Consistent Unanticipated Adaptation for Context-Dependent Applications

Nguonly Taing[1] Markus Wutzler[1] Thomas Springer[1] Nicolás Cardozo[2] Alexander Schill[1]

[1]Faculty of Computer Science, Technische Universität Dresden, Germany
[2]Systems and Computing Engineering Department, Universidad de los Andes, Colombia
[1]{firstname.lastname}@tu-dresden.de, [2]n-cardoz@uniandes.edu.co

Abstract

Unanticipated adaptation allows context-dependent applications to overcome the limitation of foreseen adaptation by incorporating previously unknown behavior. Introducing this concept in language-based approaches leads to inconsistencies as an object can have different views in different contexts. Existing language-based approaches do not address unanticipated adaptation and its associated run-time inconsistencies. We propose an architecture for unanticipated adaptation at run time based on dynamic instance binding crafted in a loosely manner to asynchronously replace adaptable entities that allow for behavioral changes of objects. To solve inconsistencies, we introduce the notion of transactions at the object level. Transactions guard the changing objects during their execution, ensuring consistent views. This allows for disruption-free, safe updates of adaptable entities by means of consistent unanticipated adaptation.

Categories and Subject Descriptors D.2.11 [*Software Architectures*]: Framework

Keywords Unanticipated Adaptation, Role, Consistency

1. Introduction

Adaptive software systems normally deal with anticipated adaptation enabling the base object to change its behavior with respect to the context. System designers are required to provide definitions of adaptable entities and their adaptations [13, 15] in advance. In highly dynamic environments it might be infeasible to foresee all possible adaptations. Changing behavior requires to shutdown the running application which causes a disruption for multi-user applications as clients need to reconnect and restart their tasks. This obviously contradicts the concept of run-time sustainability which should support both anticipated and unanticipated adaptation [17].

Arbitrarily updating a running application might lead to inconsistent states which results in a system that behaves unexpectedly. Consider a file transfer application (Figure 1) that allows multiple clients to download files. Since files to be transferred are large, the implementation of the basic `Transfer` functionality splits them into smaller chunks and sends them to the respective clients. Assume now that a client requests that the chunks should be encrypted before the transfer. Since the server runtime supports unanticipated adaptations, the file transfer server can be extended with the encryption behavior without stopping and restarting the server.

The situation is depicted in Figure 1, showing two clients using the file transfer service. `Client 1` starts a transfer using the basic transfer functionality of splitting files. While the transfer for `Client 1` is running, `Client 2` requests the transfer of encrypted chunks which triggers the incorporation of the encryption behavior into the file server. If the transfer for `Client 1` is still running, the unanticipated adaptation does not only affect the newly established transfer session for `Client 2` but also the session of `Client 1` that receives encrypted chunks after the system change. Consequently, the adaptation violates the consistency of the system.

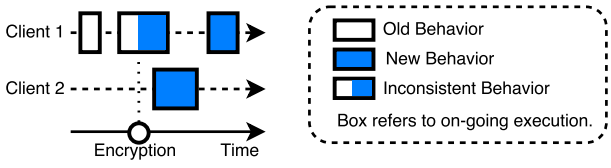

Figure 1: Inconsistency of Unanticipated Adaptation

Context-oriented Programming (COP) enables dynamic adaptation to the context [13, 15]; however, this technique lacks support for unanticipated adaptation. In COP, behavioral inconsistencies may occur when multiple layers are activated asynchronously, modifying the behavior of the ex-

COP'16 July 17–22, 2016, Rome, Italy

DOI: http://dx.doi.org/10.1145/2951965.2951966

ecuting code block. To manage these inconsistencies, COP languages such as EventCJ [8], ServalCJ [9], Subjective-C [3], and Ambience [2, 5], provide mechanisms to ensure consistent behavior execution as contexts are activated. These mechanisms prevent conflicts between adaptations known beforehand, e.g., specifying that the adaptations `GPSNavi` and `WiFiNavi` should not be active simultaneously in an indoors mapping application [8].

This paper proposes an approach for disruption-free, safe updates of adaptable entities by means of consistent unanticipated adaptation. First, we provide an architecture to realize unanticipated adaptation based on the concept of dynamic instance binding [15]. Second, we introduce the notion of *transaction* to guard the behavioral objects' change caused by asynchronous binding of unexpected adaptable entities to base instances. The transaction is inspired by the concept of *Tranquility* [16] ensuring components are in a consistent state before and after the update.

The contributions of this paper are twofold.

- A Java-based run-time architecture for unanticipated adaptation in which the adaptable entities of an application can be (re)loaded enabling behavioral change to certain instances without restarting the runtime environment.
- Harness the flexible dynamic instance binding and centralized dynamic method dispatch from our architecture to bring the concept of transaction to the object level to safely update and adapt the run-time part.

2. Background

Our approach to handle unanticipated adaptation is based on the concept of roles [14] as a means to dynamically change objects' behavior.

2.1 Roles are Implicitly Context-Dependent

A single object may contain static and dynamic parts. For example, a `Person` object has a social security number and a name. These attributes are fixed throughout the object's lifetime. Objects may also have transient properties or behaviors according to a specific role they may play. For example, a `Person` could be a student or an employee. Such transient parts are made available dynamically (e.g., *activated*, *bound*), whenever they are needed in a concrete execution context, for example in a school or workplace. In such scenarios, objects should be modeled by splitting the dynamic parts (represented as roles) from the static ones. By doing this, we achieve separation of concerns and code re-use. Object instances can present different behavior at run time by merging the dynamic parts of the object to its static parts.

This paper abstracts the role concept as views that provide implicit context-dependent behaviors. The terms adaptable entity, dynamic part, and role are used interchangeably to describe the part of the objects that modify the behavior of the static part or core object. There are several role features out of discussion but it is worth to have a look in the 26 feature list proposed by Steimann [14] and Kühn et al. [11].

2.2 Dynamic Instance Binding

This section describes a dynamic instance binding mechanism [15] that arbitrarily binds objects' static part, modeled as *players*, and their dynamic parts, presented as *roles*. The key idea is to enable the behavior of the dynamic parts in their binding to the static parts. The behavioral composition of these objects happens at the instance level which enables non-uniform adaptation for each instance.

The dynamic instance binding is inspired by the code weaving scheme of Aspect-oriented Programming (AOP), but rather than weaving players with multiple roles at the byte code or source code levels, we keep the two distinct entities independent inside our runtime. A look-up table (i.e., relational schema), is required to draw the binding information for dynamic method dispatching. When methods are invoked, the dispatcher queries and selects appropriate role instances from an instance pool and invokes them through reflection. An architecture of this framework is partially depicted in Figure 3.

Snippet 1: File Transfer

```
1 class Transfer{ void send(){...} }
2 class Encryption{ void send(){...} }
3
4 public static void main(String[] args){
5  Registry reg=Registry.getInstance();
6  Object transfer=reg.newPlayer(Transfer.class);
7  reg.invokeRole(transfer, "send"); //send raw
8  reg.bind(transfer, Encryption.class);
9  reg.invokeRole(transfer, "send"); //encrypted
10 }
```

Table 1: A Look-up Table

Id	CoreId	RoleId	Sequence	...
1	transfer	Encryption	1	...

Snippet 1 shows how behavioral adaptations are encoded and achieved in our framework using dynamic binding for the case of the file transfer application. The `Transfer` and `Encryption` objects both implement the `send()` method (Lines 1-2). Assume the `Encryption` object is given at design time. `Registry` is the mediator managing the instance pool, look-up table, and method dispatching. In Line 7, the `transfer` instance calls its `send()` method (since there is no role binding yet). After binding to an `Encryption` object in Line 8, the relation is recorded in the look-up table (Table 1). The sequence represents the ordering of bindings to dispatch polymorphic methods. The `transfer` instances calls the `send()` method again. This time the `send()` method of `Encryption` class is selected for invocation (Line 9).

3. Unanticipated Adaptive Runtime

The design decision of the dynamic instance binding, described in Section 2.2, is to achieve modularity not only at

design time but also at run time. The modular run-time enables easy replacement of adaptable entities (role) dynamically at run time generating the possibility of (re)loading existing or new role triggering *unanticipated adaptation*.

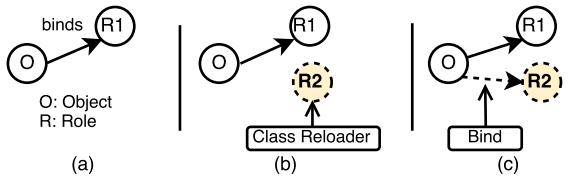

Figure 2: A Concept of Unanticipated Adaptation

A stepwise concept of bringing roles at run time is depicted in Figure 2. First, it is necessary to have a dynamic instance binding as a runtime. Second, new roles are (re)loaded via a dynamic *Class Reloader*. Last, the binding relation is constructed. The entire process does not destroy the existing core objects. So that the application states are fully preserved. This increases both flexibility and less disruption when performing updates, as changing every instance of a given type is not necessary.

3.1 Architecture

The *Unanticipated Adaptive Runtime* Architecture[1] is illustrated in Figure 3. The architecture consists of 4 main components namely *Preparation*, *Watcher Service*, *Unanticipated Adaptation* and *Dynamic Instance Binding Runtime*. In these four components, there are 7 steps in total to be completed for adapting the behavior. Step 1 takes unknown new behaviors which are required for application update. Such behaviors are coded as roles and compiled in Step 2 (*Preparation*). New roles are going to be bound to existing core objects, so that the core instances must be queried for identity from the runtime (Step 3) by a given tool. After that, the required unanticipated operations are configured in the *Adaptation.xml* in Step 4. The structure of this XML file is given in Snippet 2. Next, the *Watcher Service*, a daemon executing in a separate thread, monitors the change of XML file and fires events to an *Unanticipated Adaptation* component. In step 5, the *Unanticipated Adaptation* component handles the parsing of XML files conforming to the adaptation operations that align with the API in the *Runtime*. In step 6, the runtime performs tasks with respect to instructions defined in XML file. Finally, in step 7, classes are reloaded and bound to particular running objects granting new behaviors.

3.2 Unanticipated Adaptation in Action

The watcher service is a separately threaded daemon to monitor the change of adaptation XML file. Whenever the file change is detected, the daemon triggers the unanticipated adaptation phase to start parsing the XML configuration and to perform unanticipated adaptive operations. The structure

[1] Prototype is available at https://github.com/nguonly/lyrt-with-transaction

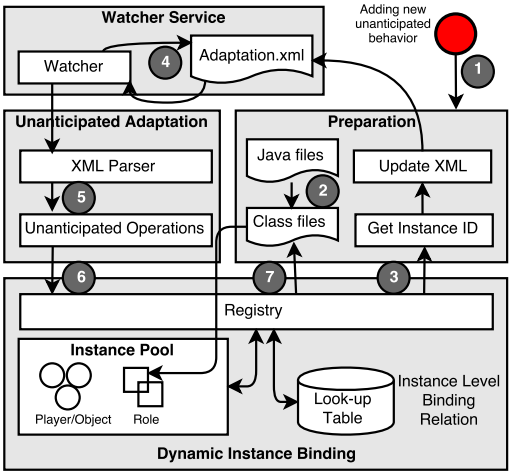

Figure 3: Unanticipated Adaptive Run-time Architecture

of the XML file is shown in Snippet 2. The `bind` and `rebind` functions are the binding operation of a core player with associated `coreId` attribute value to a `roleType` attribute that is stemmed from the *Preparation Phase*. The `bind` operation binds a core player to a new role and if the role type is already bound then it ignores the new binding. In other words, the new definition of roles is not reloading. In contrast, `rebind` operation is a bundle of the `unbind` and `bind` ones. So it always reloads the new definition of roles if developers intend to modify the role source code directly on-the-fly.

Snippet 2: A Sample Adaptation XML

```
<?xml version="1.0"?>
<adaptation>
  <rebind coreId="234" roleType="Encryption" />
  <bind coreId="456" roleType="Compression" />
  <unbind coreId="678" roleType="Encryption" />
</adaptation>
```

Snippet 3: File Transfer Example

```
public static void main(String[] args){
 Thread watcher = new WatcherService();
 watcher.start(); //monitor Adaptation.xml
 Registry reg=Registry.getInstance();
 Object transfer=reg.newPlayer(Transfer.class);
 while(!isEOF(file)){
  reg.invokeRole(transfer, "send"); //call send();
 }
}

//Convert XML to Adaptation Operations
public void parse(){
 Registry reg=Registry.getInstance();
 //parse Adaptation.xml to operation below
 reg.rebind(234, "Encryption");//234=transfer
}
```

Snippet 3 revisits the file transfer example in Figure 1. Assume now that *Encryption* is unknown at design time. There are two threads executing in parallel, the watcher service monitoring the change of XML file and the main thread executing `send()` of the `transfer` instance. In the

loop (Lines 6-8) a raw formatted chunk of a file is continuously sent with original behaviors. Supposedly, the system needs to change the behaviors by replacing from raw to encrypted format. Then the designer prepares for unanticipated adaptation, described earlier, and provides the adaptation description similar to Snippet 2. The change of XML file triggers the code in `parse()`. Subsequently, the `transfer` instance (with 234 identity) is forced to change the sending behavior from raw to encrypted format as the bound `Encryption.class` is currently active.

As noted in the code above, the `Encryption.class` is unknown during the design time. Nonetheless, Java still allows it to be (re)loaded by *Class Loader* after the program has been started. However, there are two main disadvantages; one is the unknown typed system but still it can be referenced as the *Object* type. This problem has no impact on our architecture since only core object's typed system is required. In other words, no typed system of roles is required because of reflection, the method dispatcher looks for and invokes the matched signature of a method of bound roles (cf. Section 2.2). Furthermore, the main code and core objects provided at compile time remain untouched and our architecture changes only their dynamic parts (roles). Another problem is the reloading class definition. It means that once it is loaded, no matter its definition is changed, compiled and reloaded, the first loaded version is still used. The customized `ClassReloader`, a subclass of the standard `ClassLoader`, can be a solution that allows a class to be loaded by multiple class reloaders leading to different instances with different definitions to co-exist in a JVM. This solution has been applied in many systems already such as OSGi[2].

4. Transactions for Behavioral Consistency

This section briefly reviews the concepts of *quiescence* and *tranquility* that address consistent behavioral updates at the component level, and subsequently introduces a transaction mechanism for safe adaptation of long-lasting method executions at the object level.

4.1 Quiescence and Tranquility

In component-based approaches, quiescence and tranquility are mechanisms to allow a component, denoted as a node, to be updated while keeping the whole system in a consistent behavior within a *transaction*. A transaction is a sequence of messages that must be executed atomically [16]. An adapting system needs to wait until a node reaches quiescent or tranquil state in order to update itself. The satisfying conditions for both quiescence and tranquility are described by Kramer and Magee [10] and Vandewoude et al. [16] respectively. Both present a sufficient condition to safely update a node in a consistent manner but tranquility offers less disruption.

[2] www.osgi.org

Consider the updatability of nodes Y and Z as depicted in Figure 4 [16]. In quiescence both Y and Z can be updated as long as they are in passive state which is only in times 1 and 7. Tranquility still requires Y and Z to be passive; however, it allows Z to update at all points in time except 4. Assume the system adapts by replacing Y and Z in time 5. Consequently, Z's behavior has changed from transaction *T[X]* to transaction *T[U]*. Like quiescence, tranquility allows Y to be updated only at times 1 and 7 because *T[X]* is still ongoing. Evidently, transaction *T[W]* initiated by W cannot use an updated version of Y although *T[W]* and *T[X]* are independent of each other.

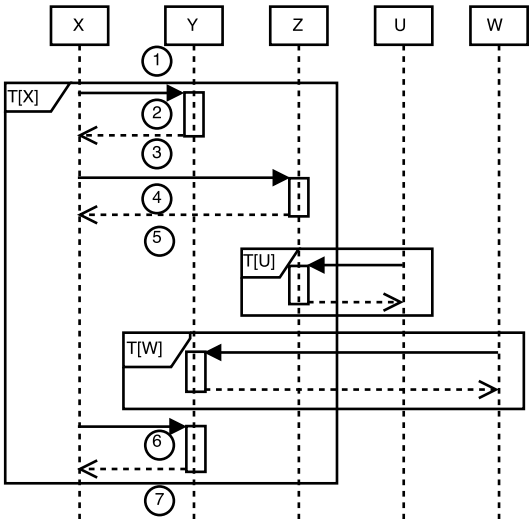

Figure 4: Updatability in a Transaction

Both quiescence and tranquility are proposed without the notion of context-dependent behavior which would enable a node to behave simultaneously different in multiple settings. Both concepts assume a node has a single view before and after it has been updated. This causes a long disruption because Y's updated behavior can be used once Y is not involved in a transaction anymore. Thus we propose another safe update mechanism for context-dependent systems allowing Y to have new behavior in *T[W]* while preserving its old behavior in *T[X]* as it did from times 2 to 5.

4.2 Transaction at Object Level

Both quiescence and tranquility are applicable for component-based approaches where the notion of transaction is explicitly provided and a component is considered a singleton entity that processes messages in a message queue. At object level there is no notion of transaction and objects cannot be considered as black-box entities. The lack of these properties makes it challenging to apply quiescence and tranquility at object level according to Ebraert et al. [6]. Consider now the example in Figure 4 at the object level where Y is an object instance. Updating Y's behavior is achieved by *Unanticipated Adaptation* as described in Section 3. However, ensur-

ing consistency and reducing disruption if an update happens at time 5 remains a challenge as depicted in Figure 1.

We tackle this challenge by introducing the notion of transaction at the object level. We provide a `Transaction` class which defines a code block (Snippet 4) that enables the developer to execute a set of methods in the sense that all engaged instances have a consistent behavior during the whole execution. It prevents engaged instances from applying updates for the current transaction.

Snippet 4: Transaction Declaration

```
try(Transaction tx = new Transaction()){
  while(!isEOF(file)){
    reg.invokeRole(transfer, "send");
  }
}
```

Transaction works on top of our dynamic instance binding and the centralized dynamic method dispatch derived from the binding information in the look-up table.

We extend the existing look-up table (Section 2.2) by adding the *binding time* and *phantom* attribute to keep track of roles bound to instances and to mark roles which have to be removed after a transaction finished. Whenever a transaction is declared, the instance of the transaction is registered in the look-up table with appropriated timestamp entering the transaction and the thread identity activating the transaction (Table 3). The implementation is shown in Snippet 5 with self-explaining comments.

Snippet 5: Transaction Implementation

```
class Transaction implements AutoClosable{
 public Transaction(){ //Transaction starts
  //1. Register this transaction with timestamp
  Registry.addTransaction(this, currentThread);
 }

 @Override
 public void close(){ //Transaction ends
  //1. Remove phantom role if any
  Registry.delPhantomRoles(this);
  //2. Remove active transaction
  Registry.delTransaction(this, currentThread);
 }
}
```

If new roles are bound to an instance which is already engaging in a transaction, these new roles will temporarily be disregarded by the method dispatcher. There might be another transaction T_2 started after the first one T_1 and after new roles have been attached to a core object $o1$ which is already part of T_1. Roles attached after T_1 has started are taken into account for method dispatch in T_2 but not in T_1 as depicted in Figure 5.

Referring to the introductory example the following binding relations are stored in the look-up table (Table 2) and the list of active transactions is given in Table 3. The `transfer` object, in transaction `tx1`, uses its original behavior to send the data in a raw format as the transaction takes place before the `Encryption` role is bound. Transaction `tx2` is initialized after the `Encryption` role is bound, hence it is included in the method dispatch for invocations within `tx2`.

Table 2: A Modified Look-up Table

Id	BindingTime	Phantom	CoreId	RoleId	...
1	T+5	nil	transfer	Encryption	...

Table 3: A List of Active Transactions

Id	EnteringTime	Trans. Id	Thread
1	T	tx1	1
2	T+7	tx2	2

Additionally, roles can also be removed or updated (reloading a bound role type). Roles that are to be unbound during a transaction are marked for removal (*phantom*) in the look-up table without destroying their instances. In another transaction they are disregarded by the method dispatch to ensure consistency. Phantom roles will be destroyed once the transaction ends. Reloading roles does not affect consistency since the dynamic class loader can load multiple versions of a given type as explained in section 3.2.

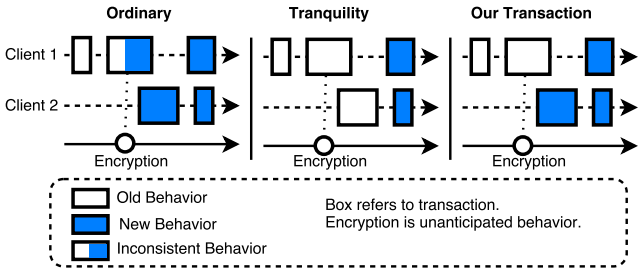

Figure 5: Comparison

Figure 5 shows a comparison of consistency and disruption. A system without transaction is prone to behavioral inconsistencies if updates occur in the middle of the transaction. Such updates are addressed by tranquility; however, in a context-dependent system there is no reason to prevent other transactions from applying new behavior. Our approach satisfies both consistent and non-disruptive behavioral updates.

5. Related Work

This work is an extension to the dynamic instance binding mechanism [15] for role-based software systems. In this paper we simplify the binding mechanism to suit generic objects in OOP, where some of objects are static or core, while others are dynamic (i.e., roles). Binding these two types of objects results in core objects having dynamic behavior.

COP enables the adaptation of systems' behavior with respect to their execution environment [13]. Existing COP languages relate to our approach in two ways: languages that deal with unanticipated adaptation and languages that assure adaptation's consistency.

In general, COP approaches do not account for unanticipated adaptation, in the sense that contexts and context-dependent behavior are defined beforehand. Notably, ContextJS [12] uses the flexibility of meta-programming to

support powerful layer activation mechanisms which could be extended to manage unanticipated adaptation. Similarly, Context Traits [7] provides a mechanism to discover and incorporates new contexts and their associated behavior at run time [1]. However, these mechanisms do not address inconsistencies that may arise from the introduction of unknown behavior.

Different types of run-time inconsistencies are addressed in COP languages. CoPNs manage the consistency between adaptations according to their defined context dependency relations [4]. Inconsistencies arising from the concurrent use of multiple activation scopes, i.e., global activation, asynchronous activation, can also be checked at run time. ServalCJ [9], and the work of Cardozo et al. [3, 5] manage the consistency between different context activation semantics. Nonetheless, the adaptable definitions must be given in advanced. Influenced by the concept of quiescence [10] and tranquility [16], our approach deals with state-behavior inconsistencies when composing adaptations with core objects. We use a transaction to ensure uniformly consistent behavior for every object executing inside a transaction. Moreover, we ensure there is no disruption to any other transaction after the adaptation.

6. Conclusion

We illustrate the issue of dealing with unanticipated adaptation where roles, adaptable entities, can be (re-)loaded arbitrarily from different threads causing object instances to change their behavior. Injecting new behavior can be achieved at run time by *dynamic instance binding*. This loosely couples core and role to grant the possibility of roles replacement at run time leading to dynamic adaptation. The inconsistencies caused by adaptation during long-lasting method executions are solved by the proposed transaction mechanism which is inspired by the concepts of quiescence and tranquility, and prevents engaged instances from being changed. This is achieved by extending the method dispatcher which performs invocations only for the behavior activated at the beginning of a transaction. Our solution enables both consistent and non-disruptive updates.

Next, we will evaluate the overhead caused by the dynamic instance binding and transaction mechanism. In addition, the investigation of different application scenarios that expose our approach to different situations in order to evaluate the limitations of the proposed consistency mechanism is planned.

Acknowledgments

This work is partially funded by Erasmus Mundus Program and the German Research Foundation (DFG) through the Research Training Group on Role-based Software Infrastructures for Continuous-Context-Sensitive Systems (RoSI) (GRK 1907).

References

[1] N. Cardozo and S. Clarke. Context slices: Lightweight discovery of behavioral adaptations. In *COP'15*, pages 2:1–2:6. ACM, July 2015.

[2] N. Cardozo, S. González, K. Mens, and T. D'Hondt. Safer context (de) activation: through the prompt-loyal strategy. In *COP'11*, page 2. ACM, 2011.

[3] N. Cardozo, S. González, K. Mens, and T. D'Hondt. Uniting global and local context behavior with context petri nets. In *COP'12*, page 3. ACM, 2012.

[4] N. Cardozo, J. Vallejos, S. González, K. Mens, and T. D'Hondt. Context petri nets: Enabling consistent composition of context-dependent behavior. In *PNSE'12*, volume 851, pages 156 – 170. CEUR, June 2012.

[5] N. Cardozo, L. Christophe, C. De Roover, and W. De Meuter. Run-time validation of behavioral adaptations. In *COP'14*, page 5. ACM, 2014.

[6] P. Ebraert, H. Schippers, T. Molderez, and D. Janssens. Safely updating running software. In *ECOOP'10 Workshop on Reflection, AOP and Meta-Data for Software Evolution*, 2010.

[7] S. González, K. Mens, M. Colacioiu, and W. Cazzola. Context traits: dynamic behaviour adaptation through run-time trait recomposition. In *AOSD'13*, pages 209–220. ACM, 2013.

[8] T. Kamina, T. Aotani, and H. Masuhara. Eventcj: a context-oriented programming language with declarative event-based context transition. In *AOSD'11*, pages 253–264. ACM, 2011.

[9] T. Kamina, T. Aotani, and H. Masuhara. Generalized layer activation mechanism through contexts and subscribers. In *Modularity'15*, pages 14–28. ACM, 2015.

[10] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[11] T. Kühn, M. Leuthäuser, S. Götz, C. Seidl, and U. Aßmann. A metamodel family for role-based modeling and programming languages. In *SLE'14*, pages 141–160. Springer, 2014.

[12] J. Lincke, M. Appeltauer, B. Steinert, and R. Hirschfeld. An open implementation for context-oriented layer composition in contextjs. *Science of Computer Programming*, 76, 2011.

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

[14] F. Steimann. On the representation of roles in object-oriented and conceptual modelling. *Data & Knowledge Engineering*, 35(1):83–106, 2000.

[15] N. Taing, T. Springer, N. Cardozo, and A. Schill. A dynamic instance binding mechanism supporting run-time variability of role-based software systems. In *Modularity Companion'16*, pages 137–142. ACM, 2016.

[16] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Transactions on Software Engineering*, 33(12):856–868, 2007.

[17] D. Weyns, M. Caporuscio, B. Vogel, and A. Kurti. Design for sustainability= runtime adaptation u evolution. *SAGRA'15*, 2015.