Software Dependability Modeling Using AADL (Architecture Analysis and Design Language)

ANA-ELENA RUGINA1, KARAMA KANOUN2,3, and MOHAMED KAANICHE2,3

1 EADS ASTRIUM, 31 Avenue des Cosmonautes, 31402 Toulouse - France
2 LAAS-CNRS, 7 av. Colonel Roche, 31077 Toulouse – France
3 Université de Toulouse; UPS, INSA, INP, ISAE ; LAAS ; F-31077 Toulouse – France

(Received on March 10, 2010 and revised on March 31, 2011)

Abstract: In traditional development processes, each type of analysis is generally based on a dedicated model, which requires substantial amount of training to be used effectively. Performing dependability evaluation along with other analyses at architectural level allows both making architectural tradeoffs and predicting the effects of architectural decisions on the dependability of an application. This paper is related to the building of dependability models for architectural software systems using the AADL (Architecture Analysis and Design Language). It presents reusable modeling patterns for fault-tolerant applications and shows how the presented patterns can be used in the context of a subsystem of a real-life application.

Keywords: Fault-tolerance, dependability modeling, AADL, reuse, patterns.

1. Introduction

Modeling software architectures has proved to be useful for promoting reuse and evolution of large applications using extensively components-off-the-shelf (COTS). In addition, performing several analyses of quality attributes such as dependability and performance on a common architectural model is particularly interesting, as this allows making architectural tradeoffs [1].

The AADL (Architecture Analysis and Design Language) [2] has received an increasing interest from the safety-critical embedded systems industry (e.g., Honeywell, Rockwell Collins, Lockheed Martin, the European Space Agency, Astrium, Airbus) during the last years. The AADL is characterized by all the properties that an architecture description language (ADL) should provide (composition, abstraction, reusability, configuration, heterogeneity, analysis) [3].

In this paper, we focus on architecture-based dependability modeling and evaluation using the AADL. Our work aims at helping engineers using the AADL for other purposes (e.g., for performance analyses), to integrate dependability modeling in their development process. This paper improves upon an initial version presented in [4].

We provide guidance on using the AADL language for modeling behaviors of fault-tolerant software systems, and show that the development of patterns is very useful to facilitate the modeling of fault tolerance behavior and to enhance the reusability of the models. We define a fault tolerance pattern as a reusable model describing a fault tolerance strategy at the architectural level. To be used in a particular system, a pattern must be instantiated and customized if necessary.

The use of patterns and, more generally, dependability modeling at architectural level:
i) favors the reduction of recurrent dependability modeling work and the understandability of the dependability model (thus reflecting the modularity of the architecture) [5] and,

ii) allows the designer to reason about fault tolerance and to assign exceptional behavior responsibilities among components [6].

At the same time, dependability measures (i.e., availability, reliability, safety) can be evaluated based on the AADL model. This allows predicting the effects of particular architectural decisions on the dependability of the system [7]. Other analyses (e.g., related to performance) may be performed on the same AADL model, which allows understanding the tradeoff between the benefits of a certain fault tolerance pattern and its impact on the applications performance [8].

From a practical point of view, the AADL model has to be transformed into a stochastic model such as a Markov chain [9] or a Generalized Stochastic Petri net [10], to obtain dependability measures such as reliability, availability, etc. In this paper, we focus on the use of patterns to facilitate the AADL model construction.

The paper is organized as follows. Section 2 surveys related work. Section 3 outlines the main concepts of the AADL and its support for dependability modeling. Section 4 gives guidance, resulting from our experience, on building dependability models for fault-tolerant duplex software systems using the AADL. Section 5 presents AADL fault tolerance patterns for three duplex software systems differing by their error detection mechanisms. Section 6 shows examples of dependability analysis results of interest for software engineers. Finally, Section 7 concludes the paper.

2. Related Work

Software architecture modeling for dependability analysis and evaluation has received a growing interest during the last two decades. Early approaches have focused on the development of analytical models to analyze the sensitivity of the application reliability to the software structure and the reliabilities of its components (see e.g., [11]; [12]) and the survey presented in [13]. More recently, the emergence of component-based software engineering approaches and architecture description languages (ADLs) led to the proliferation of research activities on software architectures and methodologies allowing the analysis and evaluation of performance- and dependability-related characteristics. Significant efforts have been focused on the Unified Modeling Language (UML). In particular, a number of recent papers consider the transformation of UML specifications (enriched e.g., with timing constraints and dependability related information) into different types of analytical models (e.g., Petri nets[14]), dynamic fault trees [15]) used to obtain dependability or performance measures.

Besides UML, various ADLs have recently received increasing attention from industry and academia. A classification of software architecture description languages including a critical analysis of their modeling capabilities (in particular compared to UML) is presented in [16]. Among ADLs, the AADL/MetaH provides advanced support for analyzing quality attributes. It also has substantial support for modeling reconfigurable architectures. These characteristics led to its serious consideration in the safety-critical embedded systems industry.

The AADL allows describing separately the analysis-related information that may be plugged into the architectural model. This feature enhances the reusability and the readability of the AADL architectural model that can be used as is for several analyses (formal verification [17], scheduling and memory requirements [18], resource allocation with the Open Source AADL Tool Environment (OSATE), research of deadlocks and un-
initialized variables with the Ocarina toolset).

The reusability of the AADL model is also enhanced by the use of a set of fault tolerance patterns. The hot standby redundancy pattern presented in [8] has been a source of inspiration for the three patterns of this paper. The pattern in [8] aims at easing the understanding of the functional architecture by clearly showing what is replicated and what the active system components are. Our patterns additionally include a customizable layer of dependability-related information (error/failure and recovery behavior) and of dynamics necessary for evaluating dependability measures. The proposed patterns can be used in the context of the iterative dependency-driven modeling approach presented in [19].

Our work complements other existing initiatives that investigated the development of fault tolerance patterns based on object-oriented approaches and UML ([20]; [21]; [22]) or other languages [23].

3. Overview of the AADL Language

In the AADL, systems are composite components modeled as hierarchical collections of interacting application components (processes, threads, subprograms, data) and a set of compute platform components (processors, memory, buses, devices). The application components are bound to the compute platform. Dynamic aspects of system architectures are captured with the AADL operational mode concept. Different operational modes of a system or system component represent different system configurations and connection topologies, as well as different sets of property values.

Each AADL system component has two levels of description: the component type and the component implementation. The component type describes how the environment sees that component (i.e., its properties and features). Examples of features are in and out ports that represent directional access points to the component. The AADL defines three types of ports: event, data, and event data (modeling respectively flows of control, data, and control and data). One or more component implementations can be associated with the same component type, matching different component implementation structures in terms of subcomponents, connections and operational modes.

An AADL architectural model can be annotated with dependability-related information (such as faults, failure modes and repair actions, error propagation, etc.) through the standardized Error Model Annex [24]. AADL error models allow modeling complex and realistic components behaviors in the presence of faults, as shown in [25]. Generic error models are defined in libraries and are associated with application components, compute platform components, as well as the connections between them. When a generic error model is associated with a component, it can be customized if necessary by setting component-specific values for the arrival rate or the probability of occurrence for error events and error propagations declared in the error model.

Error models consist of two parts: the error model type and the error model implementation. The error model type declares a set of error states, error events (inherent to the component) and error propagations. These items can be customized when the error model is associated with a specific component. Occurrence properties specify the arrival rate or the occurrence probability of events and propagations. The error model implementation declares error transitions between states, triggered by events and propagations declared in the error model type.

Figure 1 shows an error model of a component that may fail and that is restarted to regain its error free state. The component cannot be influenced by propagations coming
from its environment, as it does not declare in propagations. An out propagation is used to indicate notification of dependent components when the component fails.

Interactions between the error models of different components are determined by interactions between components of the architectural model through connections and bindings. For example, if a component has an outgoing port connection to another component, then its out propagation for that port gets mapped to the name-matching in propagation declared in the error model of the receiving component. In some cases, it is desirable to model how error propagations from multiple sources are handled. This is modeled by specifying filters and masking conditions for propagations, using Guard properties associated with features. The interested reader can refer to [25] for an extensive list of generic reusable error models and Guard properties.

<table>
<thead>
<tr>
<th>Error Model Type [independent]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>error model independent</strong></td>
</tr>
<tr>
<td><strong>features</strong></td>
</tr>
<tr>
<td>Error_Free: initial error state;</td>
</tr>
<tr>
<td>Failed: error state;</td>
</tr>
<tr>
<td>Fail: error event</td>
</tr>
<tr>
<td>{Occurrence =&gt; poisson λ};</td>
</tr>
<tr>
<td>Restart: error event</td>
</tr>
<tr>
<td>{Occurrence =&gt; poisson μ};</td>
</tr>
<tr>
<td>FailedVisible: out error propagation</td>
</tr>
<tr>
<td>{Occurrence =&gt; fixed p};</td>
</tr>
<tr>
<td>end independent;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Error Model Implementation [independent.general]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>error model implementation</strong></td>
</tr>
<tr>
<td>independent.general</td>
</tr>
<tr>
<td><strong>transitions</strong></td>
</tr>
<tr>
<td>Error_Free-&gt;[Fail]-&gt;Failed;</td>
</tr>
<tr>
<td>Failed-&gt;[Restart]-&gt;Error_Free;</td>
</tr>
<tr>
<td>Failed-&gt;[out FailedVisible]-&gt;Failed;</td>
</tr>
<tr>
<td>end independent.general;</td>
</tr>
</tbody>
</table>

Figure 1: Two-state Error Model

4. Guidelines for Modeling Dependability

In order to analyze the dependability of an application at architectural level, it is necessary to enrich the architectural model with dependability-related information relevant to the targeted measure(s). Generally, dependability models include fault assumptions, stochastic parameters for the system, description of recovery and fault tolerance mechanisms, and characteristics of phases in a phased-mission system. An AADL user describes a system’s architecture in the AADL and annotates this architectural model with error models containing relevant dependability-related information.

We first present the role of AADL operational modes and mode transitions, then discuss the use of modes versus error states, before presenting mechanisms for representing the logic connecting error states to modes.

4.1 What Operational Modes are Good for ?

An operational mode is an operational state of an AADL component. Exactly one mode is the initial mode. A component is in one mode at a time. Mode transitions model
Software Dependability Modeling Using AADL

317

dynamic operational behavior, i.e., switching between configurations of subcomponents and connections. Mode-specific property values characterize quality attributes for each operational mode. A mode transition may be triggered by: (1) an out (or in out) event port of a subcomponent of the component declaring the modes; (2) an in (or in out) event port of the component itself; (3) a local event port.

A mode transition is triggered by any event that arrives through the port named in the mode transition. A mode transition may list multiple event ports as its trigger condition. In such a case, an event through any of the ports triggers the transition (an or logic is assumed). For dependability analyses, more advanced event-based mode transition conditions (reflecting e.g., voting protocols) can be specified through Guard_Event and Guard_Transition properties, as presented in Section 4.3. The state machine formed by modes and mode transitions in a system or system component implementation must be deterministic.

4.2 Operational Modes vs. Error States

Modes of operation in phased-mission systems, as well as fault-tolerant configurations are modeled by AADL modes.

Operational modes in phased-mission systems model configurations representative of different phases in a mission. For example, in the case of an aircraft model, one may distinguish between the takeoff, cruise and landing phases. During each phase, the system would have a particular configuration with active components and connections. Also, different types of faults may affect the system in different phases.

Mode-specific fault-tolerant system configurations reflect the fault tolerance strategy chosen for the system or for particular parts of the system. For example, a fault-tolerant system formed of three components may have a nominal operational mode corresponding to a triple-redundancy configuration and another degraded operational mode corresponding to a duplex configuration.

Usually, phased-mission systems also need modes to represent fault tolerance mechanisms. In the AADL, this nesting of modes is captured by phased-mission modes in a component, which is a subcomponent of a system component whose modes represent alternative configuration of its redundant subcomponents.

The difference between modes and error states lies primarily in their semantics. Error states result from occurrences of error events (faults, repair events) while modes represent operational states of the system that may be totally independent of the occurrence of error events.

4.3 Connecting Error States to Modes

The AADL allows us to model logical error states separate of the operational mode of the running application. It also allows connecting logical error states and operational modes by translating logical error states into actions (under the form of architectural events) on the running system through Guard_Event properties. Guard_Event properties map error state configurations into architectural events that are sent through ports and thus may affect the behavior of receiving components by triggering mode transitions. An architectural event arriving through a port will unconditionally trigger a mode transition.

Sometimes one may need to constrain a mode transition in a system component to reflect specific conditions such as a voting protocol to decide on fault handling. This can be achieved through the use of Guard_Transition properties associated with mode
transitions and specifying mode transition logic expressions overriding the default or condition on events arriving through ports named in the mode transition. Guard_Event and Guard_Transition properties can be used as advanced decision mechanisms that drive reconfiguration strategies.

5. Guidelines for Modeling Dependability

In this section, we consider a fault-tolerant duplex system that uses the hot standby redundancy scheme. We present successively, in sections 5.1, 5.2 and 5.3 patterns for three architectural implementations of this system. Patterns are reusable AADL dependability models that may be instantiated directly in a system instance model or that may be extended to form other patterns. The three patterns presented in this paper extend the architectural pattern of Figure 2 that models a generic duplex system, with no assumption regarding the fault-tolerance strategy.

The generic duplex system pattern contains two identical active components, Comp1 and Comp2, modeled as threads. Both threads process the event and data input stream received by the redundant system through the port sysInput but only one components output is made visible as output of the redundant system through the out event data port sysOutput. The three hot stand-by redundancy patterns presented in this paper differ in terms of their error detection mechanisms, as follows.

![Figure 2: Architectural Pattern for Duplex System](image)

- **Figure 2: Architectural Pattern for Duplex System**

In the first pattern the error detection is achieved by means of intermediate checkpoints between the two components. The direction of the data flow is from the primary component to the one that is in standby. The redundant system has two operational modes, one in which Comp1 is the primary and another in which Comp2 is the primary. The component in standby monitors the checkpoints sent by the primary and decides to take over and change the operational mode of the system, if it detects a failure.

In the second pattern, the error detection is achieved by a separate monitoring and control component, the Controller. The outputs of both active components are connected to the output of the system, sysOutput, but only one component provides the output at a given time instant. This is modeled by two modes associated with each component. Only the component in mode primary sends data to the output of the system. The Controller initiates the mode transitions.

The third pattern models error detection by mutual observation of outputs. The outputs of both active components are connected to the output of the system. Each of the two active components monitors the output of its sibling and decides whether to provide the output or not.

In the first pattern (detection by intermediate checkpoints), modes are represented at the systems level. In the two other patterns, modes are represented at the component level.
Each of the three patterns is formed of an AADL architectural model and of error model annex subclauses that associate the dependability-related information to the components of the architectural model. These subclauses may be further refined during the development cycle to detail the internal behavior of components and component-specific occurrence of propagations.

5.1 Detection by Intermediate Checkpoints

Figure 3 shows the AADL graphical architectural model for this system implementation, using the AADL graphical notation. The system has two operational modes. In mode Comp1Primary, Comp1’s output is made visible as output of the system. In mode Comp2Primary, Comp2’s output is made visible as output of the system. Thus, the connection from Comp1 to the output event data port sysOutput of the system is active in mode Comp1Primary while the connection from Comp2 to the output event data port sysOutput of the system is active in mode Comp2Primary. Based on the input from the other component and on its own state, a component in standby can decide to take over by initiating a mode transition. Thus, the transition from Comp1Primary to Comp2Primary is triggered by the output event port IAmPrim of Comp2. If both components fail successively and if their failures are detectable, the first one restarted becomes the primary.

5.2 Detection by a Separate Controller

The system modeled in Figure 4 consists of two identical active threads (Comp1 and Comp2) and one controller component (Controller). Each of the active components can be in one of two modes: primary and standby. When a component is in primary mode, it provides the service expected from the redundant system. The Controller monitors the two components. If it detects the failure of the one in primary mode, it initiates a mode switch in each component, so that the one that is Error Free continues to provide the service. Also, if it detects the failure of both components, it waits until one of them becomes operational and orders it to go to primary mode.

5.3 Detection by Mutual Observation

Figure 5 shows the architectural model of this system. Each of the two active components can be in one of these three modes: primary, standby and reboot. Initially, one component is in primary mode while the other one is in standby. When a component is in
**primary** mode, it provides the service expected from the redundant system. The two components observe each others outputs. Based on these observations and on its own state, each component decides whether it must be the sender of output. When a failure occurs in a component, the component goes to **reboot** mode. If the failed component was in **primary** mode, the other component should take over so that the service expected continues to be provided. If both components fail one after the other, the first one restarted becomes the **primary**.

In the two previous patterns, the mode transitions of a component or system are controlled by its subcomponents or by a separate controller. In this pattern, we use self-managing components that control their own mode transitions. This is modeled by local event ports (represented as dotted ovals) triggering the mode transitions.

The two components architectural models are identical except for their initial modes, *i.e.*, one is initially in **primary** mode while the other one is in **standby** mode.

![Mutual Observation (graphical)](image)

**Figure 5:** Mutual Observation (graphical)

**Concluding Remarks**

In the previous sections we assume that all components (*Comp1, Comp2, Controller*) have the same behavior in the presence of faults (represented by the error model of Figure 1). More complex behavior can be considered for each component, by customizing the patterns. This is achieved by changing the *Model* property in the error model annex subclause associated with a particular component. Other patterns, modeling fault tolerance schemes and impacting the dependability of the system, are presented in [25].

**6. Example for Application**

We illustrate the use of AADL architectural patterns for dependability analyses on a safety-critical subsystem of the French Air Traffic Control System. This system has been studied in [26] using generalized stochastic Petri nets (GSPN) for comparing candidate architecture solutions, with respect to availability. The contribution of this paper is to show how to model it using the AADL fault tolerance patterns.

The system is formed of two fault-tolerant distributed software units that are in charge of processing respectively flight plans (FP) and radar data (RD). Two processors can host these units. We consider two candidate architectures for this subsystem, referred to as *Configuration1* and *Configuration2*, illustrated in Figure 6. Both of them use two
processors. The two components of each fault-tolerant subsystem are bound to separate processors. The difference between the two configurations lies only in the bindings of RD threads to processors. In Configuration1, the thread that initially delivers the service (Comp1) is bound to Processor2 while Comp2 is bound to Processor1. In Configuration2, the bindings are the other way round (i.e., Comp1 is bound to Processor1 while Comp2 is bound to Processor2). The thread that delivers the service in the FP exchanges data with the RD. Connections between threads bound to separate processors are bound to a bus whose failure causes the failure of the RD.

![Configuration1 and Configuration2](image)

Figure 6: Candidate Architectures

For both configurations we investigate two fault tolerance policies, FTK and FTS, defined as follows. After the failure of the software replica that delivers the service, if the other one is error free, the two software replicas switch roles. Then, the failed replica is restarted. FTK and FTS differ by the role assigned to the replicas after the restoration of the failed replica:
- if the FTK fault tolerance policy applies, the two replicas keep their current roles,
- if the FTS fault tolerance policy applies, the replicas switch roles (i.e., the system has a nominal configuration, and is brought back to this configuration after failure and restoration of the principal replica).

Figure 7 presents both candidate architectures using the AADL graphical notation. For the sake of clarity, we show the thread binding configurations in Figure 7 (a) and the bus and the connection bindings to the bus in Figure 7 (b). We assume that the error detection is achieved through intermediate checkpoints (pattern presented in Section 5.1).

The modeling effort is limited. We need to instantiate the chosen pattern, to connect together the two instances and to bind the threads to processors. Besides these rather simple actions, we need to associate an error model annex subclause with the bus connecting the two processors. The error model annex subclauses may be further customized, to consider particular reconfiguration strategies.

The AADL models of the two candidate architectures presented in Figure 9 are transformed into GSPN that are not shown in this paper due to space limitations (see 10) for details about the transformation process).

Subsection 6.1 compares the four alternatives by setting all model parameters except for the failure rate of the bus, \( \lambda_b \). Subsection 6.2 compares them with respect to the fault-tolerance policy for the FP subsystem.
6.1 Comparison with respect to the Failure Rate of the Bus

Figure 8 provides, as an example of result obtained from the GSPN processing, the unavailability of the two candidate architectures. The varying parameter $\lambda_b$ is the occurrence rate of a bus failure. $\lambda_b \leq 10^{-6}$/h corresponds to a redundant bus.

Figure 8: Unavailability of RD with respect to the Bus Failure Rate, $\lambda_b$. 

<table>
<thead>
<tr>
<th>$\lambda_b$</th>
<th>Configuration1, FTK</th>
<th>Configuration1, FTS</th>
<th>Configuration2, FTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1E-8</td>
<td>21 min 01s</td>
<td>21 min 17s</td>
<td>20 min 49s</td>
</tr>
<tr>
<td>1E-7</td>
<td>21 min 01s</td>
<td>21 min 17s</td>
<td>20 min 49s</td>
</tr>
<tr>
<td>1E-6</td>
<td>21 min 11s</td>
<td>21 min 36s</td>
<td>20 min 49s</td>
</tr>
<tr>
<td>1E-5</td>
<td>22 min 11s</td>
<td>24 min 39s</td>
<td>20 min 49s</td>
</tr>
<tr>
<td>1E-4</td>
<td>24 min 14s</td>
<td>54 min 40s</td>
<td>20 min 49s</td>
</tr>
</tbody>
</table>
The unavailability of the FP subsystem is not influenced by $\lambda_b$, the failure rate of the bus, since a broken connection from RD to FP does not cause the failure of FP. The unavailability of FP is of 2h42min/year. On the other hand, the unavailability of the RD subsystem is influenced by $\lambda_b$, as a broken connection from FP to RD causes the failure of RD. $\lambda_b$ has little influence on the unavailability for Configuration 2 and FTK, because in these cases, the communication from FP to RD is not bound to the bus most of the time. For Configuration 1 and FTS, the impact is important when $\lambda_b \geq 10^{-5}$/h. From a practical point of view, if $\lambda_b \geq 10^{-5}$/h, it is recommended to use Configuration 2 or FTK.

### 6.2 Comparison with respect to the FT Policy

Figure 9 shows that both for FP and RD, with the fault-tolerance policy FTK, the unavailability is not influenced by the use of Configuration 1 or Configuration 2. Actually, if the system is not reconfigured to find its nominal configuration after the failure and restoration of a FP replica, it will spend as much time in Configuration 1 as in Configuration 2. For RD in Configuration 1, the unavailability is higher for FTS than for FTK while in Configuration 2, the unavailability is higher for FTK than for FTS. Thus, we may conclude that for Configuration 1 the best fault-tolerance policy is FTK while for Configuration 2 the best is FTS.

<table>
<thead>
<tr>
<th>Flight Processing unit (FP)</th>
<th>Radar Data Processing unit (RD)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Configuration 1</strong></td>
<td><strong>Configuration 1</strong></td>
</tr>
<tr>
<td>FTK</td>
<td>2h42min</td>
</tr>
<tr>
<td>FTS</td>
<td>2h42min</td>
</tr>
<tr>
<td><strong>Configuration 2</strong></td>
<td><strong>Configuration 2</strong></td>
</tr>
<tr>
<td></td>
<td>2h41min</td>
</tr>
</tbody>
</table>

Figure 9: Unavailability with respect to the Fault-tolerance Policy

### 4. Conclusion

Performing several analyses on a same architectural model is particularly interesting for software engineers as having qualitative and quantitative information about a candidate architecture allows making architectural tradeoffs.

The AADL is a mature industry-standard well suited to address quality attributes. Model reusability is an essential issue in the context of complex safety-critical evolvable applications. This paper illustrated its use for dependability modeling of fault-tolerant systems. We presented patterns for modeling fault-tolerant duplex applications and we showed that they enhance the reusability and the understandability of the model. Also, we showed a pattern-based example used for evaluating the availability of two candidate architectures for a subsystem of the French Air Traffic Control System.

**Acknowledgement:** We are indebted to Peter Feiler from the Carnegie Mellon Software Engineering Institute, with whom we did a part of our work related to dependability modeling based on AADL.

**References**


**Ana-Elena Rugina** holds a Ph.D. degree since November 2007. She carried out her research in the Dependable Computing and Fault Tolerance research group of LAAS-CNRS. She also collaborated with Peter Feiler at the Carnegie Mellon Software Engineering Institute. Her Ph.D. activities focused on dependability modeling and analysis using AADL. She has joined ASTRIUM Satellites in 2007, where she is involved in on-board software R&D projects related to the introduction of new technologies in space applications.

**Karama Kanoun** is Directeur de Recherche at LAAS-CNRS, heading the Dependable Computing and Fault Tolerance Research Group. Her research interests include modeling and evaluation of computer system dependability considering hardware as well as software, and dependability benchmarking. She was Visiting Professor at the University of Illinois, Urbana Champaign, USA in 1998. She has co-directed the production of a book on Dependability Benchmarking, (Wiley and IEEE Computer Society, 2008).

**Mohamed Kaâniche** is Directeur de Recherche at LAAS-CNRS, in the Dependable Computing and Fault Tolerance Research Group. His research activities focus on the dependability and security evaluation of fault-tolerant computing systems and critical infrastructures based on analytical modeling and experimental measurement approaches. He was a Visiting Research Assistant Professor at the University of Illinois at Urbana Champaign, USA, in 1997.