Contributions to the Autonomy of Ubiquitous Software Systems

Romain Rouvoy

Université de Lille 1 SCIENCES ET TECHNOLOGIES

HABILITATION À DIRIGER DES RECHERCHES DE L'UNIVERSITÉ LILLE 1, SCIENCES ET TECHNOLOGIES

Spécialité

Informatique

École doctorale Sciences Pour l'Ingénieur (Lille)

Présentée par

Romain ROUVOY

Sujet de l'habilitation :

Contributions to the Autonomy of Ubiquitous Software Systems

soutenue le 05 décembre 2014

devant le jury composé de :

M.	*Pierre* Boulet	Président
M.	*Ivica* Crnkovic	Rapporteur
Mme	*Valérie* Issarny	Rapporteur
Mme	*Nalini* Venkatasubramanian	Rapporteur
M.	*Frank* Eliassen	Examinateur
M.	*Lionel* Seinturier	Examinateur

Acknowledgments

Contents

Introduction

This manuscript intends to provide a summary of the researches that I have been contributing to and leading for the last 8 years. While the broad scope of my research belongs to *distributed systems* and *software engineering*, this document reports on the contributions we developed along the axis of ubiquitous software systems. This particular category of systems has been catalyzed by the release of a new generation of mobile phones in 2007, while I was starting as a postdoctoral position at the University of Oslo. Since that, the emergence of smartphones, like the iPhone or the Android devices, upsets computer science by making such small computers ubiquitous and continuously connected. With 1.4 billion active smartphones by the end of 2014,[1] ubiquitous software systems are not only changing the distribution of processing capabilities, but also raise interesting research challenges. Among others, *how do we program applications that are potentially expected to run for billions of processors? How to account for computing resources that are sporadically available? How to conciliate performance and energy consumption? How to take benefit from the context in which the mobile device is immersed? How the explosion of mobile devices may affect server-side infrastructures?*

Such questions can be addressed under different perspectives by the research community and we decided to focus our efforts on the improvement of the autonomy of such ubiquitous software systems. In particular, we consider that, by giving mobile devices the capability to reason about themselves, they can autonomously decide to seamlessly adapt to the best fitting configuration depending on the current context of execution. To provide such a capability, we consider that a ubiquitous software system requires to build on three pillars: *i)* a modular programming abstraction to expose and control the reconfiguration capacities of the system under control, *ii)* a flexible context middleware solution to monitor both internal and external conditions that can affect the execution of the system, and *iii)* an open decision-making component to conciliate constraints and opportunities in order to take informed decisions on the most appropriate reconfigurations to be operated.

The state-of-practice in the development of ubiquitous software systems may tend to neglect software engineering considerations on behalf of energy or resources consumptions. However, such choices not only prevent reuse of application parts, but also penalize dynamic reconfiguration scenarios by transmitting a monolithic binary image to be loaded by the target device, inducing a undesired energy penalty. Context monitoring is a concern, which has been studied by various domains, from operating systems to human-computer interactions. Nevertheless, the context of a ubiquitous software system does not only cover the monitoring of internal events that are produced by the target device. In particular, a ubiquitous context can also be impacted by the surrounding environment, as external events being directly perceived by the mobile device or not. A flexible solution to context monitoring undoubtedly favors the quality of the decisions to be taken by the system. Finally, such decisions cannot only be based on declarative approaches that assumes a comprehensive knowledge of the operational condition of an ubiquitous software system. By definition, a ubiquitous software system can run in unexpected situations and should therefore be able to deliver the best quality of service in light of the heuristics it has been configured with.

Each of these pillars raises some key challenges to be addressed by the research community. However, more that the individual solutions we developed to address them, we consider that a valuable result of our approach lies in the combination and the complementarity of the approaches we propose. The research results that are summarized in this document are therefore oriented towards providing new abstractions to program, contextualize, and adapt ubiquitous software systems. Nonetheless, we have been continuously working on these three axis by trying to foster cross-fertilizations across the approaches we were developing. Ultimately, the path we have been following leads to the emergence of feedback control loops as an appropriate concept to address the challenge of autonomic ubiquitous software systems. We therefore consider that the experience we gained by addressing the challenges reported in this document are providing an interesting basis to leverage the

[1] `https://www.abiresearch.com/market-research/product/1004938-smartphone-technologies-and-markets`

development of autonomic ubiquitous software systems. The remainder of this document is therefore organized as follows:

Chapter 1 focuses on the first pillar and the contributions we developed to improve the elasticity of software components. In particular, we consider the different scales of component-based software engineering by reporting first on programing in the small with REMORA (cf. Section 1.2.1) as a solution to develop component-based sensor nodes. Then, we consider challenges of programing component-based systems in the large and we promote FRASCALA (cf. Section 1.2.2) as a solution to capture software architecture patterns that can be easily reused and configured. Finally, we detail the design of a middleware solution, named MACCHIATO (cf. Section 1.2.3) that intends to leverage the development of ubiquitous software systems that have the capability to stretch beyond the boundaries of the hosting mobile device;

Chapter 2 addresses the second pillar and describes the models we developed to collect and process context both in breadth and in depth. We start by reporting on the COSMOS context model (cf. Section 2.2.1) that we use to model context policies as hierarchies of context information that are correlated. From this generic model, we derive APISENSE® (cf. Section 2.2.2), as an innovative approach to gather context information at the scale of a crowd and thus offering another dimension of reasoning for ubiquitous software systems. We also derive PowerAPI (cf. Section 2.2.3) as an example of in depth context monitoring solution based on the COSMOS model. This approach illustrates how context can percolate into a system to better capture the traceability between a given context information and the software artifact it connects;

Chapter 3 covers the last pillar and more specifically the issue self-adaptation of ubiquitous applications. We first report on the contributions we developed within the MUSIC adaptation middleware (cf. Section 3.2.1) to deliver a technology-agnostic abstraction that we used to reason on the integration of third-party software services in a mobile application. This first contribution has then been extended by SPACES (cf. Section 3.2.2) to consider the discovery of adaptation policies and to sketch the design principles of feedback control loops, notably by introducing the concept of *context-as-a-Resource*. This second step has been further extended in CORONA (cf. Section 3.2.3), by collaborating to the definition of a domain-specific modeling language for the definition of feedback control loops and their projections on specific target platforms, like SCA, an industrial component model;

Chapter 4 concludes this document by providing a summary of our main achievements during the last 8 years and provides some insights on the research directions we intend to pursue along the upcoming years.

Chapter 1

Elasticity of Ubiquitous Systems

1.1 Motivations

The principles of self-adaptation of software systems rely on the capability to model and then to modify some properties of the underlying software systems in order to cope with changes observed internally or in their surrounding environment. However, providing such an intrinsic capability to alter the structure or the behavior of a software system in a controlled way is not natively supported by most of the mainstream programming languages. This statement is strengthened in the area of ubiquitous computing, where ubiquitous devices (*e.g.*, smartphones, smartwatches, sensors) are subjects to resource constraints, which usually encourage system developers to trade software engineering principles for performance or energy considerations.

Although programming models for constrained environments [AS11; LG09] have been released more recently than most of the mainstream programming languages, they still heavily build on traditional programming principles (*e.g.*, functions, objects or procedures) and offer a limited support for modularity and runtime adaptation [Fow04; Van08; MP11]. Therefore, various component-based framework approaches have been investigated to enforce the separation of concerns and to leverage the adaptation at runtime of software systems [LW07; Sei+12; Cou+08; EHL07]. In the area of ubiquitous systems, one can cite the FiGaRo [MPA08] and MADAM [Flo+06] initiatives to deliver modular yet efficient programming abstractions that can be used to support the runtime adaptation process of embedded devices, such as wireless sensors or mobile devices. These solutions provide the basics for adjusting the structure of a ubiquitous software system according to operating conditions. Yet, such middleware solutions usually exhibit a wide diversity of conceptual and programming abstractions that prevent their wider adoption by software developers. In addition to that, these approaches do not intend to promote the development of components at large and they lack of support for assembling and controlling complex software architectures. Finally, considering the diversity of execution substrates in ubiquitous systems, software components should be sufficiently runtime-agnostic to opportunistically exploit the resources made available by the surrounding environment.

The objective of this chapter is to report on our vision and our achievements in the area of the development of elastic ubiquitous software systems. By elastic, we mean ubiquitous systems that can reshape their software architecture to accommodate the available resource in a given deployment environment and at a certain point of time. To support such an elasticity, we advocate that standard component models can be ported to support the development of embedded devices both in the small and in the large. By reusing the concepts of state-of-the-art component models, the *Service Component Architecture* (SCA) [1] in particular, we believe that developers can easily approach the development of ubiquitous software systems independently of the variety of hardware platforms to be considered. Interestingly, the adoption of such a common abstraction leverages the definition of software architectures spanning several computing nodes, from sensors to smarphones and to server-side infrastructures. By reasoning on a common architecture, distributed adaptation scenarios can be envisioned to move from user-centric adaptations to collaborative optimizations.

1.2 Contributions

Our vision is supported in this chapter by reporting on three contributions we developed along the last years. The first contribution, Remora (cf. Section 1.2.1), targets challenges for programming in the small and presents a component model for programming and reconfiguring wireless sensor nodes. Then, the second contribution, FraSCAla (cf. Section 1.2.2), targets challenges for programming in the large and describes a domain-specific language for programming software architecture patterns. Finally, the third contribution, Macchiato (cf.

[1] SCA: `http://www.oasis-opencsa.org/sca`

Section 1.2.3), focuses on challenges for programming elastic software systems and reports on a mobile actor programming model.

1.2.1 Programing Elastic Software Components in the Small

Wireless Sensor Networks (WSNs) have rapidly emerged because of their applications in real-world environments. However, WSNs differ from the conventional distributed systems in many aspects. Resource scarceness is the most important uniqueness of WSNs. Sensor nodes are often equipped with a limited energy source and a processing unit with a small memory capacity. Additionally, the network bandwidth is much lower than for wired communications and radio-based operations are the dominant energy consumer within a sensor node. These factors make the way to develop WSN applications quite critical and also different from the other existing network systems. Firstly, the existing diversities in WSN hardware and software platforms have brought the same order of diversity to programming models for such platforms [SG08]. Moreover, developers' expertise in state-of-the-art programming models become useless in WSN programming as the well-established discipline of program specification is largely missing in this area. Secondly, the structure of programming models for WSNs are usually sacrificed for resource usage efficiency, thereby, the outcome of such models is usually a piece of tangled code hardly maintainable by its owner. Finally, application programming in WSNs is mostly carried out very close to the operating system, forcing developers to learn low-level system programming models. This not only diverts the programmer's focus from the application logic, but also needs low-level programming techniques, which imposes a significant burden on the programmer.

From a software composition perspective, the way to implement WSN applications is also becoming increasingly important as today's sensor software not only consists of application and system modules, but also includes various off-the-shelf, third-party software products, such as middleware services. Ideally, such integrations should be realized through a meta-level abstraction with minimum programming effort. This, in fact, indicates the capability of a WSN programming model to facilitate the development of middleware services and their integration to target application software. The ability to tune the sensor software for a particular use-case or application domain is the other major issue in this context. As sensor nodes are typically equipped with a limited memory capacity, operating system developers need to keep the size of system modules as small as possible in order to preserve enough memory space for application modules, and they also have to ensure the portability of system software to various sensor platforms. This mostly leads to software artifacts with either degraded functionality not satisfying all end-user expectations, or suffering from the lack of modularity and maintainability. One solution to tackle this problem is to consider the operating system as a collection of well-defined services deployable on a minimized kernel image so that the programmer has the ability to involve only application-required system services in the process of software installation. Therefore, this can bring a significant efficiency to resource usage in sensor nodes by avoiding installing a single monolithic operating system for any application.

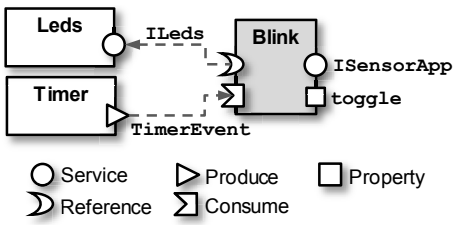
1.2.1.1 The Remora Component Model

Component-based programming provides an high-level programming abstraction by enforcing interface-based interactions between system modules and therefore avoiding any hidden interaction via direct function call, variable access, or inheritance relationships. This abstraction rather offers the capability of black-box integration of modules in order to simplify configuration and maintenance of software systems. Module reusability and provision of standard API are some other advantages of adopting *component-based software development* [Szy02; Bac+00]. Although using this paradigm in earlier embedded systems was relatively successful [Van+00; Gen+02; Han+04; Plš+08], most of the efforts in the context of WSNs remain inefficient or limited in the scope of use. The TinyOS programming model, named NesC [Gay+03], is perhaps the most popular component model for WSNs. Whereas NesC eases WSN programming, this component model remains tightly bound to the TinyOS platform. Other proposals, such as OpenCom [Cou+08] and THINK [Fas+02], are either too heavyweight for WSNs, or not able to support event-driven programming, which is of high importance in WSNs.

In this section, we therefore report on some results on Remora, a lightweight component model designed for resource-constraint embedded systems, including WSNs [Tah+10; Tah+11b]. The strong abstraction promoted by this model allows a wide range of embedded systems to exploit it at different software levels from *Operating System* (OS) to application. To achieve this goal, Remora provides a very efficient mechanism for event management, as embedded applications are inherently event-driven. WSN applications in our approach are built out of components conforming to the Remora component model. The key design principles of Remora include:

XML-based component description. The first design goal emphasizes simplicity and generality of the technique for describing Remora components. In Remora, we therefore adopt XML technologies to describe

components. The basis for the XML schema we defined is the *Service Component Architecture* (SCA) notations in order to provide a uniform component model covering components from sensors to the Internet, as well as to accelerate standardization of component-based programming in WSNs. As SCA was originally designed for large-scale systems-of-systems [OSO], REMORA extends SCA with its own architectural concerns to achieve realistic component-based programming in WSNs. The component description contains the specifications of its *services*, *references*, *producers*, *consumers*, and *properties* (cf. Figure 1.1a). A *service* can expose a REMORA interface, which is a separate XML document specifying the functions provided by the component, while a *reference* indicates the operations required by the component as an interface (cf. Listing 1.1b). Likewise, a *producer* identifies an event type generated by a component, whereas a *consumer* specifies component's interest on receiving a particular event.



(a) A simple REMORA-based application.

```
1 <componentType name="app.BlinkApp">
2   <service name="iSensorApp">
3     <interface.remora name="core.boot.api.ISensorApp"/>
4   </service>
5   <reference name="iLeds">
6     <interface.remora name="core.peripheral.api.ILeds">
7   </reference>

9   <property name="toggle" type="xsd:short">0</property>

11  <producer/>
12  <consumer operation="timerExpired">
13    <event.remora name="aTimeEvent" type="TimerEvent"/>
14  </consumer>

16  <libraries>
17    <include name="stdio" type="SystemLib"/>
18  </libraries>
19 </componentType>
```

(b) SCA description of the Blink component.

Figure 1.1: Remora application

C-like language for component implementation. REMORA components are written in a C-like language enhancing the C language with features to support component-based and structured programming. The other objective in this enhancement is to attract both embedded systems programmers and PC-based developers towards high-level programming in WSNs. The component implementation is a C-like program containing three types of operations: *i)* operations implementing the component's services, *ii)* operations processing events, and *iii)* component's private operations. Listing 1.2a presents the excerpt of the Blink implementation. This C-like code implements the single function of the interface `ISensorApp` (`runApplication`) and handles `TimerEvent`—whose structure is described in Listing 1.2b—within the `timerExpired` function. In the `runApplication` function, we specify that the `TimerEvent` generator (`aTimeEvent.producer`) is configured to generate periodically `TimeEvent` every three seconds. The last command in this function is also to notify the `TimerEvent` generator to start time measurement. When time is expired, the timer sets the attributes of `aTimeEvent` (*e.g.*, `latency`) and then the Remora framework calls the `timerExpired` function.

```
1 void runApplication(){
2     printf("--- Starting Blink Application ---");
3     short periodic = 1;
4     this.aTimeEvent.producer.configure(3*CLOCK_SECOND,
5                                        periodic);
6     this.aTimeEvent.producer.start();
7 }

9 void timerExpired(){
10    if (this.toggle == 0){
11        this.iLeds.onLeds(LEDS_RED);
12        this.toggle = 1;
13    } else {
14        this.iLeds.offLeds(LEDS_RED);
15        this.toggle = 0;
16    }
17    printf("Time elapsed after interval: %d",
18           this.aTimeEvent.latency);
19 }
```

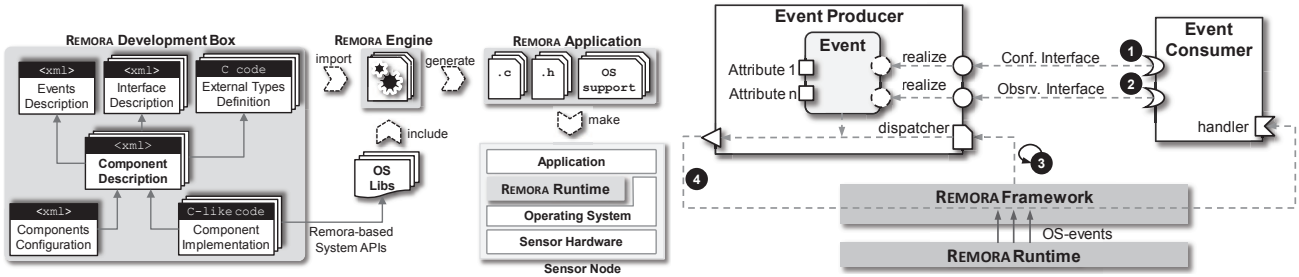
(a) C-like implementation of Blink component.

```
1 <events>
2   <remora-events>
3     <event-type name="TimerEvent"
4                 castType="unicast"
5                 observation="manual">
6       <attribute name="latency" type="xsd:int"/>
7       <configInterface>
8         <operation name="configure">
9           <in name="interval" type="xsd:int"/>
10          <in name="periodic" type="xsd:short"/>
11        </operation>
12      </configInterface>
13    </event-type>
14  </remora-events>
15  <!-- add other application event types here -->
16  <os-events>
17    <!-- describe OS-events here -->
18  </os-events>
19 </events>
```

(b) XML description of events in REMORA.

Figure 1.2: Remora application

OS abstraction layer. The REMORA component framework is integrated with the underlying operating system through a well-defined OS-abstraction layer. This thin layer can be developed for various WSN operating systems supporting the C language, such as Contiki. This feature ensures the portability of REMORA components towards different OSs. The abstraction provided by REMORA becomes more valuable when the component framework is easily configured to reuse OS-provided features, such as event processing and task scheduling. Figure 1.3a illustrates the four main phases of an application deployment. The REMORA development box encompasses artifacts supporting component specification. Events description and components configuration are used to describe system events and components assembly, respectively. Components and interfaces are also described in separate XML documents, one for each. External types are a set of C header files containing application's type definitions. The last group of elements in this box are C-like implementation files of components in which OS libraries may be called through a set of System APIs implemented by REMORA runtime components. Note that there is no hard-coded dependencies between REMORA implementers and the native API of the underlying OS (*e.g.*, Contiki) to ensure portability of REMORA components towards different OSs. In the next phase, the REMORA engine reads the elements of the development box and also OS libraries in order to generate the REMORA framework including the source code of components and OS-support code. Then, application object file will be created through OS-provided facilities and finally deployed on sensor nodes.



(a) Development process of REMORA-based applications. (b) Event management mechanism in REMORA.

Figure 1.3: Remora framework

Event handling. Event-driven programming is a common technique for programming embedded systems as memory requirements in this programming model is very low. Besides the support for events at the operating system level in embedded systems, we also need to consider event handling at the application layer. REMORA therefore proposes an high-level support of event generation and event handling, which makes it one of the key features of our contribution. In particular, REMORA achieves this goal by reifying the concept of event as a first-class architectural element simplifying the development of event-oriented scenarios. Events in our framework are categorized into two classes: *application-events* and *OS-events*. Application-level events are generated by the REMORA framework (like `TimerEvent` in the Blink application), while the latter are generated by the sensor operating system. In other words, the only difference of these two types is the source of event generation. To process OS-events at the application level, the REMORA runtime features mechanisms to observe OS-events, translate them to corresponding application-level events, and publish them through *OS-event producer* components. During the first two steps of Figure 1.3b, the event consumer can configure event generation and control event observation by calling the associated interfaces realized by the event producer component (cf. lines 4–6 in Figure 1.2a). Step 3 is dedicated to *polling* the producer component to observe event occurrence. The event producer is polled by the REMORA framework through a *dispatcher* function in the producer. The REMORA runtime listens only to application-requested OS-events, and delivers the relevant ones to the framework. The REMORA framework then forwards the event to the corresponding *OS-event producer* component by calling its dispatcher function—*e.g.*, `user_button` is a Contiki-level event that should be processed by the REMORA component UserButton. This component then generates an high-level UserButtonEvent and publishes it to the REMORA framework. Finally, in step 4, upon detecting an event in the dispatcher function, the producer component creates the associated event, fills the required attributes, and publishes it to the REMORA framework. The framework in turn forwards the event to the interesting components by calling their event handler function.

1.2.1.2 The RemoWare Reconfiguration Model

As already mentioned, WSNs are being extensively deployed today in various monitoring and control applications by enabling rapid deployments at low cost and with high flexibility. However, the nodes of a WSN are often deployed in large number and inaccessible places for long periods of time during which the sensor software, including *Operating System* (OS) and application, may need to be updated for several reasons. First, a deployed

WSN may encounter sporadic faults that were not observable prior to deployment, requiring a mechanism to detect failures and to repair faulty code [Yan+07; Cao+08]. Second, in order to maintain long-lived WSN applications, we may need to remotely patch or upgrade software deployed on sensor nodes through the wireless network. Third, the requirements from network configurations and protocols may change along the application lifespan because of the heterogeneity and distributed nature of WSN applications [Cos07]. Therefore, due to storage constraints, it is infeasible to proactively load all services supporting heterogeneity into nodes and hence requirement variations are basically satisfied through updating the sensor software. Finally, the increasing number of WSN deployments in context-aware environments makes *reconfiguration* and *self-adaptation* two vital capabilities, where a sensor application detects internal and external changes to the system, analyzes them, and seamlessly adapts to the new conditions by updating the software functionalities [Tah+08b; RC03].

When a sensor network is deployed, it may be very troublesome to manually reprogram the sensor nodes because of the scale and the embedded nature of the deployment environment, in particular when sensor nodes are difficult to reach physically. Thus, the most relevant form of updating sensor software is remote multi-hop reprogramming exploiting the wireless medium and forwarding the new code over-the-air to the target nodes [WZC06]. The early solutions in this field focused on upgrading the *full software image*. Although these provide maximum flexibility by allowing arbitrary changes to the system, they impose a significant cost by distributing wirelessly a large monolithic binary image across the network. Deluge [HC04] is one of the most popular approaches in this category, offering a functionality to disseminate code updates for applications written for TinyOS [Lev+04]. Therefore, in this section, we focus on component-based reprogramming in WSNs and reconsider Remora in order to enable compositional component reconfiguration [McK+04] in WSNs. The dynamicity of Remora is achieved by the principles of *in-situ reconfigurability*, which refer to fine-grained delimitation of static and dynamic parts of sensor software at design-time in order to minimize the overhead of post-deployment updates. This also enables programmers to tune the dynamicity of the WSN software in a principled manner and decide on the reconfigurablility degree of the target software in order to tune the associated update costs. The run-time system supporting the in-situ reconfiguration of Remora components is called RemoWare.

In-situ reconfigurability A fundamental challenge in enabling component-based reconfiguration in WSNs is how to efficiently provide this feature with minimal overhead in terms of resource usage. The ultimate goal of the in-situ reconfiguration model is to address this challenge. The main principles of the *in-situ reconfiguration model* are:

1. The reconfiguration system updates only the component code, while the component state(s) can either be left unchanged or updated by the programmer. In particular, the reconfiguration model provides the programmer with the choice of either preserving the old component state(s) or updating it(them) with user-specified values when completing the reconfiguration;
2. To preserve efficiency, the reconfiguration model enables the programmer to tune the overall overhead of the reconfiguration framework according to the degree of dynamicity of the system, as well as sensor platform capabilities. This is achieved by distinguishing between reconfigurable and non-reconfigurable components during software design before compiling and deploying the final sensor software. Using this strategy, the additional overhead imposed by defining an inherent static component as a dynamic component will be eliminated.
3. The scope of reconfiguration for a given component is limited to its internal design while preserving its architectural description. This constraint implies the in-situ reconfiguration model does not support interface-level reconfiguration. Hence, there is no need to manage a dynamically typed language at run-time, which is indeed heavyweight for sensor platforms.
4. The reconfiguration model supports adding of new components and removing existing ones. In such cases, the *binding/unbinding* mechanism for loaded/unloaded components imposes only a minimal *fixed* overhead to the system, regardless of the size of the application and the number of reconfigurable components.

In Remora, a component conforming to the above principles is referred to as an *in-situ reconfigurable component*.

Neighbor-aware binding. Dynamic binding is one of the primary requirements of any component-based reconfigurable mechanism. A major hurdle in porting state-of-the-art dynamic component models to WSNs is their binding-support constructs, which are memory-consuming, such as MicrosoftCOM's vtable-based function pointers [MIC93], CORBA Component Model's POA [OMG06], OpenCom's binding components [Cou+08], and Fractal's binding controller [Bru+06]. This is due to the fact that most of those component models are essentially designed for large-scale systems with complex requests, such as interoperability, streaming connections, operation interception, nested components, and interface introspection.

In REMORA, we aim at reducing the memory and processing overhead of the binding model, while supporting all the basic requirements for a typical dynamic binding. To this end, we propose the concept of *neighbor-aware binding* based on the principle of in-situ reconfigurability. The neighbors of a component are defined as components having *service-reference* communications or vice-versa with the component. Neighbor-aware coupling therefore refers to identifying the type of bindings between a component and its neighbors based on the reconfigurability of each side of a specific binding, as well as the service provision direction.This is viable if the reconfiguration system can distinguish static modules from dynamic modules and the programming abstraction can provide meta-information about the interaction models between modules. These issues are jointly addressed by REMOWARE, where it relies on *component-based* updates to tackle the latter and proposes a partial-dynamic/partial-static configuration model for software modules to achieve the former.

The physical locations of services provided by a dynamic component are not stable and change whenever the component is reconfigured. Therefore, to invoke a service function provided by a reconfigurable component (*dynamic functions*), either from a static or from a dynamic component, we need to provide an *indirect calling* mechanism. To do that, the direct invocations within the caller component should be replaced by a *delegator*, forwarding function calls to the correct service address in the memory. This delegator retains the list of dynamic service functions along with their current memory addresses in the *Dynamic Invocation Table* (DIT).

Component addition and removal Obviously, adding new components and removing existing ones are two basic requirements to a component-based reconfiguration framework. For instance, we may decide to unload a data logger component due to the energy overhead caused by writing log data in the external flash memory. The main concern in removing an existing component is how to handle communications referring to a removed component's services. Since a removable component should be defined as a dynamic component, its functions are dynamic and indexed in the DIT. Therefore, after removing a component we can simply map its dynamic functions to a *dummy global function* with an empty body. This function, included in the REMOWARE libraries, is designed to prevent fatal errors occurring due to the absence of removed components' functions. In this way, other components dependent to an unloaded service can continue to run without exceptions.

For newly added components, the same process of dynamic linking, discussed above, can be applied, while the main issue being how to bind the new component to components that include invocations to its services. Note that prior to loading the new component, such invocations are redirected to the above dummy global function. The same solution proposed for component removal can be used for component addition, but vice-versa. When a new component is uploaded, the DIT is searched for the name of all functions of the added component. If an entry is found, the corresponding function address is corrected. Otherwise, a new entry will be inserted to the DIT with the name and address of the new function for future binding.

In-situ program memory allocation In contrast to the full software image updating model, *modular upgrade* solutions need a reliable code memory allocation model in order to avoid memory fragmentation and minimize the wasted space in the memory. Unfortunately, this issue has received little attention in the literature. Most module-based reconfiguration models either omit to consider dynamic memory allocation in sensor nodes or rely on the capabilities of OSs for dynamic memory allocation. For instance, CONTIKI pre-allocates only 'one' contiguous fixed-size memory space at compile time in the hope that the updated module fits this pre-allocated space. We therefore propose an hybrid dynamic program memory allocation model based on the notion of *in-situ memory allocation* and *first-fit* strategy. In-situ memory allocation indicates that the updated component is tentatively flashed in its original code memory space instead of being moved to another block of memory. The immediate concern of this model is how to fit the updated component to its original memory space when it is larger than the previous version. This issue is addressed by the *pre-allocated* parameter—the extra memory that should be pre-allocated for each dynamic component. REMOWARE sets a default value for this parameter (*i.e.*, 10% in the current version) applied to all reconfigurable components, while the developer can update this value for all components or a particular one. If an updated component cannot fit in its original memory space (including pre-allocated area), the first-fit strategy comes into play by simply scanning the free spaces list until a large enough block is found. The first-fit model is generally better than best-fit because it leads to less fragmentation.

We expect this hybrid approach to be an efficient program memory allocation model due to the following reasons. Firstly, updated components do not differ significantly from the older versions. Hence, on average there will be a minor difference only between the size of the updated one and the original component. Secondly, the developer has the ability to feed the memory allocation model with information which is specific to a particular use case and specifies a more accurate tolerance of dynamic component size.

Retention of component state Retaining the state of a component during reconfiguration is of high importance, *e.g.*, for a network component buffering data packets, it is necessary to retain its state before and

after the reconfiguration. When considering state retention during the reconfiguration, it is very important to provide a *state definition* mechanism leading the programmer to a semantic model of global variables definition. In typical modular programming models the programmer may define global variables that are never required to be global (stateless variables). Therefore, the reconfiguration system is forced to retain all global variables (including stateless ones), resulting in additional memory overhead. In contrast, introducing the concept of state in component models like REMORA prevents the programmer from defining unnecessary global variables, leading to less memory overhead when the reconfiguration system implements state retention.

When an updated version of a component is being linked to the system, REMOWARE retains the previous version's state properties in the data memory and transfers them to the state properties of the updated version. This is feasible as the set of component properties never changes and the state structure of the updated component is therefore the same as previous versions. One may need to reset the value of component properties or assign a new set of values to them when reconfigured. REMOWARE addresses this by calling the predefined onLoad function—implemented by the updated component—whenever the component is successfully reconfigured. It means that if the programmer intents to set new values to component properties after the reconfiguration, he/she must implement the onLoad function.

Quiescent state for reconfiguration. Reaching a quiescent state (*i.e.*, temporarily idle or not processing any request) before initiating a component reconfiguration is a critical issue in any reconfiguration mechanism. REMOWARE, as any middleware framework, runs over the OS and therefore this issue is addressed depending on the process/task management model of the sensor OS. For example, the current REMOWARE implementation relies on CONTIKI's process management system to run the reconfiguration task. CONTIKI processes adopt a run-to-completion policy without preemption from the scheduler. Therefore, yielding of a process is explicitly requested from the functional code and when the yielding point is reached by a process, a quiescent state is also reached for the component executed within this process. As a result, the reconfiguration process is *atomic* in the sense that it cannot be preempted until completion. Using REMOWARE on OSs that offer a preemptable task model needs a careful consideration of the safe state problem.

Extensive evaluations of the REMORA component model and the REMOWARE reconfiguration framework are detailed in [TRE10; Tah+11b; Tah+11a] and [Tah+08a; Tah+09a; Tah+13], respectively.

1.2.2 Programing Elastic Software Architectures in the Large

The above section has demonstrated how software components can be applied in the small to design and implement constrained ubiquitous systems, like wireless sensor network applications, while promoting software engineering principles, such as reusability and separation of concerns. To scale out these principles, programming component-based systems in the large requires to compose software components into software architectures. However, most of the state-of-the-art *Architecture Description Languages* (ADL) that are used nowadays to build such software architectures mostly provide a declarative syntax [BR00; DHT01; Kha+01; BCL12]. Unfortunately, adopting such a declarative approach does not scale with the number of components and rather hinders the elasticity of software architectures, by encouraging software architecture to clone architectural descriptions due to the lack of support for capturing and tuning reusable software architectural patterns.

We therefore believe that software architects need a more flexible approach to cope with the definition of domain-specific architectures by leveraging general purpose ADLs. In this section, we therefore introduce the FRASCALA framework as an adaptive architectural framework that can be used to scale ADLs in order to catalyze the definition and to improve the reliability of software architectures—*i.e.*, FRASCALA is a framework for building "à la carte" ADLs. In particular, we promote a layered approach to isolate the definition of the architectural models from the language statements used to describe software architectures.

1.2.2.1 Elastic Architecture Patterns

As already introduced, ubiquitous software systems are characterized by the deployment of software components on a large number of constrained devices (*e.g.*, sensors, smartphones), which are expected to collaborate more or less directly to support the execution of a ubiquitous application. More specifically, one can observe that programming component-based ubiquitous systems at large can lead to the design of redundant, and potentially complex, pieces of architectures. Furthermore, such pieces of software architectures may require to be adapted at design-time or at runtime depending the conditions of execution. We therefore propose to implement elastic patterns as canonical forms of architecture fragments that can be stretched to accommodate the application requirements. In particular, the key limitations of existing approaches to design elastic software architecture can be summarized as follows:

Style lock-in. Architectural patterns are usually imposed by styles like *pipe and filters* [MM03], *component-based* [CSS11], or *layered architectures* [Abi+05a]. The adoption of an architectural style therefore congeals the concepts that can be used by the architect to design the application. However, by constraining the vocabulary and the semantics of the architectural constructions, the architect has to map domain-specific concepts to general purpose architectural constructions. Incidentally, this mapping breaks the traceability between domain requirements and architectural constructions, potentially diluting domain-specific knowledge (*e.g.*, constraints) into general-purpose architectural constructions.

Lack of reuse. Software architectures, like any software artifacts, can be subject to be (partly) reused from systems to systems. However, the architecture description languages provide a limited granularity for reuse. For example, the component-based architecture style promotes the component (atomic or composite) as a unit of package and reuse. Nonetheless, partial assemblies (so called fragments) can be worth to extract, capitalize and reuse whenever they address a common issue in a family of systems.

Lack of extensibility. Existing architecture description languages can often be extended with new keywords and constructions to address a specific concern. However, the design of such a language is a tedious process which requires the specification of a grammar and the implementation of the associated interpreter to convert the context policy descriptions into software architectures. Any modification to be integrated into this DSL would require to reconsider the design of the grammar as well as the implementation of the interpreter, which can be considered as cumbersome in a prototyping phase.

Lack of elasticity. Existing software architecture description languages focus on the description of the initial structure of a system with a limited support for foreseen evolutions. Even if some runtime platforms are equipped with middleware controller to adjust the architecture along time, such uncontrolled evolutions can lead to a strong divergence with regards to initial requirements.

To address the above issues, our proposition mainly advocates to move from *declaring* a software architecture to a more *imperative* style to foster the adoption of domain-specific architecture languages. In particular, we believe that, by doing so, one can quickly benefit from the programming support of general purpose languages to efficiently implement the concepts of architecture types [MM03], architectural constraints [GHR07], architectural patterns and architecture fragments, and even introduce domain-specific architecture languages. Most of these concepts are not new and some architecture languages partly address these features in the literature [ACN02; Abi+05a]. Our proposition therefore intends to conciliate these features and foster the prototyping of new constructions to improve the quality of software architectures. We therefore support the idea that the definition of elastic software architectures has to build on a flexible architecture description language.

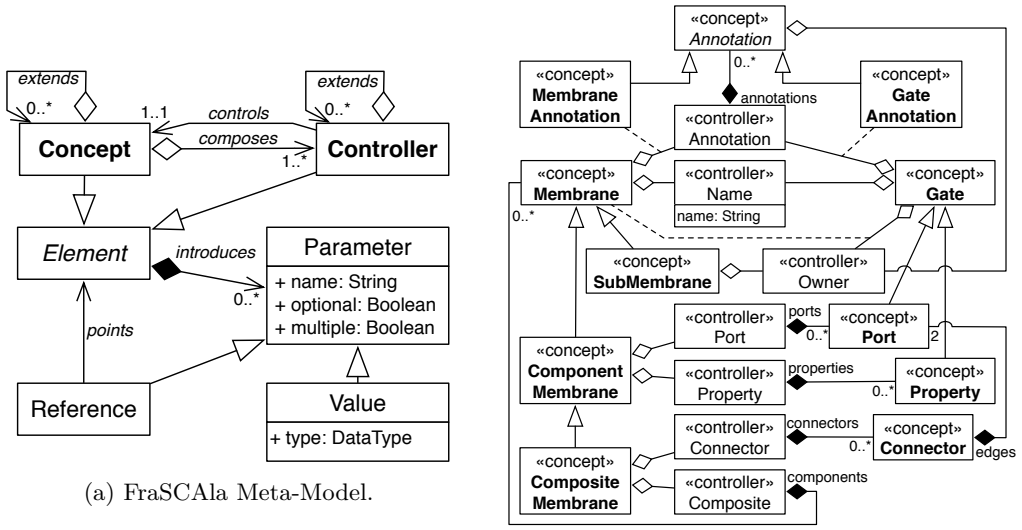
1.2.2.2 FraSCAla: An Architecture-Specific Language

FraSCAla is a framework for prototyping the definition of *Architectural Description Languages* (ADL). The objective of FraSCAla is to leverage the construction of ADLs by providing an homogeneous environment to introduce new architectural concepts according to domain-specific requirements. Although it mostly targets prototyping activities, FraSCAla promotes a structured approach to build ADLs by adopting an incremental design approach.

Design First, FraSCAla defines a *meta-model* to introduce the essential concepts required to prototype an ADL (cf. Figure 1.4a). In particular, a Concept is used to introduce a new architectural element in the model, and is structured as a composition of Controllers. A Controller can be considered as a mixin [Bru+06] isolating a specific concern of the Concept. The identification of Controllers fosters their reuse by several Concepts and it enables a Concept to be assembled "*à la carte*". This means that this meta-model composes the conceptual kernel of FraSCAla and can be used to prototype any ADL. For example, instances of this meta-model can be used to describe a specific architectural style associated to an ADL.

In the context of this section, we focus on the definition of an ADL conforming to the principles of *Component-Based Software Engineering* [CSS11]. In particular, we demonstrate that FraSCAla can be used to describe a generic component model, which can be incrementally refined towards a more concrete component model and ultimately a domain-specific component model. Figure 1.4b depicts the generic component model we designed using the stereotypes introduced by the FraSCAla meta-model. This model is strongly inspired from the Fractal component model, as described in [Bru+06].

Following this process, Each FraSCAla *model* can be further refined to fit the specificities of a given component model (*e.g.*, *Service Component Architecture* standard [Bei07]). Although not imposed by our approach, the definition of such a generic component model provides an architectural pivot, which can be used to automatically instrument concrete component models (*e.g.*, introspection, verification).



(a) FraSCAla Meta-Model.

(b) Component-based Meta-Model.

Figure 1.4: FraSCAla design

Implementation Model-Driven Engineering technologies like the *Eclipse Modeling Framework* (EMF) [Ste+09] provide advanced frameworks for describing meta-models, models as well as their graphical or textual syntaxes. However, these frameworks tend to suffer from their intrinsic complexity and do not provide much elasticity to support the definition of modular ADLs. In particular, the definition of a new concept usually requires to generate the associated classes and to modify the concrete syntaxes in order to support this concept. However, such invasive modifications conflict with our objective to build ADLs "*à la carte*" by composing architectural concepts on-demand. Furthermore, the definition of EMF models, as well as concrete syntaxes, introduces a steep learning curve, which is not expected when prototyping architectural constructions. Another direction could be to encode FraSCAla models with an executable meta-modeling language, such as Kermeta[2]. Nonetheless, extending the Kermeta language with embedded DSL constructions seems to be currently a complex engineering task. For these reasons, we investigated another approach, which consists in exploiting an advanced general-purpose programming language like Scala [Ode06; OZ05] not only to implement the FraSCAla models, but also to provide a textual syntax and to describe software architectures. This reflective approach provides an homogeneous environment to quickly define and exploit new software architectural constructions. In particular, FraSCAla benefits from Scala's support for type inference, genericity, implicit definition, trait, closures, as well as native XML support.

Listing 1.5b[3] therefore reports a Scala code excerpt of three of the component-based controllers we previously introduced. Concretely, each Controller defined in the model is encoded as a `trait`, which is a construction provided by Scala to isolate a specific concern (cf. lines 1, 5 & 10). Traits map a Parameter in the model to a variable (using the keyword `var`, cf. lines 2, 7 & 12), while generic concepts can be mapped to an abstract type (using the keyword `type`, cf. lines 6 & 11). The resulting Scala implementation defines the foundations of a component model, which remains agnostic from any technology.

Once a target model is defined, FraSCAla introduces the associated *language* by constraining this model with a concrete syntax. This concrete syntax introduces the keywords of the ADL as well as the associated operations it supports for describing software architectures. The reader can refer to [RM12] or our GitHub repository [4] for further details on this mapping.

Illustration As a matter of illustration, we use the *Service Component Architecture* (SCA) [Bei07; Sei+12] as a target architecture style that is used to deploy distributed systems. SCA is a set of OASIS's specifications for building distributed applications based on *Service-Oriented Architecture* (SOA) and CBSE principles. In Figure 1.6 we start by reporting on equivalent software architecture descriptions based on the XML-based descriptors defined by the SCA standard (cf. Figure 1.6a) and the FraSCAla-based approach we promote (cf. Figure 1.6b). Although XML-based description supports extensions to be defined, the definition of such extensions is bound to variability points allowed by the core schema. Furthermore, XML-based descriptions

[2] `http://www.kermeta.org`

[3] In all listings, texts in red are Scala or FraSCAla keywords, texts in blue are FraSCAla type and variable identifiers, while texts in green are string constants.

[4] FraSCAla: `https://github.com/rouvoy/frascala`

```scala
trait NameController {
  var name: String = _
}

trait OwnerController {
  type OWNER
  var owner: OWNER = _
}

trait AnnotationController {
  type ANNOTATION <: Annotation
  var annotations = Set[ANNOTATION]()
}
```

(a) Component-based Controllers.

```scala
trait Gate extends NameController
  with OwnerController with AnnotationController

trait PortAnnotation extends Annotation {
  type OWNER <: Port
}

trait Port extends Gate {
  type ANNOTATION <: PortAnnotation
  type OWNER <: PortController
}

trait MembraneAnnotation extends Annotation {
  type OWNER <: Membrane
}

trait Membrane extends NameController
 with AnnotationController {
  type ANNOTATION <: MembraneAnnotation
}

trait ComponentMembrane extends PortController
 with PropertyController
```

(b) Component-based Concepts.

Figure 1.5: FraSCAla framework

are purely declarative, which requires an appropriate assembly engine to parse and process the definition to implement the semantics of the model. In this example, we therefore show how we can achieve a similar expressivity by using an embedded DSL to leverage current limitations of declarative ADLs. We also show that our approach remains compatible with these declarative ADLs by providing mapping facilities to dump a FraSCAla definition into a standard SCA one. Using FraSCAla, 34 lines of code in Scala are sufficient to describe both the architecture and the implementation of an HelloWorld application, which is equivalent to an ArchJava description [Abi+05a]. The noise introduced by the Scala notation remains limited (37 characters out of the 787 for the whole definition) compared to the flexibility brought by this approach.

```xml
<composite name="Helloworld"
     xmlns="http://www.osoa.org/xmlns/sca/1.0">
  <property name="counter" value="3"/>

  <component name="Server">
    <property name="counter" value="$counter"/>
    <service name="s">
      <interface.java interface="Service"/>
    </service>
    <implementation.java class="Server"/>
  </component>

  <component name="Client">
    <property name="header" value="&gt;&gt; "/>
    <service name="r">
      <interface.java interface="Runnable"/>
    </service>
    <reference name="s" target="Server"/>
    <implementation.java class="Server"/>
  </component>

  <service name="run" promote="Client/run"/>
</composite>
```

(a) Standard SCA description.

```scala
object Helloworld extends Composite("Helloworld") {
  val cnt = property[Int]("counter") is 3

  val srv = new component("Server") {
    property[Int]("count") from cnt
    val s = service("s") exposes Java[Service]
  } uses Bean[Server]

  new component("Client") {
    property[String]("header") is ">> "
    val r = service("r") exposes Java[Runnable]
    reference("s") targets srv.s
  } uses Bean[Client]

  service("run") promotes components("Client").r
}
```

(b) FraSCAla description.

Figure 1.6: FraSCAla SCA style

Architecture patterns The last feature of FraSCAla consists in supporting the definition of *architectural patterns*. Software architecture patterns are a powerful construction for isolating reusable fragments of software architecture descriptions. By enabling the definition of architectural patterns, FraSCAla captures domain-specific architectures as advanced patterns. These patterns are then included in the base ADL as new statements, which can be immediately used to ease the definition of a complex software architecture.

SCA Intent Pattern As a first example of architectural pattern, we propose to reflect the concept of SCA *intent* [Bei07] as an SCA component. In SCA, an intent isolates a crosscutting concern, which can then be woven into one or several ports in order to intercept service invocations. This mechanism is expected to be

used for implementing non-functional services, such as transaction demarcation or authentication. However, the implementation of such capabilities usually requires the intent to interact with third part services, such as a transaction service or a key store. Therefore, an elegant way for describing an intent consists in adopting a reflective approach and to use SCA components to describe the intent itself [Sei+12].

Listing 1.7a therefore illustrates the definition of a new concept `intent`, which is defined as a composite component promoting a specific service. The implementation of the intent component (whose definition is mentioned as a parameter of the intent) is therefore expected to conform to the interface `IntentHandler` specific to the FraSCAti framework. However, the technology used to implement this intent is left open by the definition of the architectural pattern in order to accommodate the different programming languages that can be used to realize an intent.

```
1 case class intent(id: String,impl:ScaImplementation)
2   extends Composite(id) {

4  val comp = new component(id+"-intent") {
5    import org.ow2.frascati.tinfi.api.IntentHandler
6    service("intent") exposes Java[IntentHandler]
7  } uses impl

9  service("intent") promotes comp.services("intent")
10 }
```

(a) Intent Pattern.

```
1 object LoggedHelloworld extends Helloworld {
2  val log = intent("Logger", Script("logger.py"))

4  components("Client") weaves log
5  components("Server").services("s") weaves log
6  weaves(log)
7 }
```

(b) Example of Logging Intent.

Figure 1.7: Intent Pattern in SCA

Once defined, the intent pattern becomes available as a new keyword of the ADL and can be directly used to leverage the definition of more complex architectures. Listing 1.7b provides an example of architecture extension, which uses a legacy architecture definition (described in Listing 1.6b) to introduce a logging intent implemented in Python (cf. line 2). An intent can be declared once and reused in several places. In this example, the logging intent is woven within all the components of the composite component including services and references (cf. line 4), as well as all the services of the surrounding composite component (cf. line 5). Here one could note that the parameters given to the `foreach` method of `Set` are anonymous functions. From this definition, the FraSCAla framework is able to produce the set of artifacts required to deploy this architecture, including not only the architecture descriptor for the extended HelloWorld application, but also the one for the intent definition.

Delegation Chain Pattern Another example of architectural pattern, which is commonly used in the literature, is the component-oriented version of the delegation chain design pattern [Abi+05b; Gam+94]. This architectural pattern consists in chaining a list of components that provide and require compatible ports. While this kind of architectural pattern is relatively difficult to describe with a declarative approach, FraSCAla leverages functional programming to ease the definition of such parametric patterns. Listing 1.8a therefore reports on the definition of the delegation chain in FraSCAla. This pattern takes a list of component implementations as a parameter. The pattern automatically builds and composes the components by connecting the first component to the second and so on.

```
1 trait DelegationChainPattern extends Composite {
2   def delegate(id:String, itf:ScaInterface,
3                impls:List[ScaImplementation]):
4                List[component] =
5     impls match {
6       case Nil => Nil
7       case impl :: tail => {
8         val chain = delegate(id, itf, tail)

10        new component(id+"-proxy"+tail.size) {
11          service(label) exposes itf
12          val del = reference(id+"-delegate")
13          del targets chain.head.services(id)
14        } uses impl) :: chain
15 } } } }
```

(a) Delegation Chain Pattern.

```
1 object ChainedHelloworld extends Helloworld
2  with DelegationChainPattern {
3   val code = List(Bean[MessageFilter], Bean[Decorator])
4   val chain = delegate("s", Java[Service], code)

6   wire(components("Client").references("s"),
7        chain.head.services("s"))
8   wire(chain.last.references("s-delegate"),
9        components("Server").services("s"))
10 }
```

(b) Example of Delegation Chain.

Figure 1.8: Delegation chains in SCA

Once defined, the resulting architectural pattern can be used within the target architecture as described in Listing 1.8b. This code excerpt extends the Helloworld architecture we defined in Listing 1.6b to include a

delegation chain that filters and decorates the messages that are exchanged between the client and the server. The value `chain` is used to store the list of components that are created by the function `delegate()`. As the components of the delegation chain are automatically linked, the architect does only have to connect the first and the last component of the delegation chain to the client and the server components, respectively.

Discussion. By exploiting the capabilities of the Scala programming language, FraSCAla supports the definition of parametric architecture templates that can be used to isolate repetitive patterns. The two examples we provide illustrate two different approaches to define reusable architectural patterns. The definition of the delegation chain as a function is recommended when the concepts do not need to be extended and/or the pattern is a compound result (a list of components in the case of the delegation chain). The definition of the intent as a composite component is rather advised when the architect is interested in specializing a concept to introduce dedicated attributes or functions. Furthermore, the delegation chain can be used to apply the pipe and filter architectural style in a component-based software architecture, which is not supported by the state-of-the-art approaches, thus breaking the style lock-in we previously identified. Using software architecture patterns based on this delegation chain pattern automatically stretch according to the number of components to be included in the chain and the architect does not have to check the consistency of the description manually. Finally, software architecture patterns defined with FraSCAla can embed domain-specific verifications (*e.g.*, the delegation chain should be made of 10 elements at most) and rules (*e.g.*, connect the provided and required ports upon insertion of a new element), which is one of the ongoing work in this area.

1.2.3 Programing Elastic Mobile Applications at Scale

Software architectures are therefore becoming pieces of artifacts that tend to continuously evolve over time depending on evolution of user requirements, and contextual conditions. The latter is a key property of mobile systems, which should not only have the capability to run on a large diversity of devices (with different hardware characteristics), but also to adjust their behavior upon availability of resources (*e.g.*, GPS signal).

While the success of mobile devices has been catalyzed by the emergence of mobile applications, the programming models that are applied in the development of these applications do not sufficiently take into account the limitations of mobile devices. In particular, most of nowadays mobile applications (*e.g.*, social networks, RSS readers, mobile shopping) are rich client applications that interact with remote servers to download application-level data and process it locally. Although mobile device capacities keep improving, this model of distribution significantly consumes both network bandwidth and device battery as application features keep evolving. This is particularly perceived by end users, whose mobile phone's lifespan hardly exceeds a day or two.

Beyond the traditional client-side (mobile apps) or server-side (web apps) models, several research approaches have already investigated the automatic partitioning of an application according to various optimization objectives (CPU usage, network consumption), but most of them are considering pre-defined deployments of applications. This issue has already been acknowledged by the scientific community [LT11], and various solutions—mostly based on code offloading approaches—have been proposed to optimize the battery lifespan and the network bandwidth [Cue+10; GRA12; Giu+09; Yan+13].

We believe that these approaches can be further extended by including pervasive resources in the partitioning process in order to opportunistically offload network-consuming tasks on remote nodes, depending on the context and the preferences of the user. This section therefore reports on the Macchiato middleware framework as a solution to develop elastic applications that can adjust their deployment upon opportunities. By bringing data processing closer to their source, Macchiato automatically reduces the bandwidth consumption and improves the battery lifespan of the device. Incidentally, the adoption of this programming model provides the opportunity to develop applications that can keep executing beyond the boundaries of the mobile device and automatically notify the end user of relevant content, for example.

1.2.3.1 Resource-Oriented Software Computations

End users are using much more kinds of connected devices than ever before. Web applications currently cope with this ever-growing diversity of terminals by developing several versions of their websites (desktop, mobile) and are even developing dedicated mobile applications. Even if frameworks, such as Cordova[5], promote a multi-devices programming model for web applications, in practice developers often have to deal with several technologies and languages in order to build their web applications, forcing them to manage different versions. Among the key limitations of the state-of-the-art, we identify the following ones:

Lack of portability In current mobile systems, the proliferation of the ways to access to the same services results in a duplication of code with the same functionalities, thus leading to additional development costs.

[5] Apache Cordova: `http://cordova.apache.org`

For each service exposed by a given provider, client components, such as websites, Android-based and iOS-based applications, will query this service using different frameworks and therefore different programming languages. From the client side, the querying logic part, which defines equivalent functionalities, is duplicated for each framework (*e.g.*, Java for Android devices, Objective-C or Swift for iOS devices and JavaScript for browsers). This duplication of languages and frameworks generally leads to many difficulties in the development and the maintenance phases of the system.

Lack of elasticity Current programming models consider at design-time a static partitioning of tasks between the client and the server sides. However, this decision—driven by the developer's assumptions—may lead to performance bottlenecks and energy leaks. The application programming model should rather release such constraints to assume that a given task—*i.e.*, a piece of code—can be executed anywhere.

Lack of serendipity Most of pervasive applications are connected to the Internet before being connected to the physical resources surrounding them. Pervasive systems should have the capability to discover and learn from their surrounding environment to propose new features to the end user.

In this work, our intuition is that pervasive applications can partly take inspiration from the map/reduce programming model by bringing data processing tasks closer to their source and exploiting the availability of surrounding computing resources. As data consumed from the Internet by these applications are generally filtered and aggregated before being displayed to the end user, the solution we introduce seamlessly uploads processing tasks whenever it detects that the bandwidth consumption can be reduced.

The approach we adopt is therefore inspired from *Computational REST* (CREST) [Ere+07] and promotes the exposure of processing tasks a as a resource that can be remotely triggered. Beyond the approach promoted by CREST, we rather design offloadable processing tasks as mobile actors that can move across execution nodes. The implementation of this approach is summarized in Figure 1.9 and described in the rest of this section.

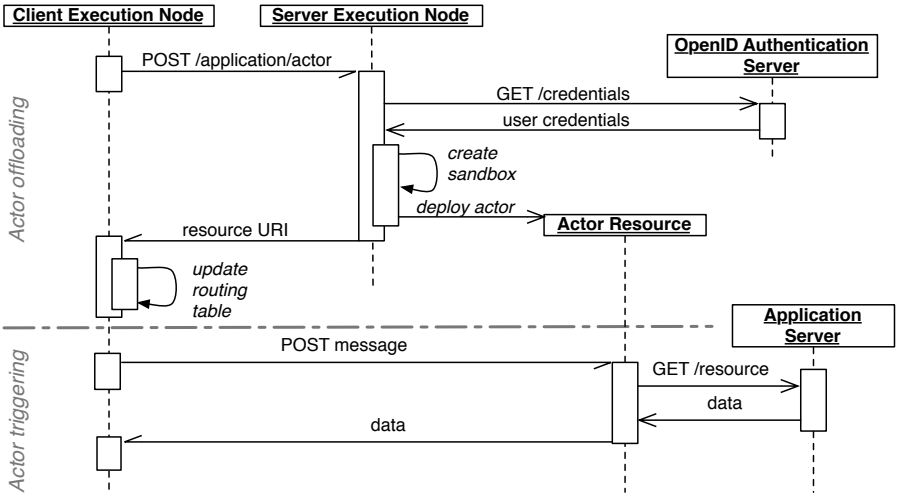


Figure 1.9: Offloading and communicating with actors.

1.2.3.2 Programing Mobile Actors with Macchiato

Development. One of the key contributions we report in this section is the implementation of an actor programming model with a scripting language. Scripting languages are becoming mainstream for the development of both server-side applications (*e.g.*, Node.js) and mobile applications (*e.g.*, PhoneGap). Indeed, their adoption is no more limited to the implementation of presentation layers, but now covers business layers using event-driven, asynchronous I/O to minimize overhead and maximize scalability. We therefore build on such programming models in order to introduce the principles of actor programming [HBS73] to improve the application elasticity. An *actor* is a first-class stateless software entity that reacts to incoming *messages*, sends messages to other actors, and even creates new actors. Each actor is identified by a unique *address*. As messages can be sent in parallel, this paradigm is concurrent by design.

Figure 1.10 illustrates part of the implementation of a custom newsreading application using the Macchiato framework. In Macchiato, an actor is an entity whose behavior is defined as a JavaScript function that takes a `message` and a `context` as parameters (lines 8–13). While the `message` encloses all the information related to the request to be processed, the `context` stores the metadata related to the execution context. Results are processed by future objects [BH77], which are automatically triggered whenever the data is made available by executing the `when` closure, thus supporting non-blocking workflows (lines 18–20). Beyond traditional actor

programming models, we borrow the concept of *binding* from component models as the communication path between two actors (line 24–25) in order to leverage the assembly of reusable actors.

```
1 macchiato.deploy(function(factory) {
2     var feeds = {
3         "Reuters" : "http://feeds.reuters.com/reuters/scienceNews",
4         "CNN"     : "http://rss.cnn.com/rss/edition_technology.rss",
5         "LeMonde" : "http://www.lemonde.fr/rss/tag/sciences.xml"
6     };

8     factory.create("heartbeat-actor").as(function(message, context) {
9         // broadcast an empty message every 10 seconds
10        context.setInterval(function() {
11            context.send({});
12        }, 10000);
13    });

15    for (var feedname in feeds) {
16        var actorId = "rss-" + feedname + "-actor";
17        factory.create(actorId).as(function(message, context) {
18            context.http.get(context.params.url).when(function(content) {
19                context.send(content);
20            });
21        }).withContext({
22            "url" : feeds[feedname]
23        });
24        factory.bind("heartbeat-actor").to(actorId);
25        factory.bind(actorId).to("display-actor");
26    }
27 });
```

Figure 1.10: Programming actors in Macchiato.

We believe that adopting JavaScript as a foundation for our middleware framework opens up for the deployment of Macchiato on a large variety of connected devices, such as mobile phones, tablet PCs, connected TVs, desktop computers, and Cloud environments. In particular, the Macchiato framework allows developers to deploy application actors from web browsers to mobile applications or application servers, without taking care of the execution environment specificities.

Atop of standard JavaScript interpreters, the Macchiato middleware implements a distributed event bus that uses Web standards, such as HTTP and WebSockets, as well as message-oriented middleware, such as Google Cloud Messaging[6], to exchange requests and data across the network (cf. Figure 1.11). The routing schemes used by the framework to distribute the messages between nodes are computed from the topology associated to the graph of actors. This approach avoids the use of expensive flooding techniques traditionally observed in distributed event based systems [BV05] to seamlessly forward events and messages within a cluster of connected devices.

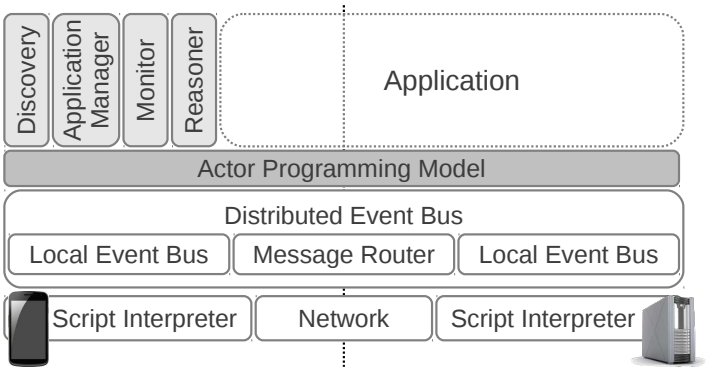


Figure 1.11: Overview of the Macchiato Middleware Architecture.

Management. Achieving the elasticity of mobile applications requires the system to be aware of its internal structure and its surrounding environment. This knowledge, which is continuously updated with information collected from monitoring activities, requires to be consolidated within a dedicated runtime model [Mor+09]. This runtime model, maintained by the mobile device, integrates the graph of actors used by the application, the deployment plan of these actors on the available nodes, and various metrics reflecting the current context of execution (*e.g.*, volume of data exchanged between actors, connectivity to other execution nodes).

[6]Google Cloud Messaging (GCM): `http://developer.android.com/google/gcm`

This runtime model is not only used by the Macchiato middleware to reason on the optimal configuration of the applications, but also to operate seamless reconfigurations on the deployed actors. Such reconfiguration operations are themselves implemented as actors located on the mobile device, which receive from the distributed event bus specific messages enclosing introspection or reconfiguration actions. A reconfiguration message is composed of a set of primitive actions: *add/remove* actors, *create/delete* a binding between actors, *offload* an actor to a remote node. This reconfiguration is executed atomically by the actor—*i.e.*, either all the reconfiguration primitives succeed or none of them is executed, leaving the system in a consistent state in case of failure of the reconfiguration process (*e.g.*, a remote execution node disappearing from the environment).

```
1 macchiato.deploy(function(factory) {
2     // removes an actor from the application
3     factory.remove("rss-LeMonde-actor");

5     // creates a new filtering actor
6     factory.create("filter-CNN-actor").as(function(msg, ctx) {
7         // filtering code
8     });

10    // injects the new actor in the application
11    factory.unbind("rss-CNN-actor").from("display-actor");
12    factory.bind("rss-CNN-actor").to("filter-CNN-actor");
13    factory.bind("filter-CNN-actor").to("display-actor");
14 }).onFail(function() { // failover code
15 }).when(function() {   // success callback
16   context.send(aMessage).to(message.actorname);
17 });
```

Figure 1.12: Supporting dynamic reconfigurations as actors with Macchiato.

We use the scripted actors as well as the runtime model not only as a foundation to develop pervasive applications, but also to develop several middleware services that are used by Macchiato to continuously optimize the configuration of the applications running on the user device. More specifically, the specificities of pervasive environments require to integrate discovery protocols, context monitoring, and continuous optimization as middleware services in order to deal with the relative instability of the execution conditions.

Discovery. Part of the runtime model maintained by Macchiato consists of a list of execution nodes that can be used to offload mobile actors. To discover these execution nodes, Macchiato includes two discovery actors based on two different techniques: a domain registry and a service discovery protocol.

On the one hand, the domain registry is used to discover the execution nodes made available from the Internet. These nodes are registered in the domain registry based on the domain they are associated to (*e.g.*, `reuters.com`). Whenever an actor submits a request to a specific URL, the Macchiato middleware searches for candidate execution nodes that are available for the associated domain. The candidate nodes for this domain are automatically added to the runtime model and attached to the related actor(s) as alternative deployment hosts. Such remote execution nodes are expected to be exposed by data providers in order to offer a better quality of service to pervasive applications exploiting their data.

On the other hand, the integration of service discovery protocols covers the discovery of execution nodes in the vicinity of the end user, such as laptop or desktop station available at home. In particular, we use the *Simple Service Discovery Protocol* (SSDP), which is part of the *Universal Plug and Play* (UPnP) standard. This protocol allows nodes to discover private execution nodes, only accessible from the local network. Such nodes are exposed to the Macchiato middleware by starting an instance. These private execution nodes are attached to the actors running on battery-powered devices to power-plugged stations in order to save battery without being impacted by the network latency.

Monitoring. For each of the execution nodes registered in the runtime model, the Macchiato middleware monitors the network quality, and injects this information into the runtime model, in order to estimate the quality of service offered by these alternative deployment hosts and to take this information into account during the optimization process. In particular, Macchiato monitors the execution of the pervasive applications to observe the messages that are exchanged between the actors. While all the actors are initially deployed on the mobile phone, some of them are interacting with remote servers. The bindings identified in the runtime model are therefore annotated with the volume of data exchanged between actors as well as with remote services. Figure 1.13 depicts a graphical representation of such a data exchange graph automatically built and updated by the Macchiato middleware services.

While the base layer of the runtime model reflects the software architecture of the pervasive application, *annotations* are used to enrich the model with contextual informations. For example, actor annotations are used

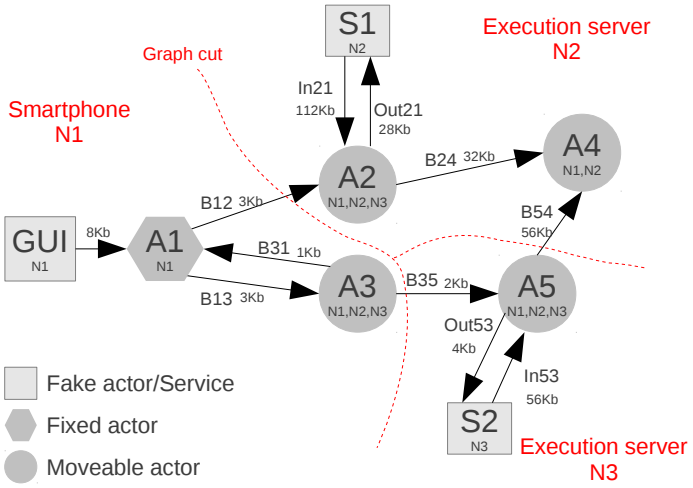


Figure 1.13: Example of application data exchange graph.

to tag an actor that can be offloaded or not, while binding annotations are used to report on the volume of data exchanged between actors. This open metadata format leverages the integration of additional informations that can be used by the MACCHIATO middleware to manage the pervasive applications it hosts. The MACCHIATO middleware services (*e.g.*, management, monitoring) are in charge of maintaining a causal connection between the runtime models and the instances of application.

The resulting graph therefore contains two kinds of actor nodes: static actors (or services) and mobile actors (with their list of potential execution nodes). It also includes weights on the graph edges to reflect the connectivity of two actors.

Optimisation. The MACCHIATO middleware reason on this runtime model to find an optimal deployment of actors given the current execution conditions. The resolution of the optimization problem consists in cutting the data exchange graph into partitions that minimizes an objective function. This function expresses the volume of data that is exchanged across the network by the various actors of the pervasive application. To solve this optimization problem, we use a boolean satisfaction and optimization (SAT) solver that we configure with a linear pseudo-boolean objective function [LP10]. The data exchange graph previously described is a directed graph $G = \{A, B\}$, where each node A_m reflects to an actor, and each edge B_o corresponds to an active binding between actors $B_{o_{src}}$ and $B_{o_{dest}}$. We also include a set of candidate execution nodes $N = \{N_1, ..., N_n\}$ and we declare a boolean variable for each pair of actor and execution node. This can be represented by the following matrix: $\begin{pmatrix} A_1N_1 & \dots & A_0N_n \\ A_mN_1 & \dots & A_mN_n \end{pmatrix}$. To find a valid application distribution, each actor should be executed on exactly one node. For each row of the boolean matrix, we therefore define the following constraint: $\sum_{j=1}^{o} A_iN_j = 1$. The objective function to be optimized by the solver can be defined as follows:

$$\min \left(\sum_{i=1}^{m} \sum_{j=1}^{n} \alpha_{ij} \times \mid A_iN_j - A_iN_j^{t-1} \mid + \sum_{i=1}^{o} \sum_{j=1}^{n} \sum_{k=1}^{n} \beta_{jk} \times \gamma_{B_i} \times \mid B_{i_{src}}N_j - B_{i_{dest}}N_k \mid \right)$$

The first part of the objective function represents the cost to migrate an actor A_i from his current location N_j^{t-1} to the location in the evaluated distribution N_j. That takes into account the connection speed between nodes and the volume of data upload required to transfer the actor. α_{ij} is the cost to migrate an actor i from its current location to execution node j. The second part in the function estimates the cost incurred by the exchange of data between actors located across the network. β_{ij} is the communication cost between execution node i and execution node j. It is computed from the monitored latency and the type of connection between nodes (Ethernet, WiFi, or 3G). γ_{B_i} is the volume of data measured on the binding B_i by the monitoring service.

The resolution of this problem is triggered along the execution of the pervasive application. Depending on the complexity of the software architecture and the number of candidate execution nodes, finding an optimal solution to the problem might become time- and resource-consuming. To overcome this issue, the solver supports the selection of a sub-optimal solution by setting a timeout. Finally, as the reasoner is automatically triggered upon changes observed in the runtime model, a steadiness parameter is used to adjust the reactivity of the middleware to the evolution of its environment. The selected solution is then compared to the current application deployment in order to generate the reconfiguration script that will offload some of the actors on the most suitable execution

nodes (cf. Figure 1.12). While the adoption of an actor programming model leverages the issue of migrating application states as actors are stateless by definition, the context of an actor is serialized as a JSON document during the offloading process. The reconfiguration script is sent to the application management actor, which will operate the reconfiguration of the pervasive application. In order to tolerate the potential failures of remote execution nodes, the application management actor keeps a copy of local actors. Whenever the distributed event bus fails to deliver a message to a remote actor, the message is automatically forwarded to the local copy of this actor.

Consumptions. We measure the network consumption of an application developed with MACCHIATO, and we compare it with two other versions of the same application: *a)* one running exclusively on the mobile phone and *b)* one statically offloading all the actors to a remote execution node. The developed application is a RSS[7] newsreader. This application periodically looks for updates on several news servers and displays only new articles retrieved from the feed. In our evaluation scenario, we simulate 20 RSS feeds, which are updated every 30 seconds.

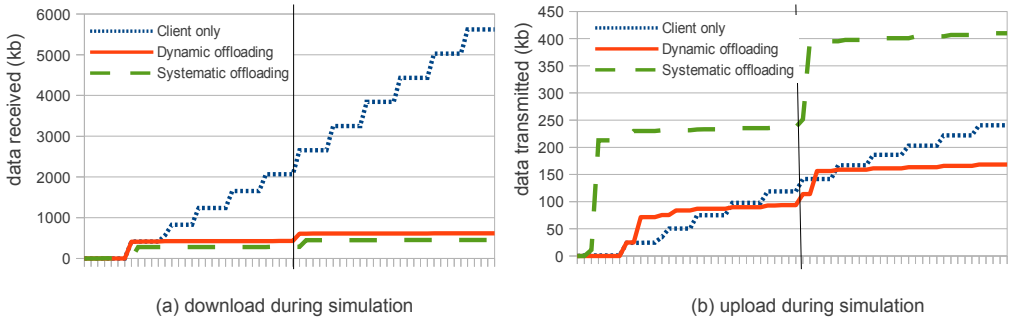


Figure 1.14: Network consumption of a RSS newsreader application.

Figure 1.14 reports on the total volume of data received (a) and sent (b) by the phone along the scenario. This covers all data exchanged over the wireless connection of the mobile phone, including payload and network overhead. While the adoption of offloading reduces the amount of data received by the phone by a factor of 9, the solution based on a static offloading strategy causes sending 2.4 times more data than the solution proposed by MACCHIATO. This intuition is further confirmed by Figure 1.15, which is a projection of the energy consumption model defined by Kalic et al [KBK12], and clearly shows that reducing the bandwidth consumption positively impacts the battery lifespan of the mobile device, with an improvement of 28% of the battery consumption compared to a static offloading strategy. While offloading an actor can be considered as penalizing the application, MACCHIATO only uploads the most network-greedy actors in order to better balance the upload/download trade-off.

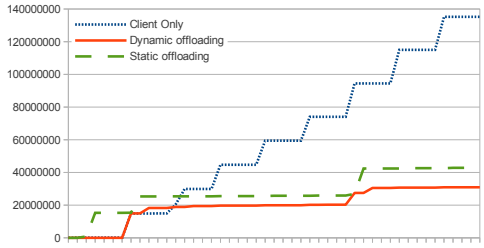


Figure 1.15: Energy consumption of a RSS newsreader application.

1.3 Synthesis

This chapter described our key contributions in the area of the design and the implementation of elastic ubiquitous systems. Beyond our contributions to the FRASCATI middleware platform [Loi+11a; Loi+11b; MRS11a; MRS11b; Sei+12], we have been investigating the adoption of the SCA standard by alternative forms of software systems. In particular, as part of a sustainable research collaboration with Frédéric Loiret at Inria and

[7]RSS specification: `http://www.rssboard.org/rss-specification`

Amirhosein Taherkordi and Frank Eliassen from the University of Oslo[8], we have been porting SCA to WSN in order to better structure the development of components in the small and to support the dynamic reconfiguration of WSN applications in the wild. The major contributions have been published at DCOSS'11 [Tah+10], in Oxford *The Computer Journal* [Tah+11b], and in ACM *Transactions on Sensor Networks* [Tah+13].

Then, at a larger scale, we considered the elasticity that can be introduced in the design of component-based software architectures. As part of a collaboration with Philippe Merle from Inria, we defined a novel approach to express software architecture patterns that can be adjusted upon requirements. This solution is built as a flexible domain-specific architecture language that can be used to describe canonical forms of SCA assemblies and stretch them according to situations. The resulting software framework, FraSCAla, has been published at CBSE'12 [RM12].

Finally, we have been considering alternative forms of components for the development of elastic mobile applications. As part of the Macchiato collaborative project[9] with LABRI, Auchan, and Webpulser, we have been developing an asynchronous component model based on the principles of actors that can automatically offload part of their tasks on remote execution nodes. The foundations of Macchiato, developed with Nicolas Petitprez and Laurence Duchien, have been published at DAIS'12 [PRD12]. Interestingly, the Macchiato middleware illustrates two other challenges of elastic computing systems that are further described in the remainder of this document: the monitoring of the environmental context of a ubiquitous system (cf. Chapter 2) as well as its self-adaptation at run-time (cf. Chapter 3).

Publications associated to this chapter

[Loi+11a] Frédéric Loiret, Romain Rouvoy, Lionel Seinturier, and Philippe Merle. "Software Engineering of Component-Based Systems-of-Systems: A Reference Framework". In: *14th ACM SIGSOFT International Symposium on Component-Based Software Engineering (CBSE'11)*. Ed. by Springer. Boulder, United States, June 2011, pp. 61–65.

[Loi+11b] Frédéric Loiret, Romain Rouvoy, Lionel Seinturier, Daniel Romero, Kevin Sénéchal, and Ales Plsek. "An Aspect-Oriented Framework for Weaving Domain-Specific Concerns into Component-Based Systems". In: *Journal of Universal Computer Science (J.UCS)* 17.5 (Mar. 2011), pp. 742–776.

[MRS11a] Philippe Merle, Romain Rouvoy, and Lionel Seinturier. "A Reflective Platform for Highly Adaptive Multi-Cloud Systems". In: *10th International Workshop on Adaptive and Reflective Middleware (ARM'2011) at the 12th ACM/IFIP/USENIX International Middleware Conference*. Lisbon, Portugal, Dec. 2011, pp. 1–7.

[MRS11b] Philippe Merle, Romain Rouvoy, and Lionel Seinturier. "FraSCAti: Adaptive and Reflective Middleware of Middleware". In: *12th ACM/IFIP/USENIX International Middleware Conference - Tutorial*. Lisbon, Portugal, Dec. 2011.

[PRD12] Nicolas Petitprez, Romain Rouvoy, and Laurence Duchien. "Connecting your Mobile Shopping Cart to the Internet-of-Things". In: *12th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'12)*. Ed. by Karl M. Göschka and Seif Haridi. Vol. 7272. LNCS. Stockholm, Sweden: Springer, June 2012, pp. 236–243.

[RM12] Romain Rouvoy and Philippe Merle. "Rapid Prototyping of Domain-Specific Architecture Languages". In: *15th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE'12)*. Ed. by Magnus Larsson and Nenad Medvidovic. Bertinoro, Italy: ACM, June 2012, pp. 13–22.

[Sei+12] Lionel Seinturier, Philippe Merle, Romain Rouvoy, Daniel Romero, Valerio Schiavoni, and Jean-Bernard Stefani. "A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures". In: *Software: Practice and Experience* 42.5 (May 2012), pp. 559–583.

[Tah+08a] Amirhosein Taherkordi, Frank Eliassen, Romain Rouvoy, and Quan Le-Trung. "ReWiSe: A New Component Model for Lightweight Software Reconfiguration in Wireless Sensor Networks". In: *On the Move to Meaningful Internet Systems: OTM 2008 Workshops*. Springer Berlin Heidelberg, 2008, pp. 415–425.

[8] successively funded by Egide PHC (Aurora), Inria Associate Team (SeaS), and Inria North European Lab (SOCS) programs: `http://seas.ifi.uio.no`

[9] funded by *Fonds Unique Interministériel* (FUI): `http://macchiato.fr`

[Tah+09a] Amirhosein Taherkordi, Romain Rouvoy, Quan Le-Trung, and Frank Eliassen. “Supporting Lightweight Adaptations in Context-aware Wireless Sensor Networks”. In: *1st International COMSWARE Workshop on Context-Aware Middleware and Services (CAMS)*. Vol. 385. ACM International Conference Proceeding Series. Dublin, Ireland: Mélanie Bouroche et al., June 2009.

[Tah+09b] Amirhosein Taherkordi, Quan Le-Trung, Romain Rouvoy, and Frank Eliassen. “WiSeKit: A Distributed Middleware to Support Application-level Adaptation in Sensor Network”. In: *9th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)*. Ed. by Twittie Senivongse and Rui Oliveira. Vol. 5523. Lecture Notes in Computer Science. Lisbon, Portugal, June 2009.

[Tah+10] Amirhosein Taherkordi, Frédéric Loiret, Azadeh Abdolrazaghi, Romain Rouvoy, Quan Le-Trung, and Frank Eliassen. “Programming Sensor Networks Using REMORA Component Model”. In: *6th IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS'10)*. Santa Barbara, California, United States, June 2010, p. 15.

[Tah+11a] Amirhosein Taherkordi, Frank Eliassen, Daniel Romero, and Romain Rouvoy. “RESTful Service Development for Resource-constrained Environments”. In: *REST: From Research to Practice*. Ed. by Erik Wilde and Cesare Pautasso. Springer, 2011, pp. 221–236.

[Tah+11b] Amirhosein Taherkordi, Frédéric Loiret, Romain Rouvoy, and Frank Eliassen. “A Generic Component-based Approach for Programming, Composing and Tuning Sensor Software”. In: *The Computer Journal* 54.2 (Feb. 2011), pp. 1–19.

[Tah+13] Amirhosein Taherkordi, Frédéric Loiret, Romain Rouvoy, and Frank Eliassen. “Optimizing Sensor Network Reprogramming via In-situ Reconfigurable Components”. In: *ACM Transactions on Sensor Networks* 9.2 (May 2013), pp. 1–37.

[TRE10] Amirhosein Taherkordi, Romain Rouvoy, and Frank Eliassen. “A Component-based Approach for Service Distribution in Sensor Networks”. In: *5th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks*. Bangalore, India: ACM, Nov. 2010, pp. 22–28.

Chapter 2

Contextualization of Ubiquitous Systems

2.1 Motivations

Ubiquitous systems are expected to run in a wide diversity of, and potentially unforeseen, operational conditions. Beyond their resilience to changes in their deployment environment, such software systems may also need to sense and monitor their surroundings to inform the end user and eventually take appropriate actions. Context monitoring has therefore emerged as a key capability in various domains to connect software systems to the underlying hardware platform or to the physical world (in the case of ubiquitous systems) [BDR07; HSK09; SW10; Per+14]. Nevertheless, providing such support is not always straightforward and raises several key challenges for the scientific community.

These challenges have been emphasized by two massive trends that recently changed the IT landscape: *Cyber-physical systems*[1] and *Cloud computing*[2]. While cyber-physical systems are now surrounding us due to a massive deployment and adoption (*e.g.*, smartphones, smartwatches, Internet of Things [IoT] devices), cloud computing has democratized the principles of *everything as a service* (XaaS) and fostered various forms of visualization through software-defined systems [Ban+11]. Incidentally, this evolution in software systems motivates the issue of the context monitoring *scalability* according to several dimensions. The first dimension refers to the capability of inferring high-level contextual situations from a large volume of raw data collected in the wild or from a device. Hardware (*e.g.*, accelerometer) or software (*e.g.*, performance counters) sensors tend to continuously produce raw data that a context monitoring solution has to quickly filter, process, and convert it into information that can be used by an application or understood by a user. The second dimension in scalability balances in-breadth and in-depth context monitoring. *In-breadth monitoring* deals with the capability to collect context data at large using, for example, a crowd of mobile devices [Lan+10; Kha+13]. This requires the development of smart mechanisms to orchestrate the retrieval of context information at large to support the analysis of collective behaviors (*e.g.*, the virtual machines of a data center, a fleet of company cars). *In-depth monitoring* addresses the capability to dig into a software-defined system to collect and process context information at any level of visualization. Similar to reflection, a software-defined system—like an application running in a virtual machine—may require introspection capabilities and access to contextual information collected by any layer of the encapsulating infrastructure (*e.g.*, hypervisor, physical machine). Furthermore, no matter the considered dimension of scalability, the resource consumption of context monitoring has to be carefully optimized as this service should not impact the performances of the overall system.

The objective of this chapter is therefore to report on our contributions in the area of scalable context monitoring and processing solutions. Our definition of scalability takes into account the various dimensions we exemplified and thus we propose novel middleware solutions to efficiently process context at large. Beyond the support for self-adaptation decision heuristics, which will be described in Chapter 3, these context-aware middleware systems can been perceived as software microscopes that clearly helps researchers and engineers to better understand the behavior of a software system. This perspective has raised the need for the development of robust software components to provide better insights on the conditions under which a software systems may be immersed and therefore one can help to adjust its capabilities accordingly.

2.2 Contributions

Building on the principles of component-based software architectures we introduced in the previous chapter, we have investigated the design and the implementation of middleware solutions to address in-breadth and in-depth

[1] `http://en.wikipedia.org/wiki/Cyber-physical_system`
[2] `http://en.wikipedia.org/wiki/Cloud_computing`

context monitoring. In particular, we report on the APISENSE® distributed platform as a solution to deal with crowd-sensing activities (cf. Section 2.2.2). Then, we introduce the PowerAPI library as a middleware solution to estimate the power consumption of software at various granularities (cf. Section 2.2.3). Both of these solutions are inspired by COSMOS, a context processing framework built with resource optimization in mind, which we introduce hereafter (cf. Section 2.2.1), and used as a reference model for developing our contributions.

2.2.1 Hierarchic Modeling of Software Context

Our work in the area of contextualization of ubiquitous systems considers that any context information requires to be processed prior to be consumed. In particular, given that context sensors tend to produce raw data, one needs to filter, aggregate, and convert such data into added-value information for an application. Figure 2.1 illustrates the complexity that can result from combining several context policies for developing a ubiquitous application. This example is part of a ubiquitous application that can be used by a family shopping in a mall with a mobile device[3]. This application allows them to share information, to consult product prices, to download discount tickets, to be notified of advertisements, to access additional information and comments about a product, or to find the location of a product or a shop in the mall. The parents want their children to remain in the mall, with their devices connected whenever possible, so that everybody knows the location of the other family members. Nevertheless, children can disconnect for some periods of time in order to save their battery. While walking in the mall, the eldest girl sees an advertisement indicating that a dressing store proposes a RFID tag-based service for helping choose clothes. All these features are based on different network technologies, such as Bluetooth or WiFi, and require the application to adapt itself depending on network connectivity and context information availability. Even if each high-level context information relates to a particular functionality and focuses on a precise set of lower-level context information, some intermediate context information may be shared and useful for several context policies. For example, the WiFi download enabled situation is associated to the functionality supporting the download of a discount ticket: It allows the application to know when the functionality is available. The detection is performed by monitoring the quality of the WiFi link. The WiFi browsing enabled situation is built upon the previous one and allows the application to enable and to configure a browsing facility to access comments about a product or to find its location in the mall. Therefore, we model context information (in the upper part of the picture) as hierarchies of context trees with the possibility of sharing sub-trees between policies.

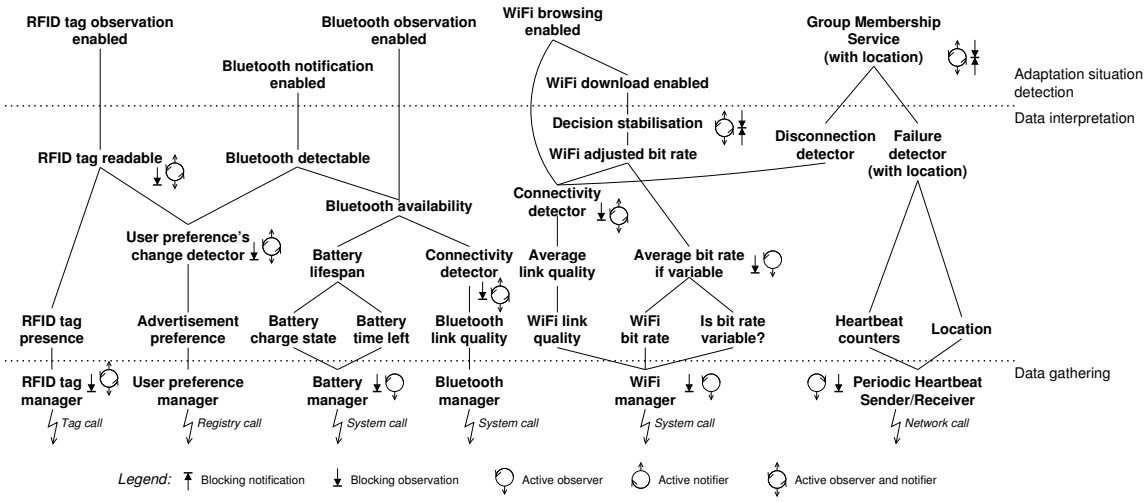


Figure 2.1: Example of Complex Context Policy.

2.2.1.1 The COSMOS Context Processing Framework

COSMOS is a component-based framework for processing context information in ubiquitous context-aware applications [CRS07; CRS08; RCS08]. COSMOS is therefore a middleware toolkit that supports the implementation of context policies, such as the one above described.

COSMOS decomposes context observation policies into fine-grained units called *context nodes*. A context node is context information modeled by a software component. COSMOS organizes context nodes into hierarchies to form *context policies*. The relationships between context nodes are sharing and encapsulation. The sharing of a context node—and, by implication, of a partial or complete hierarchy—corresponds to the sharing

[3]This scenario is a use case of the French project Cappuccino (`http://www.cappucino.fr`).

of part or all of a context policy. Context nodes at a hierarchy's leaves (the bottom-most elements in Figure 2.1, with no descendants) encapsulate raw context data obtained from collectors (such as operating system probes, sensors near the device, user preferences in profiles, and remote devices). Context nodes should provide all the inputs necessary for reasoning about the execution context, which is why COSMOS considers user preferences as context data. A context node's role is thus to isolate the inference of high-level context information from lower architectural layers responsible for collecting context data.

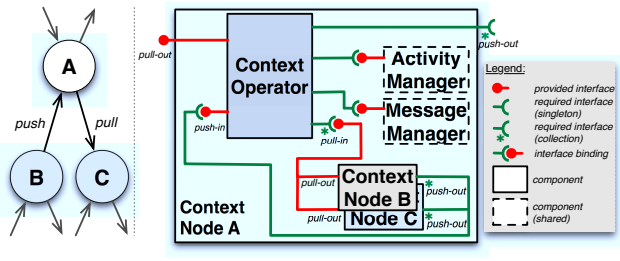
Figure 2.2a depicts the mapping from a context policy tree to the associated component-based software architecture. Context nodes are classified into two categories: leaves and other nodes. Leaves of the hierarchy are ContextNodes extended to contain one or several components that receive context information from an external entity. This external entity may be the operating system or another framework, being built with COSMOS or not, component-oriented or not. For instance, a WiFi resource manager can obtain the corresponding context information directly from the operating system (through system calls) or can encapsulate a (legacy) framework dedicated to the reification of system resources. Nodes of the graph which are not leaves are extended to contain one or several other context nodes. For example, a context node may compute the battery charge state of a terminal by encapsulating two other context nodes, the first one computing the battery charge state and the second one computing the battery time left.

Context nodes are also equipped with properties which define their behavior with respect to the context management policy:

Passive vs. active. A passive node obtains context information upon demand. A passive node must be invoked explicitly by another context node (passive or active). An active node is associated to a thread and initiates the gathering and/or the treatment of context information. The thread may be dedicated to the node or be retrieved from a pool. A typical example of an active node is the centralization of several types of context information, the periodic computation of a higher-level context information, and the provision of the latter information to upper nodes.

Observation (pull) vs. notification (push). The observation reports containing context information are encapsulated into messages that circulate from the leaves to the root of the hierarchies. When the circulation is initiated at the request of parent nodes or client applications, it is an observation. In the other case, this is a notification.

Blocking or not. During an observation or a notification, a node that treats the request can be blocking or not. During an observation, a non-blocking context node begins by requesting a new observation report from each of its child nodes, and then updates its context information before answering the request of the parent node or the client application. During a notification, a non-blocking node computes a new observation report with the new context information just being notified, and then notifies the parent node of the client application. In the case of a blocking node, an observed node provides the most up-to-date context information that it possesses without requesting child nodes, and a notified node updates its state without notifying parent nodes. In addition, a node can be configured for a unique observation or notification if its state is immutable. Finally, the observation of a node can raise exceptions, for instance when the physical resource is not present or in case of a configuration problem. On demand, the thrown exception can be masked to parent nodes or client applications, and default values can be provided in that case.



(a) Architecture of COSMOS Context Node.

```
1 object WiFiDetector extends Policy {
2   val WiFiMgr = new sensor[WirelessInterface] {
3     parameter("resourceName","eth1")
4   }.isActiveObserver.isBlockingObserver

6   val WiFiConnect = new node[ConnectivityDetector] {
7     node[Average] consumes WiFiMgr.get("link-quality")

9     val AvgWiFiBitRate = new node[AverageIf] {
10      consumes(WiFiMgr.get("bit-rate", "is-variable"))
11  } }.isActiveObserver.isActiveNotifier

13  new node[AdjustedBitRate] {
14    consumes(WiFiConnect)
15    consumes(WiFiConnect.AvgWiFiBitRate)
16 } }
```

(b) Composing a Context Policy with FraSCAla.

Figure 2.2: COSMOS Framework

COSMOS policies can be described using FractalADL [Bru+06; Lec+07] or FraSCAla. For the sake of brevity, Listing 2.2b reports only on the resulting ADL notation that can be used to describe a context policy. Therefore, as illustrated in Listing 2.2b, the architect describes the architecture of a context policy

using keywords such as `Policy`, `sensor` or `node`, which are more straightforward than generic component-based constructions.

Thanks to the definition architecture patterns, COSMOS can provide the developer with pre-defined generic context nodes that can be configured upon requirements: *sensor nodes* for collecting raw data, *memory nodes*, such as averagers, translation nodes, *data mergers* with different quality of service, *abstract or inference nodes*, such as additioners, thresholds nodes, etc. Note that in a classical context manager architecture the first nodes constitute the collectors, most of the other ones are part of the interpretation layer, while the last thresholds based ones serve to identify situations. In COSMOS, each class of nodes can be used in every layers, hence leveraging the expressiveness power of context policies.

The implementation of the COSMOS framework is based on three existing frameworks: Fractal, DREAM, and SAJE. Fractal [Bru+06] is the component model of the ObjectWeb consortium for open-source middleware. Fractal defines a lightweight, hierarchical and open component model (see `http://fractal.objectweb.org`). We use Julia [Bru+06], which is a Java implementation of Fractal. We also take advantage of the numerous tools available for this component model, such as Fractal ADL, FPath, and Fraclet (a Java 5 annotation-based programming model) [RM09], and DREAM [LQS05]—a Fractal component library for the construction of message-oriented middleware (MOM) and the fine-grained control of concurrency management with thread pools and message pools. Finally, SAJE [Cou+03] is a framework for gathering data from system resources. COSMOS is available under the GNU LGPL license and can be downloaded from `http://picoforge.int-evry.fr/projects/cosmos`. Extensive performance measurements are reported in [CRS07].

2.2.1.2 A Focus on Context Stabilization Algorithms

In order to optimize the decision making processes captured by context policies, context stabilization mechanisms can be identified as a crosscutting concern when processing context. Context regions [Ati+03] is an example of stabilization mechanism which advocates a strict partition of the context space, in oder to reduce the adaptation side-effects. However, such mechanisms solve the stabilization problem only partially. In particular, context regions do not handle properly application behavior during the transition phases from one state to another. More powerful and complex stabilization mechanisms based on machine learning algorithms are suggested in many works like [Dar07; Wu03]. Nonetheless, despite some good results in predicting application behavior, they tend to suffer from weak reactivity in the learning phase.

Classification. To address the crosscutting nature of stabilization algorithms, we propose to integrate stabilization algorithms as generic context nodes to be woven into context policies. However, as the application of different stabilization algorithms may result in different *Qualities of Context* (QoC) [SWV07], which requires stabilization mechanisms to be classified according to their properties. In particular, we based our classification on the following three criteria:

1. Algorithmic complexity refers to the maximum number of steps required by an algorithm for different configuration sets and possible inputs values;
2. Execution speed denotes the minimal time required for an algorithm consuming an input vector v_i to produce a valid output v_j.
3. Data scope describes the type (nominal, ordinal, numeric) of data that can be given as input for an algorithm.

Although only a few works in the state-of-the-art [Pad+05; TC04] directly address stabilization issues for self-adaptive applications, stabilization issues are ubiquitous in most of the works that are related to context reasoning or context fusion of information [RRM06; Pad+05; Dar07]. Most of the stabilization algorithms or techniques can be categorized into five groups:

Filtering techniques. These techniques focus on data-filtering using statistic or parametric-based algorithms. For example, in *MoCoA* [RRM06]—a framework for the management of network applications—data filtering is achieved on the basis of geographical or temporal constraints.

Threshold techniques. For algorithms of this group, the stabilization is realized by checking the system state with regards to threshold values. The *heartbeat* (HB) algorithm described in [TC04]—used for the stabilization of network connectivity failure in a mobile environment—is an example of a threshold-based technique.

Refresh techniques. The principle of these techniques is to update the system with new contextual information only when certain conditions are fulfilled. Usually these conditions are expressed in the form of time-based constraints or events/actions constraints.

Probabilistic schemas. For algorithms that belong to this group, stabilization is reached by inferring the system state from probabilities and previous states of the system, for example *Kalman Filter* [Pad+05]

or *Hidden Markov Model* [SAH07].

Uncategorized. This last group encloses all the algorithms that that does not fit in the previous categories, because they use *ad hoc* methods to handle the stabilization of the system.

When studying the stabilization mechanisms presented in the literature, we find out that they do not have the same *data scope*—*i.e.*, they do not process the same type of data. In our classification, the *data scope* criteria is based on the data type categories proposed by Mayrhofer et al. [MRF03]. Mayrhofer et al. suggest that primitive types of contextual information can be grouped in four categories: *i) Nominal data* (qualitative) includes values in a dataset on which no order relationship has been or can be defined. A special case are binary features with $S = \{0, 1\}$; *ii) Ordinal data* (rank) covers values of a set with a defined order relationship; *iii) Numerical data* (quantitative) encompasses values of an ordered set with predefined operations (an algebraic field). It can be further distinguished according to the density of values in the discrete ($S \in Z$) or continuous ($S \in R$) set; *iv) Interval data* refers to intervals instead of single values.

Table 2.1 reports our classification of stabilization mechanisms according to primitive contextual information types. We use the annotation "$\surd\surd$" to express that an algorithm targets the data group, while "$\surd$" expresses that an algorithm can target the data group by applying some simple conversions. Table 2.1 shows that depending on their implementation, stabilization algorithms usually do not have the same data scope. We use this classification to recommend the suitable stabilization mechanisms depending on the data type associate to the context node to be woven—*i.e.*, stabilized.

	Nominal	**Ordinal**	**Numerical**	**Interval**
Delta operators [BBF02]			$\surd\surd$	$\surd$
Buffer [RRM06]	$\surd\surd$	$\surd\surd$	$\surd\surd$	$\surd\surd$
Warm-up time [Bra+07b]	$\surd\surd$	$\surd\surd$	$\surd\surd$	$\surd$
Context regions [Ati+03]		$\surd\surd$	$\surd\surd$	$\surd$
Sensitivity (speed, acceleration) [Pad+05]			$\surd\surd$	$\surd$
Switch [Pad+05]	$\surd\surd$	$\surd\surd$	$\surd\surd$	$\surd\surd$
Statistical Techniques (average, variance)			$\surd\surd$	$\surd$
Filtering techniques				
Statistical filtering			$\surd\surd$	$\surd$
Parametric filtering (time, localisation)	$\surd$	$\surd$	$\surd\surd$	$\surd$
Treshold techniques				
Simple threshold		$\surd\surd$	$\surd\surd$	$\surd$
Double threshold		$\surd\surd$	$\surd\surd$	$\surd$
Double double threshold (hysteresis) [TC04]		$\surd\surd$	$\surd\surd$	$\surd$
Refresh techniques				
Parametric refresh (T)	$\surd\surd$	$\surd\surd$	$\surd\surd$	$\surd\surd$
Event/Action refresh	$\surd\surd$	$\surd\surd$	$\surd\surd$	$\surd\surd$
Probabilistic schemas				
Fuzzy logic [Dar07]	$\surd\surd$	$\surd\surd$	$\surd\surd$	$\surd\surd$
Markov chain [SAH07]		$\surd\surd$	$\surd\surd$	$\surd$
Bayesian network [SAH07]			$\surd\surd$	$\surd$
Dempster-Shaffer theory [SAH07]			$\surd\surd$	$\surd$

Table 2.1: Classification of stabilization techniques according to contextual information categories.

Composition. In order to meet the flexibility requirements for the efficient stabilization of context policies, we define a composition model for stabilization algorithms. Our composition model defines how to combine different stabilization techniques or mechanisms in order to provide a flexible stabilization mechanism for the application. The model therefore consists of two strategies of composition: *horizontal* and *vertical compositions*.

Horizontal composition. Although learning-based stabilization algorithms, like *Dempster-Shaffer Theory* (DST) [Wu03] or *Bayesian Networks* (BN) are efficient in predicting contextual changes, they introduce a latency in detecting variations in the application environment. This weakness can be compensated by associating learning-based algorithms to other algorithms with a lower latency. Horizontal composition therefore consists in executing concurrently several stabilization algorithms. The idea behind this concept is to benefit from passive and reactive stabilization techniques by combining them adequately. The detection of irregular application behavior can be obtained by combining a reactive algorithm like *Delta Operator*, and less reactive algorithm like DST algorithms, which are combined by a *Composition Rule* (CR). A CR can be a function like "$f(v_n, v_{n+1}) \Rightarrow max(v_n, v_{n+1})$", where v_n, v_{n+1} are context values, or a complex rule considering the QoC.

Hence, using this composition model can help to improve the accuracy of the stabilization process while keeping the reactivity of the system at a reasonable level.

Vertical composition. Vertical composition in our approach consists in applying two or more stabilization algorithms in sequence. In [SAH07], Sekkas et al. suggest that it can be interesting in terms of performance for stabilization of context information to apply successively several stabilization algorithms on the same sample of data. The authors compare the efficiency of using *Bayesian Networks* (BN), *Dynamic Bayesian Networks* (DBN), and *Fuzzy Logic* (FL) independently or in a combined way. In our approach, we believe that the combination of algorithms using *vertical composition* can increase the efficiency of the stabilization process. In order to limit the overhead introduced by the use of several algorithms, cheap (execution) algorithms are found at the bottom of the stabilization chain while expensive algorithms are on the top of the architecture. This association rule is mostly justified by the fact that the amount of processed data decreases from the bottom to the top, thus more costly algorithms at the top of the architecture would have to process less data, decreasing by the same way the overall cost of the stabilization process, which is tightly bound to the amount of context-information processed.

Figure 2.3 illustrates from left to right these two composition types and their combination. This combination of both composition models can be used in order to meet accuracy and efficiency properties of the stabilization process. The data source in Figure 2.3 stands for any data provider since the stabilization strategies described can be applied on any data coming from monitoring activities (before the decision making block) and on data coming from adaptation activities (after the decision making block).

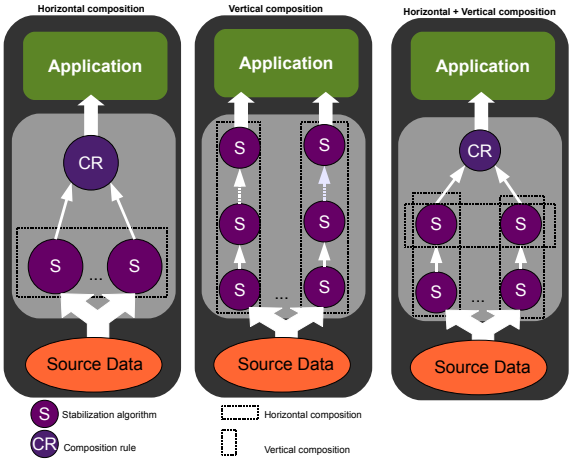


Figure 2.3: Compositions of Context Stabilization Algorithms in Context Policies.

Discussions. By integrating stabilization algorithms into context policies, we raise the quality of context and provide users or decision heuristics with stable indicators that can be used to reason on the current state of the system or the status of the surrounding environment [NRS10a; NRS09]. While these algorithms are made reusable in COSMOS, they may need to be tuned appropriately in order to be efficient.

Along our research activities, we have been applying the principles of COSMOS to several contexts, from wireless sensor networks [Tah+08b], to smart-homes [Par+12b], to mobile devices [Rom+10a; Rom+10c], and even to server-side infrastructures [Nou+12b]. The next two sections are going to further detail how we recently derive to the context model promoted by COSMOS to address the specific cases of *in-breath monitoring* of mobile devices (cf. Section 2.2.2) and *in-depth monitoring* of energy consumption (cf. Section 2.2.3).

2.2.2 In-breath Context Monitoring and Orchestration in the Wild

For years, the analysis of contextual traces has contributed to better understand user behaviors and habits [Liu+09]. For example, the *Urban Mobs* initiative[4] visualizes SMS or call activities in a city upon the occurrence of major public events. These activity traces are typically generated from GSM traces collected by the cellphone providers [Soh+06]. However, getting access to such GSM traces is often subject to constraining agreements with the mobile network operators, which restrict their publication, and have a scope limited to telecom data.

We therefore believe that cellphones represent a great opportunity to collect a wide range of crowd contextual traces. Largely adopted by populations, with more than 968 million units sold in 2013 (against 680

[4] `http://www.urbanmobs.fr`

millions in 2011) according to Gartner institute[5], smartphones have become a key companion in people's dailylife. Not only focusing on computing or communication capabilities, modern mobile devices are now equipped of a wide range of sensors enabling scientist to build a new class of datasets. Using cellphones to collect contextual traces contributed by users is reported in the literature either as *participatory sensing* [Bur+06], which requires explicit user actions to share sensors data, or as *opportunistic sensing* where the mobile sensing application collect and share data without user involvement. These approaches have been largely used in multiple research studies including traffic and road monitoring [Bia+11], social networking [Mil+10] or environmental monitoring [Mun+09]. However, developing a sensing application to collect a specific dataset over a given population in not trivial. Indeed, a participatory and opportunistic sensing application needs to cope with a set of key challenges [CHK08; Lan+10], including energy limitation, privacy concern and needs to provide incentive mechanisms in order to attract participants.

These constraints are making difficult, for scientists non expert in this field, to easily collect at large realistic datasets for their studies. But more importantly, the developed ad hoc applications may neglect privacy and security concerns, resulting in the disclosure of sensible user information. With regards to the state-of-the-art in this field, we therefore observe that current solutions lack of reusable approaches for collecting and exploiting crowd activity traces, which are usually difficult to setup and tied to specific data representations and device configurations. We therefore believe that crowd-sensing platforms require to evolve in order to become more open and widely accessible to scientific communities. In this context, we introduce APISENSE, an open crowd-sensing platform targeting multiple research communities, and providing a lightweight way to build and deploy opportunistic sensing applications in order to collect dedicated datasets.

2.2.2.1 The APISENSE® Crowd-Sensing Platform

APISENSE® extends the context model we introduced with COSMOS to deal with multiple sources of context—*i.e.*, the mobile devices—and builds on some of the programming models we described in Chapter 1 to implement a crowd-sensing platform supporting *in-breadth monitoring* of context information. The distributed architecture of APISENSE® combines a standard mobile application to be installed by participants on their own mobile phone, with a Cloud-hosted backend infrastructure to control the deployment of context policies and get access to collected context information [Par+12a; HRS13b; Qui+13; Had+14]. The APISENSE® platform therefore distinguishes between two roles. The former, called *scientist*, is a researcher who wants to define and deploy an context policy over a large population of mobile users. The platform therefore provides a set of services allowing her/him to describe context policy requirements in a domain-specific language, deploying it over a subset of participants and connect other services to the platform to extract and reuse dataset collected in other contexts (*e.g.*, *visualization*, *analysis*, *replay*). The scientist is offered a web environment to define, deploy, and store the collected dataset. In order to increase the dynamicity of context policies, the APISENSE® platform uses a script-oriented approach to describe and compose context policies. The latter is the mobile phone user, also called *participant*. The APISENSE® platform provides a mobile application allowing to download experiments, execute them in a dedicated sandbox and upload collected datasets to the APISENSE® server. The mobile application provides several mechanisms in order to keep the control of user privacy and battery lifespan.

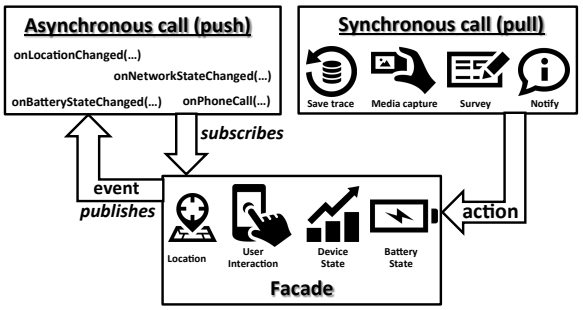
Context collection. To reduce the *learning curve*, we decided to adopt a standard scripting language in order to ease the description of context policies by the scientist. We therefore propose the APISENSE® scripting language as an extension of JavaScript and Python, which provides an efficient mean to describe an experiment without a specific knowledge of mobile device programming technologies (*e.g.*, Android SDK). The choice of JavaScript and Python was mainly motivated by their native support for JSON (*JavaScript Object Notation*), which is a lightweight data-interchange format reducing the communication overhead. For example, Listing 2.4a describes the piece of script to be written by a scientist interested in collecting geolocated GSM signal strength traces. This script is triggered whenever the position of the participant changes and builds a new contextual trace out of the collected data.

Each context policy is associated with a local database used to store the collected traces in the mobile device. Each trace collected by the policy is stored in the database before to be aggregated and sent to APISENSE® server-side infrastructure. In this example, the script requires to access two context nodes: the *location sensor node* to collect periodic updates of the device's geographical location, while the *GSM sensor node* monitors the GSM signal strength level. When the state of an active sensor is updated, the system creates an event and triggers the handlers associated to this event. As we are interested in collecting location and GSM signal strength level only when the location of the user changes, the scientist implements the function `onLocationStateChanged` as shown in Listing 2.4a. The function can access raw sensor data via the event `loc` passed as a parameter. The last step consists in describing the content of the context trace. The APISENSE®

[5] `http://www.gartner.com/newsroom/id/2665715`

```
1 // Event and action triggered when the user moves
2 $location.onLocationChanged({provider : ["network"]},
3   function(loc) {
4     return $trace.add({
5         lat:  loc.latitude(),
6         lon:  loc.longitude(),
7         time: loc.time(),
8         ss:   $gsm.signalStrengthLevel()
9     });
10 });
```

(a) GSM Signal Strength Policy.



(b) APISENSE® Scripting Language.

Figure 2.4: APISENSE® Programming Model.

scripting language provides several *helpers* in order to build traces for geolocation standard formats, such as *GPS Exchange Format* (GPX) or *Keyhole Markup Language* (KML). In addition to these helpers, the scientist is free to define a custom representation and return an object directly in JSON representation.

In addition to the sensors reported in this example, the APISENSE® scripting language supports a wide range of features to declare the data to be collected during a sensing experiments including traditional sensors proposed by smartphone technologies, such as Bluetooth, call statistics, application status (installed, running) in the case of *opportunistic sensing*, as well as a support for *participatory sensing* using surveys (filling out a form, taking a picture or a video) as illustrated in Figure 2.4b. In APISENSE, context sensors are exposed as Facades, which group *push*- and *pull*-based methods to interact with a given class of sensors (*e.g.*, accelerometer, GPS, camera, battery). By subscribing to events (*e.g.*, `onLocationChanged`), the context policy can therefore be notified of the occurrences of a specific event (whose type is bound to the subscription method) and eventually produce a context trace.

User privacy. In addition to this script, the scientist can define privacy filters to limit the volume of data to be collected on the field and to enforce the privacy of the participants. In particular, APISENSE® currently supports two types of policy filters:

Area filter allows the scientific to specify a geographic area where the data requires to be collected. In our example, this area maps to the place where the scientist is interested in collecting the GSM signal (*e.g.*, campus area). This filter guarantees the participants that no data is collected and sent to the APISENSE® server outside of this area.

Period filter allows the scientific to define a time period during which the experiment should be active and collect data. In our example, this period can be specified during the office hours in order to discard data collected during night, while the participant is expected to be at home.

Mobile application. Although our solution could be extended to other *Operating Systems*, the APISENSE® mobile application is currently based on the Android operating system[6].

A participant willing to be involved in an experiment proposed by a scientist can download and install the APISENSE® mobile application and create an account on the server-side infrastructure. Once registered, the HTTP communications between the mobile device of the participant and the infrastructure are authenticated and encrypted in order to reduce potential privacy leaks when transferring the collected datasets to the APISENSE® server. From these, the participant can connect to the *Experiment Store* component (cf. Figure 2.7), download and run one or multiple sensing experiments proposed by scientists.

Facades bridge the Android SDK with the Script engine. This abstraction covers two roles: a *security role* to prevent malicious calls of critical code and a *accessibility role* to leverage the development of context policies as illustrated in Listing 2.4a. Although the last generation of smartphones provides very powerful computing capabilities, the major obstacle to realize continuous sensing application refers to their energy restrictions. Therefore, in order to reduce the communication overhead with the remote server, which tends to be energy consuming [Sha09], datasets are uploaded only when the mobile phone is plugged. In particular, the Battery manager component monitors the battery state and triggers the Network manager component when the battery starts charging in order to send all the collected datasets to the server-side infrastructure. Additionally, this component monitors the current battery level and suspends the Script engine component when the battery level goes below a specific threshold (20% by default) in order to stop all active context policies. All the privacy rules defined by the participants are interpreted by the *Privacy Manager* component, which suspends the Script

[6]Prototype applications are also available for iOS and Windows Phone.

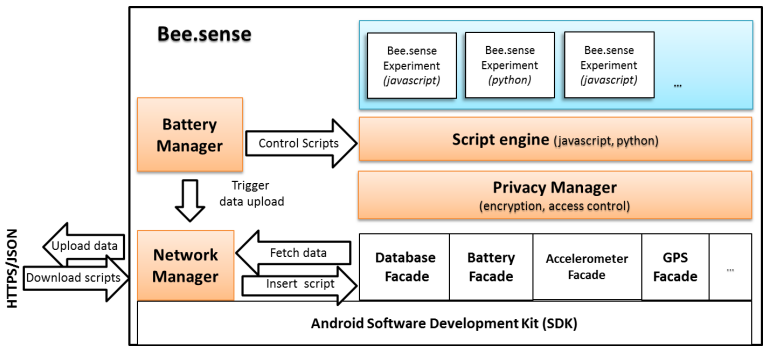


Figure 2.5: Architecture of the APISENSE® Mobile Application

engine component if one this rules is triggered. The last category of privacy rules refer to *authorization rules*, which prevent context policies to access raw context data if the user does not want to share this information. Additionally, an embedded mechanism use cryptography hashing (SHA-1) to prevent context policies to collect sensitive raw data, such as phone numbers, SMS content, or address book.

User incentives In order to attract participants, a crowd-sensing platform has to provide appropriate levers to target a critical volume of context information. As cited by [Dut+09], one key challenge when cellphone are used as research platforms is to incite users to contribute to a given experiment. Even if crowd-sensing applications represent a great interest for scientists, it does not offer any particular service to the participants, but still consumes their resources (*e.g.*, battery, bandwidth). To help scientist to motivate participants, we provide an incentive support encouraging participants to collect relevant datasets.

As the APISENSE® mobile application provides several mechanisms to control the access to sensors, the rewarding mechanism is based on the quality and the volume of context traces produced by participants. Therefore, the more sensors are activated by participants and the more context traces are uploaded to the APISENSE® server-side infrastructure, the more credits the participants receive for their contributions. The credits can be configured by the scientist in order to privilege the retrieval of specific data. For example, the scientist can allocate more credits to the GPS traces in order to balance to the energy consumption and the privacy sensitive of this sensor. The assigned credits are then used by the scientist to provide participant rankings, involvement badges, or even coupons to reward the users.

The participant is therefore free to disable some of the sensors for privacy or energy reasons (cf. Figure 2.6), but in this case, she/he will receive less credits when uploading the collected dataset. Depending on the data consistency that the scientist expects, she/he can keep or filter out partial activity traces from the dataset by executing an XQuery extraction [HRS13a].

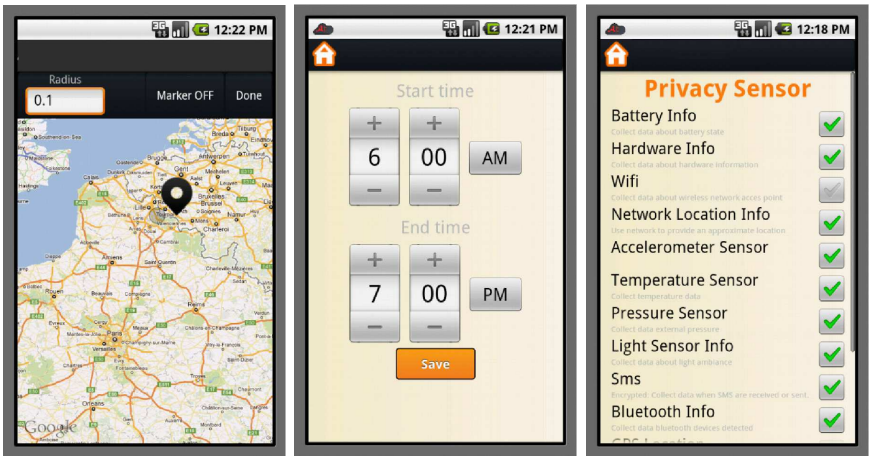


Figure 2.6: APISENSE® Privacy Filters

Server-side infrastructure. The main objective of APISENSE® is to provide to scientist an open platform, which is extensible and configurable in order to be reused in various contexts. To achieve this goal, we designed the server-side infrastructure of APISENSE® as an SCA distributed system (cf. Figure 2.7). In particular, we believe that SCA provides an flexible foundation for the APISENSE® infrastructure by accommodating a wide

diversity of programming languages and communication protocols in order to efficiently support the variety of user requirements.

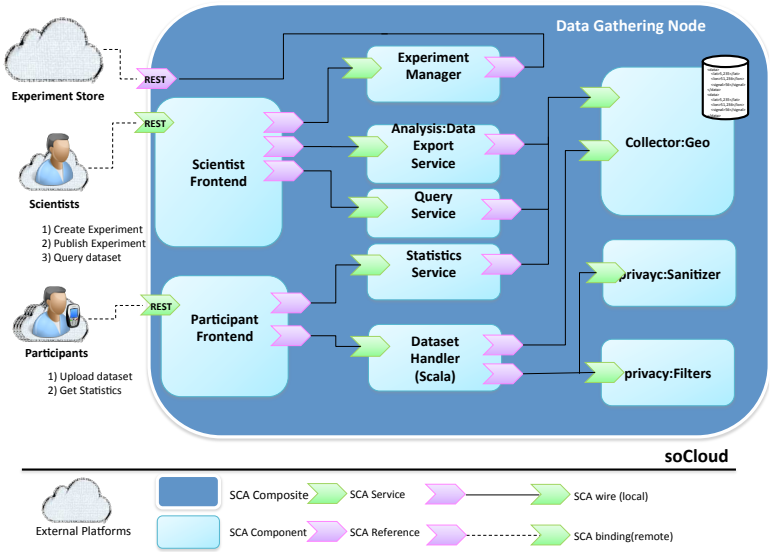


Figure 2.7: Architecture of the APISENSE® Server Infrastructure.

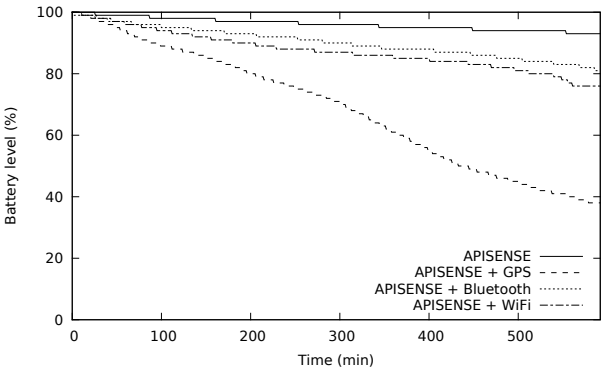
All the components building the server-side infrastructure of APISENSE® are hosted by a cloud infrastructure. The *Scientist Frontend* and *Participant Frontend* components are the endpoints for the two categories of users involved in the platform. Both components define all the services that can be invoked remotely by scientists or participants. These remote services are exposed as a REST resources. Additionally to the components hosted in the cloud, scientists can develop and deploy their own components in order to tune the platform according to their requirements. In order to ease the adoption of the APISENSE® platform and manage services provided by the platform, we developed a Web interface. Thus, this choice does not require the scientist to install any specific software on her/his computer and it leverages the burden of handling network-level configurations in order to deploy and maintain a server infrastructure to store the data collected by the participants.

Deployment model. Once a context policy is declared using the APISENSE® scripting language, a scientist can publish it into the *Experiment Store* component in order to make it available to participants. Once published, two deployment strategies can be considered for deploying experiments. The former, called pull-based approach, is a proactive deployment strategy where participants download the list of experiments from the remote server. The latter, push-based approach, propagates the experiments list updates to the mobiles devices of participants. In the case of APISENSE, the push-based strategy would induce a communication and energy overhead and, in order to leave the choice to participants to select the experiments they are interested in, we adopted the pull-based approach as a deployment strategy. Therefore, when the mobile device of a participant connects to the *Experiment Store*, it sends its characteristics (including hardware, current location, sensor available and sensors that participants want to share) and receives the list of experiments that are currently published. The scientists can configure the *Experiment Store* to limit the visibility of their experiments according the characteristics of participants. In order to reduce the privacy risk, device characteristics sent by participants are not stored by the infrastructure and scientist cannot access to this information.

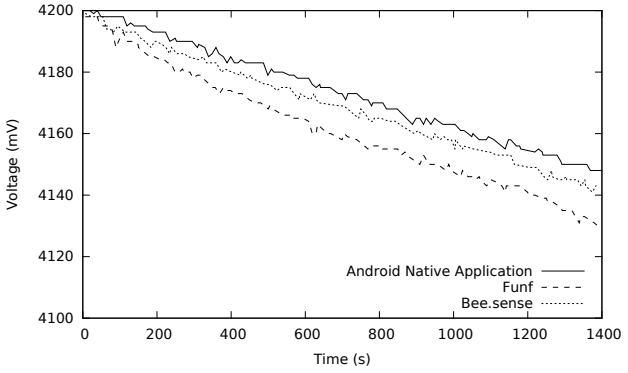
Additionally, the *Experiment Store* component is also used to update the behavior of the context policy once deployed in the wild. When an opportunistic connection is established between the mobile device and the APISENSE® server, for example when collected datasets are uploaded, the version of the context policy deployed in the mobile device is compared to the latest version published in the server. The context policy is automatically updated with the latest version without imposing participants to re-download manually the whole experiment. In order to avoid any versioning problem, each uploaded dataset includes a key encoding the version of the context policy used to collect data. Thus, scientists can configure the *Experiment Store* component in order to keep or discard datasets collected by older versions of the context policy.

Energy consumption. Our first evaluation aims at assessing the battery lifespan impact of the APISENSE® mobile application. The curve labelled as APISENSE in Figure 2.8a reports on the result of this experiment,

which has been executed on a Samsung Galaxy S based on Android 2.2. As we can observe, the baseline experiment tends to have a very small impact on the battery lifespan, thus highlighting the low overhead induced by the APISENSE® mobile application. Then, a second experiment evaluates the impact of energy-consuming sensors that can be used to collect data (cf. Figure 2.8a). For this experiment, we developed three additional scripts, which we deployed separately. The first script, labelled APISENSE® + Bluetooth, triggers a Bluetooth scan every minute and collect both the battery level as well as the resulting Bluetooth scan. The second script, APISENSE® + GPS, records every minute the current location collected from the GPS sensor, while the third script, APISENSE® + WiFi, collects a WiFi scan every minute. These experiments demonstrates that, even when stressing energy-consuming sensors, it is still possible to collect data during a normal day of work without plugging the mobile phone (40% of battery left after 10 hours of pulling the GPS sensor). Furthermore, as reported by Figure 2.8b, APISENSE® provides a better footprint than FUNF [Aha+11] in term of power consumption.



(a) Energy consumption per type of context information.



(b) Power consumption of APISENSE® and existing solutions.

Figure 2.8: APISENSE® Performances.

2.2.2.2 A Focus on Task Orchestration Algorithms

The distribution of collaborative tasks over a group of nodes has been studied by the community to address the issue of geographical coverage [ZWS04]. However, most of these works build on predictable mobility models to address this issue, while it is hardly possible to influence or predict the mobility of humans when using their mobile phone to collect data in the wild. We therefore leverage the concept of *virtual sensor* [CB10] as a solution to deal with the dynamic orchestration of context policies according to some coverage objectives (*e.g.*, geographic area, time period).

Listing 2.1 illustrates a context policy used to monitor the mobile Internet access quality at large in the area of Paris. The context policy is first described (lines 1–7), as in the previous example (cf. Listing 2.4a), and consists in collecting the mean latency of 10 ping requests to an Internet probe (line 6). Then, the script specifies that we are interested in receiving contributions from participants using a mobile connection (lines 9–11), which we rank according to their battery level (highest battery comes first, lines 12–13). Finally, we specify the geographic area coverage (Paris with an accuracy of 500 meters, line 14), the time period coverage (at least 30 minutes every hour, line 15), and the redundancy of sensing measures (2 measures are required by be meaningful, line 16).

```
1 $experiment.create().sense(function() {
2   $location.onLocationChanged(function(loc) {
3     return $trace.add({
4       operator : $gsm.operator(),
5       loc: [loc.latitude(), loc.longitude()],
6       latency : $network.ping(10,"http://...").mean});
7     });
8 }).accept(function() {
9   if (network.connectionType() != "WiFi"){
10    return {battery : $battery.level()}
11  } else return UNDEFINED;
12 }).rank(function(users) {
13    return users.sort("battery", function(x,y) { return y-x; });
14 }).geoCoverage([[50.614291,3.13282],[50.604159,3.15239]],"500 m")
15   .timeCoverage("30 min","1 hour")
16   .duplicate(2)
```

Listing 2.1: Monitoring of the Network Quality in Paris

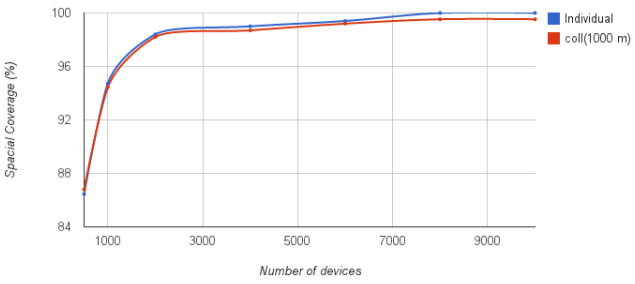
Algorithm 1 is triggered depending on the targeted coverage (methods `geoCoverage` and `timeCoverage`) specified by the script. It first checks that the redundancy property can be ensured ($|activeDevices| < duplicate$) before identifying all the active devices that are not already executing a context policy. These candidate devices are then notified to check their eligibility by executing the handler `accept` defined in Listing 2.1 and report the computed score (here the battery level). This score is used to rank all the device responses (using the handler `rank`), which are incrementally requested to execute the context policy for a given period until the expected number of redundant measures is reached.

Algorithm 1 APISENSE® Algorithm for Orchestrating Context Policies

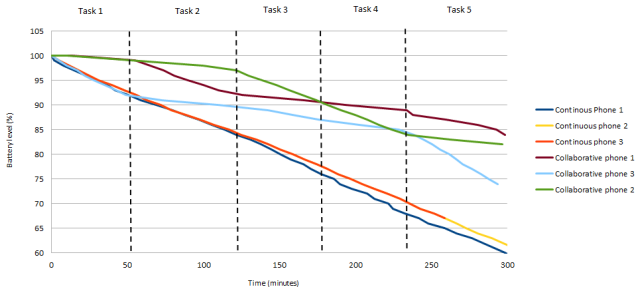
Require:
$connectedDevices$: List of connected devices
$activeDevices$: List of active devices
t : Time threshold
(t_{start}, t_{end}) : Temporal properties of the sensing task
$duplicate$: Maximum number of devices to assign the sensing task

if $|activeDevices| < duplicate$ **then**
  $candidateDevices \leftarrow connectedDevices \setminus activeDevices$
  if $|candidateDevices| \neq 0$ **then**
    $availableDevices \leftarrow \emptyset$
    broadcastTaskRequestEvent($candidateDevices$)
    repeat
      $device(id, properties) \leftarrow$ receive()
      $availableDevices \leftarrow availableDevices \cup device(id, properties)$
    until timeout t is reached
    for $i = 0 \rightarrow (|activeDevices| - duplicate)$ **do**
      $device \leftarrow$ ranking($availableDevices$)
      $activeDevices \leftarrow activeDevices \cup device$
      $availableDevices \leftarrow availableDevices \setminus device$
      notifyTaskExecutionRequest($device, t_{start}, t_{stop}$)
    end for
  end if
end if

In terms of performances, Figures 2.9a and 2.9b demonstrates that the use of context policy orchestration provides a geographic coverage that is equivalent to individual sensing, while it drastically improves the battery lifespan of involved mobile devices.



(a) Geographic coverage when using policy orchestration.



(b) Energy consumption per device depending on strategies.

Figure 2.9: APISENSE® Context Policy Orchestration Performances.

2.2.3 In-depth Context Monitoring and Processing in Real-time

Beyond the contributions we proposed for *in-breadth* context monitoring, we have also been considering *in-depth* context monitoring and processing in the area of green computing. Energy consumption of computers and software is becoming a major factor in designing, building and using sustainable technologies. The topic of energy is becoming mainstream, as more approaches, software, hardware and technologies are being proposed for energy management, optimization or measurement. *Information and Communication Technology* (or ICT) accounted for 2% of global carbon emissions in 2007 [Gar07] or 830 $MTCO_2e$ (Metric Tonne Carbon Dioxide

equivalent), and is expected to grow to 1,430 $MTCO_2e$ in 2020 [Web08]. On the other hand, in term of energy consumption, ICT consumed up to 7% of global power consumption (or 168 Gigawatt, or GW) in 2008 [Ver+10]. This number is expected to double by 2020 to 433 GW, or more than 14.5% of global power consumption [Ver+10]. These numbers show that although ICT could help to reduce energy consumption and carbon emissions of other domains, its own carbon footprint and energy consumption is predicted to rapidly grow. The need to optimize energy efficiency of ICTs is therefore a necessity for the next years and decades [NRS11].

Modern software and computer infrastructures are increasingly distributed, based on a wide diversity of devices, and require the usage of software services, either cloud-based or local. These software and devices are constantly powered up and connected, from mobiles devices, to servers in data centers, and desktop computers (*e.g.*, continuously powered during working hours or leisure time). While middleware solutions can be considered as a candidate to automatically optimize the energy consumption of such systems, *software ecodesign* remains a critical issue that has been weakly addressed over the last years, apart from mobile computing and wireless sensor networks. Nonetheless, offering such a support requires the development of appropriate tools and methodologies to understand the power consumption of software systems in depth. In particular, system administrators and developers require to understand how to split the power consumption of software along different artifact levels (processes, modules, classes, methods, etc.) in order to identify potential energy leaks and opportunities of optimizations.

We fully support this vision and we propose to apply our expertise in context-aware middleware solution to report and analyze the energy consumption of software systems at any scale: from runtime processes to method calls. We propose to advance the state-of-the-art in the area of green computing [NRS13a; NRS13b] by delivering a middleware solution that can provide accurate estimation of the power consumption in real-time without using any third-party power meter. In particular, we apply the principles of COSMOS to infer the power consumption of a software process from raw system metrics (resource utilization) provided by the operating system. Then, we show that such a context policy can be further extended to account for the power consumption of process internals like classes or methods.

2.2.3.1 The PowerAPI Middleware Solution

Methodology. For measuring the energy consumption of applications, we adopt a software-based estimation methodology, which is composed of four steps that are summarized in Figure 2.10:

1. We **collect utilization data of hardware resources**. This step is necessary for modeling the total energy consumption of the monitored hardware resources. For each resource, data is collected directly from the hardware interface (if available), or through the operating system APIs or tools;
2. We **use energy models in order to estimate the total energy consumption of the hardware resource**. These models compute the energy consumed by a hardware component based on its runtime characteristics (*e.g.*, voltage and frequency for a CPU), and its physical properties (*e.g.*, *Thermal Design Power* or TDP for a CPU);
3. We **collect resource utilization of the hardware resources by software applications**. We consider that each process (identified by a unique PID) is a separate application. Software running using multiple processes is managed by calculating the sum of the energy consumption of its processes. Resource utilization is monitored at runtime through the operating system. For example, we use the `proc` file system (or `procfs`) in Unix systems in order to collect information about the percentages of utilization of each CPU frequency by the monitored application;
4. We **apply our energy models to estimate the energy consumed by the application on a specific hardware resource**. The total energy consumption of software is therefore the sum of its energy consumption on each monitored hardware component.

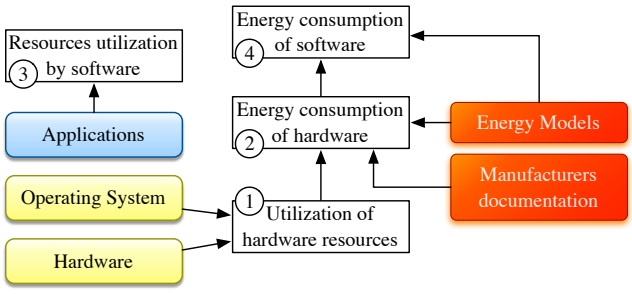


Figure 2.10: Methodology of measurement at application level.

CPU power model. We propose a power model for the CPU, which is based on the standard equation for modeling the power consumption of *Complementatry Metal Oxide Semiconductor* (CMOS) components:

$$P_{CPU}^{f,V} = c \times f \times V^2 \tag{2.1}$$

Where f is the frequency, V the CPU voltage and c a constant value depending on the hardware materials (such as the capacitance and the activity factor). Thanks to this relation, we note that power consumption does not always linearly depends on CPU utilization. This is due to *Dynamic Voltage and Frequency Scaling* (DVFS) and also to the fact that power depends on the voltage (and subsequently the frequency) of the processor. Therefore, a simple CPU utilization profiler is not enough in order to estimate the power consumption of the CPU or software. The dynamic variables in the standard model of Formula 2.1, frequency f and voltage V, are obtained through the OS at runtime, while the static variable c cannot be obtained directly. The latter is actually a set of data describing the physical CPU characteristics (*e.g.*, capacitance or activity factor). Manufacturers may provide this constant although in most cases it is missing. In order to compute this value, we use the existing relation between the overall power of a processor and its *Thermal Design Power* (or TDP) value. TDP represents the power required by the cooling system of a computer to dissipate the heat generated by the processor along execution. It is generally related to a extreme state, such as the maximum frequency and voltage. However, TDP is not a perfect estimation of the power consumption of a processor. According to [Riv+07], a factor of 0.7 is to be applied to the relation between the TDP and the power consumption. Therefore, the power consumption of the processor can be modeled as follows:

$$P_{CPU}^{f_{TDP},V_{TDP}} \simeq 0.7 \times TDP \tag{2.2}$$

Where f_{TDP} and V_{TDP} represent the frequency and the voltage of the processor within the *TDP state*, respectively. The benefit of using the TDP in our model is that TDP is a value provided by most manufacturers. Based on Formula 2.2, our model in Formula 2.1 can be used to estimate the constant c as follows:

$$P_{CPU}^{f_{TDP},V_{TDP}} = c \times f_{TDP} \times V_{TDP}^2 \simeq 0.7 \times TDP \tag{2.3}$$

thus c is modeled as:

$$c \simeq \frac{0.7 \times TDP}{f_{TDP} \times V_{TDP}^2} \tag{2.4}$$

Therefore, to compute the power consumption of a CPU, we apply the following model:

$$P_{CPU}^{f,V} = \frac{0.7 \times TDP}{f_{TDP} \times V_{TDP}^2} \times f \times V^2 \tag{2.5}$$

The Intel Pentium M processor have a TDP of 24.5 W for the maximum frequency of 1.6 GHz and voltage 1.484 V [Int04]. Thus, its constant c is estimated by Formula 2.4 to $4.86716803 \times 10^{-6}$.

CPU activity and power consumption. In order to estimate the power consumption of an application, we need to monitor its resources usage, in particular its CPU activity. We choose to identify applications by their processes, and the latter by their *Process IDentifiers* (PID). We calculate the process CPU activity as a ratio between CPU time for the PID and the global CPU time—*i.e.*, the time the processor is active for all processes—during a duration d, as follows:

$$U_{CPU}^{PID}(d) = \frac{t_{CPU}^{PID}}{t_{CPU}}(d) \tag{2.6}$$

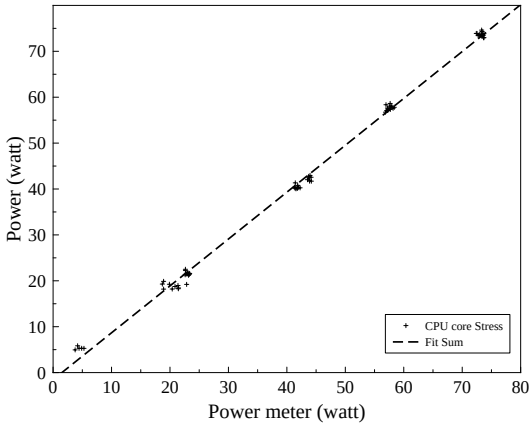
Finally, the power consumption of a process is the product of the power consumption of the CPU for all applications, with the CPU usage of the monitored PID. This product is modeled for a certain frequency as follows:

$$P_{CPU}^{f} = \frac{0.7 \times TDP}{f_{TDP} \times V_{TDP}^2} \times f \times V^2 \times \frac{t_{CPU}^{PID}}{t_{CPU}}(d) \tag{2.7}$$

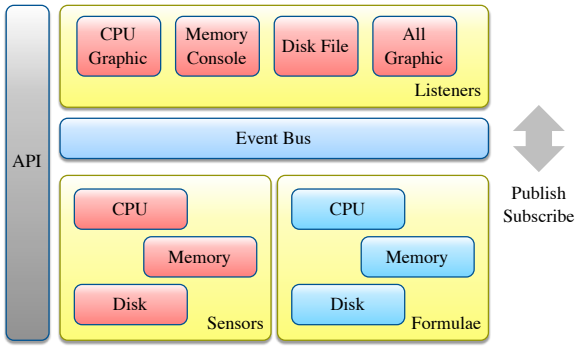
When the processor supports DVFS, the CPU power consumption for a process P_{CPU} is equal to the average of the CPU power of each frequency balanced by the CPU time of all frequencies:

$$P_{CPU} = \frac{\sum_{f \in frequencies} P_{CPU}^{f} \times t_{CPU}^{f}}{\sum_{f \in frequencies} t_{CPU}^{f}} \tag{2.8}$$

Implementation issues. We provide a system level library, called PowerAPI [Bou+13; BRS13; SR13], which implements our power models in order to measure the energy consumption of software at runtime. Each process can therefore be monitored for its power consumption with an accuracy equivalent to hardware power meters (cf. Figure 2.11a. The library also offers energy differentiation values based on hardware resources, such as giving the energy consumed by the process on the CPU, or on the network or on other supported hardware resources. PowerAPI's architecture is modular as each of its components is represented as a *power module* (see Figure 2.11b). A *sensor module* is responsible for gathering operating system related information for the module. For example, it gathers the number of bytes transmitted by the network card, and the time spent by the CPU at each of the processor frequencies (when DVFS is supported). A *formula module* uses the above power model to estimate the power consumption per process by using both information gathered by the sensor module and information based on hardware characteristics. PowerAPI is implemented in Scala[7] and is based on an event-driven architecture based on the Akka asynchrnous middleware[8].



(a) PowerAPI accuracy.



(b) PowerAPI architecture.

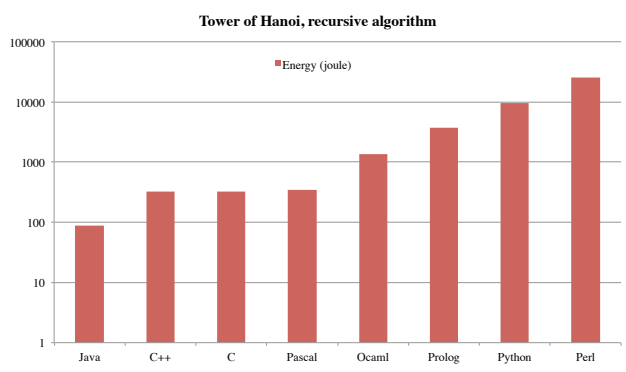
Figure 2.11: PowerAPI.

Technology impact. Thanks to PowerAPI, we can raise the awareness of software developers on the power consumption of the solution they developed. For example, we run an experiment to compare the energy consumption of several implementations of a same application, the Hanoi tower[9], which is available in 111 different versions. We measured several executions of a subset of these implementations to illustrate the impact of programming languages on the energy efficiency. Although they cannot be generalized without further investigations, these measures provide interesting insights to developers and even chief information officers on strategic decisions to be made with regard to software sustainability. In particular, while the current trend tends to encourage the adoption of scripting languages on the server side (*e.g.*, Node.js, Django), one can observe that the script interpreters may introduce a huge power overhead to the developed system. Even if this has to be balanced with other metrics like the learning curve or the maturity of the language, an orientation towards Java-based technologies would tend to minimize the energy footprint of the system. The efficiency of Java over native language like C or C++ can be explained by VM-level optimizations (in particular, JIT compilation), but the use of appropriate compiler options (*e.g.*, `-O2` or `-O3` for *gcc*) can benefit to native languages. We also investigate the impact of algorithms by comparing iterative and recursive strategies. In the specific case of the Tower of Hanoi, the recursive strategies exhibit a smaller energy footprint, independently of the options selected during the compilation. While this observations should carefully be considered and cannot be generalized as such, we rather advocate the benefit of supporting process-level power monitoring as a tool to advise developers on alternative implementations of their system.

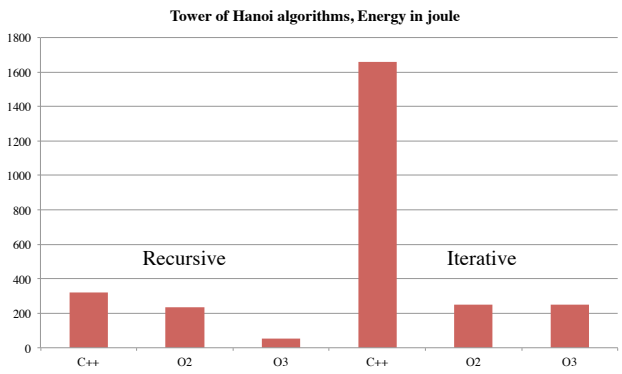
2.2.3.2 A Focus on Application Component Energy Consumption

Methodology. Beyond the evaluation of technologies and algorithmic strategies, our solution can also dive into the code of applications to spot potential energy hotspots or leaks. We provide this deeper level of power monitoring by extending the PowerAPI methodology with the following steps:

[7] `http://www.scala-lang.org`
[8] `http://akka.io`
[9] `http://www.kernelthread.com/projects/hanoi`



(a) Programming language impact.



(b) Algorithm strategies and compilation options impact.

Figure 2.12: Measuring the energy consumption of software.

1. We first collect statistics about the application's software and hardware resources utilization. Information, such as methods durations, CPU time, or the number of bytes transferred through the network card, are collected and classified at a finer grain, *e.g.*, for each method of the application.
2. Next, a correlation phase takes place to correlate the application-specific statistics with the application level energy information. Per-method energy consumption information is estimated using our power model.
3. Finally, the energy consumption per method is displayed to the user and can be exposed as a service.

Implementation. JALEN is an extension of POWERAPI implemented as a Java agent that hooks to the Java Virtual Machine during its start, and monitors and collects energy related information of the executed application. JALEN uses statistical sampling to collect metrics from the JVM on running methods, and to correlate them with our power models:

1. We first follow a two cycle approach: a *big* monitoring cycle where power consumption of software is gathered from application level monitoring using POWERAPI and a *small* monitoring cycle where statistical information is collected on each running method.
2. During the *small* monitoring cycle, we collect the number of times a method appears in our statistical sampling (measured at a higher frequency). For example, two method `AT` and `BT` are executing for 10 seconds, and the *big* cycle is 1 second and the *small* cycle is 10 milliseconds. The method `AT` is captured 7 times during the *small* cycle while `BT` is captured 3 times. Each of these methods have different execution times and CPU utilization, therefore both methods are scheduled and executed accordingly (for example, method `BT` waits for a network answer, thus the JVM executes `AT` during the wait).
3. We then correlate theses statistics with the CPU time of threads (gathered from the JVM), in order to estimate the energy consumption of methods.

JALEN can report three types of energy consumption on methods:

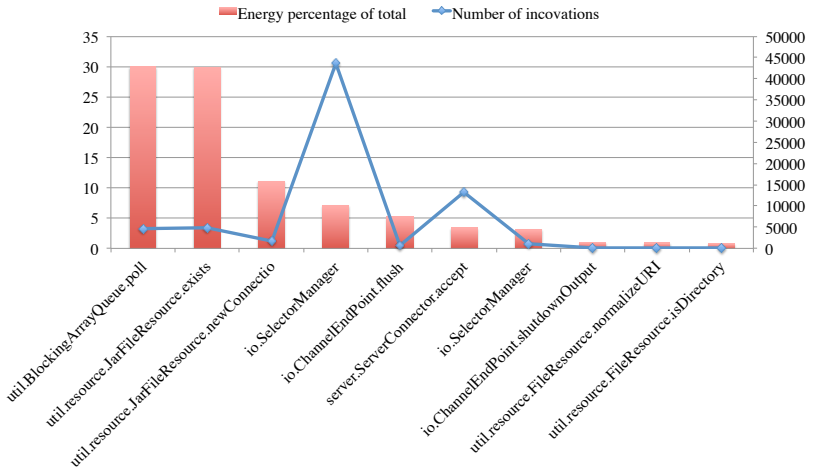
Gross energy refers to aggregated energy consumption, which means the energy consumption per method and including all the methods it invokes, including Java's JDK methods (such as `java.*` methods);

Net energy identifies the energy consumption of methods' statements as it excludes any other methods invocations from the estimation;

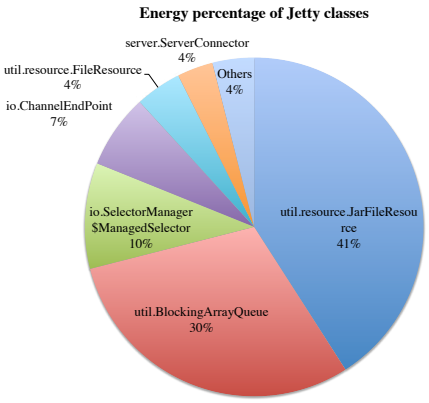
Library energy groups the net consumption of methods (filtered by canonical name) and can be used, for example, to monitor the energy consumption of a specific library used by the application.

Energy hotspots. We stress Jetty's asynchronous REST web application example (`async-rest`) using ApacheBench. The latter uses 25 concurrent users with 100,000 requests. The first observation from Figure 2.13a is that the 10 most energy consuming methods of Jetty consume the vast majority of the energy: 92.18%. Specifically, two methods consume nearly 60% of the energy: `org.eclipse.jetty.util.BlockingArrayQueue.poll` (29.92%) and `org.eclipse.jetty.util.resource.JarFileResource.exists` (29.88%). Five other methods (`util.resource.JarFileResource.newConnection`, `io.SelectorManager$ManagedSelector.select`, `io.ChannelEndPoint.flush`, `server.ServerConnector.accept`, and `io.SelectorManager$ManagedSelector.wakeup`) consume between 3% and 11%, while the energy consumption of the remaining methods is negligible (less then 1%).

In addition to detecting hotspots at the methods level, our approach can detect most energy consuming classes. Figure 2.13b outlines the 6 most consuming classes of Jetty during our experimentation. These



(a) Method-level energy consumption.



(b) Class-level energy consumption.

Figure 2.13: Measuring the energy consumption of software.

6 methods consume together 96.02% of the total energy consumed by Jetty classes. The remaining 129 classes consume the rest: 3.97%. We observe that two classes consumes more than 70% of the energy: `util.resource.JarFileResource` (40.93%) and `util.BlockingArrayQueue` (30.07%).

The detailed results of the experiments we performed with PowerAPI are available in [Nou+12a; NRS14b]

2.3 Synthesis

This chapter reports on our main contributions in the area of scalable context monitoring and processing. The roots of this work, namely COSMOS, build on a collaboration established with Denis Conan from Institut Mines-Télécom, Télécom SudParis and have been further developed as part of the Cappucino collaborative project[10] with Auchan, Norsys, and SI3SI. The results of this work have been published at DAIS'07 conference [CRS07] and IEEE Distributed Systems Online [RCS08]. Based on these foundations, we have been investigating both *in-breadth* and *in-depth* context monitoring.

As part of the PhD thesis of Nicolas Haderer [Had14], we contributed to the delivery of an open crowd-sensing platform to support the monitoring of context in breadth by leveraging the wisdom of the crowd and using the smartphones of volunteer participants. While this work has been published at Cloud'12 [Par+12a] and DAIS'13 [HRS13b] conferences, it is worth to notice that the developed solution resonates in other scientific communities. For example, we keep collaborating with Simon Charneau, Alan Ouakrat, and Vassily Rivron on the PRACTIC experiment[11], which proposed through the APISENSE® platform [Had+13a; Had+13b]. The APISENSE® platform is also already used by the telecom industry and the ip-label[12] company to monitor the quality of Internet access from mobile devices. The APISENSE® platform is currently supported by Inria through the ADT (*Action de Développement Technologique*) AntDroid (2012–2014) and CrowdLab (2014–2016).

As part of the PhD thesis of Adel Noureddine [Nou14], we considered the issue of inferring the power consumption of software components in depth. Initiated during the EconHome collaborative project[13] with OrangeLabs, STMicroelectronics, Comsis, our work on providing power consumption estimation has been released as an Open-Source Software on GitHub (`http://www.powerapi.org`). Published at the ASE'12 conference [Nou+12b] and in the Journal of Automated Software Engineering [NRS14a], this solution has also been selected by the *Web Energy Archive* (WEA) project[14], funded by ADEME, to report on the power consumed by end-user when browsing 100 selected websites. The PowerAPI library is currently supported by Inria through the ADT eSurgeon (2013–2015).

[10] funded by Fonds Unique Interministériel (FUI)
[11] `http://beta.apisense.fr/practic`
[12] `http://www.ip-label.com`
[13] funded by Fonds Unique Interministériel (FUI)
[14] `http://webenergyarchive.com`

PhD thesis supervisions associated to this chapter

[Had14] Nicolas Haderer. "APISENSE : une plate-forme répartie pour la conception, le déploiement et l'exécution de campagnes de collectes de données sur des terminaux intelligents". PhD thesis. Université Lille 1, Sciences et Technologies, Nov. 2014.

[Nou14] Adel Noureddine. "Towards a Better Understanding of the Energy Consumption of Software Systems". PhD thesis. Université Lille 1, Sciences et Technologies, Mar. 2014.

Publications associated to this chapter

[Bou+13] Aurélien Bourdon, Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. "PowerAPI: A Software Library to Monitor the Energy Consumed at the Process-Level". In: *ERCIM News* 92 (Jan. 2013), pp. 43–44.

[Bra+07b] Gunnar Brataas, Svein Hallsteinsen, Romain Rouvoy, and Frank Eliassen. "Scalability of Decision Models for Dynamic Product Lines". In: *Proceedings of the International Workshop on Dynamic Software Product Line (DSPL)*. Sept. 2007, pp. 23–32.

[BRS13] Aurélien Bourdon, Romain Rouvoy, and Lionel Seinturier. "Mesurer la consommation en énergie des logiciels avec précision". In: *01 Business & Technologies* (Jan. 2013).

[CRS07] Denis Conan, Romain Rouvoy, and Lionel Seinturier. "Scalable Processing of Context Information with COSMOS". In: *7th IFIP International Conference on Distributed Applications and Interoperable Systems*. Paphos, Cyprus, 2007, pp. 210–224.

[CRS08] Denis Conan, Romain Rouvoy, and Lionel Seinturier. "COSMOS : composition de noeuds de contexte". In: *Technique et Science Informatiques (TSI)* 27.9-10 (2008), pp. 1189–1224.

[Had+13a] Nicolas Haderer, Christophe Ribeiro, Romain Rouvoy, Simon Charneau, Vassili Rivron, Alan Ouakrat, Sonia Ben Mokhtar, and Lionel Seinturier. "Le capteur, c'est vous !" In: *L'Usine Nouvelle* 3353 (Nov. 2013), pp. 74–75.

[Had+13b] Nicolas Haderer, Romain Rouvoy, Christophe Ribeiro, and Lionel Seinturier. "APISENSE: Crowd-Sensing Made Easy". In: *ERCIM News* 93 (Apr. 2013), pp. 28–29.

[Had+14] Nicolas Haderer, Fawaz Paraiso, Christophe Ribeiro, Philippe Merle, Romain Rouvoy, and Lionel Seinturier. "A Cloud-based Infrastructure for Crowdsourcing Data from Mobile Devices". In: *Cloud-based Software Crowdsourcing*. Ed. by Wenjun Wu. Springer, 2014.

[HRS13a] Nicolas Haderer, Romain Rouvoy, and Lionel Seinturier. "A preliminary investigation of user incentives to leverage crowdsensing activities". In: *2nd International IEEE PerCom Workshop on Hot Topics in Pervasive Computing (PerHot)*. San Diego, United States: IEEE Computer Society, Mar. 2013, pp. 199–204.

[HRS13b] Nicolas Haderer, Romain Rouvoy, and Lionel Seinturier. "Dynamic Deployment of Sensing Experiments in the Wild Using Smartphones". In: *13th International IFIP Conference on Distributed Applications and Interoperable Systems (DAIS)*. Ed. by François Taïani and Jim Dowling. Vol. 7891. LNCS. Firenze, Italy: Springer, June 2013, pp. 43–56.

[Nou+12a] Adel Noureddine, Aurélien Bourdon, Romain Rouvoy, and Lionel Seinturier. "A Preliminary Study of the Impact of Software Engineering on GreenIT". In: *First International Workshop on Green and Sustainable Software*. Zurich, Switzerland, June 2012, pp. 21–27.

[Nou+12b] Adel Noureddine, Aurélien Bourdon, Romain Rouvoy, and Lionel Seinturier. "Runtime Monitoring of Software Energy Hotspots". In: *ASE - The 27th IEEE/ACM International Conference on Automated Software Engineering - 2012*. Essen, Germany, Sept. 2012, pp. 160–169.

[NRS09] Russel Nzekwa, Romain Rouvoy, and Lionel Seinturier. "Towards a Stable Decision-Making Middleware for Very-Large-Scale Self-Adaptive Systems." In: *BElgian-NEtherlands software eVOLution seminar (BENEVOL)*. Louvain-la-Neuve, Belgium, 2009.

[NRS10a] Russel Nzekwa, Romain Rouvoy, and Lionel Seinturier. "A Flexible Context Stabilization Approach for Self-Adaptive Application". In: *COMOREA - (PERCOM)*. Mannheim, Germany, Mar. 2010, pp. 7–12.

[NRS11] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. "Supporting Energy-driven Adaptations in Distributed Environments". In: *1st Workshop on Middleware and Architectures for Autonomic and Sustainable Computing*. Paris, France, May 2011, pp. 13–18.

[NRS13a] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. “A review of energy measurement approaches”. In: *ACM SIGOPS Operating Systems Review* 47.3 (Dec. 2013), pp. 42–49.

[NRS13b] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. “A Review of Middleware Approaches for Energy Management in Distributed Environments”. In: *Software: Practice and Experience* 43.9 (Sept. 2013), pp. 1071–1100.

[NRS14a] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. “Monitoring Energy Hotspots in Software”. In: *Journal of Automated Software Engineering* (2014).

[NRS14b] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. “Unit Testing of Energy Consumption of Software Libraries”. In: *Symposium On Applied Computing*. Gyeongju, Korea, Republic Of, Mar. 2014, pp. 1200–1205.

[Par+12a] Fawaz Paraiso, Nicolas Haderer, Philippe Merle, Romain Rouvoy, and Lionel Seinturier. “A Federated Multi-Cloud PaaS Infrastructure”. In: *5th IEEE International Conference on Cloud Computing*. hawaii, United States, June 2012, pp. 392–399.

[Par+12b] Fawaz Paraiso, Gabriel Hermosillo, Romain Rouvoy, Philippe Merle, and Lionel Seinturier. “A Middleware Platform to Federate Complex Event Processing”. In: *Sixteenth IEEE International EDOC Conference*. Beijing, China: Springer, Sept. 2012, pp. 113–122.

[Qui+13] Clément Quinton, Nicolas Haderer, Romain Rouvoy, and Laurence Duchien. “Towards Multi-Cloud Configurations Using Feature Models and Ontologies”. In: *1st International Workshop on Multi-Cloud Applications and Federated Clouds*. Prague, Czech Republic, Apr. 2013, pp. 21–26.

[RCS08] Romain Rouvoy, Denis Conan, and Lionel Seinturier. “Software Architecture Patterns for a Context-Processing Middleware Framework”. In: *IEEE Distributed Systems Online* 9.6 (2008), pp. 1–13.

[RM09] Romain Rouvoy and Philippe Merle. “Leveraging Component-Based Software Engineering with Fraclet”. In: *Annals of Telecommunications* 64.1-2 (Jan. 2009).

[SR13] Lionel Seinturier and Romain Rouvoy. “Informatique : Des logiciels mis au vert”. In: *J’innove en Nord Pas de Calais* (Nov. 2013).

[Tah+08b] Amirhosein Taherkordi, Romain Rouvoy, Quan Le-Trung, and Frank Eliassen. “A self-adaptive context processing framework for wireless sensor networks”. In: *Proceedings of the 3rd international workshop on Middleware for sensor networks*. ACM. 2008, pp. 7–12.

Chapter 3

Self-Adaptation of Ubiquitous Systems

3.1 Motivations

Ubiquitous software systems have the capacity to exploit the resources discovered in their vicinity to enhance the experience of end-users. While the discovery of nearby resources builds on a rich family of established service discovery protocols like *Universal Plug-and-Play* (UPnP) [Don+08], *Service Location Protocol* (SLP) [Gut99], or Bluetooth SDP [Cha+10], the integration of discovered resources is often left to the responsibility of the developer. Although some works [Rav+06; FGB11] have been focusing on unifying the discovery of heterogeneous resources, including semantic [Mok+08] and interoperability [Cap+14] issues, this still implies that a developer is expected to be aware *a priori* of potential resources to be met by the end-user and therefore to adjust the behavior of the software system depending on the availability of these resources.

However, the rise of the *Internet of Things* (IoT) and the growing connectivity of devices open up the opportunities for building opportunistic distributed systems that can optimize themselves according to specific objectives (*e.g.*, quality of service, power consumption). To go beyond the development of predictable configurations working with predefined *things*, ubiquitous software systems should develop the capability to reason on their configuration and on their environment in order to take appropriate decisions on behalf of the end-user. While automatic decision-making requires the delivery of up-to-date contextual information (cf. Chapter 2), it also requires the availability of fine-grained reconfiguration mechanisms to support the self-optimization of the system at runtime. In particular, we promote software components in the small and in the large (cf. Chapter 1) as a suitable substrate to support the self-adaptation of self-adaptive ubiquitous software systems. Based on these two building blocks that we previously described, we propose to close the loop by investigating different approaches to address the issue of decision-making in ubiquitous environments.

The objective of this chapter is to report on the contributions we developed in the area of self-adaptive ubiquitous systems. Starting from *white-box* approaches which are able to reason on the software architecture of a ubiquitous system, we shift towards *black-box* solutions that control the self-adaptation of a system at the edges, using so called *touchpoints*. This shift also highlights our perception and the associated implementation of the principles of autonomic computing community [Kep05]. In particular, we advocate the adoption of explicit *feedback control loops* as first class entities of future software systems. By making explicit the dynamics of software systems, we believe that we do not only consider different implementations of the decision-making layer—including approaches borrowed from the control theory—but we can also support the continuous evolution of self-adaptive software systems, from early system identification phases, to adaptive control strategies. This approach allows system administrators to graft and compose self-adaptive policies onto legacy software systems.

3.2 Contributions

While keeping in mind the context modeling approaches we introduced with COSMOS (cf. Section 2.2.1), we illustrate how such a model dedicated to context inference has been evolving towards a model for modeling reflective feedback control loops. We therefore start by reporting on the premises of self-adaptive ubiquitous middleware that we developed as part of the QuA and MUSIC middleware solutions (cf. Section 3.2.1). Then, we introduce the concept of ubiquitous feedback control loop (cf. Section 3.2.2) as a solution to adapt the decision-making layer to the environment. We conclude by proposing a modeling approach that bends context policy to implement an explicit control layer for legacy systems (cf. Section 3.2.3).

3.2.1 White-Box Self-Optimisation of Software Architectures

In this work, we consider ubiquitous systems designed and developed using software components (cf. Chapter 1), while raw context data is collected in real-time by a middleware solution deployed on the device (cf. Chapter 2).

Planning-based adaptation principles. Planning-based adaptation of a component-based application refers to the capability of a system to adapt to changing user needs and operating conditions by exploiting knowledge about its composition and *Quality of Service* (QoS) characteristics of its constituting components [Flo+06; GER08; Gei+09]. In this approach, this knowledge is provided in the form of a QoS-aware model, which describes the abstract composition of software components, the relevant QoS dimensions and how they are affected when varying the actual component configuration. This model is exploited by the adaptation middleware to select, connect, and deploy a configuration of component realizations providing the best utility [Bra+07a; SR07; KD07]. The utility measures the degree of fulfillment of user preferences while optimizing device resource utilization. The model describes the abstract composition as a set of *roles* collaborating through *ports*, which represent either functionality provided to or required from collaborating components. Properties and *property predictor* functions associated with the ports define how the QoS properties and resource needs of components are influenced by the QoS properties of the components they depend on. A port has a *type* defining the functionality represented by the port in terms of interfaces and protocol. Component realizations implement ports and a component realization can be used in a role if the ports match (same type). Component realizations are *atomic* or *composite*. A composite realization is itself an abstract composition and allows for recursive decomposition. Constraints are predicates over the properties of the constituting components of a composition, which restrict the possible combinations of component realizations (*e.g.*, configuration consistencies) [Flo+06; KRG08; Gei+09]. The model is represented at runtime as *plans* within the middleware. A plan reflects a component realization and describes its ports and associated property predictors as well as implicit dependencies on the hosting platform (*e.g.*, platform type and version). In the case of an atomic component realization, it also contains a reference to the class, which realizes the component. In the case of a composite realization, the plan describes the internal structure in terms of roles and ports and the connections between them. Variation is obtained by describing a set of possible alternative realizations of the roles.

Then, planning refers to the process of selecting the components that make up an application configuration providing the best possible utility to the end-user. This process will be triggered at start-up of the application and at run-time when the execution context suddenly changes. When such an adaptation process is triggered for a particular type, the planning middleware iterates over the plans associated to the roles. For each plan, it resolves the plan dependencies and evaluates the configuration suitability to the current execution context by computing the predicted properties. The predicted properties are input to the normalized utility function that computes the expected utility of the evaluated application configuration [Flo+06; GER08; Gei+09]. The utility function of an application is provided by the developer and is typically expressed as a weighted sum of dimensional utility functions where the weights express user preferences—*i.e.*, relative importance of a dimension to the user. A dimensional utility function measures user satisfaction in one property dimension.

Plans offer an interesting abstraction to reason on the quality of a wide variety of software artifacts. While we have already demonstrated the reflective nature of component-based software architectures to integrate other software engineering paradigms [Loi+11a; Loi+11b; Sei+12] (cf. Section 1.2.2), we have adopted a similar approach to reason on heterogeneous software artifacts and thus implement self-adaptive behaviors that can automatically weave aspects [RBE08; Ali+10; REB09], plug context sensors [Pas+08], reconfigure the underlying middleware [Rou+08a], or integrate dependability mechanisms [RVE08; REB09] based on the work done as part of the PhD thesis of Eli Gjørven [GER08; GRE08]. In the remainder of this section, we explore on the integration of ubiquitous services to opportunistically exploit external services whenever the quality they provide improve the overall utility of the ubiquitous application. The proposed approach therefore consists in spontaneously exploiting remote services in order to relieve the mobile device and improve the overall quality of service. This approach complements connector-based solutions [Gra+11] that enable the interoperability of services that were not designed to interact.

3.2.1.1 Discovery and Integration of Third-Party Services

Consuming ubiquitous services. In SOA-based computing environments, an application typically uses one or more services, which possibly depend on further services and so on. Thus, a large number of computers owned and administrated by different organizations may potentially be involved. This problem is aggravated when we deal with several applications running concurrently. Thus, optimizing utility over the entire set of involved computers is likely to be intractable both from a technical and administrative point of view. Therefore, we have to delineate the scope of an adaptation to be more tractable. To this end, we introduce the notion of adaptation domain and the distinction between internal and external services. An adaptation domain is

a collection of platform instances controlled by one adaptation manager. It includes one distinguished node (*e.g.*, a handheld device), which represents a permanent binding to a user. This node acts as the nucleus around which the adaptation domain forms dynamically as auxiliary nodes come and go. The movement of nucleus nodes or changes in connectivity due other phenomena causes the dynamic evolution of an adaptation domain. Adaptation domains may overlap in the sense that auxiliary nodes may be members of multiple adaptation domains. This adds to the dynamics and increases the complexity because the amount of resources the auxiliary nodes are willing to provide to a particular domain may vary depending on the needs of other served domains. The user of a nucleus node may start and stop applications or shared components, and the set of running components is adapted by the adaptation manager according to these user actions and context changes, taking into account the resource constraints. Clearly, it makes a difference whether a role is bound by instantiating a component implementation running in the adaptation domain where a system is built (private instance), by using a service provided by a component instance already running there (internal service), or by connecting to a service provided by a third party (external service). In the first two cases, the adaptation manager building the system must provision the resources and has control of the provided service level. In the latter case, the service level is outside the control of the adaptation manager, and it is necessary to negotiate a *Service Level Agreement* (SLA) with the service providers in order to compare the suitability of services by different providers and weight against deploying an internal service. External services may be provided by other adaptation domains or by third party providers (also referred to as external or legacy services) according to the following process:

Discovery of services and service levels. As already mentioned, providers make their services accessible to third parties according to specific discovery protocols. The middleware platform supports an extensible set of discovery protocols allowing the detection of services available in the service landscape. The discovery of a service triggers the retrieval of its service description, which includes information on the service capabilities, semantics, and possibly the offered service level(s) or QoS properties in form of an agreement template. The service description and, if available, the related agreement template are then converted to service plans, each one reflecting an alternative realization for the service level;

SLA negotiation. The planning phase involves the evaluation of the available plans, for selecting the composition optimizing the utility of the applications running on the device. The utility depends on the QoS properties predicted by the services, whose value can be static or dynamic. Static properties consist of fixed values that do not change over the time. Dynamic property values can change according to the current status of the service. Evaluating the actual QoS values for such properties requires a process of negotiation with the service provider. The current negotiation protocol is inspired by the WS-Agreement specification [HLW09] (for both the definition and the creation/monitoring of SLAs), where the provider enriches the service description with an agreement template and the consumer fills in the template to create and submit an agreement offer. The offer creation is driven by *Service Level Objectives* (SLO), which are conditions defined at application or configuration level and act as pre-defined criteria for negotiating an SLA contract. Once the provider has accepted the offer, the agreed property values are reflected in the plan;

SLA provisioning. Whenever a service available in the landscape is selected for use as a result of the adaptation reasoning, the middleware platform instantiates service proxies. These proxies act as local representatives of the remote services and encapsulate the communication protocol necessary to access them in a location-transparent way. They are created by a binding framework, which provides dedicated proxy factories. Each factory supports a particular communication protocol to export or import a service. During the binding phase, the SLA contract associated with the selected plan is provisioned and enforced by the involved parties, which includes the reservation of computing resources and the deployment of SLA monitoring facilities [KL03; Erl05; MPM05];

SLA monitoring. For the purpose of SLA monitoring, the service proxy is instrumented with appropriate monitoring mechanisms according to the content of the SLA contract (*e.g.*, response delay, result quality). Both parties are responsible for checking the status of the agreement and for taking proper actions in case of violation of the agreement. Thus, after the creation of an agreement, the middleware platform, at any given time, must be able to check the current state of the agreement itself. When an agreement is not fulfilled anymore, the middleware platform must terminate it and trigger a new adaptation process in order to detect a new set of available services and to select among them the best candidate to replace the one breaking the contract. SLA-enabled service providers handle the state model of an agreement and of its constituting terms, and make them accessible to consumers in form of readable properties of the agreement.

On the consumer side, the middleware architecture is responsible for checking the state of an agreement according to predefined policies (*e.g.*, at given intervals or when detecting that the expected performance of a service is degrading). By querying the service provider for the agreement state, it is possible to detect whether the agreement has been violated or not. In case of violation, the consumer terminates explicitly the agreement

by invoking a terminate operation on the provider side (since there might be costs associated to the usage of the service), and discards the related service plan, hence triggering a new adaptation process.

Providing ubiquitous services. Hosting both applications and components providing services to the outside world in an adaptation domain complicates the adaptation reasoning. In addition to the user owning the device, there are also external service consumers, which may have conflicting needs (expressed in the SLA). Fortunately, the utility function approach lends itself quite naturally to cope with such situations. Our solution is to treat shared components providing services to external clients in the same way as applications and equip them with their own utility function, computing the degree of fulfillment of active SLAs. Using the weights, the overall utility function balances the utility to the owner of the device against the utility to service clients. This information about user preferences is included in user profiles. Another difficulty is related to property prediction. For shared services, the resources needed by the component to guarantee a certain QoS often depend on the number of consumers. Hence, property predictor functions for shared services must take this into account. The key step for providing ubiquitous services includes:

Publishing of services and service levels. By publishing its description using the discovery protocols supported by the MUSIC platform, a service running on a node can be made available to other nodes within the adaptation domain. Each service description encloses the service type as well as an agreement template describing the static QoS properties that are provided by this service. QoS dimensions referring to dynamic properties of the application are unbound in order to be fixed at a later time depending on the capabilities and the processing load of the hosting node;

SLA negotiation. The MUSIC platform supports the negotiation of agreements by playing the role of a service provider. Whenever a service consumer selects one of the published services, the MUSIC platform receives an agreement offer for consuming this service. The MUSIC platform applies the negotiation heuristics to decide whether to accept or reject this offer by taking the current resource availability into account. This heuristics predicts the impact of accepting the offer with regards to agreements that have been already accepted. If the resulting impact does not trigger any violation of previous agreements, the MUSIC platform creates an agreement, which keeps track of the negotiation process;

SLA provisioning. When a service consumer requests an internal service, the MUSIC platform checks that the requested service refers to an accepted agreement. Then, the binding framework instantiates a service skeleton—*i.e.*, a local representative of the service consumer—which reflects the ongoing agreement and implements one of the supported communication protocols (*e.g.*, SOAP or RMI). Invocations received via the service skeleton are delegated to the service instance locally deployed on the node;

SLA monitoring. Depending on the negotiated properties agreed in the agreement, the service skeleton is instrumented with context sensors, which are responsible for monitoring the agreement. The MUSIC platform provides a library of sensors for observable properties (*e.g.*, invocation latency) as part of its context middleware. If one of the sensors detects a violation in one of the dimensions of the agreement, it notifies the MUSIC platform about this violation, which results in the notification of the service consumer and the termination of the agreement.

3.2.1.2 MUSIC: Optimisation Driven by the Quality of Service

To support the above-mentioned SOA principles [Erl05], we have integrated new components into the MUSIC Platform (cf. Figure 3.1), which has been developed as part of European research project. As MUSIC is independent of a particular technology, various implementations of these components can be developed (*e.g.*, Web Service, CORBA, RMI, or UPnP).

Discovery support. More specifically, the Service Discovery is responsible for publishing and discovering services using different discovery protocols. The Remoting Service is responsible for the exporting of services at the service provider side, and for the binding to these services at the service consumer side. Whenever a service is exported, it is enabled to accept requests from (remote) service consumers. Each *service description* defining the provided functionalities and containing the necessary information for the consumer to access the service can be published by the service discovery. If the service provider offers additional guarantees for the published services, *agreement templates* are published in addition to the service description.

The service discovery supports the dynamic registration of *discovery listeners*. A discovery listener can have interest for particular services and can enforce customized policies to handle them. For example, the *Remote Platform Discovery Listener* is particularly interested in finding remote instances of the MUSIC platform in order to provide information about the MUSIC platforms connected to the applications. The *SLA Discovery Listener* is interested in finding services accompanied with an SLA support. Upon the discovery of services, the service discovery notifies the registered discovery listeners by passing them the service descriptions. Since plans are the base for the Adaptation Manager to perform planning-based adaptation, the discovery listeners create

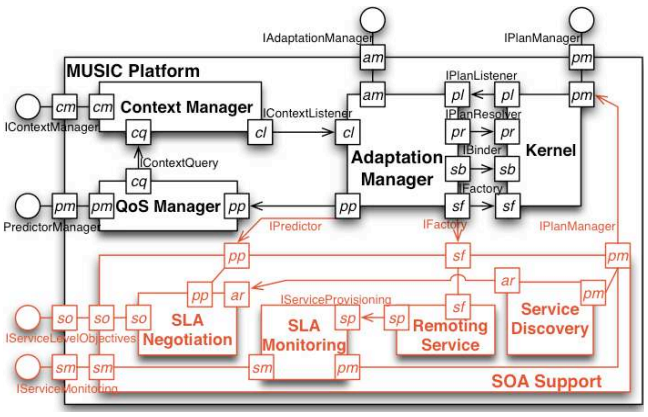


Figure 3.1: SOA configuration of the MUSIC platform.

service plans based on the service descriptions and the agreements negotiated by the SLA Negotiation. Plans for remote services are generated whenever services are discovered; hence plans are available when the adaptation manager triggers an adaptation at a later time. Plans are automatically discarded and removed from the Plan Repository whenever remote services disappear or for some reason become unavailable to the middleware.

The distributed instances of the MUSIC platform form a federation such that the service discovery on different platforms can interact with each other. Hence, MUSIC platform A can be aware of a service, which is published using a protocol supported by MUSIC platform B and not supported by A. If the remoting service on platform A supports the appropriate communication protocol, A is able to bind to that service which it would not be able to discover alone.

Agreement templates can be either static or allow for dynamic negotiation. Furthermore, a service may be offered at a predefined set of service levels. When the service discovery detects such a service, it first generates an abstract service plan enclosing structural and behavioral metadata related to the service. Then, in order to reflect the alternative service levels, the service discovery publishes an extended version of the service plan for each service level into the plan repository. Such a service level plan inherits the metadata of the service from the abstract service plan and extends it with the additional QoS properties described by the particular service level (*e.g.*, service accuracy and cost).

Adaptation support. The adaptation manager is then able to compare each available service level when applying the reasoning heuristics. Since service negotiation is a time critical factor for an efficient planning process, it should be resolved as soon as possible. In MUSIC, the negotiation is generally performed during service discovery for static QoS properties (*e.g.*, service cost) described by the service levels. The resulting static QoS property values are included into the service plan such that the predicted properties can automatically report them at a later time. However, in presence of a flexible service level [KL03; HLW09], the negotiation becomes dynamic, meaning that the SLA is negotiated during the planning process. Dynamic negotiation is particularly required when the adaptation manager needs to reason about up-to-date QoS properties (*e.g.*, current service accuracy). In this case, the predicted properties, when evaluated by the reasoning heuristics, delegate the negotiation of the requested property to the SLA negotiation. The negotiation protocol is driven by SLOs, which are predefined criteria for negotiating SLA [KL03].

Reconfiguration support. The Configuration Executor generally iterates over the plans composing the new configuration in order to reconfigure the application. As previously described, the configuration executor distinguishes between plans which refer to available services and plans which refer to services that are not available yet. In order to benefit from remote services, the configuration executor now faces a third case: If the plan refers to a remote service available in the environment, the configuration executor uses the Remoting Service to generate a specific component that will act as a service proxy. A service proxy is a local representative of the remote service. In particular, it implements the service type described by the application components and encapsulates the communication necessary to access the remote service. By invoking the service proxy, a service consumer interacts with the remote service in a location-transparent way—*i.e.*, as if the remote service is a local one.

The remoting service supports the dynamic integration of binding frameworks. During the binding phase, the SLA associated with the selected plan is provisioned and enforced by the involved parties. For the purpose of monitoring, the service proxy is instrumented with appropriate monitoring mechanisms by the component

SLA Monitoring according to the content of the SLA (*e.g.*, response delay, result quality). The SLA monitoring is responsible for checking the status of the agreement for taking proper actions in case of its violation.

Monitoring support. As an example of performing SLA monitoring in ubiquitous environments, the service proxy implements a disconnection detection algorithm. This disconnection support is inspired by the principles of ambient programming [Ded+06]. When loosing the connection to a remote service, the proxy stores the incoming service requests in a queue and returns a non-blocking *future object* to the application. The future object includes actions that are triggered whenever the connection is resolved to process the result of the request. If the connection is lost for a long period, the service proxy terminates the agreement via the component SLA negotiation. Subsequently, the SLA monitoring removes the associated service level plan from the plan repository to trigger an adaptation of the application. During the reconfiguration process, the request queue is transferred to the new component (or service proxy) that will be selected and deployed by the middleware.

Discussion. While the above approach provides an interesting abstraction level for reasoning on the underlying software architecture and the variability dimension that can be considered (components, aspects, remote services, etc.), it offers a limited visibility on the adaptation policy used to operate the control of the ubiquitous application. Furthermore, in this approach, the reasoning process is driven by the mobile device based on its own knowledge of alternative configurations and cannot import the rules, constraints, or knowledge related to the surrounding environment. In the following section, we therefore go beyond this solution by making the adaptation process ubiquitous, thus releasing and externalizing the control beyond the boundaries of the mobile device.

3.2.2 White-Box Self-Adaptation in Ubiquitous Environments

To reify the adaptation process, we employ the MAPE-K model as well as the notion of business process [Coa99; Coa99]. In particular, regarding the definition of process, we can see that the realization of context-based adaptation requires the execution of a sequence of tasks, in a specific order to meet the goal of changing the structure or/and behavior of applications according to the environment state. Furthermore, the distributed nature of the adaptation and the consequent modularization of the adaptation responsibilities (*i.e.*, monitoring, analysis and execution) bring into play different participants as well as the roles that they need to hold. Therefore, to build our solution, we start by modeling this kind of adaptation as a process.

Adaptation as a process. Figure 3.2 depicts this process following the BPMN notation [Whi04]. As it can be seen, we identify four main roles: *Information Source*, *Context Provider*, *Adaptation Orchestrator* and *Application Client*. The *Information Source* role is held by entities in the environment providing relevant information for the adaptation process. Mobile devices and sensors are examples of entities having this role. For its part, the *Context Provider* role has responsibilities associated with the collection of data from the different sources, the processing of this data and the production of context information that will enable the *Adaptation Orchestrator* for determining the required configurations on the *Client Application*. Depending on how the adaptation is tackled, the roles of *Context Provider* and *Adaptation Orchestrator* can be held by the same entity. Consequently, the functionality for executing the adaptation can be distributed between different entities:

Data gathering belongs to *monitoring* phase of the MAPE-K model and consists in collecting raw data from information sources. Once the data is gathered, this block detects if there is a significant change in the context to trigger the adaptation.

Context processing consumes the data collected in the data or context gathering tasks to produce high level information that will be used to decide the required reconfigurations.

Context gathering consists in the retrieval of the context information for identifying adaptation situations. The information can be explicitly requested from the context provider (*pull* mechanism) or the context consumer can be notified (*push* mechanism);

Identification of the new configuration for the adaptive application is computed by using the collected context information. The context processing, context gathering and identification of the new configuration tasks compose the *analysis* phase;

Determination of the configuration script is required, once the required configuration is established, to identify the atomic actions to execute in order to meet the target configuration (*e.g.*, adding, deleting and/or replacing a component). This task is part of the *planning* phase of the MAPE-K model;

Adaptation information gathering allows the collection of reconfiguration scripts. The task makes part of the *execution* phase together with the adaptation execution task;

Adaptation Execution is the final task of the adaptation process and consists in the reconfiguration of the ubiquitous application—*i.e.*, the execution of the reconfiguration script.

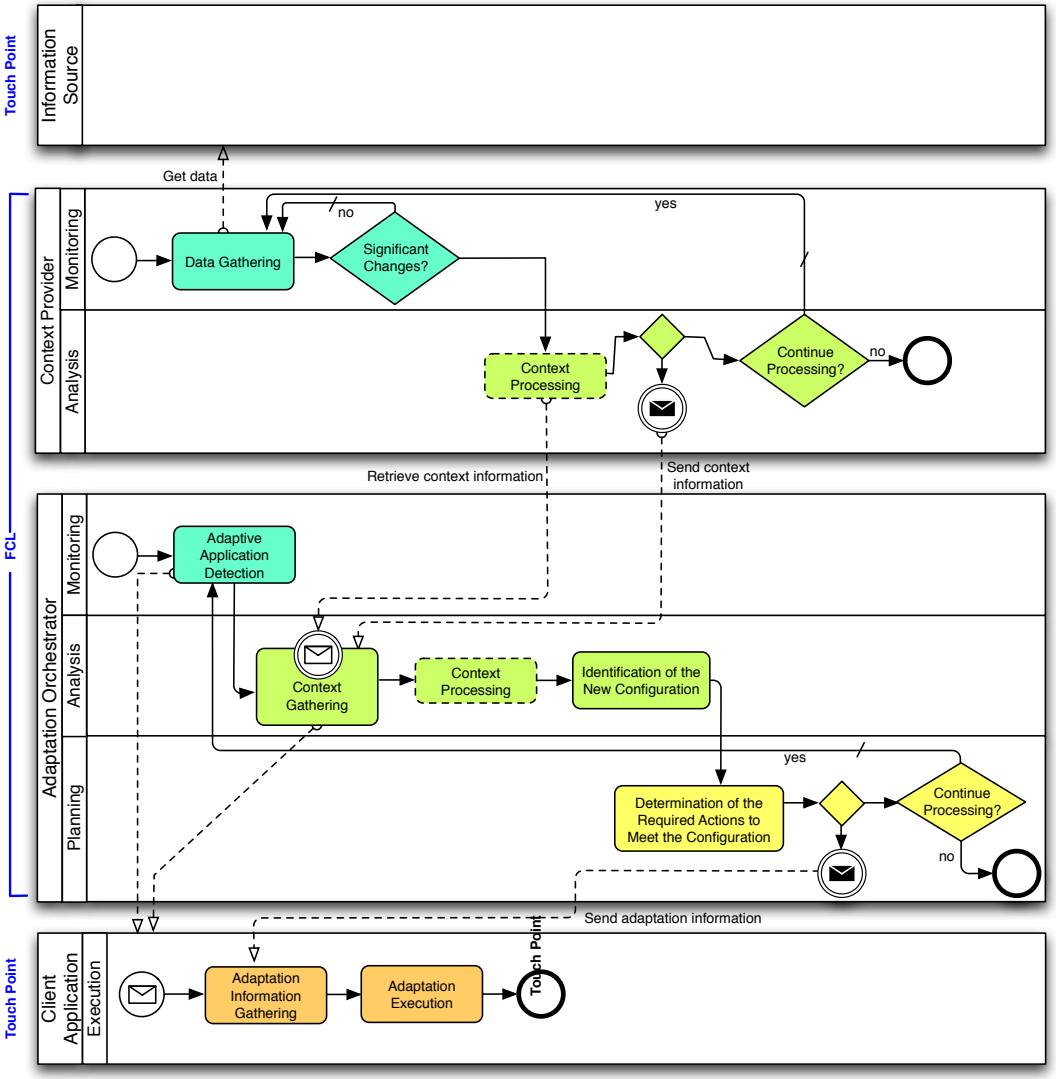


Figure 3.2: Context-Aware Adaptation Process Definition

3.2.2.1 Spaces: Enabling Ubiquitous Feedback Control Loops

In order to face the adaptation challenge in ubiquitous environments, we propose the architecture presented in Figure 3.3 to implement our ubiquitous *Feedback Control Loops* (FCLs) [Rom+10c]. We choose FCLs to support dynamic reconfigurations because they provide a clear isolation of the different steps of the adaptation process. This feature allows us to distribute the concerns in several entities and reduce the coupling between them. We qualify these FCLs as "*ubiquitous*" because they have the capacity to assemble themselves at runtime. This means that some parts of the loop can dynamically join and leave. Furthermore, the low coupling between the FCL parts promotes their integration at runtime with others ubiquitous FCLs.

The underlying motivation behind the adoption of FCLs consists in considering that the adaptation process can be considered as a flow of information that starts from raw context information and is incrementally processed and transformed by the FCL to ultimately produce reconfiguration scripts to be executed by the application under control. To support, this idea at a platform level, we apply in SPACES a RESTful approach. This solution considers *resources* as first class entities, which control interactions between resource producers and consumers. Therefore, we start the design of our solution by modeling context information as resources.

Context as a Resource. In CBSE, software connectors foster the separation and modularization of concerns. In particular, they encapsulate the transfer of control and data, and non-functional services (*e.g.*, persistency, messaging and invocation) [TMD09; Crn02] helping to keep the application functionality focused on the domain specific concerns. Therefore, we leverage on this concept to support independence of context information and communication mechanisms as well as loose-coupling between producers and consumers in our solution. The SPACES connectors expose information as resources accessible via different protocols and formats, and using

logic identifiers. This means that the SPACES connectors separate the distribution concerns from the context management tasks but they still keep a clear division of the different responsibilities associated with such distribution. In particular, SPACES connectors promote multiple implementations of the interaction mechanisms to deal with protocol heterogeneity in ubiquitous environments. For example, beyond traditional protocols, the SPACES connectors can abstract the *message-oriented* [LQS05] paradigm as well as a *peer-to-peer* [Hu+07] middleware approach as a *resource-oriented binding*.

Therefore, by encapsulating the context mediation in these connectors, we do not impact the process of the context information and we provide a first step for uncoupling the interacting entities: We hide the remote interactions from the context processing in order to integrate entities in a transparent way. Then, we need to identify, find, access and represent these resources. This is achieved by integrating the REST principles of the triangle of nouns, verbs, and content types within SPACES connectors. Nonetheless, because of the context providers and consumers mobility, we also need to consider the establishment of interactions at runtime.

Ubiquitous Resources. To address the challenges raised by the ubiquitous environment, we also include discovery protocols and *Quality of Context* (QoC) as part of the SPACES connectors. This extension leverages the discovery and the selection of context providers at runtime when required and opens the possibility of establishing spontaneous communications [ZMN05] to deal with mobility of services and clients in ubiquitous environments.

SPACES connectors promote the usage of existing discovery protocols such as UPnP, SLP and Bluetooth SDP. By using standard protocols, we make the location of providers easier since it is not necessary to develop new and complex discovery protocols. Furthermore, we foster interoperability with legacy services, which can be advertised with standard protocols in the environment. On the other hand, to benefit from the metadata describing the context information (*i.e.*, QoC attributes) and mantain the SPACES connectors flexibility in terms of interactions, we consider three design aspects of the service discovery protocols [ZMN05]:

Provider invocation: In general, the process for using located services at runtime has different steps, which include discovery, selection and access of remote providers. Regarding the access step, some protocols define the underlying communication mechanisms. For example, UPnP states SOAP [Box+00] in order to invoke operations—*i.e.*, actions in the UPnP vocabulary—on the available services. However, the adoption of a single communication mechanism in ubiquitous environments, where variability in terms of resources and protocols is the rule rather than the exception, limits the applicability of this kind of protocols. Hence, to face this lack of flexibility, SPACES connectors extend the discovery protocols by making context resources accessible via different interaction mechanisms. In this way, we offer the possibility to choose the most suitable protocols for exchanging in both, the consumer and provider sides. Furthermore, if the discovery protocol, such as SLP or Bluetooth SDP, does not define the communication mechanism, we complement it by using the supported protocols by the connectors.

Description and attribute definition: A provider description gives information about the type and operations supported by the service. Protocols, such as SLP, provide an additional service characterization by means of attributes. Therefore, in SPACES connectors we benefit from these attributes definition for expressing interaction protocols that can be used to access the provider as well as QoC information. This additional information is used in the provider selection phase.

Provider selection: In order to select the required service, SDPs apply a basic filter that considers the service type and name. Others protocols offer more specialized searches by defining restrictions on the attributes of the required providers. In ubiquitous connectors, we use these specialized searches to select providers by considering relevant properties for context information consumers—*i.e.*, QoC attributes. If the SDP does not offer the specialized search option, SPACES connectors include an additional filter to ensure that the discovered providers will satisfy the requirements.

The implementation details of SPACES connectors are reported in [Mél+10a; Mél+10b; Mél+11; Rom+10a; Rom+10b; Rom+10d; Rom+13].

Global Feedback Control Loops. In our design of a ubiquitous FCL (cf. Figure 3.3), the Controller encapsulates the functionalities required for monitoring, analyzing and planning. This means that the Controller detects the availability of new services, collects the information from the mobile devices (that join and leave the environment), processes the retrieved information and decides the required reconfigurations for the context-aware applications. These applications can be either deployed on the mobile devices or be one of the available services in the environment (*e.g.*, Multimedia Server). Consequently, the Controller requires to dynamically locate the service that operates reconfigurations of the context-aware applications. In particular, the mobile device and Multimedia Server enclose the execution part of our FCL. Moreover, the mobile device also hosts monitoring responsibilities since it notifies the Controller when changes in the provided context information occur (*e.g.*, the

battery level decreases or increases). Thus, the mobility of the different elements (mobile devices and services) in the FLC makes necessary the definition of ubiquitous FCL.

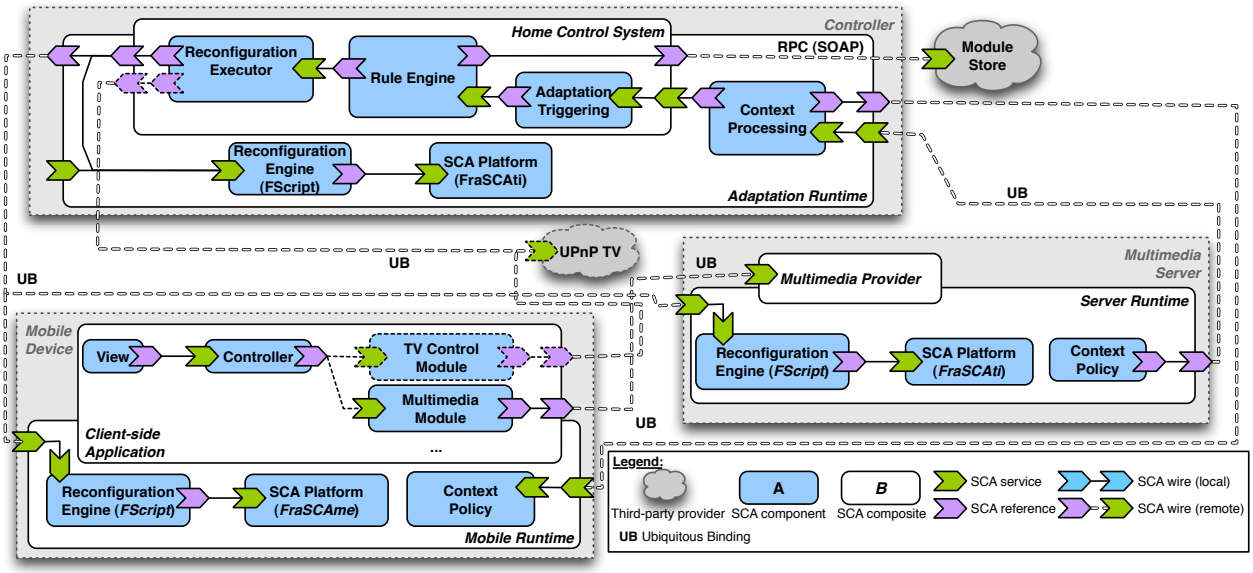


Figure 3.3: Design of an Ubiquitous Feedback Control Loop in a Smart Home Scenario.

In order to build the Ubiquitous FCLs, we use the same SCA component model that we use for REMORA (cf. Section 1.2.1) and FRASCALA (cf. Section 1.2.2). This means that the self-adaptive applications are designed as SCA applications. As already stated, the SCA selection is motivated because it structures SOA applications keeping the advantages of this approach in terms of loose-coupling and reuse. Furthermore, by using a unified model approach, such as SCA, we provide support for adaptation at platform and application layers. The SCA usage also fosters the incorporation of the SPACES connectors, which bring discovery capabilities and a data-centric approach into SCA. In particular, these bindings support the notion of *Context as a Ubiquitous Resource*.

Local Feedback Control Loops. The idea of having ubiquitous FCLs is to benefit from the most powerful entities in the environment in order to determine the required reconfigurations considering the context information and available services in the environment. However, in these global control loops we need to consider the possible communication problems between the devices hosting the adaptive application and the entities deciding the reconfigurations. In particular, the following issues should be considered:

1. What happen if the context information cannot be sent to the Adaptation Server?
2. What happen if the adaptation takes a lot of time?
3. What happen if the reconfiguration service of the adaptive application is no longer available when the required configuration is decided?
4. If the adaptation of several applications is managed at the same time, how to guarantee the consistency a distributed reconfiguration?

To deal with the first two issues we define *Local Feedback Control Loops*. Similar to the MUSIC approach (cf. Section 3.2.1), these loops introduce a certain autonomy degree in the entities hosting the adaptive applications because they provide support for making local decisions based on reactive *ECA* (Event-Condition-Action) rules [Pro+13a; Pro+13b]. However, the idea is to keep as simple as possible the local decisions because they are conceived as an auxiliary mechanism for the *Global Feedback Control Loops*. Furthermore, the entities hosting the applications have a limited knowledge about the environment. This means that these entities do not have access to all the available context sources and are not aware of the available services. Therefore, the decision-based on an incomplete knowledge should not have an important impact on the application structure. The changes should be limited to component parametrization and the disabling and enabling of functionalities of the application.

The decisions that can be made by local loops include simple application recovery and application deactivation. The data used in these simple decisions are the resource information (*e.g.*, battery level, available memory), connection state and user preferences (if they are stored in the device). These informations are independently processed and there is no consolidation of the adaptation decision as in the ubiquitous FCLs.

The global loops aggregate the decisions from the different context policies. These policies, running on the most powerful devices in the environment, collect information from different sensors, devices and consider the service availability in order to determine the application configuration. Therefore, the global loops have the capability of determining the addition, elimination or modification of the flexibility points that make part of the applications.

3.2.2.2 Self-Optimizing the Application Configuration

In the previous section, we gave an overview of our Ubiquitous FCLs and the different elements that compose them. In this section, we focus on the analysis phase of the FCL, which is associated with the *Decision Maker* from Figure 3.2. In particular, we propose an alternative to rules based on *Constraint Satisfaction Problems* (CSPs) techniques [Apt03; Gam+12; Par+12c]. This mechanism selects a new valid configuration regarding, for example, the cost associated with resource consumption (*e.g.*, memory or energy), the adaptation (*e.g.*, in terms of bindings that we need to add or remove) or QoS [Xia08] (*e.g.*, user satisfaction or response time). In this way, we provide adaptation considering not only the current context but also dimensions for providing an optimized application that guarantees a better user experience.

Our mechanism for selecting the most suitable configuration, inspired by [BBB07; NL03], assumes that an application provides a set of functionalities, each of which is reified by one or several components. Some of these functionalities are mandatory, *i.e.*, they have to be always present in the application and therefore the components that implement them represent the *application kernel*. The optional functionalities are the *flexibility points* (or *variation points*) of the architecture. We exploit these variations points in order to determine the functionalities that have to be added or modified according to the context changes.

In order to make the decision related with the new configuration, we also require some information provided by entities holding the Client Application and Decision Maker roles. The former has to keep the list of flexibility points associated with the current application configuration. This information is deployed with the application and updated each time that it is reconfigured. The latter has the list of mandatory components that define the application kernel and the different component configurations associated with each flexibility point. Furthermore, the entity holding the Decision Maker also includes the list of dependencies between the flexibility points. These dependencies define *exclude* and *require* relationships.

In our mechanism, we associate with each adaptation situation a context policy that identifies a concrete need for changing a flexibility point or functionality in the application. The results of different context policies are aggregated for defining the new required configuration—*i.e.*, the variation points that need to be modified. In Ubiquitous FCLs, several policies can be associated with the same functionality. Such policies can be triggered at the same time by context changes and can select different implementations of the point. In these cases, we need to apply our mechanism in order to decide the final new configuration of the application. Thus, by modularizing the changes of each flexibility point in context policies we simplify the selection of the new configuration.

The output of the Decision Maker component is the new configuration as a list of implementations. Each implementation is associated with a specific flexibility point. In order to determine the required actions for reaching this configuration, we apply the set differences between the current and the new configurations.

We use the difference $Conf_{initial} \setminus Conf_{target}$ to identify the optional flexibility points which implementation has to be removed. In a similar way, with $Conf_{target} \setminus Conf_{initial}$ we determine the implementation points to be added. Then, thanks to the isolation of flexibility points fostered by the context policies, we select the scripts that must be executed. In particular, each flexibility point is associated with a reconfiguration script that specifies the components to be added (resp. removed) for incorporating (resp. eliminating) a specific implementation. In the planning we also determine the order to apply the configuration. To do it, we consider the *require* relationships between the flexibility points for deciding what script should be applied first. In this analysis, we assume that there is no loops in the require dependencies between the points. If this case appears, we consider it that it is a design problem in the application and therefore we can not guarantee the application consistency. Then, in the presence of loop dependencies, we do not execute any reconfiguration.

Discussion. We consider the approaches developed in MUSIC (cf. Section 3.2.1) and Spaces (cf. Section 3.2.2) as *white-box* middleware solutions that assumes a deep and complete knowledge of the controlled software architectures to provide a self-adaptive control of ubiquitous systems. With regard to the expertise we developed along these two projects, we can report the following observations regarding FCLs:

1. **FCLs provides a relevant abstraction** to model and operate self-adaptive control steps from context monitoring to software reconfiguration;

2. Each of the building blocks of a **FCL can exhibit a strong variability** (*e.g.*, decision-making block based on utility functions, CSP models, rules, etc.);
3. The context processing model promoted by **COSMOS** (cf. Section 2.2.1) **provides a tangible foundation for modeling FCLs**;
4. The **design of FCLs requires a dedicated tool-chain** to focus on control concerns and to ease the integration of FCLs into software systems.

From these observations, we therefore believe that both MUSIC and SPACES can be leveraged into a solution that would *consider FCLs as first class entities* and thus offer a dedicated environment to design and implement autonomic control on top of legacy systems [Fri+11]. By aiming at going beyond an adaptation middleware framework, we intend to raise the level of abstraction of FCLs to *isolate the control design from implementation issues* by taking benefit from software engineering best practices. By considering legacy software systems, we raise the challenge of *black-box* adaptation as the capability to bring autonomic control on top of software systems that are not designed to be adapted and/or provide a limited visibility on their internals.

3.2.3 Black-Box Design of Feedback Control Loops

This section therefore reports on the latest approach we investigated to integrate self-adaptive policies into legacy software systems. Based on the above observations, we propose to revisit the design of FCLs by taking inspiration from the control theory community, while bringing the benefits of software engineering.

In particular, in the area of control theory, even though supporting tools, such as MATLAB, SIMULINK, or SYSWEAVER [NBR06], provide code generation capabilities, the integration of the generated controller into the target system still requires an extensive handcrafting of a non-trivial code that results in significant accidental complexities. Moreover, these tools mostly target embedded real-time systems rather than distributed enterprise systems.

We therefore propose to provide a comprehensive solution to design FCLs, and thus go beyond the design of controllers. By doing so, we propose an incremental and modular approach to support the full lifecycle of an autonomic system, from the *system identification* phase to the *design of controllers* and even the introduction of *adaptive control*.

3.2.3.1 Design of Feedback Control Loops

This section outlines our approach for integrating adaptation mechanisms into software systems through control theory centric architecture models. A more detailed description is provided in [Kri13].

Principles and Design Decisions *Generality* (applicability to a wide range of target platforms and adaptation scenarios), *visibility* (explicit FCLs, their processes and interactions), and *composability* (fine-grained reusable elements representing the FCL processes) are all well-identified requirements for FCL engineering [MPS08; ST09; Bru+09; Che+09]. In order to meet these requirements, we structure the approach around a DSML with an actor-oriented design. The key advantage of a DSML is the possibility to raise the level of abstraction at which the FCLs are described and directly use the FCL domain concepts. Moreover, DSMLs are particularly suitable for automated reasoning and implementation code synthesis [KT08]. Since FCLs are inherently concurrent, we choose an actor-oriented design [Hew77] representing the FCL processes as message-passing actors. The actor model allows to implement FCLs without worrying about thread safety, it is scalable [HO09] and seamlessly supports remote distribution.

For illustration, we use the Apache overload control FCL (cf. Figure 3.4) from Hellerstain *et al.* [Hel+04, §4.6.2][1], which can be considered as a simple adaptation mechanism. It adjusts the maximum number of simultaneous connections (MC) based on the difference between reference (MEM^*) and actual (MEM) memory usage.

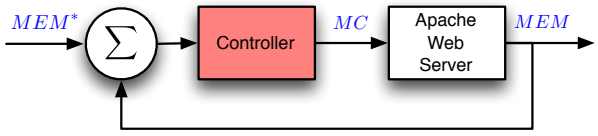


Figure 3.4: Apache overload control block diagram

[1] For simplicity, we only use the case with one controller.

Feedback Control Definition Language Combining the principles of actor-oriented components (cf. Section 1.2.3) and the COSMOS context policies (cf. Section 2.2.1), our approach is based on an actor-oriented component meta-model for representing FCLs abstractions, called *Feedback Control Definition Language* (FCDL) [KCF14]. The components are actor-like entities called *Adaptive Elements* (AE) that are connected into hierarchically composed networks that form closed FCLs.

FCDL syntax. An AE defines properties and input/output ports through which it communicates with other AEs using either data-driven (*push*) and demand-driven (*pull*) mode. Once an AE receives a message, it executes its associated behavior whose result may or may not be sent further to the connected downstream elements which in turn will cause them to react and so on and so forth. An AE can be *passive*—*i.e.*, triggered by a message—or *active*—*i.e.*, triggered by an external event (*e.g.*, a file modification). The ports and properties data values are statically typed and FCDL further supports parametric polymorphism. We recognize the following types of AE: *a sensor* (raw information collection), *an effector* (changes propagation), *a processor* (data processing and analyzing), and *a controller* (decision making). FCDL also contains a *composite* type that can be created from both atomic AEs and other composites. It can define ports, which are used to promote ports of the contained elements. Furthermore, a composite is also the primary unit of deployment. Figure 3.5 shows an FCDL model implementing the FCL from Figure 3.4. The figure uses an informal FCDL graphical notation. The `PeriodicTrigger` is an active processor. It periodically pulls memory utilization (MEM) from `SysMem` sensors and in turn pushes the value to the `Controller` that computes a new MC configuration to be applied by the `SetApacheConf` effector. The MEM^* value is modeled as a property of the controller. Conceptually,

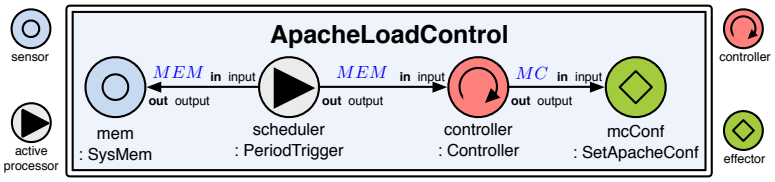


Figure 3.5: A FCDL model of Apache overload control

each AE can be seen as a target system itself, and as such it can provide sensors and effectors enabling the AE reflection. This is a crucial feature permitting to hierarchically organize multiple FCL [KC03] in an uniform way and therefore realize complex control schemes from elementary building blocks.

FCDL semantics. The execution semantics is based on the Ptolemy [Eke+03] push-pull model of computation [Zha03]. We further adapt a notion of *Interaction Contracts* (IC) to precisely define allowed interactions of AE [Cas+11]. An IC specifies what ports activate an AE, what inputs might be pulled during AE execution, and what outputs might push results. For example, the IC associated with `PeriodicTrigger` is $\langle self; \Downarrow (\texttt{input}); \Uparrow (\texttt{output}?) \rangle$. It denotes an interaction caused by a *self* activation, pulling data from the `input` port and conditionally pushing data to the `output` port. ICs allow for asserting certain architectural properties (*e.g.*, consistency, determinacy, completeness) and they denote the type of the associate activation function making the generated source code both *prescriptive* (guiding developers) and *restrictive* (limiting developers to what the architecture allows).

Local adaptation As an illustrative case study, we use ZNN, a news service [CGS09], which is one of the exemplar case studies proposed by the *Software Engineering for Adaptive and Self-Managing Systems* (SEAMS) research community. ZNN control objective is content adaptation whereby the delivered content quality (*e.g.*, degraded image quality) is reduced when the server is under heavy load. This has been well studied by Abdelzaher *et al.* [AB99; Abd00; ASB02], providing a control theoretic approach, which we integrate into ZNN using FCDL. The aim of the adaptation is to maintain the web server load at a certain pre-set value. The server content is pre-processed and stored in M trees where each one offers the same content, but of a different quality and therefore size. At runtime, a given URL request, *e.g.* `photo.jpg`, is served from either `/full/photo.jpg` or `/degraded/photo.jpg` depending on the current load of the server. Since the resource utilization is proportional to the size of the content delivered, offering the content from the degraded tree helps to reduce the server load.

Controller design Abdelzaher *et al.* [AB99; ASB02] proposes two controllers: a simple integral controller and a more sophisticated proportional integral controller. Due to the space limitations, in this section we only consider the former one, however, from the software architecture perspective, the only difference between them is the type of AE that is instantiated. The focus of FCDL is to facilitate the controller integration into software system not to develop of the controller itself. The controller input is the web server utilization

$U = aR + bW$ that is periodically computed using request rate $R = \frac{\sum r}{t}$ and delivered bandwidth $W = \frac{\sum w}{t}$, where a and b are platform constants[2] and $\sum r$, $\sum w$ are the number of requests and the amount of bytes sent over some period of time t, respectively. The controller output is the severity of the adaptation action $G = G + K_I E = G + K_I(U^* - U)$ where K_I is the controller integral gain, U^* is the target utilization (set by a system administrator) and U is the observed utilization. It determines which content tree should be used ranging from $G = M$, servicing all requests using the highest quality content tree to $G = 0$ in which case all requests are rejected.

Architecture Figure 3.6 shows one possible integration of the above controller into the target system using FCDL. For the *decision-making* part we create an AE, `IController`, that implements a general integral controller. Once a new value (U) is pushed into its input, it computes and pushes the control input (G). Both the integral gain (K_I) and the reference input (U^*) are represented as the controller properties. The *monitoring part* periodically computes server utilization U. Both the R and W can be obtained from Apache access log file. We create an active sensor, `FileTailer`, that activates every time a file content changes pushing out the modified part. The connected `AccessLogParser` extracts the number of requests r, the size of the responses w and pushes the values into the connected counters `requestCounter` and `responseSizeCounter`. To compute utilization U, the sum of requests $\sum r$ and response size $\sum w$ has to be converted into request rate R and bandwidth W—*i.e.*, the number of requests and sent bytes over certain time period t. We reuse the periodic trigger, which by pulling its input causes `LoadMonitor` to compute U using the accumulated $\sum r$, $\sum w$ sums. In the *reconfiguration part*, the `FileWriter` updates the web server URL rewrite rules reflecting the newly computed content tree.

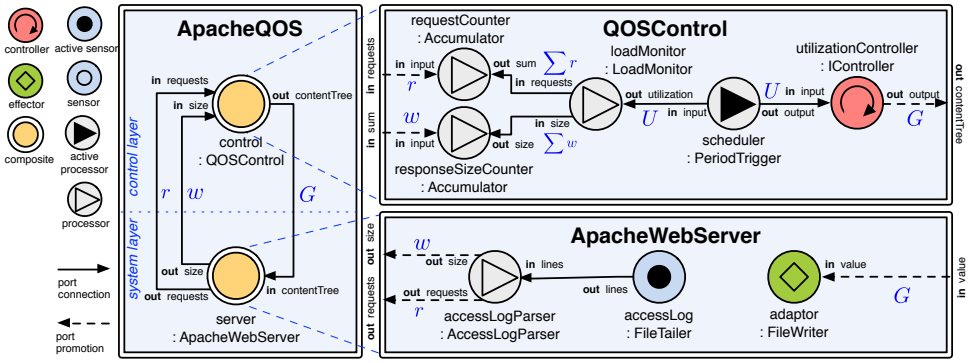


Figure 3.6: Apache content delivery control

To demonstrate composition, the presented elements are assembled into three composites `ApacheQOS`, `QOSControl` and `ApacheWebServer`, representing the main composite that will be deployed, the control, and the target system, respectively. This makes a clear separation of concerns and easy to switch from web server implementation to another.

Implementation FCDL models are implemented in a domain-specific language called *Extended Feedback Control Definition Language* (XFCDL). It is a textual DSL for authoring FCDL models that further supports modularization and AE implementation using a Java-like expression language Xbase[3]. Listing 3.1 shows an excerpt[4] of the `IController` AE. Line 1 defines a new active polymorphic processor type with data type parameter T, followed by ports declaration (lines 2-4) and property definition (line 6). Line 7 specifies an IC and line 10 provides its implementation directly in Xbase.

Distributed Adaptation Next, we extend the adaptation to cover distributed ZNN deployment on a pool of replicated servers with a load balancer.

Controller design The distributed deployment consists of a server pool S with n servers and one load balancer. Each server S_i runs locally the previously developed `ApacheQOS` FCL computing its target content tree G_i. In order to maintain the highest QoS, the load balancer dynamically schedules the arriving requests to a server $s \in S$ that provides the least degraded content: `content_tree` $(s) =$ `max` $($`content_tree` $(S))$.

[2] cf. Abdelzaher *et al.* [AB99; ASB02]
[3] A statically typed Java-like expression language `http://bit.ly/1mr36bt`
[4] The complete XFCDL code is available from the companion website `http://fikovnik.github.io/Actress/ICAC14.html`

```
1 active processor PeriodicTrigger<T> {
2   push in port output: T
3   pull in port input: T
4   self port selfport: long // self port for self-activation

6   property initialPeriod: Duration = 10.seconds
7   act activate(selfport; input; output?)

9   implementation xbase {
10    act activate { output.put(input.get) }
11  }
12 }
```

Listing 3.1: XFCDL code of `PeriodicTrigger` AE

Architecture Figure 3.7 depicts the FCL architecture representing the distributed control. The `LocalApacheQOS` runs at each of the server S_i, encapsulating the local `ApacheQOS` FCL. The `LoadBalancerControl` runs on the load balancer controlling the scheduler using the above equation.

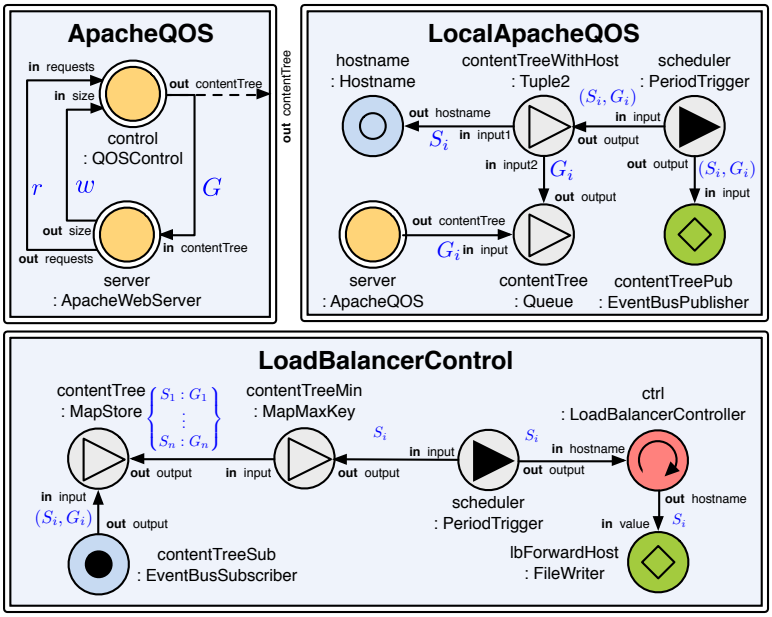


Figure 3.7: Distributed QoS Management Control FCLs

The load balancer FCL first collects the content tree (G) status of all the participating servers using distributed publish/subscribe event bus. An advantage of using an event bus is that it does not need to be *a priori* aware of all the participating servers. In FCDL, an event bus is facilitated by two AEs: the publisher (`EventBusPublisher`) and the subscriber (`EventBusSubscriber`). We use key-value tuples of servers S_i (server hostname) with their corresponding content trees G_i. The G_i is obtained from a newly promoted `ApacheQOS` port `contentTree` so that the G is available from the outside. The pushed (S_i, G_i) entries are received by the `EventBusSubscriber` and aggregated using the `MapStore` AE, which is a map storage. The server with the highest G is selected by the `MapMaxKey` AE and consequently used to update the load balancer scheduling rules.

System identification Controllers for software systems are usually driven by *"black box"* models derived from experimental runs collecting data and statistical model constructions. An experimental run consists of observing the effect of control inputs on the measured outputs. In FCDL, this can be facilitated by designing an open loop architecture in which target system touchpoints are used to set control inputs and observe/log corresponding system outputs. For example, Figure 3.8 shows an architecture model for tuning the previous controller into an open loop that can exercise the system on a range of inputs and log its outputs. Instead of connecting a controller output into the `ApacheWebServer` content tree input, we connect it directly to a value generator, a discrete sine wave.

Adaptive control An adaptive control improve FCL portability to load conditions and platform resource capacities that have not been anticipated during the system identification [AS06]. In FCDL, an adaptive control is facilitated by the model reflection. Figure 3.9 depicts an architecture of adaptive control for local content delivery adaptation FCL that reuses part of the system identification developed above.

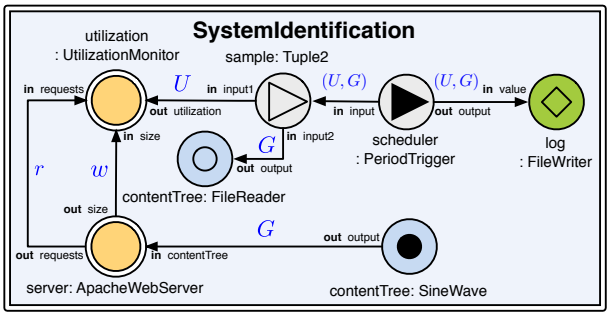


Figure 3.8: Apache content delivery control. The `UtilizationMonitor` contains the `requestCounter`, `responseSizeCounter` and `loadMonitor` elements from Figure 3.7.

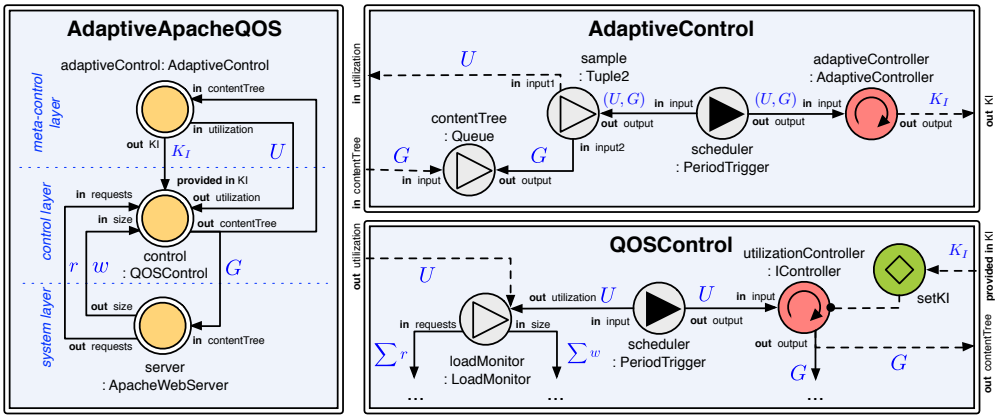


Figure 3.9: Adaptive control for Apache content delivery controller

The aim is to perform an online profiling of the target system (relation between U and G), based on which we estimate the controller parameters (K_I). First the `IController` is extended with a provided effector to allow to change K_I at runtime. Next, we reuse the part of the architecture developed for the system identification and we create an `AdaptiveController` for the parameter estimation. It can be implemented using an adaptive controller as shown by Lu *et al.* [Lu+02] or by constructing a dynamic system model as proposed by Filieri *et al.* [FHM14]. Finally, we encapsulate the corresponding elements into a new composite `AdaptiveControl` that can be placed on the top of the previous FCL developed.

Adaptive control is one example of the FCDL reflection capabilities, which can also be used to design adaptive monitoring, or to organize multiple FCLs using various control schemes, such as hierarchical control.

3.2.3.2 Corona: A Reflective Implementation of Feedback Control Loops

While FCDL provides a domain-specific modeling language to better support and formalize the description of the dynamics of software systems, the latter have to be instrumented at runtime in order to implement the self-adaptive policies described using FCDL. Corona therefore proposes a modular infrastructure to generate the middleware glue that implements the FCLs to be deployed atop of the software system to be controlled. The first contribution of Corona lies in the delivery of a component-based implementation of the FCLs, based on the SCA standard and the FraSCAti middleware platform [NRS10b]. The second contribution consists in providing a modular generation tool-chain that can accommodate different code generators and verification tools. Both of these contributions are shortly described below.

Component-based FCLs To improve the traceability of FCL entities, Corona promotes the adoption of software components as first-class entities that are used at runtime. Therefore, in Corona, FCLs consist of a set of potentially distributed components that collaborate together to maintain desired attributes for the controlled system. In particular, Corona uses the SCA standard as a supporting technology for designing and deploying the FCLs described with FCDL. Both the composition of AEs and their logic (cf. Listing 3.1) are therefore transferred to SCA components and the SCA glue is automatically generated to deliver a deployable

artifact. The composition patterns of AEs are converted to FraSCAla architectural patterns as reported in the case of COSMOS (cf. Figure 2.2).

Beyond the conformance to an industry standard, which support several programming languages and communication protocols, the choice for SCA is also motivated by the benefits to be gained from middleware platforms like FraSCAti. Indeed, in the case of Corona, FraSCAti provides a reflective support for introspecting and reconfiguring FCLs at runtime. This approach is the extension of the work reported in Section 3.2.2 and therefore proposes a systematic way to build FCLs from domain-specific models. Additionally, this approach covers the stabilization policies reported in Section 2.2.1 as a solution to stabilize both context information and control decisions. Finally, although it has not been thoroughly demonstrated yet, the proposed approach can be ported to the adaptation of WSN applications, as described in [Tah+09b], based on the lightweight Remora component model we introduced in Section 1.2.1. In particular, the support for alternative component runtime is made possible by the development of a modular tool-chain that can integrate several back-ends for the implementations of FCLs entities.

FCL generation tool-chain Corona does not only provide a mapping of FCDL descriptions to SCA components, but it also provides a supporting infrastructure to automatically generate the software artifacts that are required to deploy the FCL atop of the controlled system. Therefore, Corona provides a modular tool-chain that load the FCDL description into memory, as an EMF model, perform verifications and transformations on this internal representation before triggering source code and artifact generators to produce the above discussed component-based implementation. Nonetheless, [Kri13] proposes an alternative implementation target for FCDL descriptions based on actors and using the Akka library. Figure 3.10 reports on the tool-chain and its components, which are implemented according to the SCA standard.

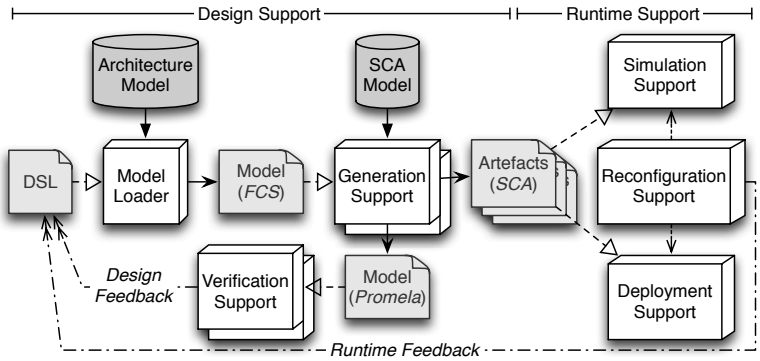


Figure 3.10: SCA architecture of the Corona tool-chain.

Among the available tool-chain components, Remora introduces a support for handling the conflicts that can be potentially raised by concurrent FCLs competing for shared resources and a support for automatically assigning the FCLs components to deployment nodes according to specific optimization objectives:

Conflict checker The *conflict checker* is a tool that implements a set of algorithms that analyze the control architecture to detect potential conflicts. The *conflict checker* provides feedbacks to developers when some conflicts are found in the control system architecture.

The verification generator is implemented as an SCA composite. It takes as input the architecture model of the FCL. This tool is automatically triggered by the Corona tool-chain before the generation of the source code. However, even if it is strongly recommended, developers can decide to follow or not the warnings generated by this tool without prejudices for the generation of the source code.

Deployment helper The *deployment helper* provides a tool to developers of FCLs to tackle the issue of the distributed deployment of the control loop. The *deployment helper* uses *Constraint Satisfaction Problem* (CSP) techniques [Apt03] to assign control elements among available resources of the managed system. The assignment of a control element to a specific host resource is done by enriching the architectural model of the control loop with adequate annotations. The *deployment helper* tool can be used for example, in the context of a developer who wants to optimize the distribution of the control loop components at runtime, knowing the network topology of the deployment infrastructure. The optimization of the distribution through the *deployment helper* can be done according to criteria like the bandwidth between host machines.

The *deployment helper* tool is implemented as an SCA composite. It requires the control loop architecture model and the network topology model as input, and generates a control loop architecture model enriched with

annotations as output. This architecture model contains annotations that drive CORONA compiler during the generation of the source code, and the deployment script.

3.3 Synthesis

This third chapter provided an overview of our contribution in the area of the design and the implementation of self-adaptive software systems. Initiated as part of the IST FP6 MUSIC project, to which I participated from 2007 to 2009, we have been working with Frank Eliassen, Eli Gjørven, Svein Hallsteinsen, Ulrich Scholz and Mickaël Beauvois on the design of technology-agnostic adaptation framework [Gjø11] that can continuously reason on the optimization of a variety of software artifacts (components, services, aspects, sensors, etc.) independently of mechanics and reconfiguration semantics imposed by the implementations of these artifacts. This work has been published in several conferences, including CBSE'08 [GER08], DAIS'08 [Oud+08], SC'08 [Rou+08b], DADS'09 [REB09] as well as a chapter in the book on *software engineering for self-adaptive systems* [Rou+09], published as an outcome of the 1st Dagstuhl seminar on Software Engineering for Self-Adaptive Systems (SefSAS).

The results of this work have then been applied as part of the CAPPUCINO collaborative project[5] to consider the self-adaptation of ubiquitous software systems distributed across a set of computing nodes. As part of Daniel Romero's PhD thesis [Rom11], we have therefore introduced the principles of *context as a resource* to extend the work done on COSMOS to deal with the full adaptation cycle of adaptive software systems and to consider the discovery of adaptation policies. This approach has also been adopted in the context of our research collaboration with the University of Oslo and published at DAIS'12 [Rom+10a; Rom+13].

This work has been further extended in the context of the SALTY collaborative project[6] and a research collaboration with Philippe Collet and Filip Krikava to provide a comprehensive solution to the design of feedback control loops. The PhD thesis of Russel Nzekwa [Nze13] provided a reference implementation for component-based FCLs and demonstrated the benefits of reasoning on domain-specific models. Beyond the publication of this work at ICAC'14 [KCR14], our results have been reported during the 3rd Dagstuhl seminar on Software Engineering for Self-Adaptive Systems (SefSAS) [Rou14].

PhD thesis supervisions associated to this chapter

[Gjø11] Eli Gjørven. "Enabling Self-Adaptation by Applying a Technology Agnostic Middleware with Support for Integration". PhD thesis. University of Oslo, Dec. 2011.

[Nze13] Russel Nzekwa. "Building Manageable Autonomic Control Loops for Large Scale Systems". PhD thesis. Université Lille 1, Sciences et Technologies, July 2013.

[Rom11] Daniel Romero. "Context as a Resource: A Service-Oriented Approach for Context-Awareness". PhD thesis. Université Lille 1, Sciences et Technologies, July 2011.

Publications associated to this chapter

[Ali+10] Mourad Alia, Mikaël Beauvois, Yann Davin, Romain Rouvoy, and Frank Eliassen. "Components and Aspects Composition Planning for Ubiquitous Adaptive Services". In: *36th EUROMICRO International Conference on Software Engineering and Advanced Applications (SEAA'10)*. Ed. by Michel Chaudron. Lille, France: ACM, 2010, pp. 1–6.

[Bra+07a] Gunnar Brataas, Jacqueline Floch, Romain Rouvoy, Pyrros Bratskas, and George A Papadopoulos. "A basis for performance property prediction of ubiquitous self-adapting systems". In: *International workshop on Engineering of software services for pervasive environments*. ACM. 2007, pp. 59–63.

[Fri+11] Marc Frincu, Norha Villegas, Dana Petcu, Hausi Muller, and Romain Rouvoy. "Self-Healing Distributed Scheduling Platform". In: *11th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*. Ed. by Carlos Varela. Newport Beach, CA, United States: IEEE, 2011, pp. 225–234.

[Gam+12] Nadia Gamez, Daniel Romero, Lidia Fuentes, Romain Rouvoy, and Laurence Duchien. "Constraint-based Self-adaptation of Wireless Sensor Networks". In: *2nd International Workshop on Adaptive Services for Future Internet*. Bertinoro, Italy, Sept. 2012, pp. 20–27.

[5] funded by Fonds Unique Interministériel (FUI)
[6] funded by Agence Nationale de la Recherche (ANR)

[GER08] Eli Gjørven, Frank Eliassen, and Romain Rouvoy. "Experiences from developing a component technology agnostic adaptation framework". In: *Component-Based Software Engineering*. Springer Berlin Heidelberg, 2008, pp. 230–245.

[GRE08] Eli Gjørven, Romain Rouvoy, and Frank Eliassen. "Cross-layer self-adaptation of service-oriented architectures". In: *Proceedings of the 3rd workshop on Middleware for service oriented computing*. ACM. 2008, pp. 37–42.

[KCR14] Filip Krikava, Philippe Collet, and Romain Rouvoy. "Integrating Adaptation Mechanisms Using Control Theory Centric Architecture Models: A Case Study". In: *ICAC - 11th International Conference on Autonomic Computing*. USENIX. Philadelphia, United States, 2014.

[Mél+10a] Rémi Mélisson, Philippe Merle, Daniel Romero, Romain Rouvoy, and Lionel Seinturier. "Reconfigurable Run-Time Support for Distributed Service Component Architectures". In: *Automated Software Engineering, Tool Demonstration*. Antwerp, Belgium, Sept. 2010, pp. 171–172.

[Mél+10b] Rémi Mélisson, Daniel Romero, Romain Rouvoy, and Lionel Seinturier. "Supporting Pervasive and Social Communications with FraSCAti". In: 28 (June 2010). Ed. by Electronic Communications of EASST, pp. 1–13.

[Mél+11] Rémi Mélisson, Daniel Romero, Romain Rouvoy, and Lionel Seinturier. "An SCA-based approach for Social and Pervasive Communications in Home Environments". In: *Scientific Annals of Computer Science* 21.1 (2011), pp. 151–173.

[NRS10b] Russel Nzekwa, Romain Rouvoy, and Lionel Seinturier. "Modelling Feedback Control Loops for Self-Adaptive Systems". In: *Third International DisCoTec Workshop on Context-Aware Adaptation Mechanisms for Pervasive and Ubiquitous Services*. Amsterdam, Netherlands, Apr. 2010, pp. 1–6.

[Oud+08] Johannes Oudenstad, Romain Rouvoy, Frank Eliassen, and Eli Gjørven. "Brokering planning metadata in a P2P environment". In: *Distributed Applications and Interoperable Systems*. Springer Berlin Heidelberg, 2008, pp. 168–181.

[Par+12c] Carlos Andrés Parra, Daniel Romero, Sébastien Mosser, Romain Rouvoy, Laurence Duchien, and Lionel Seinturier. "Using Constraint-based Optimization and Variability to Support Continuous Self-Adaptation". In: *27th ACM Symposium on Applied Computing (SAC'12), 7th Dependable and Adaptive Distributed Systems (DADS) Track*. Trento, Italy, Mar. 2012, pp. 486–491.

[Pas+08] Nearchos Paspallis, Romain Rouvoy, Paolo Barone, George A Papadopoulos, Frank Eliassen, and Alessandro Mamelli. "A pluggable and reconfigurable architecture for a context-aware enabling middleware system". In: *On the Move to Meaningful Internet Systems: OTM 2008*. Springer Berlin Heidelberg, 2008, pp. 553–570.

[Pro+13a] Lucas Provensi, Frank Eliassen, Roman Vitenberg, and Romain Rouvoy. "Improving Context Interpretation by Using Fuzzy Policies: The Case of Adaptive Video Streaming". In: *28th ACM Symposium on Applied Computing (SAC) - 8th Track on Dependable and Adaptive Distributed Systems (DADS)*. Ed. by Karl M. Göschka, Rui Oliveira, Peter Pietzuch, and Giovanni Russello. Vol. 1. Best paper award. Coimbra, Portugal: ACM, Mar. 2013, pp. 415–422.

[Pro+13b] Lucas Provensi, Frank Eliassen, Roman Vitenberg, and Romain Rouvoy. "Using fuzzy policies to improve context interpretation in adaptive systems". In: *ACM SIGAPP Applied Computing Review* 13.3 (Sept. 2013), pp. 26–37.

[RBE08] Romain Rouvoy, Mikaël Beauvois, and Frank Eliassen. "Dynamic aspect weaving using a planning-based adaptation middleware". In: *Proceedings of the 2nd workshop on Middleware-application interaction: affiliated with the DisCoTec federated conferences 2008*. ACM. 2008, pp. 31–36.

[REB09] Romain Rouvoy, Frank Eliassen, and Mikaël Beauvois. "Dynamic planning and weaving of dependability concerns for self-adaptive ubiquitous services". In: *Proceedings of the 2009 ACM symposium on Applied Computing*. ACM. 2009, pp. 1021–1028.

[Rom+10a] Daniel Romero, Gabriel Hermosillo, Amirhosein Taherkordi, Russel Nzekwa, Romain Rouvoy, and Frank Eliassen. "RESTful Integration of Heterogeneous Devices in Pervasive Environments". In: *10th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'10)*. Ed. by Frank Eliassen and Ruediger Kapitza. Vol. 6115. LNCS. Amsterdam, Netherlands, France: Springer, June 2010, pp. 1–14.

[Rom+10b] Daniel Romero, Romain Rouvoy, Lionel Seinturier, and Pierre Carton. "Service Discovery in Ubiquitous Feedback Control Loops". In: *10th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'10)*. Ed. by Frank Eliassen and Ruediger Kapitza. Vol. 6115. LNCS. Amsterdam, Netherlands, France: Springer, June 2010, pp. 113–126.

[Rom+10c] Daniel Romero, Romain Rouvoy, Lionel Seinturier, Sophie Chabridon, Denis Conan, and Nicolas Pessemier. "Enabling Context-Aware Web Services: A Middleware Approach for Ubiquitous Environments". In: *Enabling Context-Aware Web Services: Methods, Architectures, and Technologies*. Ed. by Michael Sheng, Jian Yu, and Schahram Dustdar. Chapman and Hall/CRC, May 2010, pp. 113–135.

[Rom+10d] Daniel Romero, Romain Rouvoy, Lionel Seinturier, and Frédéric Loiret. "Integration of Heterogeneous Context Resources in Ubiquitous Environments". In: *36th EUROMICRO International Conference on Software Engineering and Advanced Applications (SEAA'10)*. Ed. by Michel Chaudron. Lille, France: ACM, 2010, p. 4.

[Rom+13] Daniel Romero, Gabriel Hermosillo, Amirhosein Taherkordi, Russel Nzekwa, Romain Rouvoy, and Frank Eliassen. "The DigiHome Service-Oriented Platform". In: *Software: Practice and Experience* 43.10 (Oct. 2013), pp. 1143–1239.

[Rou+08a] Romain Rouvoy, Mikaël Beauvois, Laura Lozano, Jorge Lorenzo, and Frank Eliassen. "MUSIC: an autonomous platform supporting self-adaptive mobile applications". In: *Proceedings of the 1st workshop on Mobile middleware: embracing the personal communication device*. ACM. 2008, p. 6.

[Rou+08b] Romain Rouvoy, Frank Eliassen, Jacqueline Floch, Svein Hallsteinsen, and Erlend Stav. "Composing components and services using a planning-based adaptation middleware". In: *Software Composition*. Springer Berlin Heidelberg, 2008, pp. 52–67.

[Rou+09] Romain Rouvoy, Paolo Barone, Yun Ding, Frank Eliassen, Svein Hallsteinsen, Jorge Lorenzo, Alessandro Mamelli, and Ulrich Scholz. "Music: Middleware support for self-adaptation in ubiquitous and service-oriented environments". In: *Software engineering for self-adaptive systems*. Springer Berlin Heidelberg, 2009, pp. 164–182.

[Rou14] Romain Rouvoy. "Feedbacks Control Loops as 1st Class Entities - The SALTY Experiment". In: *Software Engineering for Self-Adaptive Systems: Assurances (Dagstuhl Seminar 13511)*. Ed. by Rogerio de Lemos, David Garlan, Carlo Ghezzi, and Holger Giese. Dagstuhl, Germany, Mar. 2014, p. 12.

[RVE08] Romain Rouvoy, Roman Vitenberg, and Frank Eliassen. "Enhancing Planning-Based Adaptation Middleware with Support for Dependability: a Case Study". In: *Electronic Communications of the EASST* 11 (2008).

[SR07] Ulrich Scholz and Romain Rouvoy. "Divide and conquer: scalability and variability for adaptive middleware". In: *International workshop on Engineering of software services for pervasive environments: in conjunction with the 6th ESEC/FSE joint meeting*. ACM. 2007, pp. 35–39.

Chapter 4

Conclusions & perspectives

This chapter concludes the thesis. We first provide a summary of the contributions reported in the manuscript and then discuss on short-term and long-term perspectives for the research in this area.

4.1 Main Results

The results reported in this manuscript are structured along three complementary axes, which we consider as the pillars of self-adaptive software systems: an *elastic component model*, an *efficient context processing middleware*, and a *robust decision-making engines*.

Elastic component models. Our contributions in this area demonstrate that the principles of *Component-Based Software Engineering* (CBSE) can be adopted at several scales of software systems: from constrained sensors, to mobile devices, to server-side infrastructures. Beyond the available standards—both in the industry and the academia—we believe that the adoption of CBSE not only improves the quality of software, but also leverages their evolution over time. In particular, we demonstrate with Remora that CBSE not only eases the development of components *in the small*, but also provides an implicit support for fine-grained reconfigurations (cf. Section 1.2.1). Thanks to Remora, WSN application developers can develop reusable building blocks that they can compose upon needs. Given the tight integration with hardware components, Remora provides a solution to compose and tune software components as they assemble the physical sensors. This modular structure also leverages the evolution over time of these software components in order to upgrade the version of driver without distributing and flushing the whole image of the WSN application. With FraSCAla, we demonstrate that components can be composed *in the large* by fostering reusable architectural patterns (cf. Section 1.2.2). Given the complexity of nowadays software systems, declarative architecture description languages do not scale anymore and rather tend to add another dimension of complexity. The approach we propose with FraSCAla captures the intention of the software architect to ensure that the requirements are met by the developers and therefore leverages the description of large-scale software architectures. Finally, we demonstrate with Macchiato that components can be spontaneously migrated across nodes to optimize the quality of service of the applications (cf. Section 1.2.3). While this contribution belongs to the long tail of offloading frameworks that have been recently published in the literature, Macchiato promotes the adoption of *Resource-as-a-Service* as a new orientation for the development of ubiquitous software. This orientation highlights the importance of data and knowledge to be exchanged between application components. Given the cost of communicating and processing data in ubiquitous devices, Macchiato considers an application as a graph of data processing components, which can be dynamically assigned to nodes discovered in the environment of the application. The introduction of dynamic reconfigurations driven by optimization heuristics emphases the nature of the other contributions we developed in our research: *context processing* and *decision making*. All theses contributions have been published in acknowledged conferences (DCOSS, CBSE) and journals (The Computer, ACM TOSN) to value the quality of our contributions. The research along this axis has been developed as part of a long-lived collaboration with the University of Oslo as well as a national research project funded by FUI (Macchiato).

Efficient context processing middleware. Context-awareness represents the second pillar of self-adaptive software systems and focuses on collecting data related to non-functional aspects of an application that might impact its execution. To address this concern, we first developed a general context inference model, named COSMOS, which intends to reflect the context inference processing chain as a graph of context nodes that incrementally transform raw metrics collected in the field into more intelligible context situations, which can be exploited by an application or a middleware solution to reason on the environment of a system (cf. Section 2.2.1). Based on this context inference model, we addressed with APISENSE® the challenges of *in-breadth* context

monitoring in the area of mobile crowd-sensing (cf. Section 2.2.2). APISENSE® enables mobile crowd-sensing *as-a-service* and proposes to collect dataset in field by orchestrating such sensing tasks across a crowd of mobile devices. In APISENSE®, crowd-sensing tasks are privacy-aware scripts that are dynamically deployed and controlled from the Cloud. By optimizing the execution crowd-sensing tasks, we demonstrate that we can reduce the energy consumed by individual mobile devices, yet ensuring a similar quality of the resulting dataset (*e.g.*, geographic or time coverage). Another dimension of context monitoring we consider in our work refers to *in-depth* monitoring in the context of automatic power consumption inference. In particular with PowerAPI, we investigate how the coarse-grained power consumption context information reported by a power meter can be distributed across the applications running on the operating system and even across the software components and the methods making these applications (cf. Section 2.2.3). We therefore propose to model the power consumption of software process and application components in order to better understand this dissipation process. The proposed power models are implemented in PowerAPI to provide online feedback on the power consumption at different levels and to locate energy hotspots. All theses contributions have been published in acknowledged conferences (ASE, DAIS) and journals (IEEE DSOnline, Wiley SPE, Springer ASE, ACM SIGOPS) to value the quality of our contributions. The research along this axis has been developed as part of a collaboration with the Telecom SudParis institute as well as a national research project funded by FUI (EconHome).

Robust decision-making engines. The third pillar of self-adaptiveness links the two previous ones and covers the intelligence to be included into a ubiquitous system in order to give it the capability to modify its structure or its behavior in order to optimize some specific objective (*e.g.*, the quality of service, the energy consumption). We first addressed this challenge in the context of MUSIC, a *white-box* adaptation middleware that seeks to optimize the utility of a ubiquitous application by evaluating alternative configurations that can be composed from software resources available in the environment (cf. Section 3.2.1). In particular, we have investigated how service-oriented resources can be discovered and incorporated in such an adaptation middleware platform to improve the utility of a mobile application. This approach has been extended to optimize the utility independently of the software artifacts that can influence it and we demonstrated the integration of aspects as another kind of artifact that can be integrated in such a technology-agnostic reasoning engine. Based on this first contribution, we have investigated the discovery of adaptive behaviors in the context of distributed software systems. This capability builds on the concept of *feedback control loop*, which we consider as an extension of a COSMOS context policy and reflect it as part of the software architecture of the considered system (cf. Section 3.2.2). By doing so, we can reason on the distribution of the adaptation process and even reify adaptation policies as ubiquitous resources that can be discovered by a mobile device, thus applying different decisions depending on the environment in which it is immersed. The principle of feedback control loop as a first class entity of software systems is further developed as part of FCDL, a domain-specific modeling language that focuses on the elicitation of self-adaptive behaviors in *black-box* legacy systems (cf. Section 3.2.3). The model proposed by FCDL fosters reuse of feedback control loop entities and is reflective to support the design and the implementation of adaptive control solutions. With Corona, we propose a reference implementation of FCDL that automatically maps feedback control loops definition to component-based software architectures based on the SCA standard. Furthermore, Corona offers a modular tool-chain to reason on the feedback control loops definition to include verifications like policy conflicts detection as well as optimizations like the deployment planning. All theses contributions have been published in acknowledged conferences (CBSE, DAIS) and journals (SPE) to value the quality of our contributions. The research along this axis has been developed as part of a collaboration with the University of Nice as well as a national research project funded by ANR (SALTY) and a European project funded by IST FP6 (MUSIC).

4.2 Perspectives

While this document reports on the results we achieved during the last 8 years of activity in the area of self-adaptive software systems, we believe that there is still a wide range of challenges to be investigated based on the experience and the solutions we developed up to now. In particular, the following paragraphs describe 6 promising research directions for the research community that we intend to address by working in collaboration with different scientific disciplines.

Power consumption of complex systems. Based on the work we developed with PowerAPI, one can observe that both hardware and software architectures are getting more and more complex to understand. Due to this complexity, it is more and more difficult to trace how the activity of software artifacts impacts the power consumption of the system as a whole. Not only application servers, but also mobile devices are now built on multi-core architectures that include power-saving technologies (dynamic voltage/frequency scaling, hyper-threading, dynamic overclocking, etc.). However, no matter the variety of optimizations made available at the

hardware level, if the system adopts greedy strategies with regards to resources, power-saving technologies are becoming inefficient. We therefore advocate that software developers and system administrators require better tools to take informed decisions on the optimization to apply in order to minimize the energy footprint of their systems. In this area, we collaborate with ADEME (*Agence De l'Environnement et de la Maîtrise de l'énergie*) and the University of Neuchâtel to develop a middleware toolkit based on PowerAPI that can be used to build *software-defined power meters*. In particular, such a middleware toolkit aims at addressing the challenge raised by the visualization of systems and to reason on the power consumption of applications hosted by virtual machines, for example. With regards to the design of an autonomic control of systems that would be driven by the energy consumption, we believe that the combination of software defined power meters with machine learning solutions can foster the emergence of smart energy saving solutions that can reduce the energy footprint of systems by learning from their past executions.

Crowd behaviors in cyber-physical environments. Based on the work we developed with APISENSE®, we are able to offer a mobile crowd-sensing platform (`http://apisense.io`) to the research community that can be used by anyone to collect realistic dataset in the field through the principles of volunteer computing. This approach already benefits to other sciences, like human sciences with PRACTIC (`http://beta.apisense.fr/practic`), and we believe and it can be adopted by other research communities. The upcoming challenge in this area therefore consists in building a citizen observatory that would raise a massive adoption to scale our approach and provide a robust scientific tool for the research communities. This involves the study and the development of incentives to motivate and inform citizens on the value of research initiatives. In particular, we intend to further contribute to the MetroScope consortium and provide a solution to monitor the quality of Internet access as it it perceived by end-users. While telecom operators are continuously monitoring the core Internet network, they fail to capture the end-to-end quality of experience of their customers, nor customers can understand the root causes of access problems they may encounter. We therefore believe that APISENSE® can offer a win-win opportunity for both telecom operators and customers to improve the quality of their Internet access by considering network measures from a wide diversity of devices, locations, and ISPs to clarify potential causes of bottlenecks. Another example of promising application refers to the analysis of user mobility patterns in the context of *smart cities*. In this area, APISENSE® can offer an open alternative to city departments to help them to improve their public transport offer, fix and optimize their road infrastructure, and monitor environmental indicators. By better connecting the citizens to the city, we believe that mobile crowd-sensing platforms can play a key role in the emergence of smart cities.

Decentralized and privacy-aware crowd-sensing algorithms. In most of the crowd-sensing experiments we considered, user privacy plays a keystone role in the adoption of crowd-sensing tasks by the participants. APISENSE® embeds privacy mechanisms to control the conditions under which data traces are produced. In particular, the participants can disable specific sensors upon preferences, define privacy areas around sensitive places (house, work, etc.), and privacy periods for disabling the crowd-sensing tasks. Nevertheless, these mechanisms are still subject to privacy leaks, in particular when reporting the location of participants. The idea in this area, in extension of the concept of virtual sensor used in APISENSE®, would therefore be to better take benefit of the wisdom of the crowd to improve the privacy of its individuals. While some theoretical algorithms have been already published in this domain, their actual implementation in the field remains an open issue. Nevertheless, the delivery of a decentralized solution to orchestrate tasks and collect data in the wild more and more appears as key feature of future crowd-sensing platforms. We therefore intend to investigate the development of *in situ* group communication layers to foster the support of collaborative crowd-sensing computations *over the air—i.e.*, before that the data reach the server-side infrastructure. By building a crowd-sensing overlay network composed of mobile devices and trusted resources (*e.g.*, a set-up box in a house), one could expect better resilience in terms of privacy and even energy consumption if we consider the results we obtained with Macchiato by sharing and offloading computation-intensive tasks.

Crowd-assisted software development. Another perspective for the work we developed in APISENSE® consists in leveraging the wisdom of the crowd to improve the quality of mobile software developments. The emergence of smartphones and app stores have throttled the development of mobile applications by promising developers to run their applications on thousands of mobile devices. Nevertheless, although *Software Development Kits* (SDK) leverage this activity, the development of mobile applications remains a complex task since it needs to take into account the variety of target devices as well as key concerns like the energy consumption of applications. In particular, even though developers can thoroughly test their application using continuous integration infrastructures, applications might still crash once deployed on users' mobile devices because of unexpected configuration settings or conditions of execution. In this area, we propose to establish an umbilical cord between the developer and the instances of her/his application deployed in the field to collect and understand potential errors that may arise in the hands of their users. The objective of this continuous connection is

to capture the quality of experience of users in order to provide a fast feedback to the developers. In particular, we target the case of automatic bug fixing by crystallizing crash reports of a community of users into test cases that can be used by developers to reproduce field errors. Such crowdsourced unit tests can then be used as an input for feeding automatic bug repair technics to generate application patches. Given the number and the diversity of mobile devices using an application, one can even consider the deployment of several patches in the field to study in the wild the best response to the proposed fix. Exploiting the wisdom of the crowd can also be considered for a community of developers to capitalize on the best practices in terms of mobile application developments. In particular, app stores can be seen as repositories of binary code that can be mined to extract a valuable knowledge that can be used to improve the quality of software developments. As part of a new collaboration with *Université du Québec À Montréal* (UQAM), we are therefore investigating the automatic analysis of design patterns and anti-patterns that are included in the mobile applications published by app stores. This activity intends to contribute to the development of smarter app stores, having the capability to advice the developers on potential errors or threats that can affect their application prior to their publication.

Adaptive Big Data processing. Based on the work we developed to improve the design of self-adaptive distributed systems with FCDL, we are targeting the specific case of adaptive big data processing. As nowadays software systems trigger a deluge of data, many big data processing platforms have emerged to provide offline or online analytics algorithms to extract indicators or knowledge out of huge mass of input data. While we intend to use some of these approaches to address the above mentioned perspective, big data processing frameworks also need a better support for elasticity. In particular, as part of the DATALYSE collaborative project[1]. We are considering the application of FCDL to apply the control theory principles to the design of a controller that protects a MapReduce framework against resource exhaustions (*e.g.*, memory, disk) and optimizing their execution by considering the availability of spare resources. In particular, we consider the case of virtualized big data platforms that build on a Cloud computing infrastructure as an opportunity to scale by provisioning the appropriate resources. Nevertheless, such an adaptive provisioning support needs to build on an accurate resource forecasting framework that can absorb the delay of resource provisioning by predicting the ideal quantity of resources that would be required in a near future to process the incoming requests. The design and the implementation of such a service requires to learn and continuously maintain a prediction model that will be used by a resource optimization controller. By composing and orchestrating such protection and optimization controllers using FCDL, we intend to deploy elastic MapReduce frameworks.

Defensive autonomic control. More generally, beyond the conviction that the application of control theory can be highly beneficial in the context of autonomic control, we think that FCDL can provide an interesting contribution in the area of defensive control of software systems. More specifically, control theory provides solid foundations for building controllers including guarantees by design (*e.g.*, convergence/stability) as long as a set of assumptions on the input signal(s) and the disturbance(s) hold. Nonetheless, if one of these assumptions breaks for some unexpected reason, then the controller might not work as expected and lead the controlled system in an unstable state. To address this issue, we can adopt the principles of assertions and defensive programming to the case of feedback control loops. The idea would be to monitor the input signals/disturbances reported by sensors in order to *i)* filter out out-of-scope values or *ii)* trigger or escalate alarms to implement some kind of *Service-Level Agreement* (SLA) between the controlled system and the controller: As long as the SLA is fulfilled, the controller can ensure the provision of exhibited assurances. Given to the reflective nature of the feedback control loops, assertions can be consumed by sensors of a meta-level feedback control loop that takes care that adaptations operate as expected. When escalating, an upper feedback control loops might decide to suspend the controller and reconfigure/replace with an alternative implementation, possibly generated *on the fly*. This issue is an example of the benefits that can be derived from combining two disciplines: *control theory* and *software engineering*.

[1] funded by the *Projet d'Investissement d'Avenir* (PIA) program: `http://www.datalyse.fr`

Bibliography

Contributions

Journals

[CRS08] Denis Conan, Romain Rouvoy, and Lionel Seinturier. "COSMOS : composition de noeuds de contexte". In: *Technique et Science Informatiques (TSI)* 27.9-10 (2008), pp. 1189–1224.

[Loi+11b] Frédéric Loiret, Romain Rouvoy, Lionel Seinturier, Daniel Romero, Kevin Sénéchal, and Ales Plsek. "An Aspect-Oriented Framework for Weaving Domain-Specific Concerns into Component-Based Systems". In: *Journal of Universal Computer Science (J.UCS)* 17.5 (Mar. 2011), pp. 742–776.

[Mél+10b] Rémi Mélisson, Daniel Romero, Romain Rouvoy, and Lionel Seinturier. "Supporting Pervasive and Social Communications with FraSCAti". In: 28 (June 2010). Ed. by Electronic Communications of EASST, pp. 1–13.

[Mél+11] Rémi Mélisson, Daniel Romero, Romain Rouvoy, and Lionel Seinturier. "An SCA-based approach for Social and Pervasive Communications in Home Environments". In: *Scientific Annals of Computer Science* 21.1 (2011), pp. 151–173.

[NRS13a] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. "A review of energy measurement approaches". In: *ACM SIGOPS Operating Systems Review* 47.3 (Dec. 2013), pp. 42–49.

[NRS13b] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. "A Review of Middleware Approaches for Energy Management in Distributed Environments". In: *Software: Practice and Experience* 43.9 (Sept. 2013), pp. 1071–1100.

[NRS14a] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. "Monitoring Energy Hotspots in Software". In: *Journal of Automated Software Engineering* (2014).

[Pro+13b] Lucas Provensi, Frank Eliassen, Roman Vitenberg, and Romain Rouvoy. "Using fuzzy policies to improve context interpretation in adaptive systems". In: *ACM SIGAPP Applied Computing Review* 13.3 (Sept. 2013), pp. 26–37.

[RCS08] Romain Rouvoy, Denis Conan, and Lionel Seinturier. "Software Architecture Patterns for a Context-Processing Middleware Framework". In: *IEEE Distributed Systems Online* 9.6 (2008), pp. 1–13.

[RM09] Romain Rouvoy and Philippe Merle. "Leveraging Component-Based Software Engineering with Fraclet". In: *Annals of Telecommunications* 64.1-2 (Jan. 2009).

[Rom+13] Daniel Romero, Gabriel Hermosillo, Amirhosein Taherkordi, Russel Nzekwa, Romain Rouvoy, and Frank Eliassen. "The DigiHome Service-Oriented Platform". In: *Software: Practice and Experience* 43.10 (Oct. 2013), pp. 1143–1239.

[RVE08] Romain Rouvoy, Roman Vitenberg, and Frank Eliassen. "Enhancing Planning-Based Adaptation Middleware with Support for Dependability: a Case Study". In: *Electronic Communications of the EASST* 11 (2008).

[Sei+12] Lionel Seinturier, Philippe Merle, Romain Rouvoy, Daniel Romero, Valerio Schiavoni, and Jean-Bernard Stefani. "A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures". In: *Software: Practice and Experience* 42.5 (May 2012), pp. 559–583.

[Tah+11b] Amirhosein Taherkordi, Frédéric Loiret, Romain Rouvoy, and Frank Eliassen. "A Generic Component-based Approach for Programming, Composing and Tuning Sensor Software". In: *The Computer Journal* 54.2 (Feb. 2011), pp. 1–19.

[Tah+13] Amirhosein Taherkordi, Frédéric Loiret, Romain Rouvoy, and Frank Eliassen. "Optimizing Sensor Network Reprogramming via In-situ Reconfigurable Components". In: *ACM Transactions on Sensor Networks* 9.2 (May 2013), pp. 1–37.

Book chapters

[Had+14] Nicolas Haderer, Fawaz Paraiso, Christophe Ribeiro, Philippe Merle, Romain Rouvoy, and Lionel Seinturier. "A Cloud-based Infrastructure for Crowdsourcing Data from Mobile Devices". In: *Cloud-based Software Crowdsourcing*. Ed. by Wenjun Wu. Springer, 2014.

[Rom+10c] Daniel Romero, Romain Rouvoy, Lionel Seinturier, Sophie Chabridon, Denis Conan, and Nicolas Pessemier. "Enabling Context-Aware Web Services: A Middleware Approach for Ubiquitous Environments". In: *Enabling Context-Aware Web Services: Methods, Architectures, and Technologies*. Ed. by Michael Sheng, Jian Yu, and Schahram Dustdar. Chapman and Hall/CRC, May 2010, pp. 113–135.

[Rou+09] Romain Rouvoy, Paolo Barone, Yun Ding, Frank Eliassen, Svein Hallsteinsen, Jorge Lorenzo, Alessandro Mamelli, and Ulrich Scholz. "Music: Middleware support for self-adaptation in ubiquitous and service-oriented environments". In: *Software engineering for self-adaptive systems*. Springer Berlin Heidelberg, 2009, pp. 164–182.

[Tah+11a] Amirhosein Taherkordi, Frank Eliassen, Daniel Romero, and Romain Rouvoy. "RESTful Service Development for Resource-constrained Environments". In: *REST: From Research to Practice*. Ed. by Erik Wilde and Cesare Pautasso. Springer, 2011, pp. 221–236.

Conferences

[Ali+10] Mourad Alia, Mikaël Beauvois, Yann Davin, Romain Rouvoy, and Frank Eliassen. "Components and Aspects Composition Planning for Ubiquitous Adaptive Services". In: *36th EUROMICRO International Conference on Software Engineering and Advanced Applications (SEAA'10)*. Ed. by Michel Chaudron. Lille, France: ACM, 2010, pp. 1–6.

[CRS07] Denis Conan, Romain Rouvoy, and Lionel Seinturier. "Scalable Processing of Context Information with COSMOS". In: *7th IFIP International Conference on Distributed Applications and Interoperable Systems*. Paphos, Cyprus, 2007, pp. 210–224.

[Fri+11] Marc Frincu, Norha Villegas, Dana Petcu, Hausi Muller, and Romain Rouvoy. "Self-Healing Distributed Scheduling Platform". In: *11th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*. Ed. by Carlos Varela. Newport Beach, CA, United States: IEEE, 2011, pp. 225–234.

[GER08] Eli Gjørven, Frank Eliassen, and Romain Rouvoy. "Experiences from developing a component technology agnostic adaptation framework". In: *Component-Based Software Engineering*. Springer Berlin Heidelberg, 2008, pp. 230–245.

[HRS13b] Nicolas Haderer, Romain Rouvoy, and Lionel Seinturier. "Dynamic Deployment of Sensing Experiments in the Wild Using Smartphones". In: *13th International IFIP Conference on Distributed Applications and Interoperable Systems (DAIS)*. Ed. by François Taïani and Jim Dowling. Vol. 7891. LNCS. Firenze, Italy: Springer, June 2013, pp. 43–56.

[KCR14] Filip Krikava, Philippe Collet, and Romain Rouvoy. "Integrating Adaptation Mechanisms Using Control Theory Centric Architecture Models: A Case Study". In: *ICAC - 11th International Conference on Autonomic Computing*. USENIX. Philadelphia, United States, 2014.

[Loi+11a] Frédéric Loiret, Romain Rouvoy, Lionel Seinturier, and Philippe Merle. "Software Engineering of Component-Based Systems-of-Systems: A Reference Framework". In: *14th ACM SIGSOFT International Symposium on Component-Based Software Engineering (CBSE'11)*. Ed. by Springer. Boulder, United States, June 2011, pp. 61–65.

[Nou+12a] Adel Noureddine, Aurélien Bourdon, Romain Rouvoy, and Lionel Seinturier. "A Preliminary Study of the Impact of Software Engineering on GreenIT". In: *First International Workshop on Green and Sustainable Software*. Zurich, Switzerland, June 2012, pp. 21–27.

[Nou+12b] Adel Noureddine, Aurélien Bourdon, Romain Rouvoy, and Lionel Seinturier. "Runtime Monitoring of Software Energy Hotspots". In: *ASE - The 27th IEEE/ACM International Conference on Automated Software Engineering - 2012*. Essen, Germany, Sept. 2012, pp. 160–169.

[NRS14b] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. "Unit Testing of Energy Consumption of Software Libraries". In: *Symposium On Applied Computing*. Gyeongju, Korea, Republic Of, Mar. 2014, pp. 1200–1205.

[Oud+08] Johannes Oudenstad, Romain Rouvoy, Frank Eliassen, and Eli Gjørven. "Brokering planning metadata in a P2P environment". In: *Distributed Applications and Interoperable Systems*. Springer Berlin Heidelberg, 2008, pp. 168–181.

[Par+12a] Fawaz Paraiso, Nicolas Haderer, Philippe Merle, Romain Rouvoy, and Lionel Seinturier. "A Federated Multi-Cloud PaaS Infrastructure". In: *5th IEEE International Conference on Cloud Computing*. hawaii, United States, June 2012, pp. 392–399.

[Par+12b] Fawaz Paraiso, Gabriel Hermosillo, Romain Rouvoy, Philippe Merle, and Lionel Seinturier. "A Middleware Platform to Federate Complex Event Processing". In: *Sixteenth IEEE International EDOC Conference*. Beijing, China: Springer, Sept. 2012, pp. 113–122.

[Par+12c] Carlos Andrés Parra, Daniel Romero, Sébastien Mosser, Romain Rouvoy, Laurence Duchien, and Lionel Seinturier. "Using Constraint-based Optimization and Variability to Support Continuous Self-Adaptation". In: *27th ACM Symposium on Applied Computing (SAC'12), 7th Dependable and Adaptive Distributed Systems (DADS) Track*. Trento, Italy, Mar. 2012, pp. 486–491.

[Pas+08] Nearchos Paspallis, Romain Rouvoy, Paolo Barone, George A Papadopoulos, Frank Eliassen, and Alessandro Mamelli. "A pluggable and reconfigurable architecture for a context-aware enabling middleware system". In: *On the Move to Meaningful Internet Systems: OTM 2008*. Springer Berlin Heidelberg, 2008, pp. 553–570.

[PRD12] Nicolas Petitprez, Romain Rouvoy, and Laurence Duchien. "Connecting your Mobile Shopping Cart to the Internet-of-Things". In: *12th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'12)*. Ed. by Karl M. Göschka and Seif Haridi. Vol. 7272. LNCS. Stockholm, Sweden: Springer, June 2012, pp. 236–243.

[Pro+13a] Lucas Provensi, Frank Eliassen, Roman Vitenberg, and Romain Rouvoy. "Improving Context Interpretation by Using Fuzzy Policies: The Case of Adaptive Video Streaming". In: *28th ACM Symposium on Applied Computing (SAC) - 8th Track on Dependable and Adaptive Distributed Systems (DADS)*. Ed. by Karl M. Göschka, Rui Oliveira, Peter Pietzuch, and Giovanni Russello. Vol. 1. Best paper award. Coimbra, Portugal: ACM, Mar. 2013, pp. 415–422.

[REB09] Romain Rouvoy, Frank Eliassen, and Mikaël Beauvois. "Dynamic planning and weaving of dependability concerns for self-adaptive ubiquitous services". In: *Proceedings of the 2009 ACM symposium on Applied Computing*. ACM. 2009, pp. 1021–1028.

[RM12] Romain Rouvoy and Philippe Merle. "Rapid Prototyping of Domain-Specific Architecture Languages". In: *15th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE'12)*. Ed. by Magnus Larsson and Nenad Medvidovic. Bertinoro, Italy: ACM, June 2012, pp. 13–22.

[Rom+10a] Daniel Romero, Gabriel Hermosillo, Amirhosein Taherkordi, Russel Nzekwa, Romain Rouvoy, and Frank Eliassen. "RESTful Integration of Heterogeneous Devices in Pervasive Environments". In: *10th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'10)*. Ed. by Frank Eliassen and Ruediger Kapitza. Vol. 6115. LNCS. Amsterdam, Netherlands, France: Springer, June 2010, pp. 1–14.

[Rom+10b] Daniel Romero, Romain Rouvoy, Lionel Seinturier, and Pierre Carton. "Service Discovery in Ubiquitous Feedback Control Loops". In: *10th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'10)*. Ed. by Frank Eliassen and Ruediger Kapitza. Vol. 6115. LNCS. Amsterdam, Netherlands, France: Springer, June 2010, pp. 113–126.

[Rom+10d] Daniel Romero, Romain Rouvoy, Lionel Seinturier, and Frédéric Loiret. "Integration of Heterogeneous Context Resources in Ubiquitous Environments". In: *36th EUROMICRO International Conference on Software Engineering and Advanced Applications (SEAA'10)*. Ed. by Michel Chaudron. Lille, France: ACM, 2010, p. 4.

[Rou+08b] Romain Rouvoy, Frank Eliassen, Jacqueline Floch, Svein Hallsteinsen, and Erlend Stav. "Composing components and services using a planning-based adaptation middleware". In: *Software Composition*. Springer Berlin Heidelberg, 2008, pp. 52–67.

[Tah+09b] Amirhosein Taherkordi, Quan Le-Trung, Romain Rouvoy, and Frank Eliassen. "WiSeKit: A Distributed Middleware to Support Application-level Adaptation in Sensor Network". In: *9th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)*. Ed. by Twittie Senivongse and Rui Oliveira. Vol. 5523. Lecture Notes in Computer Science. Lisbon, Portugal, June 2009.

[Tah+10] Amirhosein Taherkordi, Frédéric Loiret, Azadeh Abdolrazaghi, Romain Rouvoy, Quan Le-Trung, and Frank Eliassen. "Programming Sensor Networks Using REMORA Component Model". In: *6th IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS'10)*. Santa Barbara, California, United States, June 2010, p. 15.

Workshops

[Bra+07a] Gunnar Brataas, Jacqueline Floch, Romain Rouvoy, Pyrros Bratskas, and George A Papadopoulos. "A basis for performance property prediction of ubiquitous self-adapting systems". In: *International workshop on Engineering of software services for pervasive environments*. ACM. 2007, pp. 59–63.

[Bra+07b] Gunnar Brataas, Svein Hallsteinsen, Romain Rouvoy, and Frank Eliassen. "Scalability of Decision Models for Dynamic Product Lines". In: *Proceedings of the International Workshop on Dynamic Software Product Line (DSPL)*. Sept. 2007, pp. 23–32.

[Gam+12] Nadia Gamez, Daniel Romero, Lidia Fuentes, Romain Rouvoy, and Laurence Duchien. "Constraint-based Self-adaptation of Wireless Sensor Networks". In: *2nd International Workshop on Adaptive Services for Future Internet*. Bertinoro, Italy, Sept. 2012, pp. 20–27.

[GRE08] Eli Gjørven, Romain Rouvoy, and Frank Eliassen. "Cross-layer self-adaptation of service-oriented architectures". In: *Proceedings of the 3rd workshop on Middleware for service oriented computing*. ACM. 2008, pp. 37–42.

[HRS13a] Nicolas Haderer, Romain Rouvoy, and Lionel Seinturier. "A preliminary investigation of user incentives to leverage crowdsensing activities". In: *2nd International IEEE PerCom Workshop on Hot Topics in Pervasive Computing (PerHot)*. San Diego, United States: IEEE Computer Society, Mar. 2013, pp. 199–204.

[MRS11a] Philippe Merle, Romain Rouvoy, and Lionel Seinturier. "A Reflective Platform for Highly Adaptive Multi-Cloud Systems". In: *10th International Workshop on Adaptive and Reflective Middleware (ARM'2011) at the 12th ACM/IFIP/USENIX International Middleware Conference*. Lisbon, Portugal, Dec. 2011, pp. 1–7.

[NRS09] Russel Nzekwa, Romain Rouvoy, and Lionel Seinturier. "Towards a Stable Decision-Making Middleware for Very-Large-Scale Self-Adaptive Systems." In: *BElgian-NEtherlands software eVOLution seminar (BENEVOL)*. Louvain-la-Neuve, Belgium, 2009.

[NRS10a] Russel Nzekwa, Romain Rouvoy, and Lionel Seinturier. "A Flexible Context Stabilization Approach for Self-Adaptive Application". In: *COMOREA - (PERCOM)*. Mannheim, Germany, Mar. 2010, pp. 7–12.

[NRS10b] Russel Nzekwa, Romain Rouvoy, and Lionel Seinturier. "Modelling Feedback Control Loops for Self-Adaptive Systems". In: *Third International DisCoTec Workshop on Context-Aware Adaptation Mechanisms for Pervasive and Ubiquitous Services*. Amsterdam, Netherlands, Apr. 2010, pp. 1–6.

[NRS11] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. "Supporting Energy-driven Adaptations in Distributed Environments". In: *1st Workshop on Middleware and Architectures for Autonomic and Sustainable Computing*. Paris, France, May 2011, pp. 13–18.

[Qui+13] Clément Quinton, Nicolas Haderer, Romain Rouvoy, and Laurence Duchien. "Towards Multi-Cloud Configurations Using Feature Models and Ontologies". In: *1st International Workshop on Multi-Cloud Applications and Federated Clouds*. Prague, Czech Republic, Apr. 2013, pp. 21–26.

[RBE08] Romain Rouvoy, Mikaël Beauvois, and Frank Eliassen. "Dynamic aspect weaving using a planning-based adaptation middleware". In: *Proceedings of the 2nd workshop on Middleware-application interaction: affiliated with the DisCoTec federated conferences 2008*. ACM. 2008, pp. 31–36.

[Rou+08a] Romain Rouvoy, Mikaël Beauvois, Laura Lozano, Jorge Lorenzo, and Frank Eliassen. "MUSIC: an autonomous platform supporting self-adaptive mobile applications". In: *Proceedings of the 1st workshop on Mobile middleware: embracing the personal communication device*. ACM. 2008, p. 6.

[SR07] Ulrich Scholz and Romain Rouvoy. “Divide and conquer: scalability and variability for adaptive middleware”. In: *International workshop on Engineering of software services for pervasive environments: in conjunction with the 6th ESEC/FSE joint meeting*. ACM. 2007, pp. 35–39.

[Tah+08a] Amirhosein Taherkordi, Frank Eliassen, Romain Rouvoy, and Quan Le-Trung. “ReWiSe: A New Component Model for Lightweight Software Reconfiguration in Wireless Sensor Networks”. In: *On the Move to Meaningful Internet Systems: OTM 2008 Workshops*. Springer Berlin Heidelberg, 2008, pp. 415–425.

[Tah+08b] Amirhosein Taherkordi, Romain Rouvoy, Quan Le-Trung, and Frank Eliassen. “A self-adaptive context processing framework for wireless sensor networks”. In: *Proceedings of the 3rd international workshop on Middleware for sensor networks*. ACM. 2008, pp. 7–12.

[Tah+09a] Amirhosein Taherkordi, Romain Rouvoy, Quan Le-Trung, and Frank Eliassen. “Supporting Lightweight Adaptations in Context-aware Wireless Sensor Networks”. In: *1st International COMSWARE Workshop on Context-Aware Middleware and Services (CAMS)*. Vol. 385. ACM International Conference Proceeding Series. Dublin, Ireland: Mélanie Bouroche et al., June 2009.

[TRE10] Amirhosein Taherkordi, Romain Rouvoy, and Frank Eliassen. “A Component-based Approach for Service Distribution in Sensor Networks”. In: *5th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks*. Bangalore, India: ACM, Nov. 2010, pp. 22–28.

Dissemination

[Bou+13] Aurélien Bourdon, Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. “PowerAPI: A Software Library to Monitor the Energy Consumed at the Process-Level”. In: *ERCIM News* 92 (Jan. 2013), pp. 43–44.

[BRS13] Aurélien Bourdon, Romain Rouvoy, and Lionel Seinturier. “Mesurer la consommation en énergie des logiciels avec précision”. In: *01 Business & Technologies* (Jan. 2013).

[Had+13a] Nicolas Haderer, Christophe Ribeiro, Romain Rouvoy, Simon Charneau, Vassili Rivron, Alan Ouakrat, Sonia Ben Mokhtar, and Lionel Seinturier. “Le capteur, c’est vous !” In: *L’Usine Nouvelle* 3353 (Nov. 2013), pp. 74–75.

[Had+13b] Nicolas Haderer, Romain Rouvoy, Christophe Ribeiro, and Lionel Seinturier. “APISENSE: Crowd-Sensing Made Easy”. In: *ERCIM News* 93 (Apr. 2013), pp. 28–29.

[Mél+10a] Rémi Mélisson, Philippe Merle, Daniel Romero, Romain Rouvoy, and Lionel Seinturier. “Reconfigurable Run-Time Support for Distributed Service Component Architectures”. In: *Automated Software Engineering, Tool Demonstration*. Antwerp, Belgium, Sept. 2010, pp. 171–172.

[MRS11b] Philippe Merle, Romain Rouvoy, and Lionel Seinturier. “FraSCAti: Adaptive and Reflective Middleware of Middleware”. In: *12th ACM/IFIP/USENIX International Middleware Conference - Tutorial*. Lisbon, Portugal, Dec. 2011.

[Rou14] Romain Rouvoy. “Feedbacks Control Loops as 1st Class Entities - The SALTY Experiment”. In: *Software Engineering for Self-Adaptive Systems: Assurances (Dagstuhl Seminar 13511)*. Ed. by Rogerio de Lemos, David Garlan, Carlo Ghezzi, and Holger Giese. Dagstuhl, Germany, Mar. 2014, p. 12.

[SR13] Lionel Seinturier and Romain Rouvoy. “Informatique : Des logiciels mis au vert”. In: *J’innove en Nord Pas de Calais* (Nov. 2013).

Bibliography for the Elasticity of Ubiquitous Systems

[Abi+05a] Marwan Abi-Antoun, Jonathan Aldrich, David Garlan, Bradley R. Schmerl, Nagi H. Nahas, and Tony Tseng. “Modeling and implementing software architecture with Acme and ArchJava”. In: *Proceedings of the 27th International Conference on Software Engineering (ICSE)* (2005).

[Abi+05b] Marwan Abi-Antoun, Jonathan Aldrich, David Garlan, Bradley R. Schmerl, Nagi H. Nahas, and Tony Tseng. “Modeling and implementing software architecture with acme and archJava”. In: *ICSE*. ACM, 2005, pp. 676–677.

[ACN02] J. Aldrich, C. Chambers, and D. Notkin. "ArchJava: connecting software architecture to implementation". In: *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002* (2002).

[AS11] Building Applications and Android Sdk. *The Android Developer ' s Cookbook*. 2011, p. 355.

[Bac+00] Felix Bachmann, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau. *Technical Concepts of Component-Based Software Engineering*. Tech. rep. CMU/SEI-2000-TR-008. Pittsburgh, PA, USA: Carnegie Mellon Software Engineering Institute, May 2000.

[BCL12] Hongyu Pei Breivold, Ivica Crnkovic, and Magnus Larsson. *A systematic review of software architecture evolution research*. 2012.

[Bei07] Beisiegel, M. et al. *Service Component Architecture*. `http://www.osoa.org`. 2007.

[BH77] Henry C Baker Jr and Carl Hewitt. "The incremental garbage collection of processes". In: *ACM SIGART Bulletin*. Vol. 12. 64. ACM. 1977, pp. 55–59.

[BR00] T. Batista and N. Rodriguez. "Dynamic reconfiguration of component-based applications". In: *Software Engineering for Parallel and Distributed Systems, 2000. Proceedings. International Symposium on* (2000).

[Bru+06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. "The FRACTAL component model and its support in Java". In: *Softw., Pract. Exper.* 36.11-12 (2006), pp. 1257–1284.

[BV05] Roberto Baldoni and Antonino Virgillito. "Distributed event routing in publish/subscribe communication systems: a survey". In: *DIS, Universita di Roma La Sapienza, Tech. Rep* (2005), p. 5.

[Cao+08] Qing Cao, Tarek Abdelzaher, John Stankovic, Kamin Whitehouse, and Liqian Luo. "Declarative tracepoints: a programmable and application independent debugging system for wireless sensor networks". In: *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*. Raleigh, NC, USA: ACM, 2008, pp. 85–98.

[Cos07] Paolo et al. Costa. "The RUNES Middleware for Networked Embedded Systems and its Application in a Disaster Management Scenario". In: *PERCOM '07: Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 69–78.

[Cou+08] Geoff Coulson, Gordon Blair, Paul Grace, Francois Taiani, Ackbar Joolia, Kevin Lee, Jo Ueyama, and Thirunavukkarasu Sivaharan. *A generic component model for building systems software*. 2008.

[CSS11] Ivica Crnkovic, Judith A. Stafford, and Clemens A. Szyperski. "Software Components beyond Programming: From Routines to Services". In: *IEEE Software* 28.3 (2011), pp. 22–26.

[Cue+10] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. "MAUI: making smartphones last longer with code offload". In: *Proceedings of the 8th international conference on Mobile systems, applications, and services*. ACM. 2010, pp. 49–62.

[DHT01] E.M. Dashofy, A. van der Hoek, and R.N. Taylor. "A highly-extensible, XML-based architecture description language". In: *Proceedings Working IEEE/IFIP Conference on Software Architecture* (2001).

[EHL07] Clement Escoffier, Richard S. Hall, and Philippe Lalanda. "IPOJO: An extensible service-oriented component framework". In: *Proceedings - 2007 IEEE International Conference on Services Computing (SCC)*. 2007, pp. 474–481.

[Ere+07] Justin R Erenkrantz, Michael Gorlick, Girish Suryanarayana, and Richard N Taylor. "From representations to computations: the evolution of web architectures". In: *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM. 2007, pp. 255–264.

[Fas+02] Jean-Philippe Fassino, Jean-Bernard Stefani, Julia L. Lawall, and Gilles Muller. "Think: A Software Framework for Component-based Operating System Kernels". In: *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference (ATEC)*. Berkeley, CA, USA: USENIX Association, 2002, pp. 73–86.

[Flo+06] Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjørven. "Using architecture models for runtime adaptability". In: *IEEE Software* 23 (2006), pp. 62–70.

[Fow04] Martin Fowler. *Inversion of Control Containers and the Dependency Injection pattern*. 2004.

[Gam+94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st ed. Addison-Wesley Professional, Nov. 1994.

[Gay+03] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. "The nesC language: A holistic approach to networked embedded systems". In: *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation (PLDI)*. San Diego, California, USA: ACM, 2003, pp. 1–11.

[Gen+02] Thomas Genssler, Alexander Christoph, Michael Winter, Oscar Nierstrasz, Stéphane Ducasse, Roel Wuyts, Gabriela Arévalo, Bastiaan Schönhage, Peter O. Müller, and Christian Stich. "Components for embedded software: the PECOS approach". In: *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*. Grenoble, France: ACM, 2002, pp. 19–26.

[GHR07] Simon Giesecke, Wilhelm Hasselbring, and Matthias Riebisch. "Classifying architectural constraints as a basis for software quality assessment". In: *Advanced Engineering Informatics* 21 (2007), pp. 169–179.

[Giu+09] Ioana Giurgiu, Oriana Riva, Dejan Juric, Ivan Krivulev, and Gustavo Alonso. "Calling the cloud: enabling mobile phones as interfaces to cloud applications". In: *Middleware 2009*. Springer, 2009, pp. 83–102.

[GRA12] Ioana Giurgiu, Oriana Riva, and Gustavo Alonso. "Dynamic software deployment from clouds to mobile devices". In: *Middleware 2012*. Springer, 2012, pp. 394–414.

[Han+04] Hans Hansson, Mikael Akerholm, Ivica Crnkovic, and Martin Torngren. "SaveCCM - A Component Model for Safety-Critical Real-Time Systems". In: *Proceedings of the 30th EUROMICRO Conference*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 627–635.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. "A universal modular actor formalism for artificial intelligence". In: *Proceedings of the 3rd international joint conference on Artificial intelligence*. Morgan Kaufmann Publishers Inc. 1973, pp. 235–245.

[HC04] Jonathan W. Hui and David Culler. "The dynamic behavior of a data dissemination protocol for network programming at scale". In: *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*. New York, NY, USA: ACM, 2004, pp. 81–94.

[KBK12] Goran Kalic, Iva Bojic, and Mario Kusek. "Energy consumption in android phones when using wireless communication technologies". In: *MIPRO, 2012 Proceedings of the 35th International Convention*. IEEE. 2012, pp. 754–759.

[Kha+01] R. Khare, M. Guntersdorfer, P. Oreizy, N. Medvidovic, and R.N. Taylor. "xADL: enabling architecture-centric tool integration with XML". In: *Proceedings of the 34th Annual Hawaii International Conference on System Sciences* (2001).

[Lec+07] Matthieu Leclercq, Ali Erdem Özcan, Vivien Quéma, and Jean-Bernard Stefani. "Supporting Heterogeneous Architecture Descriptions in an Extensible Toolset". In: *ICSE*. IEEE, 2007, pp. 209–219.

[Lev+04] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. "TinyOS: An operating system for sensor networks". In: *Ambient Intelligence*. Berlin, Germany: Springer Verlag, 2004, 15—148.

[LG09] Philip Levis and David Gay. "TinyOS Programming". In: *ReVision* 28 (2009), p. 2006.

[LP10] Daniel Le Berre and Anne Parrain. "The Sat4j library, release 2.2 system description". In: *Journal on Satisfiability, Boolean Modeling and Computation* 7 (2010), pp. 59–64.

[LT11] Eemil Lagerspetz and Sasu Tarkoma. "Mobile search and the cloud: The benefits of offloading". In: *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2011 IEEE International Conference on*. IEEE. 2011, pp. 117–122.

[LW07] Kung-Kiu Lau and Zheng Wang. *Software Component Models*. 2007.

[McK+04] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. "Composing Adaptive Software". In: *Computer* 37.7 (2004), pp. 56–64.

[MIC93] MICROSOFT COM. `www.microsoft.com/com`. 1993.

[MM03] Nikunj R. Mehta and Nenad Medvidovic. *Composing architectural styles from architectural primitives*. 2003.

[Mor+09] Brice Morin, Olivier Barais, Gregory Nain, and Jean-Marc Jezequel. "Taming dynamically adaptive systems using models and aspects". In: *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society. 2009, pp. 122–132.

[MP11] Luca Mottola and Gian Pietro Picco. *Programming wireless sensor networks*. 2011.

[MPA08] Luca Mottola, Gian Pietro Picco, and Adil Amjad Sheikh. "FiGaRo: Fine-grained software reconfiguration for wireless sensor networks". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 4913 LNCS. 2008, pp. 286–304.

[Ode06] Martin Odersky. "The Scala experiment: can we provide better language support for component systems?" In: *POPL*. ACM, 2006, pp. 166–167.

[OMG06] OMG. *CORBA Component Model Specifications*. `http://www.omg.org/spec/CCM/4.0`. 2006.

[OZ05] Martin Odersky and Matthias Zenger. "Scalable component abstractions". In: *OOPSLA*. ACM, 2005, pp. 41–57.

[Plš+08] Aleš Plšek, Frédéric Loiret, Philippe Merle, and Lionel Seinturier. "A component framework for java-based real-time embedded systems". In: *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware (Middleware)*. Leuven, Belgium: Springer-Verlag, 2008, pp. 124–143.

[RC03] Anand Ranganathan and Roy H. Campbell. "A middleware for context-aware agents in ubiquitous computing environments". In: *Middleware '03: Proceedings of the ACM/IFIP/USENIX International Conference on Middleware*. Rio de Janeiro, Brazil: Springer-Verlag, 2003, pp. 143–161.

[SG08] Ryo Sugihara and Rajesh K. Gupta. "Programming models for sensor networks: A survey". In: *ACM Transactions on Sensor Networks (TOSN)* 4.2 (2008), pp. 1–29.

[Ste+09] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. 2nd. Addison-Wesley Professional, 2009.

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming, 2nd edition*. Boston, MA, USA: Addison-Wesley, 2002.

[Van+00] Rob Van Ommering, Frank Van der Linden, Jeff Kramer, and Jeff Magee. "The Koala Component Model for Consumer Electronics Software". In: *Computer* 33.3 (2000), pp. 78–85.

[Van08] Robbie Vanbrabant. *Google guice: Agile lightweight dependency injection framework*. 2008, pp. 1–181.

[WZC06] Qiang Wang, Yaoyao Zhu, and Liang Cheng. "Reprogramming wireless sensor networks: challenges and approaches". In: *IEEE Network* 20.3 (2006), pp. 48–55.

[Yan+07] Jing Yang, Mary Lou Soffa, Leo Selavo, and Kamin Whitehouse. "Clairvoyant: a comprehensive source-level debugger for wireless sensor networks". In: *SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems*. Sydney, Australia: ACM, 2007, pp. 189–203.

[Yan+13] Seungjun Yang, Yongin Kwon, Yeongpil Cho, Hayoon Yi, Donghyun Kwon, Jonghee Youn, and Yunheung Paek. "Fast Dynamic Execution Offloading for Efficient Mobile Cloud Computing". In: *IEEE International Conference on Pervasive Computing and Communications (PerCom)*. Vol. 18. 2013, p. 22.

Bibliography for the Contextualization of Ubiquitous Systems

[Aha+11] N. Aharony, W. Pan, C. Ip, I. Khayal, and A. Pentland. "Social fMRI: Investigating and shaping social mechanisms in the real world". In: *Pervasive and Mobile Computing* (2011).

[Ati+03] Michael Atighetchi, Partha P. Pal, Christopher C. Jones, Paul Rubel, Richard E. Schantz, Joseph P. Loyall, and John A. Zinky. "Building Auto-Adaptive Distributed Applications: The QuO-APOD Experience". In: *Distributed Computing Systems Workshops*. IEEE, May 2003, pp. 104–109.

[Ban+11] Prith Banerjee, Richard Friedrich, Cullen Bash, Patrick Goldsack, Bernardo A. Huberman, John Manley, Chandrakant Patel, Parthasarathy Ranganathan, and Alistair Veitch. "Everything as a service: Powering the new information economy". In: *Computer* 44 (2011), pp. 36–43.

[BBF02] Céline Boutros Saab, Xavier Bonnaire, and Bertil Folliot. "PHOENIX: A Self Adaptable Monitoring Platform for Cluster Management". In: *Cluster Computing* 5.1 (2002), pp. 75–85.

[BDR07] Matthias Baldauf, Schahram Dustdar, and Florian Rosenberg. *A survey on context-aware systems*. 2007.

[Bia+11] James Biagioni, Tomas Gerlich, Timothy Merrifield, and Jakob Eriksson. "EasyTracker: automatic transit tracking, mapping, and arrival time prediction using smartphones". In: *9th Int. Conf. on EmbeddedNetworked Sensor Systems*. Seattle, WA, USA: ACM, Nov. 2011.

[Bur+06] J.A. Burke, D. Estrin, M. Hansen, A. Parker, N. Ramanathan, S. Reddy, and M.B. Srivastava. "Participatory Sensing". In: (2006).

[CB10] NM Chowdhury and Raouf Boutaba. "A survey of network virtualization". In: *Computer Networks* 54.5 (2010), pp. 862–876.

[CHK08] D. Cuff, M. Hansen, and J. Kang. "Urban Sensing: Out of the Woods". In: *Communications of the ACM* 51.3 (2008).

[Cou+03] L. Courtrai, F. Guidec, N. Le Sommer, and Y. Mahéo. "Resource Management for Parallel Adaptive Components". In: *IEEE IPDPS Workshop on Java for Parallel and Distributed Computing*. Nice, France, Apr. 2003, pp. 134–141.

[Dar07] Waltenegus Dargie. "The Role of Probabilistic Schemes in Multisensor Context-Awareness". In: *5th IEEE International Conference on Pervasive Computing and Communication workshops (PerCom'07)*. IEEE, 2007, pp. 27–32.

[Dut+09] P. Dutta, P.M. Aoki, N. Kumar, A. Mainwaring, C. Myers, W. Willett, and A. Woodruff. "Common Sense: Participatory Urban Sensing Using a Network of Handheld Air Quality Monitors". In: *7th ACM Conf. on Embedded Networked Sensor Systems*. ACM. 2009.

[Gar07] Gartner. "Green IT: The New Industry Shockwave". In: *Gartner*. Presentation at Symposium/ITXPO Conference, 2007.

[HSK09] Jong-yi Hong, Eui-ho Suh, and Sung-Jin Kim. *Context-aware systems: A literature review and classification*. 2009.

[Int04] Intel. *Enhanced Intel SpeedStep Technology for the Intel Pentium M Processor*. White Paper. Mar. 2004.

[Kha+13] Wazir Zada Khan, Yang Xiang, Mohammed Y Aalsalem, and Quratulain Arshad. "Mobile Phone Sensing Systems: A Survey". In: *IEEE Communications Surveys & Tutorials* 15 (2013), pp. 402–427.

[Lan+10] Nicholas D. Lane, Emiliano Miluzzo, Hong Lu, Daniel Peebles, Tanzeem Choudhury, and Andrew T. Campbell. "A survey of mobile phone sensing". In: *IEEE Communications Magazine* 48 (2010), pp. 140–150.

[Liu+09] L. Liu, C. Andris, A. Biderman, and C. Ratti. "Uncovering Taxi Driver's Mobility Intelligence through His Trace". In: *IEEE Pervasive Computing* (2009).

[Mil+10] Emiliano Miluzzo, Nicholas D. Lane, Hong Lu, and Andrew T. Campbell. "Research in the App Store Era: Experiences from the CenceMe App Deployment on the iPhone". In: *1st Int. Work. Research in the Large: Using App Stores, Markets, and other wide distribution channels in UbiComp research*. 2010.

[MRF03] Rene Mayrhofer, Harald Radi, and Alois Ferscha. "Recognizing and Predicting Context by Learning from User Behavior". In: *International Conference on Advances in Mobile Multimedia*. Sept. 2003, pp. 25–35.

[Mun+09] M. Mun, S. Reddy, K. Shilton, N. Yau, J. Burke, D. Estrin, M. Hansen, E. Howard, R. West, and P. Boda. "PEIR, The Personal Environmental Impact Report, as a Platform for Participatory Sensing Systems Research". In: *7th Int. Conf. on Mobile Systems, Applications, and Services*. ACM. 2009.

[Pad+05] Amir Padovitz, Arkady Zaslavsky, Seng Wai Loke, and Bernard Burg. "Maintaining Continuous Dependability in Sensor-Based Context-Aware Pervasive Computing Systems". In: *38th Annual Hawaii International Conference on System Sciences (HICSS'05)*. IEEE Computer Society, 2005.

[Per+14] Charith Perera, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. "Context aware computing for the internet of things: A survey". In: *IEEE Communications Surveys and Tutorials* 16 (2014), pp. 414–454. arXiv: 1305.0982.

[Riv+07] Suzanne Rivoire, Mehul A. Shah, Parthasarathy Ranganathan, and Christos Kozyrakis. "JouleSort: a balanced energy-efficiency benchmark". In: *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*. SIGMOD '07. Beijing, China: ACM, 2007, pp. 365–376.

[RRM06] Couto Antunes da Rocha, Ricardo, and Endler Markus. "Middleware: Context Management in Heterogeneous, Evolving Ubiquitous Environments". In: *IEEE Distributed Systems Online* 7.4 (2006).

[SAH07] Odysseas Sekkas, Christos B. Anagnostopoulos, and Stathes Hadjiefthymiades. "Context Fusion Through Imprecise Reasoning". In: *IEEE International Conference on Pervasive Services*. IEEE, 2007, pp. 88–91.

[Sha09] J. Sharkey. "Coding for life–battery life, that is". In: *Google IO Developer Conf.* 2009.

[Soh+06] Timothy Sohn, Alex Varshavsky, Anthony LaMarca, Mike Y. Chen, Tanzeem Choudhury, Ian E. Smith, Sunny Consolvo, Jeffrey Hightower, William G. Griswold, and Eyal de Lara. "Mobility Detection Using Everyday GSM Traces". In: *8th Int. Conf. on Ubiquitous Computing*. Vol. 4206. LNCS. Orange County, CA, USA: Springer, 2006.

[SW10] Aamna Saeed and Tabinda Waheed. "An extensive survey of context-aware middleware architectures". In: *2010 IEEE International Conference on Electro/Information Technology, EIT2010*. 2010.

[SWV07] Kamran Sheikh, Maarten Wegdam, and Marten Van Sinderen. "Middleware support for quality of context in pervasive context-aware systems". In: *Proceedings - Fifth Annual IEEE International Conference on Pervasive Computing and Communications Workshops, PerCom Workshops 2007*. 2007, pp. 461–466.

[TC04] Lynda Temal and Denis Conan. "Failure, Connectivity and Disconnection Detectors". In: *1st French-speaking Conference on Mobility and Ubiquity Computing (UbiMob'04)*. ACM, 2004, pp. 90–97.

[Ver+10] W. Vereecken, W. Van Heddeghem, D. Colle, M. Pickavet, and P. Demeester. "Overall ICT footprint and green communication technologies". In: *Proceedings of the 4th International Symposium on Communications, Control and Signal Processing*. 2010, pp. 1–6.

[Web08] Molly Webb. *SMART 2020: enabling the low carbon economy in the information age, a report by The Climate Group on behalf of the Global eSustainability Initiative (GeSI)*. GeSI, 2008.

[Wu03] Huadong Wu. "Sensor Data Fusion for Context-aware Computing using Dempster-shafer Theory". PhD thesis. Carnergie Mellon University, Pittsburgh, 2003.

[ZWS04] Shu Zhou, M-Y Wu, and Wei Shu. "Finding optimal placements for mobile sensors: wireless sensor network topology adjustment". In: *Emerging Technologies: Frontiers of Mobile and Wireless Communication, 2004. Proceedings of the IEEE 6th Circuits and Systems Symposium on*. Vol. 2. IEEE. 2004, pp. 529–532.

Bibliography for the Self-Adaptation of Ubiquitous Systems

[AB99] Tarek F. Abdelzaher and Nina T. Bhatti. "Web server QoS management by adaptive content delivery". In: *7th International Workshop on Quality of Service*. IWQoS. London: IEEE, 1999, pp. 216–225.

[Abd00] Tarek F. Abdelzaher. "Modeling and performance control of Internet servers". In: *39th IEEE Conference on Decision and Control*. Vol. 3. IEEE, 2000, pp. 2234–2239.

[Apt03] Krzysztof Apt. *Principles of Constraint Programming*. New York, NY, USA: Cambridge University Press, 2003.

[AS06] Tarek F. Abdelzaher and John A. Stankovic. "Feedback Control Architecture and Design Methodology for Service Delay Guarantees in Web Servers". In: *IEEE Transactions on Parallel and Distributed Systems* 17.9 (Sept. 2006), pp. 1014–1027.

[ASB02] Tarek F. Abdelzaher, Kang G. Shin, and Nina T. Bhatti. "Performance guarantees for Web server end-systems: a control-theoretical approach". In: *IEEE Transactions on Parallel and Distributed Systems* 13.1 (2002), pp. 80–96.

[BBB07] Mikaël Beauvois, Djamel Belaïd, and Guy Bernard. "A Planning Framework for Dynamic Configuration in Mobile Environments". In: *GIIS'07: 1st International Workshop on Seamless Services Mobility (SSMO)*. 2007.

[Box+00] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk, Satish Thatte, and Dave Winer. *Simple Object Access Protocol (SOAP) 1.1*. `http://www.w3.org/TR/2000/NOTE-SOAP-20000508`. May 2000.

[Bru+09] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, H. Kienle, Marin Litoiu, H. Müller, M. Pezzè, and Mary Shaw. "Engineering Self-Adaptive Systems Through Feedback Loops". In: *Software Engineering for Self-Adaptive Systems* (2009), pp. 48–70.

[Cap+14] Mauro Caporuscio, Marco Funaro, Carlo Ghezzi, and Valérie Issarny. "\emph{ubi}REST: {A} RESTful Service-Oriented Middleware for Ubiquitous Networking". In: *Advanced Web Services*. Ed. by Athman Bouguettaya, Quan Z Sheng, and Florian Daniel. Springer, 2014, pp. 475–500.

[Cas+11] Damien Cassou, Emilie Balland, Charles Consel, and Julia Lawall. "Leveraging software architectures to guide and verify the development of sense/compute/control applications". In: *33rd International Conference on Software Engineering*. ICSE. New York, New York, USA: ACM Press, 2011, p. 431.

[CGS09] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. "Evaluating the effectiveness of the Rainbow self-adaptive system". In: *4th ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS. IEEE, May 2009, pp. 132–141.

[Cha+10] Goutam Chakraborty, Kshirasagar Naik, Debasish Chakraborty, Norio Shiratori, and David Wei. "Analysis of the Bluetooth device discovery protocol". In: *Wireless Networks* 16 (2010), pp. 421–436.

[Che+09] Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Di Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, Jon Whittle, Rogério Lemos, et al. "Software Engineering for Self-Adaptive Systems: A Research Roadmap". In: *Software Engineering for Self-Adaptive Systems*. Lecture Notes in Computer Science 5525 (June 2009). Ed. by Betty Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, pp. 1–26.

[Coa99] Workflow Management Coalition. *Workflow Management Coalition Terminology Glossary*. 1999.

[Crn02] Ivica Crnkovic. *Building Reliable Component-Based Software Systems*. Norwood, MA, USA: Artech House, Inc., 2002.

[Ded+06] Jessie Dedecker, Tom van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. "Ambient-Oriented Programming in AmbientTalk". In: *ECOOP 2006 – Object-Oriented Programming*. Vol. 4067. 2006, pp. 230–254.

[Don+08] Andrew Donoho, Jose Costa-requena, Tom Mcgee, Alan Messer, Andrew Fiddian-green, and John Fuller. "UPnP^{TM} Device Architecture 1.1". In: *Architecture* (2008), pp. 1–136.

[Eke+03] J. Eker, J.W. Janneck, E.A. Lee, J. Ludvig, S. Neuendorffer, and S. Sachs. "Taming heterogeneity - the Ptolemy approach". In: *IEEE* 91.1 (Jan. 2003), pp. 127–144.

[Erl05] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. 2005, p. 760.

[FGB11] Carlos A Flores-Cortés, Paul Grace, and Gordon S Blair. "SeDiM: {A} Middleware Framework for Interoperable Service Discovery in Heterogeneous Networks". In: *TAAS* 6.1 (2011), p. 6.

[FHM14] Antonio Filieri, Henry Hoffmann, and Martina Maggio. "Automated Design of Self-Adaptive Software with Control-Theoretical Formal Guarantees". In: *Proc. 36th International Conference on Software Engineering*. ICSE. Hyderabad, 2014.

[Gei+09] Kurt Geihs, Paolo Barone, Frank Eliassen, Jacqueline Floch, Rolf Fricke, Eli Gjørven, Svein O. Hallsteinsen, Geir Horn, Mohammad Ullah Khan, Alessandro Mamelli, George A. Papadopoulos, Nearchos Paspallis, Roland Reichle, and Erlend Stav. "A comprehensive solution for application-level adaptation". In: *Software - Practice and Experience* 39.4 (2009), pp. 385–422.

[Gra+11] Paul Grace, Nikolaos Georgantas, Amel Bennaceur, Gordon S. Blair, Franck Chauvel, Valérie Issarny, Massimo Paolucci, Rachid Saadi, Betrand Souville, and Daniel Sykes. "The CONNECT architecture". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 6659 LNCS. 2011, pp. 27–52.

[Gut99] Erik Guttman. "Service location protocol: automatic discovery of IP network services". In: *IEEE Internet Computing* 3 (1999), pp. 71–80.

[Hel+04] J.L. Hellerstein, Y. Diao, S. Parekh, and D.M. Tilbury. *Feedback control of computing systems*. Wiley Online Library, 2004, p. 451.

[Hew77] Carl Hewitt. "Viewing control structures as patterns of passing messages". In: *Artificial Intelligence* 8.3 (June 1977), pp. 323–364.

[HLW09] Sebastian Hudert, Heiko Ludwig, and Guido Wirtz. “Negotiating SLAs-An approach for a generic negotiation framework for WS-agreement”. In: *Journal of Grid Computing* 7 (2009), pp. 225–246.

[HO09] Philipp Haller and Martin Odersky. “Scala Actors: Unifying thread-based and event-based programming”. In: *Theoretical Computer Science* 410.2-3 (Feb. 2009), pp. 202–220.

[Hu+07] Xiaoming Hu, Yun Ding, Nearchos Paspallis, Pyrros Bratskas, George A Papadopoulos, Paolo Barone, and Alessandro Mamelli. “A Peer-to-Peer based infrastructure for Context Distribution in Mobile and Ubiquitous Environments”. In: *Proceedings of 3rd International Workshop on Context-Aware Mobile Systems (CAMS'07)*. Vilamoura, Algarve, Portugal, Nov. 2007.

[KC03] Jeffrey O. Kephart and David M. Chess. “The Vision of Autonomic Computing”. In: *Computer* 36.1 (Jan. 2003), pp. 41–50.

[KCF14] Filip Krikava, Philippe Collet, and Robert France. “ACTRESS: Domain-Specific Modeling of Self-Adaptive Software Architectures”. In: *Symposium on Applied Computing (SAC), track on Dependable and Adaptive Distributed Systems (DADS)*. ACM. Gyeongju (Korea), Mar. 2014.

[KD07] Jeffrey O. Kephart and Rajarshi Das. “Achieving self-management via utility functions”. In: *IEEE Internet Computing* 11 (2007), pp. 40–48.

[Kep05] Jeffrey 0. Kephart. “Research challenges of autonomic computing”. In: *Software Engineering* (2005).

[KL03] Alexander Keller and Heiko Ludwig. “The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services”. In: *Journal of Network and Systems Management* 11 (2003), pp. 57–81.

[KRG08] Mohammad Ullah Khan, Roland Reichle, and Kurt Geihs. “Architectural constraints in the model-driven development of self-adaptive applications”. In: *IEEE Distributed Systems Online* 9 (2008).

[Kri13] Filip Krikava. “Domain-Specific Modeling Language for Self-Adaptive Software System Architectures”. PhD thesis. University of Nice Sophia-Antipolis, 2013.

[KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Press, 2008.

[LQS05] Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. “DREAM: A Component Framework for Constructing Resource-Aware, Configurable Middleware”. In: *IEEE Distributed Systems Online (DSO)* 6.9 (Sept. 2005), pp. 1–12.

[Lu+02] Ying Lu, Tarek Abdelzaher, Chenyang Lu, and Gang Tao. “An adaptive control framework for QoS guarantees and its application to differentiated caching”. In: *10th International Workshop on Quality of Service*. IWQoS. IEEE, 2002, pp. 23–32.

[Mok+08] Sonia Ben Mokhtar, Davy Preuveneers, Nikolaos Georgantas, Valérie Issarny, and Yolande Berbers. “EASY: Efficient semAntic Service discoverY in pervasive computing environments with QoS and context support”. In: *Journal of Systems and Software* 81 (2008), pp. 785–808.

[MPM05] Graham Morgan, Simon Parkin, and C Molina-Jimenez. “Monitoring middleware for service level agreements in heterogeneous environments”. In: *Challenges of Expanding Internet: E-Commerce, E-Business, and E-Government. IFIP International Federation for Information Processing* 189 (2005), pp. 79–93.

[MPS08] Hausi Müller, Mauro Pezzè, and Mary Shaw. “Visibility of control in adaptive systems”. In: *Proceedings of the 2nd international workshop on Ultra-large-scale software-intensive systems - ULSSIS '08*. ULSSIS. New York, New York, USA: ACM Press, 2008, pp. 23–26.

[NBR06] Dionisio De Niz, Gaurav Bhatia, and Raj Rajkumar. “Model-Based Development of Embedded Systems: The SysWeaver Approach”. In: *12th IEEE Real-Time and Embedded Technology and Applications Symposium*. RTAS c. 2006.

[NL03] Sandeep Neema and Akos Ledeczi. “Constraint-guided self-adaptation”. In: *Proceedings of the 2nd international conference on Self-adaptive software: applications*. IWSAS'01. Balatonfüred, Hungary: Springer-Verlag, 2003, pp. 39–51.

[Rav+06] Pierre Guillaume Raverdy, Oriana Riva, Agnès De La Chapelle, Rafik Chibout, and Valérie Issarny. “Efficient context-aware service discovery in multi-protocol pervasive environments”. In: *Proceedings - IEEE International Conference on Mobile Data Management*. Vol. 2006. 2006.

[ST09] Mazeiar Salehie and Ladan Tahvildari. “Self-adaptive software: Landscape and research challenges”. In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 4.2 (2009), pp. 1–42.

[TMD09] Richard N. Taylor, Nenad Medvidovic, and Irvine E. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, Jan. 2009.

[Xia08] XiPeng Xiao. *Technical, Commercial and Regulatory Challenges of QoS: An Internet Service Model Perspective*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[Zha03] Yang Zhao. *A Model of Computation with Push and Pull Processing*. Tech. rep. Technical Memorandum UCB/ERL M03/51, University of California, Berkeley, 2003.

[ZMN05] Fen Zhu, Matt W. Mutka, and Lionel M. Ni. "Service Discovery in Pervasive Computing Environments". In: *IEEE Pervasive Computing* 4.4 (2005), pp. 81–90.