Design-Driven Development Methodology for Resilient Computing

Quentin Enard, Miruna Stoicescu, Emilie Balland, Charles Consel, Laurence Duchien, Jean-Charles Fabre, Matthieu Roy

Design-Driven Development Methodology for Resilient Computing

Quentin Enard[1] Miruna Stoicescu[3, 4] Emilie Balland[1]
Charles Consel[1] Laurence Duchien[2] Jean-Charles Fabre[3, 5] Matthieu Roy[3, 4]

[1] Inria, Université de Bordeaux, 200 avenue de la Vieille Tour, F-33400 Talence, France
[2] Université de Lille, LIFL, Inria, Cité Scientifique, F-59655 Villeneuve d'Ascq, France
[3] CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
[4] Univ de Toulouse, LAAS, F-31400 Toulouse, France
[5] Univ de Toulouse, INP, LAAS, F-31400 Toulouse, France

ABSTRACT

Resilient computing is defined as the ability of a system to stay dependable when facing changes. To mitigate faults at runtime, dependable systems are augmented with fault tolerance mechanisms such as replication techniques. These mechanisms have to be systematically and rigorously applied in order to guarantee the conformance between the application runtime behavior and its dependability requirements.

The main contribution of this paper is to propose a design-driven development methodology for resilient computing systems. Our approach consists of refining the design with specifications dedicated to the dependability concerns. This design is then leveraged to support the development of the application while ensuring the traceability of the dependability requirements along the application life-cycle, including runtime adaptation.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*domain-specific architectures, languages, patterns*

Keywords

Design, Resilience, Component-based Architectures, Adaptive Fault Tolerance, Generative Programming

1. INTRODUCTION

Critical systems have to face changes, either to meet new user requirements or because of changes in the execution context. Often, these changes are made at runtime because the system is too critical to be stopped. Such systems are called *resilient systems*. They have to guarantee dependability despite runtime evolution. For example, in the domain of pervasive computing, building management systems (*e.g.,* anti-intrusion, fire protection system, access control) have to be resilient as they are in charge of people safety and have to run in a continuous way.

To mitigate faults at runtime, dependable systems are augmented with fault tolerance mechanisms such as replication techniques or degraded modes of operation [2]. However, these mechanisms cover a large spectrum of areas ranging from hardware to distributed platforms, to software components. As a consequence, the need of fault-tolerance expertise is spread throughout the software development process, making it difficult to trace the dependability requirements. The fault tolerance mechanisms have to be systematically and rigorously applied in order to guarantee the conformance between the application runtime behavior and the dependability requirements. This integration becomes even more complex when taking into account runtime adaptation. Indeed, a change in the execution context of an application may require to adapt the fault tolerance mechanisms [8]. For example, a decrease of the network bandwidth may require to change the replication mechanism for one requiring less network bandwidth (*e.g.,* Leader-Follower Replication instead of Primary-Backup Replication).

Without a clear separation of the functional and fault-tolerance concerns, ensuring dependability becomes a daunting task for programmers, whose outcome is unpredictable. In this context, design-driven development approaches are of paramount importance because the design drives the development of the application while ensuring the traceability of the dependability requirements. However, because most existing approaches are general purpose, their guidance is limited, causing inconsistencies to be introduced in the design and along the development process. This situation calls for an integrated development process centered around a conceptual framework that allows to guide the development process of a resilient application in a systematic manner.

In this paper, we propose a novel approach that relies on a design language which is extended with fault-tolerance declarations. To further raise the level of abstraction, our development approach revolves around the Sense-Compute-Control (SCC) paradigm commonly used in pervasive computing [19]. The design is then compiled into a customized programming framework that drives and supports the de-

CBSE'13, June 17–21, 2013, Vancouver, BC, Canada.

velopment of the application. To face up changes in the execution context, our development methodology relies on a component-based approach, allowing fine-grained runtime adaptation [17]. This design-driven development approach ensures the traceability of the dependability requirements and preserves the separation of concerns, despite runtime evolution.

2. PROBLEM STATEMENT

To illustrate the requirements of resilient computing, we choose an anti-intrusion application. This application is responsible for securing a room with video cameras and alarms. To detect an intrusion, the application controls a video camera and periodically analyzes pictures of the environment. When an intrusion is detected, an alarm is triggered and the pictures are recorded in a database that can be consulted by a human operator to understand the situation or identify the intruder. Because this application is critical for the security of the building, intrusion detection should be ensured despite the presence of faults like hardware defects.

2.1 Traceability

Developing a resilient application requires to demonstrate that the dependability requirements result in effective measures such as fault prevention (*e.g.,* coding rules), fault removal (*e.g.,* static verification) or fault tolerance (*e.g.,* replication) [2]. In this paper, we focus on fault tolerance (FT) measures. These measures have to cover both error detection and recovery. They are generally implemented by fault tolerance mechanisms (FTMs) such as replication techniques or degraded modes of operation. For example, the application may use two video cameras to tolerate the potential crash of one of them.

The selection of an appropriate mechanism depends of (1) the kind of faults that need to be tolerated (*e.g.,* physical faults, design faults), (2) application assumptions (*e.g.,* determinism, state accessibility) and (3) the system configuration and resources (*e.g.,* network bandwidth) [16]. As these parameters are fixed along the development process and can evolve at runtime, dependability requirements have to be traceable at each stage of the application life-cycle, from the design to the deployment, to runtime evolution. However, ensuring this traceability is a tedious task and support should be provided to the developers to ensure the conformance of the application with the requirements throughout the application life-cycle.

2.2 Separation of concerns

A resilient application has to face changes in its execution context. For example, an intruder may switch off the main power supply in order to deactivate the security measures. In this case, the anti-intrusion system has to rely on alternate power sources like batteries and the application should adapt itself to reduce its power consumption. FTMs can impact dramatically the power consumption and the application may have to adapt its FTM at runtime. For example, the Leader-Follower Replication (LFR) mechanism requires less network bandwidth than the Primary-Backup Replication (PBR) mechanism but increases power consumption (all the replicas process the requests). In our example, we choose the following adaptation strategy. When running on the main power supply, the LFR mechanism is used for optimizing network bandwidth; but when running on battery, the PBR mechanism is used to optimize the power consumption.

Such runtime adaption makes the development of resilient applications even more challenging. Indeed, runtime adaptation requires the developers to deal with fault tolerance and adaptation concerns while implementing the application logic. For example, the developers of the anti-intrusion application have to deal with the monitoring of the power supply of video cameras, the use of PBR and LFR mechanisms, and how to switch from one mechanism to another. Without a clear separation of these concerns, the development of such applications becomes burdensome. In this context, a design-driven development approach ensures the separation of concerns at design time, by clearly identifying the components in charge of fault tolerance and runtime adaptation. Then the generation of programming support preserves this separation of concerns along the application life-cycle.

3. OUR APPROACH

Our approach relies on DiaSuite, a design-driven methodology dedicated to the SCC paradigm [5]. This paradigm originates from the *Sense/Compute/Control* architectural pattern, promoted by Taylor *et al.* [19]. This pattern ideally fits applications that interact with an external environment. SCC applications are typical of domains such as home/building automation, robotics, automotive and avionics.

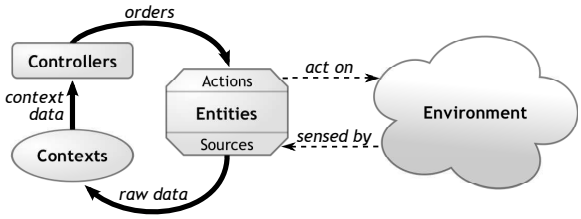

Figure 1: The SCC paradigm

As depicted in Figure 1, this architectural pattern consists of three types of components: (1) *entities* correspond to devices, whether hardware or software, and interact with the external environment through their sensing and actuating capabilities; (2) *context components* refine (filter, aggregate and interpret) raw data sensed by the entities; (3) *controller components* use this refined information to control the environment by triggering actions on entities.

The DiaSuite tool suite leverages the SCC paradigm to support each stage of the development process, from design to deployment. In this work, we extend DiaSuite to take into account resilience. The Figure 2 presents an overview of our approach and the support provided for designing, implementing and deploying a resilient application.

3.1 Design

At the design stage, the DiaSpec language provides SCC-specific declarations (stage ①) [4]. Figure 3 presents an extract of the DiaSpec specification of the anti-intrusion application. The `ImageProcessing` context component analyzes the pictures taken by the camera and provides an image where the shapes of moving objects are highlighted. A component is defined by its *interaction contract* [4]. For example, the contract of the `ImageProcessing` context component specifies that when a signal from the `Timer` entity is

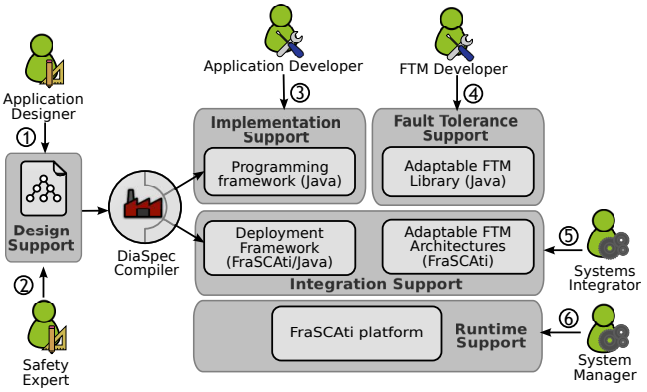

Figure 2: Overview of the approach

```
context ImageProcessing as Image {
    when provided signal from Timer;
    get image from Camera [require availability];
    always publish;
}

context Intrusion as Image {
    when provided ImageProcessing;
    maybe publish;
}

controller AlarmController {
    when provided Intrusion;
    do Trigger on Alarm [require availability],
       Log on Database;
}
```

Figure 3: Extract the anti-intrusion application specification

received, the context may access to the images provided by the `Camera` entity and systematically publishes a new value of type `Image`. This value corresponding to the analyzed image is then processed by the `Intrusion` context component to determine if there is an intrusion. In case of intrusion, the `AlarmController` component uses the `Trigger` action on `Alarm` entities and record the image on `Database` using the `log` action.

In this work, we extend the DiaSpec design language, allowing a safety expert to refine the interaction contract of an SCC component with fault-tolerance declarations (stage ②). These declarations reflect the fault taxonomy. In this paper, we focus on physical faults so we provide two annotations: `require availability` and `require correctness`, corresponding to the tolerance of respectively crash faults and value faults. The design language could be easily extended with new annotations corresponding to other types of faults (*e.g.,* development faults) or other dependability attributes (*e.g.,* integrity, confidentiality). The choice between these annotations is made accordingly to the safety and cost-benefit analysis. In the anti-intrusion example, the safety expert has considered that the `Camera` and `Alarm` entities are required to tolerate only crash faults. As shown in Figure 3, the interaction contracts of the `ImageProcessing` and `AlarmController` components are enriched with the `require availability` annotation as these components depend on respectively the `Camera` and `Alarm` entities. These FT declarations are then compiled into programming constraints, ensuring the traceability of the dependability requirements and guiding the systems integrator in the selection of FTMs.

Concerning runtime adaptation strategies, our approach consists of layering the design of a resilient application into the logic of the functional layer and a supervisory layer in charge of the monitoring and reconfiguration of the application (*e.g.,* runtime adaptation of the FTMs). We use the SCC paradigm to uniformly describe both the functional and supervisory layers. This approach ensures separation of concerns, without introducing new concepts in the design language. As shown in Figure 4, the `power` source is used by the `FTSTrategy` context component to determine whether the camera is on main power supply or on battery and triggers an adaptation of the FTM accordingly. This runtime adaptation is realized by the `AdaptationController` component that triggers the `AdaptFT` action on `Camera`.

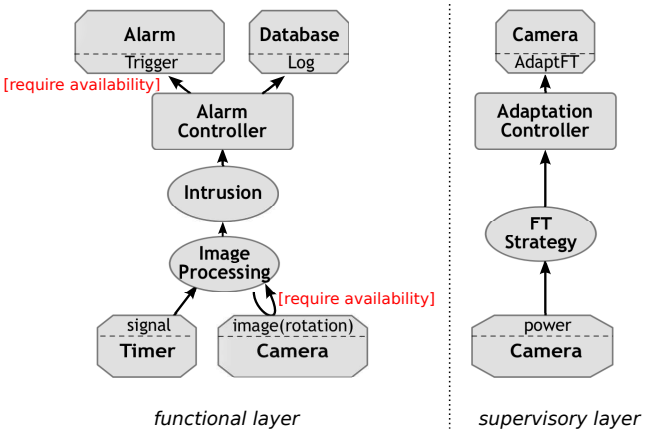

Figure 4: Functional and supervisory layers of the anti-intrusion application

3.2 Implementation

As shown in Figure 2, we distinguish two roles in the implementation stage: the application developer is in charge of implementing the application logic while the FTM developer is in charge of implementing adaptable FTMs.

Application logic. From a DiaSpec description, a programming framework is generated to guide and support the application developer (stage ③). This generative approach ensures the conformance between the design and the implementation [4]. Similarly, the FT declarations are used to enforce the dependability requirements in a transparent way for the application developer, by generating type constraints. For example, the **require availability** declaration in the `ImageProcessing` interaction contract generates a Java class named `AvailableCamera`, as shown in Figure 5. This Java type enforces the developer to provide a FTM, through the parameter of type `AvailabilityFTM<AbstractCamera>`. The `ImageProcessing` context can only bind cameras that tolerate crash faults (*i.e.,* of type `AvailableCamera`).

Fault tolerance. The role of the FTM developer is to provide implementations of FTMs, in compliance with the FTM interfaces used in the programming framework (stage ④). The hierarchy of FTM interfaces used in the generated programming framework reflects the fault model of the FT decla-

```
public class AvailableCamera
  extends AbstractCamera {

  private AvailabilityFTM<AbstractCamera> m;

  public AvailableCamera
      (AvailabilityFTM<AbstractCamera> m) {
    this.m= m;
    m.apply();
  }

  @Override
  public Image getImage() {
    m.getService().getImage();
  }

 ...
}
```

Figure 5: Extract of the `AvailableCamera` class

rations, allowing the application developer to abstract over the concrete implementations of the FTMs, and inversely the FTM developer to abstract over the application implementation.

Following the classification of FTMs [16], we provide two FTM interfaces: `AvailabilityFTM` and `CorrectnessFTM`, corresponding to the tolerance of respectively crash faults and value faults. If the fault model of the design language was extended, new interfaces would be provided. Based on these interfaces, the FTM developer provides a hierarchy of FTM implementations. Figure 6 shows an extract of the implementation of the LFR mechanism based on the adaptable FraSCAti pattern [17]. FraSCAti [14] is a service-oriented component-based middleware platform that implements the OASIS's *Service Component Architecture* (SCA) specifications [14]. The main originality of FraSCAti is to bring FRACTAL-based reflective computing features (*i.e.,* introspection and reconfiguration) to SCA [3]. In our approach, the reflective capabilities of FraSCAti are leveraged to implement runtime adaptation of the FTMs.

In Figure 6, the generic architecture of LFR is given by the `FraSCAtiLFRComposite` field. This implementation depends only on the `StatefulService` interface, allowing a clear separation between the functional and FT components. The `apply` method enables the requests to the DiaSpec service to be intercepted by the FT components. This method is called by the constructor of the `AvailableCamera` class, as shown in Figure 5. The `switchTo` method defines the adaptation from this mechanism to the mechanism given in parameter, if such adaptation exists [17]. In this example, the implementation of this method consists of a call to a FraSCAti adaptation script through the `adaptation` field.

3.3 Deployment & Runtime

The DiaSpec design is leveraged to automatically generate a deployment framework. This framework guides the systems integrator to combine the functional and non-functional developments (stage ⑤). In particular, the systems integrator has to provide instances of the DiaSpec components with suitable FTMs (*e.g.,* of type `AvailableCamera` as shown in Figure 5). These type constraints ensure the traceability of the dependability requirements at runtime.

The implementation of the FTMs depends on a specific execution platform (stage ⑥). In this paper, we illustrate this methodology with the FraSCAti middleware. A DiaSpec

```
public class FraSCAtiLFR
              <T extends StatefulService>
    implements AvailabilityFTM<T> {

  private FraSCAtiLFRComposite lfr;
  private FraSCAtiAdaptationComposite adaptation;
  private T master;
  private T slave;


  public FraSCAtiLFR(T master, T slave) {
    this.master = master;
    this.slave = slave;
    lfr = new FraSCAtiLFRComposite(master, slave);
  }

  @Override
  public T getService() {
    return master;
  }

  @Override
  public void apply() {
    master.addInterceptor(lfr.getMaster());
    slave.addInterceptor(lfr.getSlave());
  }

  @Override
  public void switchTo(AdaptableFTM m) {
    adaptation.changeFTM(this, m);
  }

  ...
}
```

Figure 6: Extract of the `FraSCAtiLFR` class

design is automatically mapped into a FraSCAti architecture. Each DiaSpec component is encapsulated into a FraSCAti component that provides the *DiaSpecService* interface. For runtime adaptation, this service interface is refined into *StatefulService* corresponding to DiaSpec components that provide methods for state handling. For example, Figure 7 illustrates the resulting FraSCAti architecture corresponding to an `AvailableCamera` instance used by the `ImageProcessing` context component. The `Camera` entity is encapsulated in a FraSCAti component providing a *StatefulService* interface. When the `FraSCAtiLFR` mechanism is deployed on two instances of `Camera`, the FraSCAti components corresponding to the cameras are composed with the `FraSCAtiLFRComposite` components (*e.g.,* the `Heartbeat` component). In this figure, we only detail the FT components involved in the `Master` composite as the `Slave` composite is similar.

As a result of our approach, the separation of concerns is ensured at design time and preserved by the programming and deployment frameworks along the application life-cycle. The integration of these concerns is delegated to the systems integrator during the deployment stage and is supported by the generated deployment framework. Note that this stage preserves the separation of concerns as the deployment architecture allows runtime adaptation of the FTMs without impacting the functional components.

Even if the methodology is illustrated with the FraSCAti middleware, our approach is general enough to target any execution platform providing the runtime adaptation capabilities required by the FTM developer [17]: (1) access to components' state and properties, (2) control over components' lifecycle (start, stop), and (3) control over interactions between components (*i.e.,* creating or removing bindings).

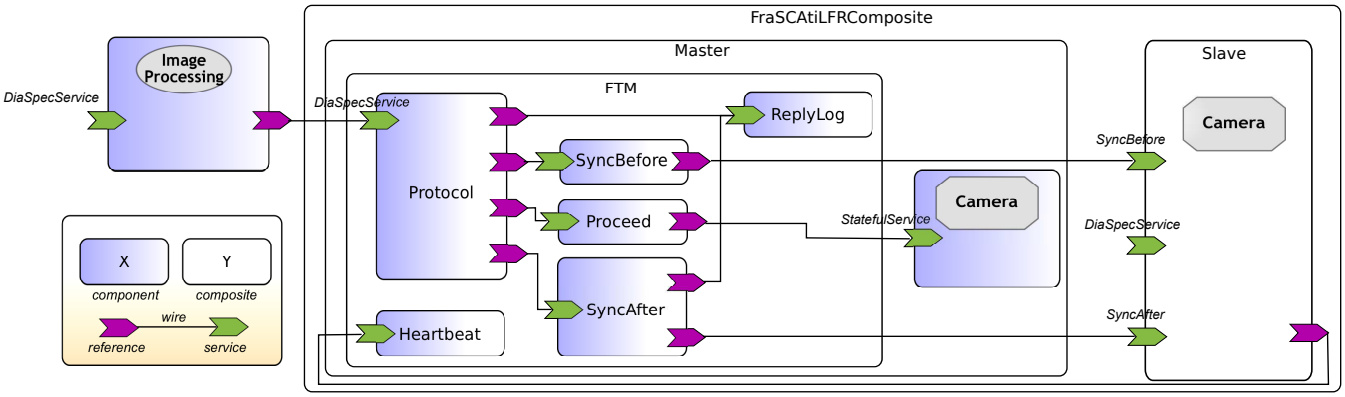

Figure 7: FraSCAti architecture of an `AvailableCamera` instance based on the LFR mechanism

4. RELATED WORK

We now review existing approaches for the development of resilient applications, focusing on three aspects: (1) fault tolerance support, (2) conformance between the design and the implementation and (3) runtime adaptation support.

4.1 Fault tolerance support

In the domain of pervasive computing, a lot of research has been devoted to providing programming support for fault tolerance [6]. For example, the one.world project [11] proposes a check-pointing mechanism that allows developers to capture the execution state of a component, and to restore it after a failure, such as power loss. It also enhances the robustness of pervasive computing systems by providing transaction-level persistence. This line of work focuses on providing implementation-level support but does not offer a development methodology ensuring the traceability of high-level dependability requirements.

In the domain of web services, Edstrom and Tilevich have proposed an approach for reusing and extending FTMs in RESTful applications [7]. This approach relies on a language named FTDL, allowing to describe fault-tolerant RESTful applications. From this specification, a compiler generates platform-specific code modules rendering the application fault-tolerant. Similarly to our approach, these modules are hidden to the application developer, promoting separation of concerns. In this work, they focus on dependability but do not cover runtime adaptation. However, with minimal efforts, this work could be leveraged to provide a library of FTMs for the DiaSuite back-end targeting Web Services.

In the domain of component-based architectures, a lot of research has been devoted to integrate dependability concerns. For example, Rubira *et al.* have proposed an approach that enriches a component-based architecture with exceptional behaviors [12]. Similarly to our approach, they promote separation of concerns but do not generate dedicated programming support.

4.2 Conformance

Conformance between design and implementation has been extensively studied in the context of software architectures. Among the approaches that focus on the conformance between the design and the implementation, ArchJava [1], ComponentJ [13] and ACOEL [15] propose to directly enrich the Java programming language with architecture-specific constructs. These approaches strongly couple the design and implementation of an application. To overcome these limitations, Archface leverages concepts from Aspect-Oriented Programming (AOP) [20]. However, by describing component interactions with implementation-level mechanisms such as pointcuts, designers have to anticipate the structure of the implementations, blurring the separation between design and implementation.

On the contrary, Zheng and Taylor present a development approach allowing a clear separation between design and implementation [21]. Similarly to our approach, they rely on code generation to ensure by construction the conformance between the design and the implementation. However, their approach is rather general purpose and does not cover dependability concerns. In this paper, we demonstrate how the use of a specific architectural pattern can guide further the development process of resilient applications.

4.3 Runtime adaptation support

Runtime adaptation has been extensively studied in the context of component-based architectures. For example, FraSCAti has been leveraged to provide QoS-driven runtime reconfiguration [18]. Another example is the Rainbow framework that leverages ACME architectures as runtime models [10]. This framework is composed of three layers. The system layer provides probes for gathering data from the executing system. The architecture layer provides gauges that aggregate information from the probes and trigger adaptations based on the runtime model. This layer implements the adaptation strategies, similarly to the SCC supervisory layer in our development methodology. Finally, the translation infrastructure is in charge of mapping the architectural model to the system. Contrary to our approach, they provide development support only for the adaptation strategies and do not cover the development of the applications or the FTMs. However their adaptation infrastructure provides high-level definitions of adaptation strategies, which could be reused for the implementation of the SCC supervisory layer.

Most of these approaches provide general-purpose adaptation mechanisms and few focus on the adaptation of FTMs. A notable exception is the work of Fraga *et al.*, which defines a CORBA component-based model allowing coarse-grained adaptation of FTMs [9]. In comparison, our adaptable FTM patterns allow fine-grained adaptation, minimizing the exe-

cution cost [17]. In any case, our development methodology is general enough to target any execution platforms providing runtime adaptation facilities, facilitating the reuse of existing FTM libraries.

5. CONCLUSION

In this paper, we have proposed a design-driven development approach that systematically processes dependability requirements throughout the application life-cycle. This approach ensures the traceability of the dependability requirements and preserves the separation of concerns, despite runtime evolution. To overcome changes in the execution context, our development methodology relies on a component-based approach, allowing fine-grained runtime adaptation.

We are currently expanding this work in several directions. Work is in progress to add and compose several non-functional layers (*e.g.,* fault tolerance, safety and security). Another direction is to extend the FTM library, following our "design for adaptation" approach [17]. On one hand, runtime transitions between FTMs are performed with a minimal set of changes. On the other hand, this approach greatly facilitates the evolution and customization of the library. Indeed, new FTMs can be built by composing or tuning existing ones.

6. REFERENCES

[1] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: connecting software architecture to implementation. In *ICSE'02: Proceedings of the 24th International Conference on Software Engineering*, pages 187–197. IEEE, 2002.

[2] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[3] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The FRACTAL Component Model and its Support in Java. *Software: Practice and Experience – Special issue on Experiences with Auto-adaptive and Reconfigurable Systems*, 36(11-12):1257–1284, Aug. 2006. John Wiley & Sons.

[4] D. Cassou, E. Balland, C. Consel, and J. Lawall. Leveraging software architectures to guide and verify the development of Sense/Compute/Control applications. In *ICSE'11: Proceedings of the 33rd International Conference on Software Engineering*, pages 431–440. ACM, 2011.

[5] D. Cassou, J. Bruneau, C. Consel, and E. Balland. Towards a tool-based development methodology for pervasive computing applications. *Transactions on Software Engineering*, 38(6):1445–1463, 2012.

[6] S. Chetan, A. Ranganathan, and R. Campbell. Towards fault tolerant pervasive computing. *Technology and Society Magazine*, 24(1):38–44, 2005.

[7] J. Edstrom and E. Tilevich. Reusable and extensible fault tolerance for RESTful applications. In *TrustCom'12: Proceedings of the 11th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 737–744. IEEE, 2012.

[8] J.-C. Fabre, M.-O. Killijian, and T. Pareaud. Towards on-line adaptation of fault tolerance mechanisms. In *EDCC'10: Eighth European Dependable Computing Conference*, pages 45–54. IEEE, 2010.

[9] J. Fraga, F. Siqueira, and F. Favarim. An adaptive fault-tolerant component model. In *WORDS'03: Proceedings of the 9th International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 179–186. IEEE, 2003.

[10] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[11] R. Grimm. One.world: Experiences with a pervasive computing architecture. *Pervasive Computing*, 3(3):22–30, 2004.

[12] C. M. F. Rubira, R. de Lemos, G. R. Ferreira, and F. Castor Filho. Exception handling in the development of dependable component-based systems. *Software: Practice and Experience*, 35(3):195–236, 2005.

[13] J. a. C. Seco and L. Caires. A basic model of typed components. In *ECOOP'00: Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 108–128. Springer-Verlag, 2000.

[14] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani. A component-based middleware platform for reconfigurable service-oriented architectures. *Software: Practice and Experience*, 42(5):559–583, 2012.

[15] V. C. Sreedhar. Mixin'up components. In *ICSE'02: Proceedings of the 24th International Conference on Software Engineering*, pages 198–207. ACM, 2002.

[16] M. Stoicescu, J.-C. Fabre, and M. Roy. Architecting resilient computing systems: Overall approach and open issues. In *SERENE'11: Proceedings of the Third International Workshop on Software Engineering for Resilient Systems*, volume 6968 of *LNCS*, pages 48–62. Springer-Verlag, 2011.

[17] M. Stoicescu, J.-C. Fabre, and M. Roy. From design for adaptation to component-based resilient computing. In *PRDC'12: Proceedings of the 18th Pacific Rim International Symposium on Dependable Computing*, pages 1–10. IEEE, 2012.

[18] G. Tamura, R. Casallas, A. Cleve, and L. Duchien. QoS contract-aware reconfiguration of component architectures using E-Graphs. In *FACS'10: Proceedings of the 7th International Workshop on Formal Aspects of Component Software*, volume 6921 of *LNCS*, pages 34–52. Springer-Verlag, 2010.

[19] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[20] N. Ubayashi, J. Nomura, and T. Tamai. Archface: a contract place where architectural design and code meet together. In *ICSE '10: Proceedings of the 32nd International Conference on Software Engineering*, pages 75–84. ACM, 2010.

[21] Y. Zheng and R. N. Taylor. Enhancing architecture-implementation conformance with change management and support for behavioral mapping. In *ICSE'12: Proceedings of the 2012 International Conference on Software Engineering*, pages 628–638. IEEE, 2012.