An Analysis of Language-Level Support for Self-Adaptive Software

GUIDO SALVANESCHI, Technische Universität Darmstadt
CARLO GHEZZI and MATTEO PRADELLA, Politecnico di Milano

Self-adaptive software has become increasingly important to address the new challenges of complex computing systems. To achieve adaptation, software must be designed and implemented by following suitable criteria, methods, and strategies. Past research has been mostly addressing adaptation by developing solutions at the software architecture level. This work, instead, focuses on finer-grain programming language-level solutions. We analyze three main linguistic approaches: metaprogramming, aspect-oriented programming, and context-oriented programming. The first two are general-purpose linguistic mechanisms, whereas the third is a specific and focused approach developed to support context-aware applications. This paradigm provides specialized language-level abstractions to implement dynamic adaptation and modularize behavioral variations in adaptive systems.

The article shows how the three approaches can support the implementation of adaptive systems and compares the pros and cons offered by each solution.

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

General Terms: Languages, Design

Additional Key Words and Phrases: Context, self-adaptive software, context-oriented programming, autonomic computing

ACM Reference Format:
Salvaneschi, G., Ghezzi, C., and Pradella, M. 2013. An analysis of language-level support for self-adaptive software. *ACM Trans. Auton. Adapt. Syst*. 8, 2, Article 7 (July 2013), 29 pages.
DOI:http://dx.doi.org/10.1145/2491465.2491466

1. INTRODUCTION

Over the last few years, runtime adaptation to changing conditions has become a common requirement for many software applications and for a wide spectrum of computing systems. At one end, mobile devices and sensor networks increased their computational power and became extremely common, introducing completely new dimensions for adaptation like energy consumption, connection availability, and spatial position. At the other end, datacenters increased in complexity to a level that demands for self-management.

Self-adaptive software [Salehie and Tahvildari 2009] and autonomic computing [Kephart 2005; Kephart and Chess 2003] offer promising approaches to deal with these issues. Research in these areas led to a specialized yet interdisciplinary

This research has been partially funded by the European Community's IDEAS-ERC Programme, Project 227977 (SMSCom).
Authors' addresses: G. Salvaneschi (corresponding author), Software Technology Group, Technische Universität Darmstadt, Hochschulstr. 10, 64289 Darmstadt, Germany; email: guido.salva@gmail.com; C. Ghezzi and M. Pradella, DEEPSE Group, DEIB, Politecnico di Milano, Piazza L. Da Vinci, 32 Milano, Italy.

DOI:http://dx.doi.org/10.1145/2491465.2491466

community [Oreizy et al. 2008], and the involved fields range from artificial intelligence to control theory and to software engineering. This research has been addressing both the theoretical foundations of adaptation and, more pragmatically, how adaptation techniques can be applied to solve the problems at hand. The autonomic *self-** properties [Huebscher and McCann 2008] are an example of the goals pursued by researchers in autonomic computing. To achieve these goals, solutions to engineer self-adaptive behaviors are often sought at the software architecture level, including middleware and component-based design. Accordingly, architectural approaches to dynamic adaptation have been extensively studied by researchers [Kramer and Magee 2007; Oreizy et al. 1998; White et al. 2004].

As a complementary approach, researchers also adopted specialized programming paradigms to implement adaptive systems, such as metaprogramming and Aspect-Oriented Programming (AOP). Recently, Context-Oriented Programming (COP) was proposed to provide ad hoc language-level abstractions for adaptive software [Hirschfeld et al. 2008]. This research was mostly driven by the programming language community and suggested that COP can support self-adaptive applications better than traditional paradigms [Kamina et al. 2011; Salvaneschi et al. 2012b]. Language-level solutions like COP, AOP, and metaprogramming bring a significant contribution. For example, changes are supported at fine-grain level, while architectures mostly work at the component-level granularity. They can concisely specify interception points, allowing transparent monitoring of existing applications. Finally, they provide general-purpose abstractions, improving cross-framework expertise and knowledge reuse.

Although language-level solutions have appeared in the literature, and often implemented as prototypes, with a few exceptions [Dowling et al. 2000; McKinley et al. 2004] little effort has been devoted to systematizing and comparing the alternative options. In summary, the current state of the affairs leaves many questions open: Which programming paradigm is suitable to implement adaptive systems? Which features are really required? Is it possible to adopt a unique paradigm providing all the needed abstractions?

In this article, we wish to pave the way for a discussion on these issues. First, we present language-level adaptation and we analyze its advantages over the other techniques. Then, we describe the state-of-the-art in language-level approaches to the implementation of self-adaptive systems. Finally, and more importantly, we compare the existing techniques—metaprogramming, AOP, and COP—and discuss their features along a number of significant directions. We show how COP's dedicated approach leads to certain improvements, and highlight the need for integration where other techniques are more effective.

This article is structured as follows. In Section 2 we describe the scope of this work. In Section 3 we present the three linguistic approaches analyzed in the article, which are then compared in Section 4, highlighting also some research challenges. Section 5 briefly illustrates some other related approaches. Finally, Section 6 presents a summary and some concluding remarks.

2. LANGUAGE-LEVEL ADAPTATION

Software adaptation can be achieved at different levels of abstraction. The survey on self-adaptive software by Salehie and Tahvildari [2009] refers to this issue as the *artifact and granularity* analysis direction. The identified alternatives are: parameters, method, aspect, component, application, architecture, system, and datacenter. Since we are interested in language-level adaptation, our analysis roughly lies at the method/aspect granularity level. According to the preceding classification, the lowest granularity level supports adaptations that can be expressed through parametrization. In this work we focus on more *complex* adaptations that concern behaviors, like alternative algorithms. This difference has been described by McKinley et al. [2004] as *parameter adaptation* versus *compositional adaptation*. For convenience, we adopt the terminology of the COP literature [Hirschfeld et al. 2008], and we refer to alternative complex behaviors as *behavioral variations*.

From a design perspective, complex behavioral changes are more challenging to deal with. Examples of behavioral modifications are the replacement of a method body, the insertion of a *before* advice to a method call, and the dynamic change of an object's class. Parametric change requires less support from a software design standpoint; the same computation simply executes with different input parameters. Parameter-based adaptation is out of the scope of this work. Noticeably, a lot of research is ongoing. Current research challenges include the design of an autonomic manager capable of the correct parameter selection [Hellerstein 2009; Sharifi et al. 2011] and runtime models to keep the strategy planning effective [Epifani et al. 2009].

Supporting behavioral variations at the language level is appealing with respect to dealing with adaptation at a higher architectural level. First, language-level approaches provide means to adapt at a very fine-grain detail. Second, many language concepts (e.g., polymorphism and late binding) are well known to programmers and can be easily specialized to support adaptation [Ghezzi et al. 2011]. In addition, programming languages can take advantage of tools like compilers and type checkers to enforce safety constraints. Third, introducing adaptation does not impose the burden of frameworks for software components or the adoption of ad hoc middleware. Finally, language-level approaches offer features, like method interception, or quantification, that in higher-level solutions are simply not available.

On the negative side, general-purpose language-level adaptation mechanisms (like AOP and metaprogramming) can lead to deep modifications in the code semantics and the final application can become cumbersome to understand. In addition, because they are general purpose, these mechanisms require to be tailored to the specific setting of self-adaptive systems. This process is usually driven on a per-application basis: many solutions are specific for a single software, so hardly adaptable to other systems. In conclusion, limiting complexity and providing standard abstractions to promote expertise, as COP has been doing, is an important improvement.

Since these requirements must be evaluated in the context of adaptive applications, we refer to the MAPE-K loop model (Figure 1), commonly accepted in the autonomic computing community [Kephart and Chess 2003]. An *autonomic element* is composed by an *autonomic manager* and a *managed element*. The autonomic manager controls the managed element and is responsible for the autonomic behavior. The most important aspects of the autonomic manager are summarized in the MAPE-K acronym: *monitoring*, *analyzing*, *planning*, *execution*, and global *knowledge*. The autonomic manager collects information about the managed element through *sensors* and modifies its behavior through *effectors*. In this article, we assume that the managed element is a software artifact. The autonomic manager affects the computation of the managed element by triggering the activation of the behavioral variations. Then behavioral variations combine to produce the overall behavior of the autonomic element. Effectors are implemented through the activation of variations. Sensors are probes in the managed element. Indeed, sensing includes reading the value of relevant variables in the application, but also inspecting its structure, or querying a class to obtain the list of its methods or determining the class hierarchy. In this work, we are especially interested in the support that a language or a paradigm can provide for actuators (activation of variations) and for sensors (inspection of the managed application). All the internal activities of the manager element and how they are implemented, for example,

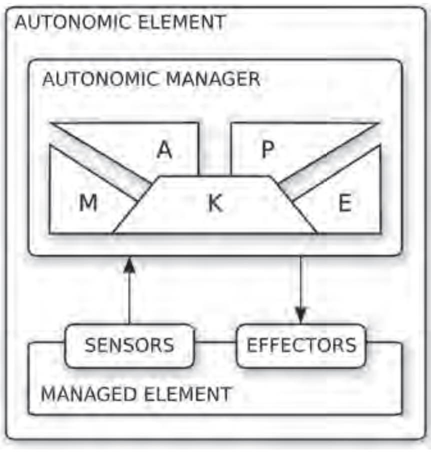

Fig. 1. The reference model of a self-adaptive system with language-level adaptation.

Event-Condition-Action (ECA) rules are out of the scope of this article. The interested reader can refer to Huebscher and McCann [2008].

3. PROGRAMMING MODELS

In this section, we introduce the three main language approaches that are assessed in this article: metaprogramming, AOP, and COP. For each technique, we describe the main features it offers and how it can improve the design of self-adaptive systems.

3.1. Metaprogramming

Computational reflection [Maes 1987; Smith 1984] is the ability of a program to reason about itself, by observing the ongoing computation and possibly modifying its behavior. Metaprogramming refers to the use of computational reflection, that is, programming at the meta level. A different use of the term metaprogramming refers generically to programs capable of processing other programs. With this meaning, metaprogramming includes, for example, compilers. However, in this article we are interested in the abstractions available to the developer, so we consider metaprogramming only in the sense of using a reflective interface on the program. Like traditional programming abstracts over low-level activities, metaprogramming must access the required information and alter the program functionalities hiding the unnecessary details. A Meta-Object Protocol (MOP) provides the abstractions for metaprogramming, that is, it is the meta-interface used to access the meta-level computation [Kiczales and Rivieres 1991]. Since the meta level is *causally connected* to the base level, a change in the meta level reflects in a behavioral change of the base level. In this article, we are interested in the way metaprogramming can provide abstractions to support monitoring, and dynamic activation of behavioral variations.

It must be noted that computational reflection is an extremely general mechanism. It addresses the most diverse needs such as runtime class definition, reverse engineering of private interfaces, and separate compilation [Lamm 2001]. For this reason, when reflection is used in the scope of dynamic adaptation, it must be tailored to this purpose. So, developers specialize the use of reflective abstractions to enable inspection and runtime change.

In languages which natively offer a metaprogramming interface, adaptation can be directly built on top of the reflective services. Conversely, when a reflective API is not available, or when it is not powerful enough, the designers of adaptive systems extend

```java
// Old API:
public class Channel {
  public boolean send (Message m) { ... }
  public Message receive() { ... }
}
public class Message { ... } // Data str.

// New API:
public class NewChannel {
  public boolean write(Block b) { ... }
  public Block read() { ... }
}
public class Block { ... } // Data str.


// Protocol:
package com.foo.adapters;
protocol RedirectChannel {
   local:
     reify Invocation: RedirectExecute;
}

// Dynamic association:
Meta.associate("com.acme.comms.Channel",
   "com.foo.adapters.RedirectChannel");

```

```java
// Invocation metaobject:
public class RedirectExecute
                       extends MExecute {
  private NewChannel nc;

  public Object execute(Object o,
                        Object[] args,
                        Method m)
       throws InvocationTargetException {
    if(nc == null)
      nc = new NewChannel( ... );
    String mname = m.getName();
    if(mname.equals("send")) {
      Message msg = (Message)args[0];
       ... // Convert msg to new Block
      return (nc.write(blk));
    }

    else if (mname.equals("receive")) {
      Block blk = nc.read();
     ... // Convert blk to new Message
      return (msg);
    }
    else return(proceed(o,args,m));
  }
}
```

Fig. 2. An example of the Iguana/J language.

the language with the required features [Hsieh et al. 1996]. Alternatively, they resort to general-purpose extensions which introduce the reflective support. For example, Xu and Zorzo implemented an adaptable fault-tolerant system [Xu et al. 1996] using the Open C++ extension of C++ [Chiba 1995]. Ledoux [1997] proposed an adaptable ORB object based on Neoclasstalk [Rivard 1996], a reflective kernel for Smalltalk.

Noticeably, the term *reflection* has been used to refer not only to language abstractions, but also to a class of systems capable of self-inspection and self-modification. For example, reflective middleware borrows the idea of reflection from programming languages: they reify the middleware behavior and allow self-analysis and dynamic change. Often, these frameworks provide ad hoc means to implement reflective functionalities, and do not necessarily rely on the meta-object protocol of the underlying language. In this case, the meta level controls the behavior of entities which do not necessarily belong to the language runtime or to the virtual machine. Instead, they are at a higher level of abstraction and are specialized for the application domain. Some approaches discussed in the following sections, like Open ORB [Blair et al. 2001] and CARISMA [Capra et al. 2003], belong to this category. From a practical standpoint, the resulting solutions are therefore more similar to APIs in traditional programming: they provide access to the middleware internal structures and operational behavior. Conceptually, however, the user can access the system at a higher level of abstraction, since all the relevant entities are modeled via meta-objects.

Example. In Figure 2, we show an example of the Iguana/J language [Redmond and Cahill 2006]. Iguana/J is a reflective extension to Java which supports dynamic modification of running applications. The example, taken from Redmond and Cahill [2006], shows the implementation of an adaptive communication system that dynamically switches between `Message` data structures and `Block` data structures to hold the exchanged information. Adaptation is achieved by redirecting the method calls to `Channel` objects to `NewChannel` objects. Iguana/J allows reification of a number of language operations like object creation, field reading, and method dispatching. In Iguana/J, a coherent set of meta-object classes representing a new behavior is called a `protocol`. In the example, the `RedirectChannel` protocol (line 18) reifies method invocation (referenced by the `Invocation` identifier) into the `RedirectExecute` class. The `local` keyword regulates class associations and creates an instance of the `RedirectExecute` class for each instance of the `Channel` class. The `Meta.associate` statement dynamically applies a protocol to a given class. The actual behavioral modification is implemented in the class `RedirectExecute`. The `RedirectExecute` class extends the `MExecute` (Meta-Execute) class and redefines the `execute` method which is responsible for altering method executions. The `send` and `receive` methods are intercepted (lines 39 and 45) and redirected to a `NewChannel` class which uses a `Block` instance to hold the information exchanged in the communication.

Applications. Reflection and MOPs have been actively used to support dynamic adaptation in various fields. Most of the approaches belong to the research areas of operating systems, distributed systems, and mobile and ubiquitous computing.

With some limitations, many production operating systems present forms of dynamic adaptation. For example, it is possible to supply parameters at boot time. More interestingly, at runtime, the administrator can install and remove kernel modules [Denys et al. 2002]. Usually this approach adopts "hooks" in the system, so the changes are constrained to predefined places. Gowing and Cahill [1995] proposed the use of MOPs to support dynamic adaptation and extension of the operating system. Their approach, namely *extension protocols*, supports nonpredetermined change but still preserves system security and integrity. Madany et al. [1992] discuss metaprogramming extensions to C++ in the scope of operating systems. For example, they advocate the need for runtime definition of inheritance relationships to support late time specialization. In the Apertos object-oriented operating system [Itoh et al. 1995; Yokote 1992] an object is associated to a group of meta-objects (a meta space), which determine its semantics. An object can change the meta-objects to modify its behavior. This solution introduces a high degree of flexibility: for example, it is possible to select the proper network protocol to deliver a message, to dynamically assign resources and enforce real-time constraints, or to manage the memory at object granularity.

In distributed systems, the need for flexibility and runtime adaptation raises naturally due to the changing quality of the network communication and to the dynamic reconfiguration of services and hosts. In this context, it has been proposed to augment existing frameworks—like Java RMI and CORBA—with a reflective interface. The purpose is to access the adaptation capabilities of the system and dynamically adjust policies and mechanisms for distribution [Eliassen et al. 1999]. *Reflective middleware* [Kon et al. 2002] has been proposed as a means to dynamically adapt to the environment through a metaprogramming interface. For example, reflection is used to modify proxy objects and force them to operate locally or remotely [Ledoux 1997]. Open ORB [Blair et al. 2001] is a middleware platform developed at Lancaster University. Open ORB provides several reflective interfaces to access different aspects of dynamic adaptation. These multiple metamodels include *interceptors*, to introduce monitoring, and *resources* to adapt the resource usage and the management policies. Dynamic TAO [Roman et al. 1999] is a CORBA reflective system which supports various forms of dynamic reconfigurations, including performance optimization, presence of new hardware and software components, and error recovery.

Specific middleware solutions have been developed to support mobile and ubiquitous computing. Applications in these new areas introduce new technical challenges since the need to respond to changes in the environment often requires a dynamic reconfiguration of the system. CARISMA [Capra et al. 2003] is an adaptable middleware that exploits reflection to enhance the development of context-aware mobile applications. MobiPADS [Chan and Chuang 2003] is an applicative layer for context-aware mobile computing. In MobiPADS, adaptation policies are available to the programmer through meta-objects, which also provide access to subscribe to contextual events. ReMMoC [Grace et al. 2003] is a reflective platform to support interoperability among heterogeneous services. Dynamic adaptations include the binding of new services and the discovery protocol. In addition, a meta-level API allows introspection of the platform structure.

Behavioral Change. Metaprogramming facilities can be classified as *inspection* and *modification*. Modification refers to semantic change; inspection deals with the observation of a program execution.

Self-adaptive systems based on metaprogramming rely on modification to operate dynamic changes. Possible modifications include redefining method bodies, intercepting method calls, modifying the dispatching algorithm, and augmenting an object with new methods or fields. Ledoux [1997] adopts dynamic class change to modify the behavior of objects. Remote and local method invocations are managed transparently by switching between classes which implement alternative method lookup protocols. Dowling et al. [2000] leverage reflection to change the binding between method name and method implementation. Hsieh et al. [1996] use reflection to enable dynamic linking of modules. Xu et al. [1996] intercept method executions to inject the code to implement fault tolerance. In addition, this mechanism enables dynamic adaptation of the fault-tolerance scheme. In a previous paper we have shown how dynamic class change can be used to support runtime adaptation in a caching system [Ghezzi et al. 2011].

Monitoring. Inspection is the way metaprogramming can be used to implement *sensors* and monitor the execution of a self-adaptive system. Observing the current state of the application is a basic functionality provided even by the most limiting metaprogramming models. For example, in Java, the programmer can query classes to know their attributes and inspect the associated value.

Monitoring provides means to access the running application and collect information about current values. For example, the metaprogramming support of Java allows one to inspect an object by accessing even private fields. Dawson et al. [2008] propose to use Java dynamic proxies to monitor self-adaptive applications by intercepting method calls. Dynamic proxies are part of the reflection facilities of Java because they can provide an interface which is dynamically selected. In this way, it is possible to proxy objects only known at runtime. Interestingly, via proxy chaining, this mechanism also supports multiple-monitors composition.

Beside monitoring, inspection can support the analysis of the *structure* of the program environment. Structural inspection addresses the need for collecting information about the current design of the system. This includes which modules are loaded, how the class hierarchy is structured, and which functions are implemented. Hsieh et al. [1996] modified the Modula-3 language to support the inspection of the relationships among code structures, such as the interfaces implemented by a module.

Variations and Separation. Reflection has been used to separate different concerns, especially before the introduction of AOP. Dowling et al. [2000] compare different language-level techniques to support software dynamic adaptation: reflection, Dynamically Linked Libraries (DLL), and design patterns. According to their study, computational reflection offers significant advantages in separating functional code from adaptation code.

Metaprogramming supports separation of concerns thanks to the distinction between the meta layer and the base layer. Since specific functionalities can be implemented in the meta layer, *what* an object does is separated from *how* it behaves. For example, Stroud and Wu [1996] implement a library of atomic data types which enforces separation of concerns using MOPs. The application code is implemented at the base level, and the synchronization/recovery code is at the meta level. Similarly, Xu et al. [1996] carried out an experimental study on control structures for fault tolerance. They compared a pure C++ implementation and the use of an external reflective library. The version based on metaprogramming can keep the fault-tolerance code better separated by implementing it in the meta layer. In this way, fault tolerance can be introduced without adding complexity to the base code. Additionally, different fault-tolerance schemes can be selected dynamically and applied transparently to the existing application.

Another technique to enforce separation of concerns via metaprogramming is based on *multiple meta-object protocols*. In this case, separation of concerns is supported through the implementation of several distinct metaspace models. Each meta space accounts for different aspects (e.g., fault tolerance, concurrency, and persistence), a technique initially introduced in Okamuray et al. [1992]. For example, the Open ORB 2 middleware [Blair et al. 2001] exposes an *interception* metamodel to insert pre- and postbehavior at the interfaces. A distinct metaspace, the *resources* metamodel, instead, provides access to the configuration of the resource management.

3.2. Aspect-Oriented Programming (AOP)

Aspect-oriented programming was proposed to handle separation of concerns [Kiczales et al. 1997; Tarr et al. 1999]. With AOP, the functionalities that are orthogonal to the main modularization direction, such as logging, persistence, synchronization, and failure handling, can be kept separate, improving modularity and maintainability. After separate development, the concerns are composed to achieve the complete functionality. AOP allows one to specify points in the program execution (*joinpoints*) in which the control is transferred to the code implementing the separate concern (*advice*). AOP languages support the specification of joinpoints through convenient expressions (*pointcuts*) which quantify over joinpoints sets. An *aspect weaver* merges the advice and the base application. Over the years, cross-cutting concerns have become an important issue of software design. Industrial-strength tools have been developed to support this principle, such as JBoss AOP[1], Spring AOP[2], and the AspectJ framework[3]. For a comprehensive survey of the existing AOP techniques, the interested reader can refer to Brichau and Haupt [2005].

Example. In Figure 3, we provide an example of the JAsCo language [Suvée et al. 2003]. JAsCo is an AOP extension of Java which combines ideas from aspect orientation and component-based software development. In JAsCo an *aspect bean* captures a cross-cutting behavior, and *connectors* bind aspects to the base code. This design decouples orthogonal features from their context and encourages component reuse. The example, taken from Vanderperren et al. [2005], implements an incremental backup that can be dynamically activated on selected resources. Line 1 defines an aspect bean, which declares a `Backup` hook in the base code. In this case, the execution of the `triggeringmethod` method is hooked (line 7). To keep aspect beans generic with respect to their execution context, at this stage, the `triggeringmethod` method is abstract and it is still not bound to any concrete implementation. In the `isApplicable` clause, aspect beans can declare a runtime condition that enables the injection of the aspect behavior. Finally, the `before` clause defines the injected behavior.

[1]http://www.jboss.org/jbossaop
[2]http://www.springsource.org
[3]http://www.eclipse.org/aspectj

```
 1 class DataStorePersistence {
 2 
 3   hook Backup {
 4     int i = 0;
 5 
 6     Backup(triggeringmethod(..args)) {
 7       execute(triggeringmethod);
 8     }
 9     isApplicable() {
10       //true when changed since last visit
11     }
12     before() {
13       FileOutputStream fw =
14         new FileOutputStream("state"+i++);
15       ObjectOutputStream writer =
16         new ObjectOutputStream(fw);
17       writer.writeObject(getDataMethod());
18       writer.close();
19     }
20     refinable Object getDataMethod();
21   }
22 }
```

```
23 refining DataStorePersistence.Backup for
24   DataStore {
25     public Object getDataMethod() {
26       DataStore store =
27           thisJoinPointObject;
28       return store.getData();
29   }
30 }
31 
32 
33 connector PersistenceConnector {
34   DataStorePersistence.Backup hook =
35       new DataStorePersistence.Backup(
36         * DataStore.set*(*));
37 }
38 
39 
40 
41 
42 
43 
44 
```

Fig. 3. Persistence in the JAsCo language.

When the aspect refers to the base code, it relies on abstract *refinables* like the `getDataMethod` method (line 17). Refinables are bound to concrete methods in *refinements*: Line 23 shows a refinement of the `DataStorePersistence` aspect for the `DataStore` class. The refinement provides a concrete implementation of the `getDataMethod` method (line 23). Refinements are late-bound, that is, at runtime, the most specific refinement for the object is executed. Line 33 defines a connector which deploys the `DataStorePersistence` aspect. The `triggeringmethod` parameter defined in the `DataStorePersistence` aspect bean is bound to each setter method of the `DataStore` class, that is, each occurrence of the regular expression in line 33. As a result, before the execution of a setter, the control transfers to the body of the `before` clause (line 12).

Applications. Several contributions dating back to the early days of aspects are motivated by supporting software dynamic adaptation. Among the others, the issue of dynamic adaptability can be found in the papers presenting PROSE [Popovici et al. 2002, 2003], JAC [Pawlak et al. 2001, 2004], and AspectWerkz [Boner 2004; Vasseur 2004]. Not surprisingly, many researchers experimented with AOP to implement self-adaptive systems. Yang et al. [2002] used AspectJ: in their approach, the program is first prepared to adaptation by instrumenting it with convenient interception points. At runtime, an adaptation kernel based on ECA rules intercepts the execution and triggers the adaptive behavior. Greenwood and Blair introduced dynamic AOP in autonomic computing [Greenwood and Blair 2003]. The motivation for their choice is threefold. First, they observed that many concerns that ask for adaptation are also cross-cutting. Second, they advocated the benefits of encapsulating the adaptations that are required in an autonomic system. Third, they wished to use the support given by dynamic AOP for aspect application and removal. A subsequent work [Greenwood and Blair 2006] introduces a distinction between *monitoring* aspects and *effector* aspects. The former are in charge of inspecting the application, the latter are behavioral modifications. TOSKANA [Engel and Freisleben 2005] is a toolkit for deploying dynamic aspects into an operating system kernel. TOSKANA was used to modify NetBSD to support self-configuration, self-healing, self-optimization, and self-protection properties. JEARS [Bachara et al. 2010] is a framework for autonomic Web applications: sensor and effectors are implemented as aspects and can be deployed and removed dynamically through a user interface. Besides traditional applications, AOP has been used to introduce adaptability in *Service-Oriented Applications* (SOAs) in general, and *Web Services* (WSs) in particular. Cibrán et al. actively worked on AOP in adaptive systems (e.g., Cibrán et al. [2007]). They showed how aspects can address several issues in WS adaptability, including monitoring, policy selection, and WS composition. Other researchers combined AOP with the WS orchestration language BPEL to obtain automatic synthesis of adaptable WSs [Charfi and Mezini 2004; Courbis and Finkelstein 2005].

Behavioral Change. To enable modularization of cross-cutting concerns, AOP provides means to intercept the execution flow in the base program and redirect it to an advice. In self-adaptive systems, this feature supports dynamic behavioral change and inspection of the running system. Behavioral change must be triggered when certain events in the execution of the program occur. Similarly, inspection requires that only certain points of interest are observed. Through pointcuts, AOP allows one to explicitly refer to and quantify over those execution points. Pointcuts can be either static or dynamic.

Static pointcuts are events in the program execution that are determinable during compilation. For example, in AspectJ, the `execution` pointcut transfers control to an advice every time a method matching a given expression is called. Static pointcuts do not support runtime adaptation directly. However, they allow intercepting the execution flow and introduce a layer of indirection: after the interception, other mechanisms can be used to dynamically select the context-dependent behavior. For example, in TRAP/J [Sadjadi et al. 2004] AOP is used to intercept method calls, and behavioral change is triggered by metaprogramming.

Dynamic pointcuts, instead, designate an execution point that cannot be decided at compile time. For example, in AspectJ, the `if` and the `cflow` pointcuts belong to this category [Laddad 2009]. The `if` pointcut activates an advice when a Boolean condition is satisfied; the `cflow` pointcut performs the activation along the current control flow. In adaptive systems, dynamic pointcuts are fundamental to defer to runtime the adoption of a certain monitoring scope or to trigger a behavioral variation. The expressive power of the pointcut language is important to control the adaptation effectively. In the case of `cflow`, for example, it is possible to achieve nonlocal adaptation, since the change propagates along all the execution flow into nested method calls. Dynamic pointcuts are a fundamental mechanism used by dynamic AOP, and advanced aspect languages like CaesarJ provide even more elaborate activation strategies [Aracic et al. 2006].

Weaving is the process of binding advices with the rest of the code. Starting from the first compile-time weavers [Kiczales et al. 2001], several solutions exploiting other biding times have been proposed, arguing that to meet all the possible requirements, the whole spectrum is required [Bollert 1999]. Whereas weaving is not strictly related to dynamic capabilities, as we shortly discuss, it has important implications in the development process of an adaptive system.

In compile-time weaving (sometimes referred to as *static* weaving) aspects are merged with the codebase during the compilation process. For example, the AspectJ `ajc` static weaver postprocesses the bytecode from the Java compiler and introduces the hooks for the advices. Other compile-time weavers directly operate on the source code.

Load-time weaving enhances the code when the class is loaded in the virtual machine. AspectJ also supports this kind of weaving: via the `javaagent` option of the Java VM, which specifies a preprocessor for the classes to be loaded. The introduction of hooks for the advices is often performed by using bytecode manipulation libraries like

ASM [Bruneton et al. 2002] and BCEL [Dahm and Berlin 1998]. Load-time weaving addresses some adaptation scenarios that cannot be managed with compile-time weaving. For example, with load-time weaving, it is not necessary to distribute a woven version of an existing library. Instead, the library can be adapted just before loading. Another application of load-time weaving are execution models that include code generation. For example, JSPs are compiled on-the-fly at the first request, so compile-time weaving is simply not possible. These examples show that load-time weaving may overcome some technical limitations of static weaving. Conversely, load-time weaving is difficult to exploit as a mechanism for the activation of behavioral variations. In this perspective, load-time weaving can be considered as a deferred form of static weaving.

In runtime weaving, advices are woven during the execution of the application, without the need for recompilation or rebooting. Runtime weaving relies on virtual machine support, like breakpoints, to intercept joinpoint events [Popovici et al. 2002]. JIT compilers can be used to insert advice hooks [Popovici et al. 2003; Sato et al. 2003], and hot swap allows replacing the running code [Boner 2004; Vasseur 2004]. To achieve better performance, VM modifications have been proposed to specifically support this feature [Bockisch et al. 2006a; Nicoara et al. 2008]. The advantage of runtime weaving is to perform optimizations based on information that is known only at runtime. For example, if the aspects are not known in advance, with static weaving and load-time weaving, all the possible joinpoints must be instrumented to intercept the control flow and execute the advice. Depending on the aspect, only some hooks must be really activated and runtime checks can impose a high performance overhead. Instead, runtime weaving can insert hooks only where they are needed and avoid unnecessary checks [Sato et al. 2003]. For these reasons, runtime weaving is an important technology to efficiently support late binding of variations in self-adaptive systems.

Dynamic AOP refers to activating, configuring, and removing aspects dynamically. With dynamic AOP, programmers can plug and unplug aspects during the execution of the program. Among the others, AspectWerkz [Boner 2004; Vasseur 2004] JAC [Pawlak et al. 2001, 2004], JBoss AOP, CaesarJ [Aracic et al. 2006], PROSE [Popovici et al. 2002], and JAsCo [Suvée et al. 2003] support dynamic AOP in various forms.

Dynamic AOP is especially important for self-adaptive systems since it natively supports dynamic modification of behavior when aspects are dynamically activated. A typical example is caching. Caching is a cross-cutting concern, since cache access and cache invalidation are scattered across the code. So caching is correctly modeled by AOP. Since there is no optimal strategy for all configurations and all applications, dynamic AOP is needed to switch to a better caching strategy when the hit rate is too low [Ségura-Devillechaise et al. 2003]. Dynamic AOP also supports runtime configuration of the activation scope. In CaesarJ, the developer uses different language primitives to activate an aspect on different portions of the application. For example, per-instance, per-class, per-thread, application-wide, and VM-wide activations are possible. Dynamic configuration of the activation scope is important in adaptive systems to enable behavioral variations at the desired granularity. For example, in self-healing systems, monitoring should be activated only on the relevant components, since global monitoring can introduce unacceptable overhead.

The relation between weaving strategy and dynamic AOP deserves some observations. Ignoring optimization, dynamic support for AOP and weaving strategy are, in principle, orthogonal. An AOP implementation can hook all the possible joinpoints statically and postpone at runtime their activation depending on the configuration of the active advices. Not surprisingly, many dynamic AOP systems require preruntime class preparation [Aracic et al. 2006; Boner 2004; Pawlak et al. 2001; Popovici et al. 2002].

Monitoring. With AOP, it is possible to modify the execution of an application by inserting additional or alternative behavior. Monitoring is a simple application of this feature: the injected behavior simply detects information about the running program. Since monitoring and logging are typically cross-cutting aspects, AOP is an extremely effective way to accomplish this task. Remarkably, aspect languages include proper abstractions to transfer data from the intercepted point to the advice. A common example is accessing the parameters of a method call to perform a security check [Win et al. 2001]. Another advantage of AOP is that quantification allows one to concisely activate monitoring on several execution points. These features make AOP convenient to implement monitoring in self-adaptive systems.

Greenwood and Blair [2006] use AOP to monitor the execution time of getter methods. If the execution time exceeds a threshold, an ECA rule is triggered and caching is activated. When the requirements of an application change dynamically, monitoring must be part of the adaptable functionalities. In this case, dynamic AOP is an ideal solution. For example, Cibrán et al. [2007] use dynamic AOP to monitor a pool of WSs and select those compliant with certain policies. Since policies can be added at runtime, new monitoring features must be introduced dynamically. Janik and Zielinski [2010a] present a reconfigurable monitoring system based on AOP. Monitoring can be enabled dynamically through aspect activation and deactivation. This also limits the number of resources that are monitored simultaneously.

Variations and Separation. In many systems, the set of modifications that coherently determine a behavioral adaptation cross-cut the application structure. For example, in a mobile device, adapting to a context with low bandwidth availability can require several countermeasures. First, the network management system must be modified to increase the number of times a transmission is tried again in case of failure. Additionally, the data model must be adapted to use local information instead of a remote service (e.g., a map from a database instead of a dynamically updated one). Finally, the interface must be modified to inform the user. So, the adaptation to the low-bandwidth context impacts on different functionalities of the application.

Several researchers observed the cross-cutting nature of dynamic adaptations. Orthogonal concerns include network availability [Kamina et al. 2011], user activity [Appeltauer et al. 2010b], device location [Popovici et al. 2003], and access control policies [Popovici et al. 2002]. Proper modularization of cross-cutting concerns is the fundamental motivation behind AOP, so adaptive applications can clearly benefit from AOP design. Remarkably, McKinley et al. [2004] in their analysis on self-adaptive systems take into consideration separation of concerns and identify AOP as an enabling technology to design adaptive software.

3.3. Context-Oriented Programming (COP)

COP was recently introduced to provide ad hoc language-level abstractions for context-aware software. Without proper programming support, context-aware applications are cumbersome to design. In the absence of specific COP constructs, context adaptation would be achieved by spreading across the application conditional statements that trigger dynamic changes. The basic program logic is thus tangled with the possible adaptations, and this leads to applications that are hard to understand and maintain. Another issue is that adaptations rarely occur individually. More often, multiple context conditions coexist at the same time and adaptations must be combined accordingly. In traditional languages, this combination must be performed by dedicated code. COP, instead, addresses these issues by providing abstractions that modularize otherwise scattered software adaptations. Additionally, COP directly supports variations activation and runtime combination.

COP defines *context* as *any computationally accessible information* [Hirschfeld et al. 2008]. So, the programmer can represent any dynamic variability along the direction of context. This pragmatic approach makes COP ideal for self-adaptive software, since the adaptations can be easily modeled as contexts and triggered in response to detected changes.

COP extensions have been proposed for several languages, including Java, Python, Ruby, JavaScript, Common Lisp, and Scheme [Appeltauer et al. 2009]. Since the new abstractions need to fit into the underlying programming model, the concrete solutions are not completely homogeneous. In addition, researchers have investigated amendments to the original COP model to better address specific design issues. This adds further variability to the available COP solutions. Despite this variety, some concepts are essential and are supported in all implementations. The support of *abstraction for behavioral variations* is the distinguishing feature of COP: variations are represented in the language by dedicated abstractions. These abstractions are usually first class, so they can be referenced, assigned to parameters, and returned by functions. Additionally, variations provide means to modify the application in different places, potentially scattered across the code. This feature is fundamental to support cross-cutting adaptations. *Activation* distinguishes many COP approaches from traditional AOP techniques: variations can be dynamically enabled and disabled. When a variation is activated, the behavior it models starts modifying the application. Variation deactivation returns to the original application behavior. Finally, *behavior combination* supports the reaction to simultaneous contextual conditions. If multiple variations are active at the same time, the software behavior is given by the combination of all the active variations. According to the cross-cutting nature of certain variations, the combination occurs at each execution point.

COP languages provide alternative activation mechanisms depending on the scope on which behavioral variations are enabled. From a practical standpoint, the programmer has the choice of extending the activation on different portions of the application. Indeed, this aspect also has deeper consequences: conceptually, different activation mechanisms coincide with different *context models*.

In most COP languages, including ContextJ [Hirschfeld et al. 2008], which is discussed shortly, context is associated to control flow, so threads can live in different contexts. In this case, the behavioral variations propagate their effect along the dynamic extent of the activation block. In self-adaptive systems this model is useful when context is per thread and it is fetched before a long sequence of operations: an adaptation is planned and remains the same for the whole subsequent computation of that thread.

A different approach is adopted in the Ambience language [González et al. 2007], where context is shared across all the application. So, if a thread activates a variation, the change is seen by all the other threads. This solution has the advantage that the communication between the autonomic manager and the managed elements is simplified: the thread implementing the autonomic manager can directly trigger a variation activation on the whole program. Yet, this model requires care, since the activation is completely asynchronous and it is easy to activate conflicting or inconsistent variations.

ContextErlang [Salvaneschi et al. 2012b], a COP extension of Erlang[4], explores another alternative in the design space. The Erlang concurrency model is based on agents which exchange messages and have no shared memory [Hewitt et al. 1973]. ContextErlang improves Erlang agents to account also for context awareness. In ContextErlang, variations are activated on context-aware agents which encapsulate

[4]http://erlang.org

```
 1 public class Account{
 2   private int accountNumber;
 3   private float balance;
 4   ...
 5   public void credit(float amount){
 6     balance = balance + amount ;
 7   }
 8   public void debit(float amount){
 9     balance = balance - amount ;
10   }
11   layer EncryptionLayer{
12     public void credit(int am){
13       proceed(RSA.decrypt(am));
14     }
15     public void debit(int am){
16       proceed(RSA.decrypt(am));
17     }
18   }
19   layer LoggingLayer{
20     after public void credit(int am){
21       Logger.logCredit(this, am);
22     }
23     after public void debit(int am){
24       Logger.logDebit(this, am);
25     }
26   }
27 }
28
```

```
29 public class TransferSystem{
30   public void transfer(Account from,
31                        Account to,
32                        float amount){
33     from.debit(amount);
34     to.credit(amount);
35   }
36  ...
37   layer EncryptionLayer{
38     public void transfer(Account from,
39                          Account to,
40                          int amount){
41       without(EncryptionLayer){
42         proceed(from, to,
43           RSA.encrypt(amount));
44   }} }
45   layer LoggingLayer{
46    after public void transfer(Account from,
47                               Account to,
48                               int amount){
49      Logger.logTransfer(from, to, amount);
50 } }}
51 ------------------------------------------
52 public void transfer100(Account from,
53                         Account to){ ...
54   with(LoggingLayer, EncryptionLayer){
55     transferSystem.transfer(from, to, 100);
56 } }
```

Fig. 4. An example of the ContextJ language.

the current context along computation and state. This solution is especially effective to model self-adaptive systems in which the context is associated to different entities in the application. For example, clients using different communication protocols can be modeled as context-aware agents and can be individually adapted.

Although most COP languages enforce a single activation model, an interesting perspective is to open the activation model and leave to the programmer the freedom of designing the activation mechanism that better fits her needs. ContextJS, proposed by Lincke et al. [2011], is an attempt in this direction. ContextJS is an open implementation which provides an interface to customize the activation strategy. For example, global, dynamically scoped, and even per-object activation can be easily implemented. Beside those known solutions, the programmer can implement her own custom activation models.

Example. To be more concrete, in Figure 4, we present a COP example taken from Appeltauer et al. [2011]. The example is written in the ContextJ language, a COP extension to Java. It is taken from Appeltauer et al. [2011] and implements an adaptable bank account system. Most COP languages represent behavioral variations via *layers*. In ContextJ, layers are defined by the `layer` keyword. Layers contain a sequence of methods which implement an alternative behavior of the application. In the example, the `EncriptionLayer` layer and the `LoggingLayer` layer define an alternative behavior of the `Account` class and of the `TransferSystem` class.

The `Account` class defines a variation of the `credit` method which encrypts the credit value (line 12) and a variation which logs the operation (line 20). If no layer is active, the standard definition of `credit` is executed. The `with` statement dynamically activates a layer in the scoped block. In case of multiple activations, they are executed in reverse order, starting from the innermost activated layer and proceeding towards the outermost layers. Variants of the `with` statement accept a collection of layers which can be unknown at compile time. A `without` statement can be used to temporarily disable layers. Layers combine through `proceed`, which calls the method in the next active layer or the basic method if no active layers are left. The `proceed` method call is similar to its homonymous call in AOP and to `super` in many OO languages. The effect of the `with` statement propagates along the control flow. So it is possible to adapt not only the local execution, but also to propagate changes over the dynamic extent. Since layers are first-class entities, they can be stored in variables and activated later. Other forms of combinations are also possible. Besides `around` methods, which are executed in place of the original definition, layers can provide `before` and `after` methods.

The example presented here only accounts for the basic functionalities provided by COP. In the following sections, we describe the most important variants of this model. The interested reader can find a comprehensive overview in Appeltauer et al. [2009] and Salvaneschi et al. [2012a].

Applications. Since COP is a recent technique, the number of existing applications is limited. However, several scenarios for self-adaptive software have been already explored and COP abstractions proved to be an effective solution.

In the area of desktop applications, Appeltauer et al. [2010b] proposed CJEdit, an adaptable development environment. This program dynamically switches from a code-editing mode to a documenting mode depending on the user activity. Lincke et al. [2011] developed the LivelyKernel application. In LivelyKernel, the user can draw new shapes inside a virtual desktop and figures in the workbench adapt their appearance depending on the surrounding entities.

Mobile applications are a classic scenario for adaptive software. González et al. [2007] developed the CityMaps mobile application. CityMaps can modify its behavior depending on environmental conditions. For example, it displays a static map when no connection is available; when the GPS sensor is active, instead, the map is dynamically updated with the user position. Kamina et al. [2011] present the pedestrian navigation system, an adaptable android application capable of switching at runtime between the WiFi and the GPS to provide a better user experience when more information is available.

In the field of server-side software, we implemented a chat server which adapts to the user state [Salvaneschi et al. 2012b]. For example, when users are offline, a corresponding variation is activated and the messages are stored and delivered later. Finally, in Salvaneschi et al. [2011a] we designed an autonomic Web application which modifies the quality of the Web pages depending on the available network bandwidth.

Behavioral Change. Most COP languages adopt explicit mechanisms to trigger variations. Adaptive systems monitor the events from the external world and from the application execution, then modify their behavior depending on the collected data. Concretely, in COP, the programmer can rely on specific primitives (like the `with` statement) to activate variations. In the design of adaptive systems, explicit activation is often adopted when the events come from the external environment. Conversely, when the source of the events is the application itself, that is, events are points in the execution flow, explicit activation can be cumbersome to apply. Since each activation point must be explicitly managed, the programmer can incur in the code scattering problems that COP should specifically avoid. To solve this issue, COP researchers borrowed quantification from AOP and introduced in their languages expressions that can refer to execution points in the program. For example, in JCOP [Appeltauer et al. 2010b] variations are enabled on the control flow depending on pointcut-like predicates. Since pointcuts quantify over several execution points, the activation code can be properly modularized. EventCJ [Kamina et al. 2010, 2011] is a Java extension which supports per-object layer activation. In EventCJ, the programmer can declaratively define variations transitions on objects. When an event associated to a point in the program execution is reached, the associated transitions are triggered.

Monitoring. Interestingly, the contamination of COP with pointcut-like expressions from AOP is the first attempt to provide COP with dedicated abstractions for program monitoring. However, COP does not address external events or complex event combination. In the aforementioned approaches, the events observed in the execution are immediately bound to variation activation and do not constitute a general mechanism for accessing the application's state. For this reason, at the time being, monitoring is probably the area of adaptive systems where COP has more needs to be coupled with other techniques. For example, in Rho et al. [2011] COP is coupled with a context provisioning system. The framework supports queries on the current context which are continuously evaluated, and COP provides means to dynamically adapt the application when the result changes.

Variations and Separation. From a modularization standpoint, ContextJ layers, discussed in the previous sections, are defined inside classes. The advantage of this approach is that the alternative behavior is immediately available in the same code unit of the basic behavior. Other COP languages (for example, ContextL [Costanza and Hirschfeld 2005]) support different modularization conventions and allow variations declaration outside the lexical scope of the module they augment. This design improves extensibility, since adaptive systems can be provided with new behaviors without modifying the existing codebase. A survey of the modularization solutions investigated by COP can be found in Salvaneschi et al. [2012a].

4. DISCUSSION AND CHALLENGES

In this section, we provide a comparative discussion of the linguistic mechanisms described earlier, by focusing on how they impact on some key aspects of the development of self-adaptive applications.

4.1. Modularization and Extensibility

In adaptive software, modularization includes adding new behavioral variations to the system. A desirable property is that, when the basic system is extended, the variations automatically extend to the new portion. Composition enables reuse of existing code units, like behavioral variations, by combining them to achieve complex behavior. Future extensions of the adaptive system are then immediately available through composition of existing features.

In many adaptive systems, the code implementing the adaptations is kept separate from the codebase. This solution has a number of advantages. In the maintenance phase, the code is more readable and modifications are localized. In the development phase, separate teams can work on different adaptations separately. For example, programmers specialized in fault tolerance can implement recovery strategies independently of security experts focusing on access control. It has been argued that, in general, a clear separation between functional logic developers and adaptability developers is highly desirable [Janik and Zielinski 2010b].

Behavioral variations often plug into different basic functionalities of the application [Cibrán et al. 2007; McKinley et al. 2004]. This aspect is aggravated by the fact that often adaptation involves nonfunctional requirements, like performance or security, which are known to be cross-cutting, and can lead to scattering and tangling code if not properly modularized [Kiczales et al. 1997].

In the context of metaprogramming, cross-cutting adaptation is supported by injecting orthogonal functionalities in several places of the application, for example, by intercepting method executions in a way similar to AOP. However, since metaprogramming does not provide elaborate pointcut languages to express interception concisely, this is more tedious than in AOP. Another issue stems from the fact that metaprogramming does not adopt a standard model for behavioral variations, so modularization of the adaptive behavior has no uniform granularity. This may easily conflict with the way object-oriented software is organized, for example, it is possible to replace entire classes or single methods indiscriminately. Another serious drawback is that the behavior of a program can be modified in ways that break encapsulation; for example, by accessing private attributes or by modifying the internal structure of classes. Even if some of these features can be extremely convenient for monitoring, they are potentially dangerous. Powerful metaprogramming models can overcome these limitations, for example, by altering the dispatching algorithm to account for adaptations, but custom modifications are inevitably complex and error prone.

In AOP, extensibility along the adaptation direction is achieved by adding new aspects to the application. Thanks to quantification, existing aspects apply also to any extension of the basic system. This mechanism, however, must be used with care, to avoid undesirable extensions. Also, many AOP frameworks define pointcuts as syntactic expressions, and therefore suffer the *fragile pointcut problem* [Koppen and Storzer 2004]: due to the tight coupling introduced by syntactic expressions, small modifications of the system or even refactoring can cause advice matching failures or erroneous match to certain joinpoints. A key benefit of AOP is instead automatic composition of aspects defined on the same pointcut.

AOP provides specific support for cross-cutting adaptive concerns and it is probably the strongest solution along this direction. Not surprisingly, separation of adaptation concerns is among the main motivations of the popularity of AOP in self-adaptive systems [David and Ledoux 2006]. However, pointcut languages require specialized skills by the programmers, and complicates code comprehension.

AOP can be an effective way to introduce adaptive capabilities into an existing application. First, aspects are specifically designed to be separated from the existing code. Second, AOP enforces *obliviousness*, that is, the basic program is not aware of the aspects, like in most OO languages classes are not aware of subclasses. Third, the AOP developer only needs to specify the execution points in which advice must be triggered. Therefore, self-adaptive behavior can be added without modifying the structure of the existing application. In principle, AOP can be used to add self-adaptive behavior to existing software even without accessing the source code, like in legacy systems [Yang et al. 2002]: the application can be augmented with probes through bytecode instrumentation or by using virtual machines with dedicated support for aspect weaving. *Transparent shaping* [Sadjadi et al. 2005] is a programming model to enable adaptation in existing programs, which combines AOP to support separation of adaptive features and metaprogramming to control the dynamic reconfiguration. For example, TRAP/J [Sadjadi et al. 2004] allows one to intercept method calls and redirect the execution flow to the adaptation logic.

In COP, the class-in-layer approach offers the best support for modularity, because layers are organized independently of the existing codebase, so new layers can be added to an adaptive application without modifying the existing code. Since in adaptive systems behavioral variations need to dynamically combine, incorrect configurations should be detected and avoided. For example, the *low-bandwidth* and the *high-bandwidth* variations should not be active at the same time. COP emphasizes the importance of these constraints. Some languages use reflection to dynamically enforce combination restrictions [Costanza and Hirschfeld 2007]. Other approaches introduce a domain-specific language to express constraints on layers and throw an exception when they are violated [Costanza and D'Hondt 2008]. Alternatively, it is possible to actively enable certain layers to fulfill the constraints [González et al. 2010]. Another approach is to encapsulate variations in an abstract data type and permit only legal operations [Salvaneschi et al. 2012b]. The problem of constraints among concerns has been explored in AOP as well [Durr et al. 2005; Nagy et al. 2004, 2005; Rashid et al. 2003].

COP modularization abstractions take into consideration the lesson of AOP concerning the cross-cutting nature of some functionalities. COP supports cross-cutting concerns through the concept of layer, which is orthogonal to classes, the main modularization direction in OO. However, COP focuses more on the representation of alternative behavior. Thus, separation is no more mandatory, and some languages adopt a layer-in-class approach, favoring side-by-side placement of basic behavior and adaptive variations. In this case, the application is often more readable, because each method is defined together with all its possible variations.

A point of debate is whether the separation of the adaptation features should include the code performing the dynamic activation, as in David and Ledoux [2006] and Cibrán et al. [2007]. These approaches aim at reducing the tangling of the base code by adaptation-related concerns, including the activation of variations. Activation in the base code is probably a more natural solution when the software is designed from scratch with adaptation in mind. Metaprogramming provides access to activation inside the metalayer, but it is also possible to access the metalayer from the base code. Some AOP languages keep activation separate. In AspectJ, activation can be performed by conditional pointcuts declared in the aspect, but other frameworks support activation through an API which can be even accessed remotely. CaesarJ keeps the activation code in the basic application and the programmer of the base code is responsible for enabling an aspect and selecting its influence scope. COP provides both approaches: for example, in ContextJ, layers are activated by `with` statements in the base code; in ContextPy, each layer can activate itself depending on an external condition.

4.2. Adaptation to Unforeseen Situations

Systems need to be self-adaptive to automatically react to changes in the requirements or in the environment in which they are embedded. Indeed, many modern applications operate in an open world where requirements are not stable [Baresi et al. 2006]. The surrounding environment behaves in a way that is hard to predict when the software is developed and can also change dynamically. In this scenario, anticipating all possible changes is simply not possible. Systems should be therefore *open*, to support the dynamic insertion of new functionalities without compromising service availability. Many off-the-shelf runtime environments address this issue and provide means to dynamically load code modules. For example, Java customized classloaders can be used to fetch classes from a remote network location. The Erlang VM supports *hot code replacement*, which dynamically swaps a module implementation without restarting the system. These environments, in combination with distribution frameworks such as RMI, RPC, or CORBA, can be used to transfer behavioral adaptations to remote nodes and dynamically activate the new functionalities.

Self-adaptive systems can take advantage of the dynamic loading in combination with metaprogramming, since code can be loaded via these services and then easily manipulated via reflection. For example, methods can be entirely replaced by updated versions coming from the network. Metaprogramming interfaces, however, do not provide a unique level of granularity and the change can be rather unstructured, limited only by the expressive power of the metaprogramming protocol. An extreme case is the one involving the *eval* function, which executes arbitrary code provided as data. In languages supporting this feature, *eval* makes it extremely easy to modify the existing codebase, as new features can be simply transmitted as data over the network and installed. Of course, running in the interpreter code provided as a data bypasses safety guarantees and is often a security hazard.

Some AOP frameworks specifically address the problem of unforeseen adaptation. In PROSE [Popovici et al. 2002, 2003] aspects can be sent to the aspect manager through a remote interface (JVMAI, Java Virtual Machine Aspect Interface). Aspects are instantiated and initialized in a first VM, then marshaled and sent over the network to the target VM, where aspects are woven and become effective. Similarly, JAC [Pawlak et al. 2001, 2004] supports remote uploading of aspect components in a distributed environment. Remote transmission of aspects has been explored also for monitoring previously unobserved portions of code. For example, in the AOP-based monitoring framework proposed by Janik and Zielinski [2010a], aspects can be downloaded and dynamically applied.

In COP, little attention has been given to unforeseen adaptation. Concerning this issue, we believe that dynamic layer loading could be easily integrated in many COP languages, for example, this could be done by using the dynamic loading capabilities of the underlying runtime system. However, to the best of our knowledge, only ContextErlang [Salvaneschi et al. 2012b] currently addresses this issue.

4.3. Performance Impact

Adaptive systems monitor the environment, make decisions based on input data, and perform dynamic adaptation to meet the requirements under the changed conditions. Inevitably, these activities have a performance cost. Focusing on the managed element, three main sources of overhead can be identified. First, there is an *activation* cost to enable the behavioral variation. Second, once the behavioral variation is active, its execution is often less efficient than the basecode, since the mechanism to support dynamic linking introduces a level of indirection. Finally, monitoring can penalize performance.

Comparing the performance of the paradigms presented in this article is extremely difficult, because of the variety of the approaches and of the underlying languages and implementations. Surprisingly, very few research efforts have been directed to comparing the available solutions with concrete code examples, highlighting differences, advantages, and disadvantages. A remarkable exception is Dowling et al. [2000], which unfortunately is becoming outdated and does not include either AOP or COP. Besides synthetic examples, we need empirical studies on real applications showing the impact of design choices. Many papers describe specific implementation of autonomic systems, but very few discuss the concrete solutions used to achieve autonomic behavior. Nevertheless, experimental analysis is fundamental to evaluate the impact of a technology on medium- to large-size projects and in the long term.

Metaprogramming is usually considered to be expensive in terms of performance. Dowling et al. [2000] indicate this as one of the limitations of metaprogramming compared to DLLs and design patterns to implement self-adaptive software. However, this should not be overgeneralized. For example, the ABCL/R2 reflective language [Masuhara et al. 1992] shows performances similar to C on concurrent programs. Furthermore, it must be noted that optimization techniques can mitigate the performance problems of metaprogramming in the Java virtual machine[5]. Finally, an important remark is that reflection techniques do not necessarily operate at runtime. Compile-time metaprogramming (e.g., macroexpansion) alters the code during compilation.

As AOP became more and more popular, performance issues were increasingly taken into account and several studies are available. Albeit being somewhat outdated,

[5]http://java.sun.com/products/hotspot/whitepaper.html#performance

the results of the AWbench AOP benchmark[6] represent an indicative reference for the performance of many AOP languages. From a methodological standpoint, Haupt and Mezini [2004] propose a benchmark suite for AOP based on the Java Grande benchmark frameworks [Bull et al. 1999, 2000]. Other performance analysis can be found in the papers describing specific implementations, such as Bockisch et al. [2006a, 2006b] and Haupt et al. [2005]. Even if it is hard to obtain a general overview of AOP performance, the existing benchmarks allow us to draw some conclusions. For example, AOP based on bytecode instrumentation, as in AspectJ, performs better than solutions which rely on dynamic proxies like Spring. Compile-time weavers usually produce faster code than load-time weavers [Nicoara et al. 2008]. Finally, virtual machines are in the general case less efficient than static approaches [Nicoara et al. 2008]. However, they provide advantages in terms of flexibility, since the application can be modified at runtime, and exhibit considerable speedup for dynamic pointcuts, particularly slow in traditional AOP implementations [Bockisch et al. 2006a, 2006b]. However, highly optimized compilers like the aspect bench compiler (abc) can produce comparably efficient code [Avgustinov et al. 2005], when the whole program is available for analysis; but this precludes the use of Java dynamic loading capabilities to achieve self-adaptation. As most of the performance evaluations in AOP focus on steady state, that is, once aspects are already installed, fewer results are available on the overhead of dynamic activation [Bockisch et al. 2006b]. They show how existing approaches have very high activation and deactivation costs and how dedicated VMs can support fast aspect deployment and still provide competitive steady state performance.

Performance of COP languages is extremely variable and research on optimizations is actively ongoing. Languages based on efficient metaprogramming support such as ContextL [Costanza and Hirschfeld 2005] or on compilers, like ContextJ, are reported to experience approximately a 1/3 slowdown compared to the basic language on method dispatching [Appeltauer et al. 2009]. Unfortunately other languages are less efficient. For example, in JavaScript some virtual machines perform aggressive optimizations, but in the COP extension optimizations are inhibited by contextual dispatching, and the slowdown compared to the base language is up to two orders of magnitude [Lincke et al. 2011]. However, these values are observed in microbenchmarks targeting only contextual method dispatching, usually a minor fraction in a real application. For example, in the context of Web applications, we experienced no observable difference (from a client perspective) between the Java and the ContextJ implementations of a Tomcat application [Salvaneschi et al. 2011a]. A systematic performance evaluation of COP languages, including activation costs, was carried on by Appeltauer et al. [2009]. Other comparative microbenchmarks are reported in Appeltauer et al. [2011], Kamina et al. [2011], and Salvaneschi et al. [2012b]. Another performance optimization is proposed by Costanza et al. [2006] where layers are internally represented as classes. In this way, they increase efficiency by leveraging existing dispatching optimizations for multiple inheritance. Further improvements are achieved by caching layer combinations. Krahn et al. [2012] designed an optimization of ContextJS based on caching and method inlining. Finally, Appeltauer et al. [2010a] propose an optimization based on the INVOKEDYNAMIC recently introduced bytecode instruction [Rose 2009].

4.4. Impact on the Development Process

First of all, simplicity is an important factor that can facilitate the acceptance of a technology. The mechanisms to implement dynamic adaptation should therefore be

[6]http://docs.codehaus.org/display/AW/AOP+Benchmark

easy to use by application developers. However, simplicity often conflicts with expressive power, so a reasonable trade-off must be found.

Metaprogramming is usually considered to be hard to master. First, programmers must learn the protocol used to manipulate language abstractions. Second, they have to deal with potentially complex semantic changes: since reflective features can deeply modify the behavior of program entities, applications become harder to understand.

AOP can be considered as a form of constrained metaprogramming; so, not surprisingly, its usage complexity is in general lower. A source of complexity is that programmers are required to learn an ad hoc language to define aspects. On the other hand, some AOP frameworks are based on coding conventions and do not extend the underlying language, but these solutions can be verbose and hard to read, since the semantics of pointcuts and advice is forced into the syntax of the base language [Pawlak et al. 2004]. Also, the semantics of code written in AOP languages which support a rich set of pointcut expressions can be hard to understand.

Compared to metaprogramming and AOP, COP further reduces the complexity for programmers. Essentially, COP specifically addresses adaptive systems, so developers do not deal with unnecessary features, and only have to learn how to design applications with COP abstractions. Moreover, adaptation features are not application specific: programmers do not face the problem of how to apply generic features, like reflection or AOP, in each project they may be involved in.

Another fundamental aspect of the development process involves tool ecosystems or Integrated Development Environments (IDE). Moreover, in a real-world context, programs are not developed in isolation. New applications must interoperate with the existing ones and frequently share the same environment or VM. Programs are often designed in broader frameworks like J2EE or EJB and compatibility with existing libraries is fundamental. Component containers provide support for aspects like persistence and security, and new technologies must integrate with preexisting standards. As a result, it is important to consider how a new technology relates to these issues: for example, if it requires a modified runtime environment—which is often not acceptable in a production setting—or if a syntax extension breaks tool compatibility. Most of these aspects may be transient: if a technology succeeds, proper tool support becomes eventually available. Nevertheless, in practice, this is an important issue for acceptance.

In the case of metaprogramming based on the standard language mechanisms, since there is no language extension, tool compatibility is preserved. The implementation of other paradigms by using metaprogramming shares the same advantages. For example, AspectS [Hirschfeld 2003], an AOP extension of Squeak (an implementation of Smalltalk), is based on the Smalltalk meta-object protocol. In AspectS, aspect weaving and removal employs meta-object composition. In COP, many implementations rely on metaprogramming: among the others, ContextS (Smalltalk MOP), ContextPy and PyContext (Python decorators), and ContextL (Lisp MOP) are based on the reflective features of the base language. While these solutions are always compatible with existing supports for the development process, tools are often not aware of the adaptive features. For example, when the behavior of an application is inspected through a debugger, the programmer needs to step through the scaffolding introduced by metaprogramming, since the debugger has no high-level notion of behavioral variations.

Maintaining the syntax of the base language is a minimal requirement to preserve tool compatibility. For this reason, in some AOP framework aspects are defined in the basic language [Pawlak et al. 2004] or by using annotations, like in Spring. JavaCtx [Salvaneschi et al. 2011b] is an attempt to minimize the impact on the development process by expressing COP abstractions in plain Java. Then AspectJ is used to modify the semantics of the program: behavioral modifications are compiled to aspects that are woven into the application.

Most mature tools, like AspectJ or PROSE, provide IDE extensions (e.g., the AJDT plugin[7]), which support the new language and are aware of the specific abstractions. COP is also moving in that direction. For example, EventCJ and JCOP are currently provided with an Eclipse plugin. Approaches based on macroexpansion or source-to-source compilation generate code that can be processed by a standard compiler. For example, metaprogramming is implemented by a preprocessor in Open C++ [Chiba 1995] and Open Java [Tatsubori et al. 1999]. Some AOP approaches also employ source-to-source compilation, like Aspect C++ [Spinczyk et al. 2002]. In COP, ContextJ and EventCJ are implemented as source-to-source compilers. More radical choices include custom compilers and VM support. Adopting a modified version of the compiler can encounter resistance in a production environment, but may allow one to perform specific optimizations that are otherwise not possible. JCOP is a context-oriented compiler which directly emits Java bytecode [Appeltauer et al. 2010b].

5. OTHER RELATED APPROACHES

In this section, we consider some other related approaches that are relevant for defining self-adaptive software, but are less directly comparable with those considered here in details: metaprogramming, AOP, and COP.

The agent-oriented programming paradigm [Shoham 1993] has been proposed for multi-agent systems [Shoham and Leyton-Brown 2008], a software engineering approach to implement applications open to unforeseen conditions and that can successfully model human reasoning and team behavior. The agent-oriented paradigm adopts agents as building elements and provides abstractions to model social and cognitive behaviors. Agents are specified in terms of concepts like communication, beliefs, plans, goals, and actions. Agent-oriented languages include JADE [Bellifemine et al. 2007], AgentSpeak [Rao 1996], JACK [Winikoff 2005], Jason [Bordini et al. 2005], and 2APL [Dastani 2008]. These languages support agent structure, agent interaction, and messaging. They provide declarative abstractions, to specify the agent beliefs and plans and ad hoc semantics to model agent behavior, for example, to express reasoning and planning. Complex systems are supported by transparent distribution and interfacing with mainstream languages. The latter feature is used, for example, to model the interaction with the environment or to implement internal actions in an imperative way.

The paradigms discussed in this article tackle the problem of extending existing (mainstream) languages with the flexibility required by self-adaptive systems. For example, AOP, COP, and metaprogramming introduce new directions of variability to model behavioral adaptation, they support interception to inject monitoring code, and reify programming abstractions to dynamically modify the execution. However, those paradigms are not bound to any specific software engineering style. On the other hand, agent-oriented languages specifically address the multi-agent style. Another difference is that AOP, COP, and metaprogramming augment the semantics of the base language, while agent-based languages enforce an ad hoc semantics that for convenience can interface with a general-purpose language. An interesting line for future research is to bridge the gap between those paradigms and making the features provided by both available in the same language.

The tuple-based programming model focuses on accessing data in a distributed system in a flexible and compact way [Gelernter 1985]. Tuple spaces provide a

[7]http://www.eclipse.org/ajdt

repository of tuples that can be accessed to publish data. Clients can retrieve data via pattern matching. Tuple-based models have been successfully applied to a number of adaptive scenarios, including sensor networks [Whitehouse et al. 2004], mobile applications [Murphy et al. 2006], and bio-inspired computing [Menezes and Tolksdorf 2003]. Recently, tuple-based models have been used to support context-aware adaptation and situated computing. In the TOTA approach [Mamei and Zambonelli 2009], tuples automatically propagate in the network according to user-defined patterns and agents can access the close tuples to tune their behavior. A similar approach [Viroli et al. 2011] proposes chemically inspired tuple spaces. This solution supports pervasive applications by modeling computational patterns based on proximity, competition, and situatedness. Compared to the programming paradigms presented in this article, tuple-based models position more closely to adaptive middlewares, while AOP, COP, and metaprogramming focus on abstractions that enrich a language to support adaptation.

6. CONCLUSIONS

In this article, we discussed the main language-level approaches used to implement self-adaptive systems, namely metaprogramming, AOP, and COP. We presented the main contributions and we compared the advantages of each solution. We also identified a number of research challenges that call for further investigation.

As we observed, self-adaptation may be achieved in different ways and at different levels. We focused here on linguistic mechanisms that provide explicit support to self-adaptation. At this stage, it would be hard to provide conclusive arguments regarding the benefits we achieve through the use of specific linguistic support. This will only be possible through practical use and empirical observation of application developments. We argue, however, that the current evolution of the technology favors language-level techniques. For example, sensor networks and mobile devices are more and more widespread. These devices usually have a relatively simple programming model, and applications are not large. In this scenario, architectural solutions like component-based adaptation are overkilling. In addition, in the last few years, we observed a concentration in the mobile area around only few platforms. These platforms are based on a set of libraries and a common language, typically Java or Objective-C. This homogeneity can further promote the use of standard language-level means to support dynamic adaptation.

We envisage an interesting research direction in combining the advantages of the techniques presented in this article. With a few exceptions (e.g., Sadjadi et al. [2004]), the language-level mechanisms described in this article have been applied in isolation. However, considerable advantages can be achieved by the contamination of paradigms. For example, obliviousness is a fundamental property of AOP to support interception without modifying the basecode. This feature is fundamental to add autonomic capabilities to legacy code. This approach can be coupled with COP explicit activation, which appears more natural when adaptation is designed upfront.

Another interesting hybridization can be envisaged with design patterns [Gamma et al. 1993]. Mixing patterns via dedicated language syntax is not a recent idea. For example, the Iterator pattern is a successful case of hybridization between patterns and language abstractions. Library programmers apply standard patterns to provide access to collections, and client programmers can rely on language constructs to access data at a higher level of abstraction. A combination of COP and patterns could benefit from the conciseness of COP dedicated language support and the familiarity that most developers have with patterns.

REFERENCES

Appeltauer, M., Hirschfeld, R., Haupt, M., Lincke, J., and Perscheid, M. 2009. A comparison of context-oriented programming languages. In *Proceedings of the International Workshop on Context-Oriented Programming (COP'09)*. ACM, New York, 1–6.

Appeltauer, M., Haupt, M., and Hirschfeld, R. 2010a. Layered method dispatch with invokedynamic: An implementation study. In *Proceedings of the 2nd International Workshop on Context-Oriented Programming (COP'10)*. ACM, New York, 4:1–4:6.

Appeltauer, M., Hirschfeld, R., Masuhara, H., Haupt, M., and Kawauchi, K. 2010b. Event-specific software composition in context-oriented programming. In *Proceedings of the 9th International Conference on Software Composition (SC'10)*. B. Baudry and E. Wohlstadter Eds., Lecture Notes in Computer Science Series, vol. 6144, Springer, 50–65.

Appeltauer, M., Hirschfeld, R., Haupt, M., and Masuhara, H. 2011. ContextJ: Context-oriented programming with Java. *Inf. Media Technol. 6*, 2, 399–419.

Aracic, I., Gasiunas, V., Mezini, M., and Ostermann, K. 2006. An overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development*. I. A. Rashid and M. Aksit Eds., Lecture Notes in Computer Science Series, vol. 3880, Springer, 135–173.

Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J. 2005. Optimising AspectJ. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. ACM, New York, 117–128.

Bachara, P., Blachnicki, K., and Zielinski, K. 2010. Framework for application management with dynamic aspects J-EARS case study. *Inf. Softw. Technol. 52*, 67–78.

Baresi, L., Di Nitto, E., and Ghezzi, C. 2006. Toward open-world software: Issue and challenges. *Comput. 39*, 10, 36–43.

Bellifemine, F. L., Caire, G., and Greenwood, D. 2007. *Developing Multi-Agent Systems with JADE*. Wiley Series in Agent Technology. John Wiley and Sons.

Blair, G. S., Coulson, G., Andersen, A., Blair, L., Clarke, M., Costa, F., Duran-Limon, H., Fitzpatrick, T., Johnston, L., Moreira, R., Parlavantzas, N., and Saikoski, K. 2001. The design and implementation of Open ORB 2. *IEEE Distrib. Syst. Online 2*.

Bockisch, C., Arnold, M., Dinkelaker, T., and Mezini, M. 2006a. Adapting virtual machine techniques for seamless aspect support. *SIGPLAN Not. 41*, 10, 109–124.

Bockisch, C., Kanthak, S., Haupt, M., Arnold, M., and Mezini, M. 2006b. Efficient control flow quantification. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06)*. ACM, New York, 125–138.

Bollert, K. 1999. On weaving aspects. In *Proceedings of the Workshop on Aspect-Oriented Programming (ECOOP'99)*.

Boner, J. 2004. AspectWerkz - Dynamic AOP for Java. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*.

Bordini, R. H., Hübner, J. F., and Vieira, R. 2005. Jason and the Golden Fleece of agent-oriented programming. In *Multi-Agent Programming: Languages, Platforms and Applications*. R. H. Bordini, M. Dastani, J. Dix, and A. E. F. Seghrouchni Eds., Multiagent Systems, Artificial Societies, and Simulated Organizations, vol. 15, Springer, New York, 3–37.

Brichau, J. and Haupt, M. 2005. Survey of aspect-oriented languages and execution models. Tech. rep. AOSD-Europe-VUB-01, AOSD-Europe.

Bruneton, E., Lenglet, R., and Coupaye, T. 2002. ASM: A code manipulation tool to implement adaptable systems. In *Proceedings of the Adaptable and Extensible Component Systems*. 1–12.

Bull, J. M., Smith, L. A., Westhead, M. D., Henty, D. S., and Davey, R. A. 1999. A methodology for benchmarking Java Grande applications. In *Proceedings of the ACM Conference on Java Grande (JAVA'99)*. ACM, New York, 81–88.

Bull, J. M., Smith, L. A., Westhead, M. D., Henty, D. S., and Davey, R. A. 2000. A benchmark suite for high performance Java. *Concurr. Pract. Exper. 12*, 6, 375–388.

Capra, L., Emmerich, W., and Mascolo, C. 2003. CARISMA: Context-aware reflective middleware system for mobile applications. *IEEE Trans. Softw. Engin. 29*, 10, 929–945.

Chan, A. T. S. and Chuang, S.-N. 2003. MobiPADS: A reflective middleware for context-aware mobile computing. *IEEE Trans. Softw. Eng. 29*, 12, 1072–1085.

Charfi, A. and Mezini, M. 2004. Aspect-oriented web service composition with AO4BPEL. In *Proceedings of the European Conference on Web Services (ECOWA'94)*. L.-J. Zhang and M. Jeckle Eds., Lecture Notes in Computer Science Series, vol. 3250, Springer, 168–182.

Chiba, S. 1995. A metaobject protocol for C++. In *Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*. ACM, New York, 285–299.

Cibrán, M. A., Verheecke, B., Vanderperren, W., Suvée, D., and Jonckers, V. 2007. Aspect-oriented programming for dynamic web service selection, integration and management. *World Wide Web 10*, 3, 211–242.

Costanza, P. and D'Hondt, T. 2008. Feature descriptions for context-oriented programming. In *Proceedings of the 12th International Conference on Software Product Lines (SPLC'08)*. 9–14.

Costanza, P. and Hirschfeld, R. 2005. Language constructs for context-oriented programming: An overview of ContextL. In *Proceedings of the Symposium on Dynamic Languages (DLS'05)*. ACM Press, New York, 1–10.

Costanza, P. and Hirschfeld, R. 2007. Reflective layer activation in contextL. In *Proceedings of the ACM Symposium on Applied Computing (SAC'07)*.

Costanza, P., Hirschfeld, R., and De Meuter, W. 2006. Efficient layer activation for switching context-dependent behavior. In *Proceedings of the 7th Joint Conference on Modular Programming Languages (JMLC'06)*. Springer, 84–103.

Courbis, C. and Finkelstein, A. 2005. Towards aspect weaving applications. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*. ACM, New York, 69–77.

Dahm, M. and Berlin, F. U. 1998. Byte code engineering with the BCEL api. Tech. rep. B-17-98, Freie Universitt Berlin - Institut fr Informatik.

Dastani, M. 2008. 2APL: A practical agent programming language. *Auton. Agents Multi-Agent Syst. 16*, 3, 214–248.

David, P.-C. and Ledoux, T. 2006. An aspect-oriented approach for developing self-adaptive fractal components. In *Proceedings of the 5th International Conference on Software Composition (SC'06)*. Springer, 82–97.

Dawson, D., Desmarais, R., Kienle, H. M., and Müller, H. A. 2008. Monitoring in adaptive systems using reflection. In *Proceedings of the International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'08)*. ACM Press, New York, 81–88.

Denys, G., Piessens, F., and Matthijs, F. 2002. A survey of customizability in operating systems research. *ACM Comput. Surv. 34*, 450–468.

Dowling, J., Schäfer, T., Cahill, V., Haraszti, P., and Redmond, B. 2000. Using reflection to support dynamic adaptation of system software: A case study driven evaluation. In *Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering: Reflection and Software Engineering*. Papers from OORaSE 1999. Springer, 169–188.

Durr, P., Staijen, T., Bergmans, L., and Aksit, M. 2005. Reasoning about semantic conflicts between aspects. In *Proceedings of the European Interactive Workshop on Aspects in Software*.

Eliassen, F., Andersen, A., Blair, G. S., Costa, F., Coulson, G., Goebel, V., Ivind Hansen, Kristensen, T., Plagemann, T., Rafaelsen, H. O., Saikoski, K. B., and Yu, W. 1999. Next generation middleware: Requirements, architecture, and prototypes. In *Proceedings of the 7th IEEE Workshop on Future Trends in Distributed Computing Systems*. IEEE Computer Society Press, 60–65.

Engel, M. and Freisleben, B. 2005. Supporting autonomic computing functionality via dynamic operating system kernel aspects. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD'05)*. ACM, New York, 51–62.

Epifani, I., Ghezzi, C., Mirandola, R., and Tamburrelli, G. 2009. Model evolution by run-time parameter adaptation. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*. IEEE Computer Society, 111–121.

Gamma, E., Helm, R., Johnson, R. E., and Vlissides, J. M. 1993. Design patterns: Abstraction and reuse of object-oriented design. In *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP'93)*. Springer, 406–431.

Gelernter, D. 1985. Generative communication in Linda. *ACM Trans. Program. Lang. Syst. 7*, 1, 80–112.

Ghezzi, C., Pradella, M., and Salvaneschi, G. 2011. An evaluation of the adaptation capabilities in programming languages. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'11)*. ACM Press, New York, 50–59.

González, S., Mens, K., and Heymans, P. 2007. Highly dynamic behaviour adaptability through prototypes with subjective multimethods. In *Proceedings of the Symposium on Dynamic Languages (DLS'07)*. ACM Press, New York, 77–88.

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

Gowing, B. and Cahill, V. 1995. Making meta-object protocols practical for operating systems. In *Proceedings of the 4th International Workshop on Object-Orientation in Operating Systems*. 52–55.

Grace, P., Blair, G., and Samuel, S. 2003. ReMMoC: A reflective middleware to support mobile client interoperability. In *Proceedings of the OTM Confederated International Conferences, CoopIS, DOA, and ODBASE*. R. Meersman, Z. Tari, and D. Schmidt Eds., Lecture Notes in Computer Science Series, vol. 2888, Springer, 1170–1187.

Greenwood, P. and Blair, L. 2003. Using dynamic aspect-oriented programming to implement an autonomic system. In *Proceedings of the Dynamic Aspect Workshop (DAW'04)*.

Greenwood, P. and Blair, L. 2006. A framework for policy driven auto-adaptive systems using dynamic framed aspects. In *Transactions on Aspect-Oriented Software Development II*. Springer, 30–65.

Haupt, M. and Mezini, M. 2004. Micro-measurements for dynamic aspect-oriented systems. In *Proceedings of the 5th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Nerworked World*. M. Weske and P. Liggesmeyer Eds., Lecture Notes in Computer Science Series, vol. 3263, Springer, 277–305.

Haupt, M., Mezini, M., Bockisch, C., Dinkelaker, T., Eichberg, M., and Krebs, M. 2005. An execution layer for aspect-oriented programming languages. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE'05)*. ACM Press, New York, 142–152.

Hellerstein, J. L. 2009. Configuring resource managers using model fuzzing: A case study of the .NET thread pool. In *Proceedings of the 11th IFIP/IEEE International Conference on Symposium on Integrated Network Management (IM'09)*. IEEE Press, 1–8.

Hewitt, C., Bishop, P., and Steiger, R. 1973. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI'73)*. Morgan Kaufmann Publishers, San Francisco, CA, 235–245.

Hirschfeld, R. 2003. AspectS - aspect-oriented programming with Squeak. In *Revised Papers from the International Conference on NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World (NODe'02)*. Springer, 216–232.

Hirschfeld, R., Costanza, P., and Nierstrasz, O. 2008. Context-oriented programming. *J. Object Technol. 7*, 3.

Hsieh, W., Fiuczynski, M., Garrett, C., Savage, S., Becker, D., and Bershad, B. 1996. Language support for extensible operating systems. In *Proceedings of the Workshop on Compiler Support for System Software*. 127–133.

Huebscher, M. C. and McCann, J. A. 2008. A survey of autonomic computing – Degrees, models, and applications. *ACM Comput. Surv. 40*, 7:1–7:28.

Itoh, J.-I., Lea, R., and Yokote, Y. 1995. Using meta-objects to support optimisation in the Apertos operating system. In *Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS'95)*. USENIX Association, Berkeley, CA, 11–11.

Janik, A. and Zielinski, K. 2010a. Aaop-based dynamically reconfigurable monitoring system. *Inf. Softw. Technol. 52*, 380–396.

Janik, A. and Zielinski, K. 2010b. Adaptability mechanisms for autonomic system implementation with aaop. *Softw. Pract. Exper. 40*, 209–223.

Kamina, T., Aotani, T., and Masuhara, H. 2010. Designing event-based context transition in context-oriented programming. In *Proceedings of the 2nd International Workshop on Context-Oriented Programming (COP'10)*. ACM Press, New York, 2:1–2:6.

Kamina, T., Aotani, T., and Masuhara, H. 2011. EventCJ: A context-oriented programming language with declarative event-based context transition. In *Proceedings of the 10th International Conference on Aspect-Oriented Software Development (AOSD'11)*. ACM Press, New York, 253–264.

Kephart, J. O. 2005. Research challenges of autonomic computing. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*. ACM Press, New York, 15–22.

Kephart, J. O. and Chess, D. M. 2003. The vision of autonomic computing. *Comput. 36*, 41–50.

Kiczales, G. and Rivieres, J. D. 1991. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. 1997. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*. Lecture Notes in Computer Science, vol. 1241, Springer, 220–242.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. 2001. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*. J. Knudsen Ed., Lecture Notes in Computer Science Series, vol. 2072, Springer, 327–354.

Kon, F., Costa, F., Blair, G., and Campbell, R. H. 2002. The case for reflective middleware. *Comm. ACM 45*, 33–38.

Koppen, C. and Storzer, M. 2004. PCDiff: Attacking the fragile pointcut problem. In *European Interactive Workshop on Aspects in Software (EIWAS'04)*. K. Gybels, S. Hanenberg, S. Herrmann, and J. Wloka Eds., Krahn, R., Lincke, J., and Hirschfeld, R. 2012. Efficient layer activation in ContextJS. In *IEEE Conference on Creating, Connecting and Collaborating through Computing (C5)*.

Kramer, J. and Magee, J. 2007. Self-managed systems: An architectural challenge. In *Proceedings of the Conference on Future of Software Engineering (FOSE'07)*. IEEE Computer Society, 259–268.

Laddad, R. 2009. *AspectJ in Action: Enterprise AOP with Spring Applications* 2nd Ed. Manning Publications, Greenwich, CT.

Lamm, E. 2001. Component libraries and language features. In *Proceedings of the 6th International Conference Leuven on Reliable Software Technologies (Ada-Europe'01)*. D. Craeynest and A. Strohmeier Eds., Lecture Notes in Computer Science, vol. 2043, Springer, 215–228.

Ledoux, T. 1997. Implementing proxy objects in a reflective ORB. In *Proceedings of the Workshop on CORBA: Implementation, Use and Evaluation (ECOOP'97)*.

Lincke, J., Appeltauer, M., Steinert, B., and Hirschfeld, R. 2011. An open implementation for context-oriented layer composition in ContextJS. *Sci. Comput. Program. 76*, 12, 1194–1209.

Madany, P. W., Islam, N., Kougiouris, P., and Campbell, R. H. 1992. Reification and reflection in C++: An operating systems perspective. Tech. rep. UIUCDCS-R-92-1736, University of Illinois at Urbana-Champaign.

Maes, P. 1987. Concepts and experiments in computational reflection. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*. ACM Press, New York, 147–155.

Mamei, M. and Zambonelli, F. 2009. Programming pervasive and mobile computing applications: The TOTA approach. *ACM Trans. Softw. Eng. Methodol. 18*, 4, 15:1–15:56.

Masuhara, H., Matsuoka, S., Watanabe, T., and Yonezawa, A. 1992. Object-oriented concurrent reflective languages can be implemented efficiently. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'92)*. ACM Press, New York, 127–144.

McKinley, P. K., Sadjadi, S. M., Kasten, E. P., and Cheng, B. H. C. 2004. Composing adaptive software. *Comput. 37*, 56–64.

Menezes, R. and Tolksdorf, R. 2003. A new approach to scalable Linda-systems based on swarms. In *Proceedings of the ACM Symposium on Applied Computing (SAC'03)*. ACM Press, New York, 375–379.

Murphy, A. L., Picco, G. P., and Roman, G.-C. 2006. LIME: A coordination model and middleware supporting mobility of hosts and agents. *ACM Trans. Softw. Eng. Methodol. 15*, 3, 279–328.

Nagy, I., Bergmans, L., and Aksit, M. 2004. Declarative aspect composition. In *Software-Engineering Properties of Languages for Aspect Technologies (SPLAT!): in conjunction with the 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*.

Nagy, I., Bergmans, L., and Aksit, M. 2005. Composing aspects at shared join points. In *Proceedings of International Conference NetObjectDays (NODe'05)*. Springer.

Nicoara, A., Alonso, G., and Roscoe, T. 2008. Controlled, systematic, and efficient code replacement for running Java programs. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems (Eurosys'08)*. ACM Press, New York, 233–246.

Okamuray, H., Ishikawayy, Y., and Tokoroy, M. 1992. AL-1/D: A distributed programming system with multi-model reflection framework. In *Workshop on New Models for Software Architecture*.

Oreizy, P., Medvidovic, N., and Taylor, R. N. 1998. Architecture-based runtime software evolution. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*. IEEE Computer Society, 177–186.

Oreizy, P., Medvidovic, N., and Taylor, R. N. 2008. Runtime software adaptation: Framework, approaches, and styles. In *Companion of the 30th International Conference on Software Engineering (ICSE'08)*. ACM Press, New York, 899–910.

Pawlak, R., Seinturier, L., Duchien, L., and Florin, G. 2001. JAC: A flexible solution for aspect-oriented programming in Java. In *Proceedings of the 3rd International Conference REFLECTION*. A. Yonezawa and S. Matsuoka Eds., Lecture Notes in Computer Science, vol. 2192, Springer, 1–24.

Pawlak, R., Seinturier, L., Duchien, L., Florin, G., Legond-Aubry, F., and Martelli, L. 2004. JAC: An aspect-based distributed dynamic framework. *J. Softw. Practice Exper. 34*, 12, 1119–1148

Popovici, A., Gross, T., and Alonso, G. 2002. Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD'02)*. ACM Press, New York, 141–147.

Popovici, A., Alonso, G., and Gross, T. 2003. Just-in-time aspects: Efficient dynamic weaving for Java. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*. ACM Press, New York, 100–109.

Rao, A. S. 1996. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Proceedings of the 7th European Workshop on Modeling Autonomous Agents in a Multi-Agent World: Agents Breaking Away (MAAMAW'96)*. Springer, 42–55.

Rashid, A., Moreira, A., and Araújo, J. 2003. Modularisation and composition of aspectual requirements. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*. ACM Press, New York, 11–20.

Redmond, B. and Cahill, V. 2006. Supporting unanticipated dynamic adaptation of application behaviour. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP'02)*. B. Magnusson Ed., Lecture Notes in Computer Science, vol. 2374, Springer, 29–53.

Rho, T., Appeltauer, M., Lerche, S., Cremers, A. B., and Hirschfeld, R. 2011. A context management infrastructure with language integration support. In *Proceedings of the 3rd International Workshop on Context-Oriented Programming (COP'11)*. ACM Press, New York, 3:1–3:6.

Rivard, F. 1996. A new smalltalk kernel allowing both explicit and implicit metaclass programming. In *Proceedings of the Workshop on Extending the Smalltalk Language (OOPSLA'96)*.

Roman, M., Kon, F., and Campbell, R. 1999. Design and implementation of runtime reflection in communication middleware: The dynamicTAO case. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshops on Electronic Commerce and Web-based Applications/Middleware*. 122–127.

Rose, J. R. 2009. Bytecodes meet combinators: Invokedynamic on the JVM. In *Proceedings of the 3rd Workshop on Virtual Machines and Intermediate Languages (VMIL'09)*. ACM Press, New York, 2:1–2:11.

Sadjadi, S. M., McKinley, P. K., Cheng, B. H. C., and Stirewalt, R. E. K. 2004. TRAP/J: Transparent generation of adaptable Java programs. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'04)*.

Sadjadi, S. M., McKinley, P. K., and Cheng, B. H. C. 2005. Transparent shaping of existing software to support pervasive and autonomic computing. In *Proceedings of the Workshop on Design and Evolution of Autonomic Application Software (DEAS'05)*. ACM Press, New York, 1–7.

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

Salvaneschi, G., Ghezzi, C., and Pradella, M. 2011a. Context-oriented programming: A programming paradigm for autonomic systems. CoRR abs/1105.0069. http://arxiv.org/vc/arxiv/papers/1105/1105.0069vl.pdf.

Salvaneschi, G., Ghezzi, C., and Pradella, M. 2011b. JavaCtx: Seamless toolchain integration for context-oriented programming. In *Proceedings of the 3rd International Workshop on Context-Oriented Programming (COP'11)*. ACM Press, New York, 4:1–4:6.

Salvaneschi, G., Ghezzi, C., and Pradella, M. 2012a. Context-oriented programming: A software engineering perspective. *J. Syst. Softw. 85*, 8, 1801–1817.

Salvaneschi, G., Ghezzi, C., and Pradella, M. 2012b. 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 Press, New York, 191–202.

Sato, Y., Chiba, S., and Tatsubori, M. 2003. A selective, just-in-time aspect weaver. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE'03)*. Springer, 189–208.

Ségura-Devillechaise, M., Menaud, J.-M., Muller, G., and Lawall, J. L. 2003. Web cache prefetching as an aspect: Towards a dynamic-weaving based solution. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*. ACM Press, New York, 110–119.

Sharifi, A., Srikantaiah, S., Mishra, A. K., Kandemir, M., and Das, C. R. 2011. Mete: Meeting end-to-end QoS in multicores through system-wide resource management. *SIGMETRICS Perform. Eval. Rev. 39*, 1, 13–24.

Shoham, Y. 1993. Agent-oriented programming. *Artif. Intell. 60*, 1, 51–92.

Shoham, Y. and Leyton-Brown, K. 2008. *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press, Cambridge, UK.

Smith, B. C. 1984. Reflection and semantics in lisp. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'84)*. ACM Press, New York, 23–35.

Spinczyk, O., Gal, A., and Schröder-Preikschat, W. 2002. AspectC++: An aspect-oriented extension to the c++ programming language. In *Proceedings of the 40th International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications (CRPIT'02)*. 53–60.

Stroud, R. J. and Wu, Z. 1996. Using metaobject protocols to satisfy non-functional requirements. In *Advances in Object-Oriented Metalevel Architectures and Reflection*. C. Zimmerman Ed.

Suvée, D., Vanderperren, W., and Jonckers, V. 2003. JAsCo: An aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*. ACM Press, New York, 21–29.

Tarr, P., Ossher, H., Harrison, W., and Sutton, Jr., S. M. 1999. N degrees of separation: Multidimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*. ACM Press, New York, 107–119.

Tatsubori, M., Chiba, S., Itano, K., and Killijian, M.-O. 1999. OpenJava: A class-based macro system for Java. In *Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering: Reflection and Software Engineering*. 117–133.

Vanderperren, W., Suvée, D., Verheecke, B., Cibrán, M. A., and Jonckers, V. 2005. Adaptive programming in JAsCo. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD'05)*. ACM Press, New York, 75–86.

Vasseur, A. 2004. Dynamic aop and runtime weaving for Java – How does AspectWerkz address it? In *Proceedings of the DAW Dynamic Aspects Workshop*. 135–145.

Viroli, M., Casadei, M., Montagna, S., and Zambonelli, F. 2011. Spatial coordination of pervasive services through chemical-inspired tuple spaces. *ACM Trans. Auton. Adapt. Syst. 6*, 2, 14:1–14:24.

White, S., Hanson, J., Whalley, I., Chess, D., and Kephart, J. 2004. An architectural approach to autonomic computing. In *Proceedings of the International Conference on Autonomic Computing*. 2–9.

Whitehouse, K., Sharp, C., Brewer, E., and Culler, D. 2004. Hood: A neighborhood abstraction for sensor networks. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services (MobiSys'04)*. ACM Press, New York, 99–110.

Win, B. D., Vanhaute, B., and Decker, B. D. 2001. Security through aspect-oriented programming. In *Proceedings of the 1st Annual Working Conference on Network Security: Advances in Network and Distributed Systems Security*. 125–138.

Winikoff, M. 2005. JACK intelligent agents: An industrial strength platform. In *Multi-Agent Programming, Languages: Platforms and Applications*, Springer, 175–193.

Xu, J., R, B., and Zorzo, A. F. 1996. Implementing software-fault tolerance in C++ and Open C++: An object-oriented and reflective approach. In *Proceedings of the International Workshop on Computer Aided Design, Test, and Evaluation for Dependability*. 224–229.

Yang, Z., Cheng, B. H. C., Stirewalt, R. E. K., Sowell, J., Sadjadi, S. M., and McKinley, P. K. 2002. An aspect-oriented approach to dynamic adaptation. In *Proceedings of the 1st Workshop on Selfhealing Systems (WOSS'02)*. ACM Press, New York, 85–92.

Yokote, Y. 1992. The Apertos reflective operating system: The concept and its implementation. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'92)*. ACM Press, New York, 414–434.

Received July 2012; revised November 2012; accepted December 2012