the new sequence of active variations in the agent [*self var change*]. The names of the variations to be activated must be in the set of the already loaded modules names L of the node the agent belongs to. Otherwise, the variations are required to be in the declarations of the node. In this case, they are (lazily) loaded by adding them to L.

As we already noted, the formal semantics includes only the *immediate* version of functions for variation manipulation. However, in practical applications it is often required that a synchronous or asynchronous request for a

19

variation change is sent from another agent to the agent to be affected by the change. In the semantics, this can be easily achieved by adding a `varChange($\overline{vml}$)` method to the callback module of the receiving agent. This method simply executes the `selfChange($\overline{vml}$)` operation.

Variation sending. The `varSend` primitive allows an agent on a node n_1 to add to a remote node n_2 a variation, which was previously unknown to n_2, i.e., which is not in the declarations D of n_2 [*variation sending*]. The variation name is immediately added to the set of loaded module names L of n_2. This behavior formalizes eager code loading.

4.4. Program Execution

The execution of a CONTEXTERLANGLITE system starts with an initial agent running alone in the main node. All the other nodes contain no active agents. The computation is triggered by the initial agent spawning new agents, possibly on other nodes. The initial system configuration is a set of nodes N_1 ... N_k and an initial agent P_1:

$$Conf = N_1 \; N_2 \; ... \; N_i \; ... \; N_k \; P_1$$

Each node $N_i = \langle L, n_i \rangle_N$ is associated with a node declaration in the program, where n_i is the node identifier stated in the declaration. N_1 is the main node and n_1 its identifier. The initial configuration of each node has an empty set of loaded module names L.

$$N_i = \langle \varnothing, n_i \rangle_N \quad \forall i : i \in 1 \; ... \; k$$

The initial agent P_1, whose unique identifier is w_1, starts the computation in node n_1. To keep the notation uniform with the other agents, the callback module field is initialized with a fictitious name `main`, which however is never used in the execution.

$$P_1 = \langle p_1, \texttt{main}, \varnothing, \varnothing, \epsilon, mainexp, \epsilon \rangle_P \qquad p_1 = \langle w_1, n_1 \rangle_R$$

The inbox is initialized to the empty set, the *sender* field and the *susp* field are set to ϵ. The expression *mainexp* is set as the active task of the process, triggering the all subsequent computation.

4.5. Properties

In this section we consider our guiding principles for managing context in a concurrent and distributed environment, presented in Section 3.4, and show that they are enforced by the semantics. For convenience, we start from the second principle, i.e., *Agent authority*, that states *an agent retains ultimate authority on its current context*. We formalize it as the following statement:

Statement 1. Agent authority.
$\forall \; w, n, cml, A, A', M, M', p_{\epsilon susp}, p'_{\epsilon susp}, Conf, Conf', e, e_{\epsilon susp}, e', e'_{\epsilon susp}$
if $\;\nexists e_\square, vml \;\; such\; that\; e = e_\square[\texttt{selfChange}(\overline{vml})]$
and $\;\; \langle \langle w, n \rangle_R, cml, A, M, p_{\epsilon susp}, e, e_{\epsilon susp} \rangle_P, Conf \to \langle \langle w, n \rangle_R, cml, A', M', p'_{\epsilon susp}, e'_\epsilon, e'_{\epsilon susp} \rangle_P, Conf'$
then $\;\; A = A'$.

This result is a direct consequence of the fact that in the semantics the only way of modifying A is by [*self var change*].

The first principle, i.e., *Non interference*, states that *the context of a running computation cannot be altered by a contextual message*. We formalize it as in the following statement:

Statement 2. Non interference.
$\forall \; w, n, cml, A, A', M, M', p_{\epsilon susp}, p'_{\epsilon susp}, Conf, Conf', e, e_{\epsilon susp}, e', e'_{\epsilon susp}$
if $\;\nexists e_\square, vml \;\; such\; that\; e = e_\square[\texttt{selfChange}(\overline{vml})]$
and $\;\; \langle \langle w, n \rangle_R, cml, A, M \cdot msg, p_{\epsilon susp}, e, e_{\epsilon susp} \rangle_P, Conf \to \langle \langle w, n \rangle_R, cml, A', M', p'_{\epsilon susp}, e'_\epsilon, e'_{\epsilon susp} \rangle_P, Conf'$
where msg is either $\langle p_2, \texttt{varChange}(\overline{vml}) \rangle_M$ *or* $\langle \texttt{varChange}(\overline{vml}) \rangle_M$
then $\;\; A = A'$.

Proof. We know that if $e \neq \epsilon$, then the computation is still "running". Hence, according to the semantics, we can apply neither [*sync exec*], nor [*async exec*]. We also know that e does not contain any `selfChange` subterm, so we cannot apply [*self var change*]. Therefore, $A' = A$. $\qquad\square$

4.6. Discussion

The development of CONTEXTERLANG's formal semantics is a key contribution of this work. As we showed above, it allowed us to reason about all the corner cases in the execution of CONTEXTERLANG programs. In particular, the critical aspect is the consistency of the execution of CONTEXTERLANG programs across the change of variations. The formalization shows that execution of adapted code and behavioral changes do not interfere.

In addition, the semantics was an invaluable tool to reason about the implications of our design choices. For example, the need for immediate functions for variations activation (Section 3.4) firstly emerged in the development of the semantics. Interestingly, this issue then practically arose in the development of ContextChat and we finally formulated it as the *agent authority* principle stated in the previous sections.

Finally, to get more confidence on the soundness of our formalization, we used Maude [30] to develop an executable prototype. Tool support helped us to test the semantics, checking its correct execution. A semantics with evaluation contexts, such as the one we introduced here, requires some machinery to be properly encoded in rewriting logic [31]. Existing tools provide context evaluation out of the box, such as PLT Redex [32] and K-Maude [33]. However we preferred to apply some simplifications and remove evaluation contexts. This choice is motivated by efficiency considerations and by the possibility of fine-grain tuning in anticipation of the use of our prototype for verification, a topic that has been already explored in Maude [34]. For example, we chose an imperative style with storage for the execution of method bodies and a storage stack for nested `proceed` calls in a way that resembles function calls in assembly languages. At the price of a less elegant model and of some expressive limitations, this approach allowed us to get rid of evaluation contexts.

5. Validation

In this section, we discuss how we empirically validated CONTEXTERLANG. First, to demonstrate that CONTEXTERLANG is effective in the development of real-world applications we describe here two prototypes: one is the ContextChat extensively presented in the previous sections, and the other is an autonomic storage server that will be analyzed next. Second, to show that the design of CONTEXTERLANG is applicable in general to languages supporting the Actor Model, we implemented CONTEXTSCALA, a COP language based on Scala. Finally, to provide empirical evidence of its usability, we studied the critical performance aspects of CONTEXTERLANG and CONTEXTSCALA through a micro-benchmark and compared performance with other COP languages. Then we reimplemented the autonomic storage server in plain Erlang and compared its performances with the CONTEXTERLANG version.

5.1. The Adaptive Storage Server Case Study

The second case study presented in this paper concerns the development of an adaptive storage server. The storage server is an autonomic application providing storage a space for generic resources such as web pages or serialized data structures. The application behaves like a key-value map: keys allow one to retrieve resources or modify their value. Resources can be stored in memory or on disk. Autonomicity ensures 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 who 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 [35]. 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 validity of the design choices of CONTEXTERLANG. Since the `on_disk` and the `in_memory` variations are mutually exclusive, they can be 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 (Section 3.3). Note that moving the autonomic capabilities to a centralized engine would require the adaptation to be driven by context-related messages.

```
class OnDisk extends Variation[OnDisk] {          -module(on_disk).
  def setValue(i: Any): Any = { ... }             handle_call({set,Value}, From, State) -> ...
  def getValue(): Any = { ... }                       Ret;
    ...                                           handle_call({get}, From, State) -> ...
}                                                     {reply, Reply, State1}.
                                                   ...


class InMemory extends Variation[InMemory] {       -module(in_memory).
  def setValue(i: Any): Any = { ... }             handle_call({set,Value}, From, State) -> ...
  def getValue(): Any = { ... }                       Ret;
    ...                                           handle_call({get}, From, State) -> ...
}                                                     {reply, Reply, State1}.
                                                   ...




class ResourceAgent() extends ContextAgent {       -module(resource_agent).
    setActiveVariations(List('OnDisk))            -behavior(context_agent).
}                                                  start() ->
                                                    gen_server:start_link({local, ?MODULE},
                                                                          ?MODULE, [], []).
                                                   ...


object StorageServer extends App {                 -module(storage_server).
  val system = ActorSystem("StorageServer")        store(Key,Value) ->
                                                        resource:set(Key,Value).
  def store(key: Key, value: Any): Any = {
    val actor = system.actorFor(key)              lookup(Key) ->
    sendMsg(actor,setValue(value))                    resource:get(Key).
  }                                                 ...
  def lookup(key: Key) = { ... }
    ...
}
case class setValue(i: Any) extends Msg[setValue]
case class getValue() extends Msg[getValue]
```

Figure 15: Fragments of the autonomic storage server in ContextScala (left), and their ContextErlang counterparts (right).

5.2. Validating the Design: ContextScala

To demonstrate that the design principles of ContextErlang are applicable in general to actor-based languages, we implemented ContextScala, a COP language that applies the concepts of ContextErlang to the Scala programming language. We implemented ContextScala on top of the Akka[7] framework. Akka is a toolkit for distributed and highly concurrent fault-tolerant applications in Scala. Like the OTP platform, Akka is based on the Actor model, provides means to distribute actors on different hosts, monitor their behavior and obtain notifications of failures.

Figure 15 (left) shows part of the autonomic storage server implemented in ContextScala and the respective ContextErlang counterparts (right). Clearly, ContextScala applications look different from ContextErlang applications. First, the language abstractions of ContextErlang, originally designed for Erlang, have to be mapped to Scala. Second, the design of ContextScala needs to comply with the Akka design principles. More generally, our goal in the development of ContextScala was to demonstrate the generality of the ContextErlang model, not to maximize the similarity with the Erlang implementation and syntax. Here, we outline the design of ContextScala and summarize the main differences with ContextErlang.

Variations are modeled as classes that extend the Variation[T] class. Like functions in ContextErlang variation modules, methods in ContextScala variations implement chunks of the variation's functionality. For example, in Figure 15, the setValue and the getValue methods in the OnDisk class implement the behavior of the context-aware agent when the disk is used as a storage. An advantage of this design is that variations can be instantiated and store local state. However, this feature comes at the cost of creating an instance of the variation for each agent.

---
[7]http://akka.io

CONTEXTERLANG and CONTEXTSCALA differ in the way they represent messages. In OTP and CONTEXTERLANG, messages are modeled as tuples. According to the Akka best practices, instead, in CONTEXTSCALA messages are encoded as Scala case classes, which are immutable, and can be pattern matched. Figure 15 shows the `setValue` and the `getValue` messages used to assign – respectively, store – a value into an agent of the autonomic storage. CONTEXTSCALA agents inherit from the `ContextAgent` class, which provides the functionalities of context-aware agents, including reacting to contextual messages and dispatching among variations. Internally, `ContextAgent` extends `akka.actor.Actor`, the standard actors in the Akka toolkit. As a result, CONTEXTSCALA context-aware agents can interoperate with the rest of the Akka infrastructure in the same way CONTEXTERLANG context-aware agents are compatible with the OTP.

The implementation of CONTEXTSCALA is quite different from CONTEXTERLANG. Erlang is a dynamically typed language, while Scala is statically typed, which makes it harder to implement a custom dispatching mechanism for method calls. Internally, CONTEXTSCALA relies on reflection to overcome the restrictions imposed by the type system. Obviously, this comes at the cost of reducing safety. For example, run time casts are needed to refine the type of the return value for variations methods. We note, however, that the Akka actor system already weakens safety compared to traditional applications, since messages are pattern-matched at run time and the type of the return value cannot be checked statically.

Despite the obvious differences due to the underlying languages, CONTEXTSCALA and CONTEXTERLANG share the same key design principles and formal semantics. In CONTEXTSCALA, like in CONTEXTERLANG, developers shape adaptive applications around the concept of context-aware agents and, like in CONTEXTERLANG, adaptations are organized through variations that are activated and deactivated dynamically. Furthermore, CONTEXTSCALA enforces the agent authority and the non interference principles, thanks to the same design inherited from CONTEXTERLANG.

5.3. 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 even in the absence of specific optimizations. All the tests we report hereafter 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, Scala 2.10.

Microbenchmark. We compare the overhead introduced by CONTEXTERLANG with respect to other COP implementations [36]. 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 [37] for AOP micro-benchmarking: compare methods performance without aspects (i.e., a non-advised method) and with aspects deployed.

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 3 to 8). Each method and each partial method increments a global 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 would be meaningless 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 16 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 [38] highlighted a huge performance impact of COP and motivated research on possible optimizations [39]. 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

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
ContextScala	301.25 (Akka)	502.30	795.03	980.15	1120.13	1302.18	1521.96

Table 1: Performance of COP languages in the microbenchmark. All values are in milliseconds.
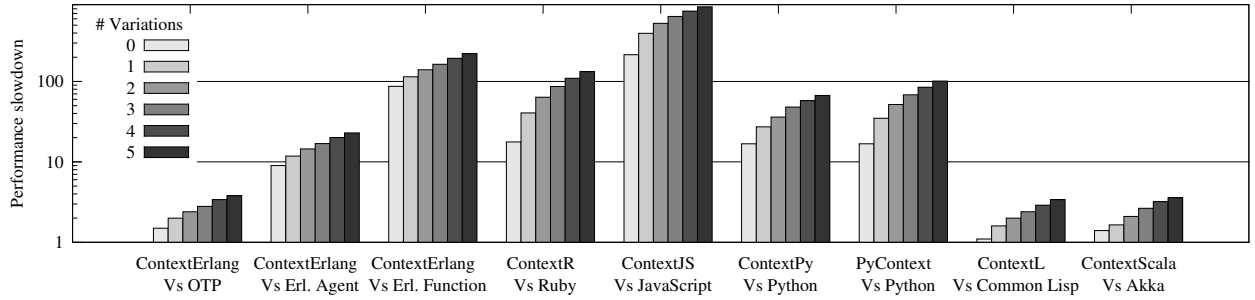
Figure 16: Performance of layered methods compared to the basic methods in various COP languages.

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 16 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 16. Nevertheless, ContextJS is among the fastest COP extension in our test (Table 1).

To overcome the limitations of micro-benchmarking, we estimated the overhead of CONTEXTERLANG in the adaptive storage server. 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. 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 17 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.

6. Related Work

Our work touched several research areas. For each of them, in the sequel, we briefly discuss related work.

Self-adaptive software. An overview of self-adaptive software, the existing technology and the open research challenges can be found in [2]. According to that work, adaptation mechanisms can be classified along the *artifact & granularity* analysis direction, which include parameters, method, aspect, component, application, architecture, system and data center. In that classification, COP abstractions roughly lie at the method/aspect granularity level. The problem of dynamic software adaptation has been extensively tackled from a software architecture standpoint [4, 5, 6]. Architecture-based adaptation frameworks include Rainbow [40] and the Fractal component system [41]. McKinley
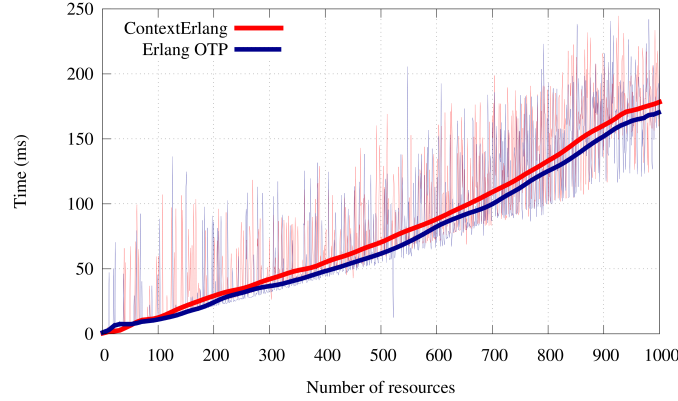
Figure 17: Performance comparison for the autonomic storage server.

et al. [9] analyze the features required by self-adaptive software and identify AOP, metaprogramming and component-based architecture as enabling technologies. Special-purpose programming paradigms like metaprogramming and AOP have been employed for a long time by researchers to support self-adaptive software. An analysis and comparison of metaprogramming, AOP and COP in this context can be found in [36].

Context-oriented programming. COP has been recently explored, starting from the pioneering work on ContextL [16, 23] 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 [7, 15, 19, 21, 42]. A comparison of the existing COP languages with a performance evaluation of the available solutions can be found in [38]. Our recent work [8] surveys the available solutions and compares them from a software engineering standpoint.

ContextErlang [1, 22] 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 [43]. Ambience – designed simultaneously with ContextL – is alternative approach to layer-based COP languages, leveraging multimethods dispatching and context objects. In [43] 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 [44], in which we presented the initial implementation of ContextErlang as a promising solution. As already discussed (Section 2) Kamina et al. [21] also tackled this issue in the EventCJ Java COP extensions.

Jcop [19] 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

25

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 AOP [10], 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 [21, 19, 42]. However, although some AOP frameworks exist support dynamic aspect activation, such as Prose [45], 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 [16, 19, 7].

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 [18] offer a linguistic support for this mechanism, obtaining better encapsulation of crosscutting concerns and decoupling from other code.

EScala [46] supports not only events that are imperatively triggered by the programmer, but also implicit events that are transparently raised at method boundaries, in the style of AOP. Ptolemy [47] also 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.

Formal approaches in COP. Recently, researchers started investigating the use of formal models to study COP systems. Schippers *et al.* [48] present a semantics of *layers* using a delegation-based machine devoted to the modelling of crosscutting concerns. Schippers *et al.* [49] use a graph-transformation approach for the operational semantics of a delegation-based OO language with actors and layer activation. Kamina *et al.* [21] propose to describe a COP system as a finite state automaton. States model context conditions of the system and arcs model context transitions. The automaton is translated into Promela and verified with the SPIN model checker [50]. The works by form Clarke and Sergey [51] and by Hirschfeld on ContextFJ [52] describe core calculi that extend Featherweight Java [53] to encompass COP abstractions. A further extension of ContextFJ also admits changes in the interface of objects [54]. To the best of our knowledge, none of those formalization takes into account distribution and concurrency, as Context-ERlangLite does.

Other language-level techniques. Subjective dispatch [55] 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 [56] 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.

Roles describe objects' state and behavior in a certain context [57]. Pradel and Odersky [58] propose Scala Roles, a library for Scala that provides support for augmenting objects with roles. As in ContextScala, new functionalities can be added at run time. In contrast to Scala Roles, where roles apply to objects, in ContextScala, variations apply to Context-aware agents, which also encapsulate Actor-like concurrency.

Several modularization approaches have been proposed in literature, such as traits [59] and mixins [60], which offer an alternative to the modularization mechanism provided by classes. Multiclass-granularity solutions include mixin layers [61] and delegation layers [62]. Compared to COP, these approaches focus on composition of functionalities and not on activation and deactivation of behavioral variations.

7. Conclusions

In this work, we presented ContextErlang, a COP language that supports several features commonly required by adaptive systems. The design of ContextErlang integrates dynamic adaptation and modularization of behavioral variations with asynchronous activation and distribution. In addition, ContextErlang supports unforeseen adaptation and variation constraints – a way to discipline dynamic activation of multiple variations. The key design decision of ContextErlang has been to integrate the actor model with COP abstractions, a solutions which, to the best of our knowledge, has never been proposed before. We provided an implementation that fits into the OTP, the *de facto* standard for real-world Erlang applications. The paper also presented ContextErlang's formalization with a core calculus that specifies the exact semantics. Finally, it provided an extensive empirical assessment of its potential benefits in the development of self-adaptive software.

Currently, programmers of adaptive systems leverage software architectures or design patterns to implement adaptation features. Some solutions also rely on language-level approaches for specific tasks. For example AOP is used to intercept the program execution at certain joinpoints and redirect the control flow depending on the current context [36]. We believe that addressing the requirements of adaptive software with a coherent language design can encourage programmers of adaptive-systems to adopt language-level solutions and benefit of their expressivity, conciseness and safety.

References

[1] G. Salvaneschi, C. Ghezzi, M. Pradella, ContextErlang: introducing context-oriented programming in the actor model, in: Proceedings of the 11th annual international conference on Aspect-oriented Software Development, AOSD '12, ACM, New York, NY, USA, 2012, pp. 191–202.

[2] M. Salehie, L. Tahvildari, Self-adaptive software: Landscape and research challenges, ACM Trans. Auton. Adapt. Syst. 4 (2009) 14:1–14:42.

[3] J. O. Kephart, D. M. Chess, The vision of autonomic computing, Computer 36 (2003) 41–50.

[4] P. Oreizy, N. Medvidovic, R. N. Taylor, Architecture-based runtime software evolution, in: ICSE '98: Proceedings of the 20th international conference on Software engineering, IEEE Computer Society, Washington, DC, USA, 1998, pp. 177–186.

[5] R. N. Taylor, N. Medvidovic, P. Oreizy, Architectural styles for runtime software adaptation, in: 3rd European Conference on Software Architecture (ECSA), IEEE, 2009, pp. 171–180.

[6] J. Kramer, J. Magee, Self-managed systems: an architectural challenge, in: 2007 Future of Software Engineering, FOSE '07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 259–268.

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

[8] G. Salvaneschi, C. Ghezzi, M. Pradella, Context-oriented programming: A software engineering perspective, Journal of Systems and Software 85 (2012) 1801 – 1817.

[9] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, B. H. C. Cheng, Composing adaptive software, Computer 37 (2004) 56–64.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold, An overview of AspectJ, in: J. Knudsen (Ed.), ECOOP 2001 – Object-Oriented Programming, volume 2072 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 2001, pp. 327–354. 10.1007/3-540-45337-718.

[11] A. Popovici, T. Gross, G. Alonso, Dynamic weaving for aspect-oriented programming, in: Proceedings of the 1st international conference on Aspect-oriented software development, AOSD '02, ACM, New York, NY, USA, 2002, pp. 141–147.

[12] A. Popovici, G. Alonso, T. Gross, Just-in-time aspects: efficient dynamic weaving for Java, in: Proceedings of the 2nd international conference on Aspect-oriented software development, AOSD '03, ACM, New York, NY, USA, 2003, pp. 100–109.

[13] R. Pawlak, L. Seinturier, L. Duchien, G. Florin, JAC: A flexible solution for aspect-oriented programming in Java, in: A. Yonezawa, S. Matsuoka (Eds.), Reflection, volume 2192 of *Lecture Notes in Computer Science*, Springer, 2001, pp. 1–24.

[14] R. Pawlak, L. Seinturier, L. Duchien, G. Florin, F. Legond-Aubry, L. Martelli, JAC: An aspect-based distributed dynamic framework (2004).

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

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

[17] G. Salvaneschi, C. Ghezzi, M. Pradella, Context-oriented programming: A programming paradigm for autonomic systems, CoRR abs/1105.0069 (2011).

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

[19] M. Appeltauer, R. Hirschfeld, H. Masuhara, M. Haupt, K. Kawauchi, Event-specific software composition in context-oriented programming, in: B. Baudry, E. Wohlstadter (Eds.), Software Composition, volume 6144 of *LNCS*, 2010.

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

[21] K. Tetsuo, A. Tomoyuki, 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.

[22] C. Ghezzi, M. Pradella, 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 (2010).

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

[24] P. Costanza, T. D'Hondt, Feature descriptions for context-oriented programming, in: Software Product Lines, 12th International Conference (SPLC), 2008, pp. 9–14.

[25] S. González, N. Cardozo, K. Mens, A. Cádiz, J.-C. Libbrecht, J. Goffaux, Subjective-C: Bringing context to mobile platform programming, in: Proceedings of the International Conference on Software Language Engineering, 2010, Lecture Notes in Computer Science, Springer-Verlag, Eindhoven, The Netherlands.

[26] J. Schäfer, A. Poetzsch-Heffter, JCoBox: Generalizing active objects to concurrent components, in: 24th European Conference on Object-Oriented Programming (ECOOP 2010), LNCS, Springer, 2010, pp. 275–299.

[27] F. S. DeBoer, D. Clarke, E. B. Johnsen, A complete guide to the future, in: Proc. 16th European Symposium on Programming (ESOP07), volume 4421 of LNCS, Springer-Verlag, 2007, pp. 316–330.

[28] M. Flatt, S. Krishnamurthi, M. Felleisen, A programmer's reduction semantics for classes and mixins, in: Formal Syntax and Semantics of Java, Springer-Verlag, London, UK, 1999, pp. 241–269.

[29] M. Felleisen, R. Hieb, The revised report on the syntactic theories of sequential control and state, Theor. Comput. Sci. 103 (1992) 235–271.

[30] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. L. Talcott, All About Maude — A High-Performance Logical Framework. How to Specify, Program and Verify Systems in Rewriting Logic, volume 4350 of *Lecture Notes in Computer Science*, Springer, 2007.

[31] T. F. Serbanuta, G. Rosu, J. Meseguer, A rewriting logic approach to operational semantics, Information and Computation 207 (2009) 305 – 340. Special issue on Structural Operational Semantics (SOS).

[32] M. Felleisen, R. B. Findler, M. Flatt, Semantics Engineering with PLT Redex, The MIT Press, 2009.

[33] T. F. Şerbanuţă, G. Roşu, K-Maude: A rewriting based tool for semantics of programming languages, in: P. C. Ölveczky (Ed.), Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010, volume 6381 of *Lecture Notes in Computer Science*, pp. 104–122.

[34] S. Eker, J. Meseguer, A. Sridharanarayanan, The Maude LTL model checker, in: F. Gadducci, U. Montanari (Eds.), Fourth Workshop on Rewriting Logic and its Applications, WRLA '02, volume 71 of *Electronic Notes in Theoretical Computer Science*, Elsevier, 2002.

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

[36] G. Salvaneschi, C. Ghezzi, M. Pradella, An analysis of language-level support for self-adaptive software, ACM Trans. Auton. Adapt. Syst. 8 (2013) 7:1–7:29.

[37] M. Haupt, M. Mezini, Micro-measurements for dynamic aspect-oriented systems, in: M. Weske, P. Liggesmeyer (Eds.), Object-Oriented and Internet-Based Technologies, volume 3263 of *LNCS*, Springer Berlin / Heidelberg, 2004.

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

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

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

[41] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, J.-B. Stefani, The FRACTAL component model and its support in Java: Experiences with auto-adaptive and reconfigurable systems, Softw. Pract. Exper. 36 (2006) 1257–1284.

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

[43] S. González, K. Mens, P. Heymans, Highly dynamic behaviour adaptability through prototypes with subjective multimethods, in: Proceedings of the 2007 symposium on Dynamic languages, DLS '07, pp. 77–88.

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

[45] A. Popovici, T. Gross, G. Alonso, Dynamic weaving for aspect-oriented programming, in: Proceedings of the 1st international conference on Aspect-oriented software development, AOSD '02.

[46] V. Gasiunas, L. Satabin, M. Mezini, A. Núñez, J. Noyé, EScala: modular event-driven object interactions in Scala, in: Proceedings of the tenth international conference on Aspect-oriented software development, AOSD '11, ACM, New York, NY, USA, 2011, pp. 227–240.

[47] H. Rajan, G. T. Leavens, Ptolemy: A language with quantified, typed events, in: J. Vitek (Ed.), ECOOP 2008, Cyprus, volume 5142 of *LNCS*, Berlin, pp. 155–179.

[48] H. Schippers, D. Janssens, M. Haupt, R. Hirschfeld, Delegation-based semantics for modularizing crosscutting concerns, in: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, OOPSLA '08, ACM, New York, NY, USA, 2008, pp. 525–542.

[49] H. Schippers, T. Molderez, D. Janssens, A graph-based operational semantics for context-oriented programming, in: International Workshop on Context-oriented Programming at ECOOP'10.

[50] G. Holzmann, Spin model checker, the: primer and reference manual, Addison-Wesley Professional, first edition, 2003.

[51] D. Clarke, I. Sergey, A semantics for context-oriented programming with layers, in: International Workshop on Context-Oriented Programming, COP '09, ACM, New York, NY, USA, 2009, pp. 10:1–10:6.

[52] R. Hirschfeld, A. Igarashi, H. Masuhara, ContextFJ: a minimal core calculus for context-oriented programming, in: Proceedings of the 10th

international workshop on Foundations of aspect-oriented languages, FOAL '11, ACM, New York, NY, USA, 2011, pp. 19–23.

[53] A. Igarashi, B. C. Pierce, P. Wadler, Featherweight Java: A minimal core calculus for Java and GJ, in: ACM Transactions on Programming Languages and Systems, pp. 132–146.

[54] A. Igarashi, R. Hirschfeld, H. Masuhara, A type system for dynamic layer composition, in: In Proceedings of the Workshop on the Foundations of Object-oriented Languages (FOOL), co-located with the Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA) 2012, ACM, Tucson, Arizona, USA.

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

[56] D. Batory, J. N. Sarvela, A. Rauschmayer, Scaling step-wise refinement, in: Proceedings of the 25th International Conference on Software Engineering, ICSE '03, IEEE Computer Society, Washington, DC, USA, 2003, pp. 187–197.

[57] F. Steimann, On the representation of roles in object-oriented and conceptual modelling, Data Knowl. Eng. 35 (2000) 83–106.

[58] M. Pradel, M. Odersky, Scala Roles - A lightweight approach towards reusable collaborations, in: International Conference on Software and Data Technologies (ICSOFT '08).

[59] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, A. P. Black, Traits: A mechanism for fine-grained reuse, ACM Trans. Program. Lang. Syst. 28 (2006) 331–388.

[60] G. Bracha, W. Cook, Mixin-based inheritance, in: Proc. OOPSLA 90, ACM Press, 1990, pp. 303–311.

[61] Y. Smaragdakis, D. Batory, Implementing layered designs with mixin layers, in: In ECCOP 98: Proceedings of the 12th European Conference on Object-Oriented Programming, Springer, 1998, pp. 550–570.

[62] K. Ostermann, Dynamically composable collaborations with delegation layers, in: Proceedings of the 16th European Conference on Object-Oriented Programming, ECOOP '02, Springer-Verlag, London, UK, 2002, pp. 89–110.